QUANTUM VARIATIONAL MONTE CARLO STUDIES OF SYSTEMS OF CONFINED FERMIONS

by

Christian Fleischer

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2017

Contents

Chapter 1

Introduction

Part I

Theory

Chapter 2

Variational Monte Carlo

The Variational Monte Carlo method the main computational method used in this thesis. It is based on the variational principle in quantum mechanics. The variational principle states that, given a Hamiltonian H and a trial wave function ψ_T, the expectation value $\langle H \rangle$ defined as [6]

$$E[H] = \langle H \rangle = \frac{\int d\mathbf{R} \Psi_T^*(\mathbf{R}, \boldsymbol{\alpha}) H(\mathbf{R}) \Psi_T(\mathbf{R}, \boldsymbol{\alpha})}{\int d\mathbf{R} \Psi_T^*(\mathbf{R}, \boldsymbol{\alpha}) \Psi_T(\mathbf{R}, \boldsymbol{\alpha})} \tag{2.0.1}$$

is an upper bound to the ground state energy E_0 of the Hamiltonian, i.e.

$$E_0 \leq \langle H \rangle. \tag{2.0.2}$$

Here our trial wave function is dependant on some variational parameters $\boldsymbol{\alpha}$, and the goal of the VMC method is to vary these parameters until we find the lowest possible value of $\langle H \rangle$ in order to get an estimate for the ground state energy E_0. In general, the integrals we have to compute to find $\langle H \rangle$ are multi-dimensional ones, so using traditional integration methods like Gauss-Legendre is too computationally expensive. Therefore we turn to Monte Carlo methods.

2.1 Metropolis Sampling

2.1.1 Monte Carlo Integration with Brute Force Metropolis Sampling

This description of Monte Carlo integration and the Metropolis algorithm follows Ref. [4]. For a given trial wave function we define a probability distribution function (PDF)

$$P(\mathbf{R}) = \frac{|\Psi_T(\mathbf{R})|^2}{\int |\Psi_T(\mathbf{R})|^2 d\mathbf{R}} \tag{2.1.1}$$

Together with the local energy given in Eq.(5.4.1) we have that the approximation to the expectation value of the Hamiltonian is

$$\begin{aligned} E[H(\alpha)] &= \int P(\mathbf{R}) E_L(\mathbf{R}) d\mathbf{R} \\ &\approx \frac{1}{N} \sum_{i=1}^{N} P(\mathbf{R}_i, \boldsymbol{\alpha}) E_L(\mathbf{R}_i, \boldsymbol{\alpha}), \end{aligned} \tag{2.1.2}$$

where N is the number of Monte Carlo samples, $\mathbf{R_i}$ are the positions of the particles at step i and $\boldsymbol{\alpha}$ are the variational parameters.

The Metropolis algorithm is used to sample the probability distribution by a stochastic process. We define $\mathbf{P}_i^{(n)}$ to be the probability for finding the system in the state i at step n, and the algorithm is then

- Sample a possible new state j with some probability $T_{i\to j}$
- Accept the new state with probability $A_{i\to j}$ and use it as the next sample, or with probability $1 - A_{i\to j}$, reject the move and use the original state i as sample again.

We want to ensure that $\mathbf{P}_i^{(n\to\infty)} \to p_i$, so that regardless of the initial distribution, the method converges to the correct distribution. To ensure this we demand that the transition probability T and the acceptance probability A, fulfill the detailed balance requirement

$$\frac{A_{i\to j}}{A_{j\to i}} = \frac{p_i T_{i\to j}}{p_j T_{j\to i}}, \tag{2.1.3}$$

where $p_i = P(\mathbf{R}_i)$ and $p_j = P(\mathbf{R}_j)$. The Metropolis algorithm then uses the following ratio of probabilities to determine whether or not to accept a move

$$\frac{p_j}{p_i} = \frac{T_{i\to j} A_{i\to j}}{T_{j\to i} A_{j\to i}} \tag{2.1.4}$$

When using the Metropolis algorithm we can either use brute force or importance sampling. If we use the brute force Metropolis algorithm, we assume that $T_{i\to j} = T_{j\to i}$. Then the ratio used by the Metropolis algorithm is only dependant on the acceptance probabilities, i.e. the ratio is given by

$$w = \frac{p_j}{p_i} = \frac{A_{i\to j}}{A_{j\to i}} = \frac{|\Psi_T(\mathbf{R}_j)|^2}{|\Psi_T(\mathbf{R}_i)|^2} \tag{2.1.5}$$

The algorithm for estimating the ground state energy given a set of variational parameters α is then

- Fix the number of Monte Carlo steps and create an initial state $\mathbf{R}$ using the given variational parameters $\boldsymbol{\alpha}$. Set a step size $\Delta\mathbf{R}$ and calculate $|\Psi_T^{\alpha}(\mathbf{R})|^{\mathbf{2}}$.
- Initialize the local energy and start the Monte Carlo calculations.
  - Choose a random particle and update its position in order to create a trial state: $\mathbf{R}_p = \mathbf{R} + r\Delta\mathbf{R}$ where r is a random variable $r \in [0, 1]$

  - Calculate $|\Psi_T^{\alpha}(\mathbf{R_p})|^{\mathbf{2}}$ and use the Metropolis algorithm to accept or reject the move by calculating the ratio w. If $w \geq s$, where s is a random number $s \in [0, 1]$, the new state is accepted. Otherwise we keep the old state.
  - If the new state is accepted, then we set $\mathbf{R} = \mathbf{R}_p$.
  - Update the local energy.

- Finish and compute the final estimate for the ground state energy.

2.1.2 Metropolis-Hastings Algorithm (Importance Sampling)

For the description of the Metropolis-Hastings Algorithm we again follow Ref. [4]. When using importance sampling the walk in coordinate space is biased by the trial wave function, so the walker is more likely to move towards regions where the trial wave function is large. The trajectory in coordinate space is generated by the Fokker-Planck equation and the Langevin equation. To find the new positions in coordinate space we solve the Langevin equation using Euler's method. The Langevin equation

$$\frac{\partial x(t)}{\partial t} = DF(x(t)) + \eta, \tag{2.1.6}$$

where η is a random variable and D is the diffusion coefficient, gives the new position

$$y = x + DF(x)\Delta t + \xi\sqrt{\Delta t}, \tag{2.1.7}$$

where ξ is gaussian random variable, Δt is a chosen time step and $F(x)$ is the function responsible for drifting the walker towards regions where the wave function is large. Δt is treated as a parameter and $\Delta t \in [0.001, 0.01]$ generally yields fairly stable values of the ground state energy. The diffusion coefficient D is equal to $1/2$ and comes from the $1/2$ factor in the kinetic energy operator. The Fokker-Planck equation describes the process of isotropic diffusion characterized by a time-dependent probability density $P(x, t)$ and is given by

$$\frac{\partial P}{\partial t} = \sum_i D\frac{\partial}{\partial \mathbf{x}_i}\left(\frac{\partial}{\partial \mathbf{x}_i} - \mathbf{F}_i\right)P(\mathbf{x}, t), \tag{2.1.8}$$

where $\mathbf{F_i}$ is the i^{th} component of the drift term. By setting the left hand side to zero we obtain the convergence to a stationary probability density. For the resulting equation to be satisfied all the terms of the sum have to be equal to zero, i.e.

$$\frac{\partial^2 P}{\partial \mathbf{x}_i^2} = P\frac{\partial}{\partial \mathbf{x}_i}\mathbf{F}_i + \mathbf{F}_i\frac{\partial}{\partial \mathbf{x}_i}P. \tag{2.1.9}$$

The drift vector $\mathbf{F}$ should be on the form $\mathbf{F} = g(\mathbf{x})\frac{\partial P}{\partial \mathbf{x}}$, which gives us

$$\frac{\partial^2 P}{\partial \mathbf{x}_i^2} = P\frac{\partial g}{\partial P}\left(\frac{\partial P}{\partial \mathbf{x}_i}\right)^2 + Pg\frac{\partial^2 P}{\partial \mathbf{x}_i^2} + g\left(\frac{\partial P}{\partial \mathbf{x}_i}\right)^2. \tag{2.1.10}$$

The condition of stationary density requires that the terms cancel each other out, and that is only possible if $g = \frac{1}{P}$. This leads to

$$\mathbf{F} = 2\frac{1}{\Psi_T}\nabla\Psi_T, \tag{2.1.11}$$

which is the so-called *quantum force*. By pushing the walker towards regions where the trial wave function is large, this term increases the efficiency of the simulation compared to the brute force Metropolis algorithm where the walker has the same probability of moving in every direction.

From the Fokker-Planck equation we get a transition probability given by the Green's function

$$G(y, x, \Delta t) = \frac{1}{(4\pi D\Delta t)^{3N/2}} \exp(-(y - x - D\Delta t F(x))^2/4D\Delta t), \tag{2.1.12}$$

which means that the ratio used in the Metropolis algorithm

$$w = \frac{|\Psi_T(\mathbf{R}_j)|^2}{|\Psi_T(\mathbf{R}_i)|^2} = q(y, x) = \frac{|\Psi_T(y)|^2}{|\Psi_T(x)|^2}, \tag{2.1.13}$$

is now replaced by

$$q(y, x) = \frac{G(x, y, \Delta t)|\Psi_T(y)|^2}{G(y, x, \Delta t)|\Psi_T(x)|^2}. \tag{2.1.14}$$

Using this ratio and Eq.(2.1.7), the algorithm is called the Metropolis-Hastings algorithm.

2.1.3 Metropolis Sampling with a Slater Determinant

When our trial wave function contains a Slater determinant we can manipulate it to improve the performance of the code by avoiding calculating the entire determinant at every Metropolis step. The ratio used in Metropolis sampling is given by (Ref. [4])

$$R = \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} \frac{\Psi_C^{\text{new}}}{\Psi_C^{\text{old}}}, \tag{2.1.15}$$

where $|\hat{D}|$ is the Slater determinant and Ψ_C is the correlation part of the wave function, while "new"and "old" refer to the position before and after a proposed move. If we move only one electron at the time, only a single row in the Slater determinant changes. By doing the calculations in section A.2(Appendix A.2), we can calculate the Slater determinant part of R using the following formula

$$R_{SD} = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}), \tag{2.1.16}$$

where $\phi_j(\mathbf{r}_i^{\text{new}})$ are the single particle wave functions evaluated at the new position, and $d_{ji}^{-1}(\mathbf{r}^{\text{old}})$ are the elements on the i-th column of the inverse Slater matrix $\hat{D}^{-1}$. In addition we need to maintain the inverse matrix by using an updating algorithm whenever a move is accepted. This updating algorithm is also covered in section A.2,

and the equations used are

$$d_{kj}^{-1}(\mathbf{r}^{\text{new}}) = \begin{cases} d_{kj}^{-1}(\mathbf{r}^{\text{old}}) - \frac{d_{ki}^{-1}(\mathbf{r}^{\text{old}})}{R} \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{new}}) d_{lj}^{-1}(\mathbf{r}^{\text{old}}) & \text{if } j \neq i \\ \frac{d_{ki}^{-1}(\mathbf{r}^{\text{old}})}{R} \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{old}}) d_{lj}^{-1}(\mathbf{r}^{\text{old}}) & \text{if } j = i \end{cases}$$

For the correlation part of R we simply have

$$R_C = \frac{\Psi_C^{\text{new}}}{\Psi_C^{\text{old}}} = \prod_{i<j}^{N} \exp\left(f_{ij}^{\text{new}} - f_{ij}^{\text{old}}\right) \tag{2.1.17}$$

$$= \exp\left(\sum_{i<j}^{N} f_{ij}^{\text{new}} - f_{ij}^{\text{old}}\right), \tag{2.1.18}$$

where

$$f_{ij} = \frac{a r_{ij}}{(1 + \beta r_{ij})}. \tag{2.1.19}$$

For $i, j \neq k$, where k is the index of the moved electron, we get $f_{ij}^{\text{new}} - f_{ij}^{\text{old}} = 0$. Therefore we can simplify the expression to

$$\begin{aligned} R_C &= \exp\left(\sum_{i=1}^{k-1} f_{ik}^{\text{new}} - f_{ik}^{\text{old}} + \sum_{j=k+1}^{N} f_{kj}^{\text{new}} - f_{kj}^{\text{old}}\right) \\ &= \exp\left(\sum_{i=1, i\neq k}^{N} f_{ik}^{\text{new}} - f_{ik}^{\text{old}}\right). \end{aligned} \tag{2.1.20}$$

2.1.4 Slater Determinant and Quantum Force

As described in (importance sampling section), when using importance sampling, we need to find the *quantum force* F, which is given by

$$\mathbf{F} = 2\frac{1}{\Psi_T} \nabla \Psi_T. \tag{2.1.21}$$

We therefore need the gradient of the wave function. For the Slater determinant part, we can again use that moving only one particle changes only one row in the determinant, in order to reduce computation time. As described in section A.2, we then get

$$\frac{\vec{\nabla}_i |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \vec{\nabla}_i \phi_j(\mathbf{r}_i) d_{ji}^{-1}(\mathbf{r}), \tag{2.1.22}$$

Following the calculations in A.2 we also get that the gradient for the correlation part is given by

$$\frac{\nabla_k \Psi_C}{\Psi_C} = \sum_{j \neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{a}{(1+\beta r_{kj})^2}. \tag{2.1.23}$$

2.2 Splitting the Slater Determinant

This section on splitting the Slater determinant follows Ref. [4]. The Slater determinant is on the form

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, , \mathbf{r}_3, \mathbf{r}_4, \alpha, \beta, \gamma, \delta) = \frac{1}{\sqrt{4!}} \begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4) \\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}, \tag{2.2.1}$$

which is zero because the spatial wave functions for the spin up and spin down states are equal. We rewrite the Slater determinant as the product of a spin up Slater determinant and a spin down Slater determinant and get

$$\begin{aligned} \Phi(\mathbf{r}_1, \mathbf{r}_2, , \mathbf{r}_3, \mathbf{r}_4, \alpha, \beta, \gamma, \delta) = &\det\uparrow(1,2)\det\downarrow(3,4) - \det\uparrow(1,3)\det\downarrow(2,4) \\ &- \det\uparrow(1,4)\det\downarrow(3,2) + \det\uparrow(2,3)\det\downarrow(1,4) \\ &- \det\uparrow(2,4)\det\downarrow(1,3) + \det\uparrow(3,4)\det\downarrow(1,2), \end{aligned} \tag{2.2.2}$$

where

$$\det\uparrow(1,2) = \frac{1}{\sqrt{2}} \begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) \end{vmatrix}, \tag{2.2.3}$$

and

$$\det\downarrow(3,4) = \frac{1}{\sqrt{2}} \begin{vmatrix} \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}. \tag{2.2.4}$$

This still gives a total determinant equal to zero. However, we want to avoid summing over spin variables when the interaction is independent of spin. With regards to the variational energy we can use the following approximation to this Slater determinant

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, , \mathbf{r}_3, \mathbf{r}_4, \alpha, \beta, \gamma, \delta) \propto \det\uparrow(1,2)\det\downarrow(3,4), \tag{2.2.5}$$

and in general we can use the approximation

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N) \propto \det\uparrow \det\downarrow, \tag{2.2.6}$$

where the approximation to the Slater determinant is the product of a spin up part with the electrons with spin up only and a spin down part with the electrons with spin down. This ansatz is not antisymmetric when exchanging electrons with opposite spins, however the expectation value we get for the energy is the same as we get for the full

Slater determinant, provided that the Hamiltonian is independent of spin. By factorizing the full determinant $|\hat{D}|$ into two smaller ones we can reduce the computation time. We identify the two determinants with an $\uparrow$ and a $\downarrow$

$$|\hat{D}| = |\hat{D}|_{\uparrow} \cdot |\hat{D}|_{\downarrow} \tag{2.2.7}$$

Combining the dimensionality of the smaller determinants yields the dimensionality of the full determinant. Doing the factorization allows us to calculate the ratio R as well as update the inverse Slater matrix separately for the two determinants

$$\frac{|\hat{D}|^{\text{new}}}{|\hat{D}|^{\text{old}}} = \frac{|\hat{D}|^{\text{new}}_{\uparrow}}{|\hat{D}|^{\text{old}}_{\uparrow}} \cdot \frac{|\hat{D}|^{\text{new}}_{\downarrow}}{|\hat{D}|^{\text{old}}_{\downarrow}}, \tag{2.2.8}$$

which reduces the computation time by a constant factor. The time reduction is greatest when the system has an equal amount of spin up and spin down electrons, so that both of the factorized determinants are half the size of the full determinant. This is the case for the ground state of closed-shell systems (i.e. systems with $2, 6, 12, 20, \ldots$ electrons filling up the $1, 2, 3, 4, \ldots$ lowest shells).

```
1  void ManyElectrons::setUpSlaterDet() {
2      // Function for setting up the Slater determinant at the begining of the
           simulation.
3      int n = 0;
4      int nx = 0;
5      int ny = 0;
6      m_quantumNumbers = zeros<mat>(m_halfNumberOfParticles, 2);
7      for (int p=0; p < m_halfNumberOfParticles; p++) {
8          m_quantumNumbers(p, 0) = nx;     m_quantumNumbers(p, 1) = ny;
9          if (ny == n) {
10             n++;
11             nx = n;
12             ny = 0;
13         }
14         else {
15             nx--;
16             ny++;
17         }
18     }
19
20     m_a = zeros<mat>(m_numberOfParticles, m_numberOfParticles);
21     int half = m_halfNumberOfParticles;
22     for (int i=0; i < m_numberOfParticles; i++) {
23         for (int j=0; j < m_numberOfParticles; j++) {
24             if ( ((i < half) && (j < half)) || ((i >= half) && (j >= half)) ) {
                   m_a(i,j) = 1./3; }
25             else { m_a(i,j) = 1.; }
26         }
27     }
28
29     m_spinUpSlater = zeros<mat>(m_halfNumberOfParticles, m_halfNumberOfParticles)
           ;
30     m_spinDownSlater = zeros<mat>(m_halfNumberOfParticles,
           m_halfNumberOfParticles);
31
32     for (int i=0; i < m_halfNumberOfParticles; i++) {
33         for (int j=0; j < m_halfNumberOfParticles; j++) {
34             nx = m_quantumNumbers(j, 0);
35             ny = m_quantumNumbers(j, 1);
36             double xSpinUp = m_system->getInitialState()->getParticles()[i]->
                   getPosition()[0];
```

```
37              double ySpinUp = m_system->getInitialState()->getParticles()[i]->
                    getPosition()[1];
38              double xSpinDown = m_system->getInitialState()->getParticles()[i+
                    m_halfNumberOfParticles]->getPosition()[0];
39              double ySpinDown = m_system->getInitialState()->getParticles()[i+
                    m_halfNumberOfParticles]->getPosition()[1];
40              m_spinUpSlater(i,j) = evaluateSingleParticleWF(nx, ny, xSpinUp,
                    ySpinUp);
41              m_spinDownSlater(i,j) = evaluateSingleParticleWF(nx, ny, xSpinDown,
                    ySpinDown);
42          }
43      }
44
45      m_spinUpSlaterInverse = m_spinUpSlater.i();
46      m_spinDownSlaterInverse = m_spinDownSlater.i();
47  }
```

Setting up the (Split) Slater Determinant in C++

2.3 The Steepest Descent Method (Gradient Descent)

We want to optimize the variational parameters to minimize the estimation of the ground state energy. To do this we use the Steepest Descent method, also called Gradient Descent[1]. We have two variational parameters we need to optimize; α, and β. To optimize the parameters we treat our estimate to the ground state energy as a function of the parameters. Steepest Descent is a way to minimize a function parameterized by some parameters, by updating the parameters in the opposite direction of the gradient of the function w.r.t. to the parameters. [13] At each step the move is proportional to a chosen step length γ, and for α, one step is then

$$\alpha_{i+1} = \alpha_i - \gamma \bar{E}_\alpha, \tag{2.3.1}$$

where

$$\bar{E}_\alpha = \frac{d\langle E_L[\alpha]\rangle}{d\alpha} \tag{2.3.2}$$

is the gradient. We then have that $E_L[\alpha_{i+1}] \leq E_L[\alpha_i]$. In order to find $\bar{E}_\alpha$ we also need the derivative of the trial wave function, which we define as

$$\bar{\psi}_\alpha = \frac{d\psi[\alpha]}{d\alpha}. \tag{2.3.3}$$

By using the chain rule and the hermiticity of the Hamiltonian we get an expression for the gradient $\bar{E}_\alpha$ (Ref. [1])

$$\bar{E}_\alpha = 2\left(\left\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]} E_L[\alpha]\right\rangle - \left\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]}\right\rangle \langle E_L[\alpha]\rangle\right), \tag{2.3.4}$$

so we need to compute

$$\left\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]} E_L[\alpha]\right\rangle, \tag{2.3.5}$$

[1]Steepest Descent and Gradient Descent are sometimes referred to as different methods in literature. We will use Steepest Descent to refer to the method used in this thesis.

and

$$\Big\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]} \Big\rangle, \tag{2.3.6}$$

in addition to $\langle E_L[\alpha]\rangle$. We do the same for β as well.

To optimize the parameters we first guess an initial value for each. Then, using few Monte Carlo cycles, we run the simulation and sample the expectation values Eq.(2.3.5), Eq.(2.3.6) and the expectation value for the ground state energy (as usual). Using the expectation values we calculate the gradient Eq.(2.3.4) and find new parameters using Eq.(2.3.1). We repeat this until we have found optimal parameters to some desired precision. Finally, we do a large-scale Monte Carlo simulation (many cycles) using the optimal parameters to find a good estimate to the ground state energy. We find the absolute value of the difference between a new value and an old value for each variational parameter. We then sum up these absolute values (one for each parameter) and stop the optimization when the sum is below a chosen tolerance, or when a set number of maximum iterations has been reached.

2.4 Statistical Error Estimation with Blocking

Here follows a brief explanation of the blocking method based on Ref. [4]. For a more detailed explanation consult the reference. At each Monte Carlo step in our simulation we sample a local energy and the mean of these samples $\langle H\rangle$ is our estimate for the ground state energy. If we assume that the n samples are uncorrelated our best estimate for the standard deviation of the mean $\langle H\rangle$ is given by

$$\sigma = \sqrt{\frac{1}{n}(\langle H^2\rangle - \langle H\rangle^2)}. \tag{2.4.1}$$

However, this is a too optimistic estimate of the error in our calculations because the samples are correlated. Therefore we need to rewrite our expression for the standard deviation to

$$\sigma = \sqrt{\frac{1+2\tau/\Delta t}{n}(\langle H^2\rangle - \langle H\rangle^2)}, \tag{2.4.2}$$

where τ is the correlation time, i.e. the time between a given sample and the next uncorrelated sample. Δt is the time between each sample. If $\Delta t \gg \tau$ the estimate in Eq.(2.4.1) still holds, however, usually $\Delta t < \tau$. When using the blocking method we divide the sequence of samples into blocks, and then calculate the mean and variance of each block separately. Finally we calculate the total mean and variance of all of the blocks. The size of the blocks has to be large enough that sample j of block i is not correlated with sample j of block $i+1$. For this, the correlation time τ would be a good choice, however, τ is too expensive to compute.

Instead we can plot the standard deviation as a function of block size. As long as the block size is so small that the blocks are correlated the standard deviation will increase with increasing block size. However, once the block size is large enough that the blocks are uncorrelated, we reach a plateau. Therefore, when the standard deviation stops increasing, the plateau value of the standard deviation will be a good estimate of the error in our results.

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

def readData(filename):

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

    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 = 1
    energies = readData("energiesN%i.dat" %N)

    deltaBlockSize = 100
    minBlockSize = 10
    maxBlockSize = 10000
    numberOfSizes = (maxBlockSize-minBlockSize)/deltaBlockSize + 1
    largestBlockSize = minBlockSize + (numberOfSizes-1)*deltaBlockSize #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.))
    plt.plot(blockSizes, standardDeviation)
    plt.xlabel("Block Size")
    plt.ylabel(r"Standard Deviation $\sigma$")
    plt.title("N=%i" %N)
    plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
    plt.show()
```

Blocking in Python

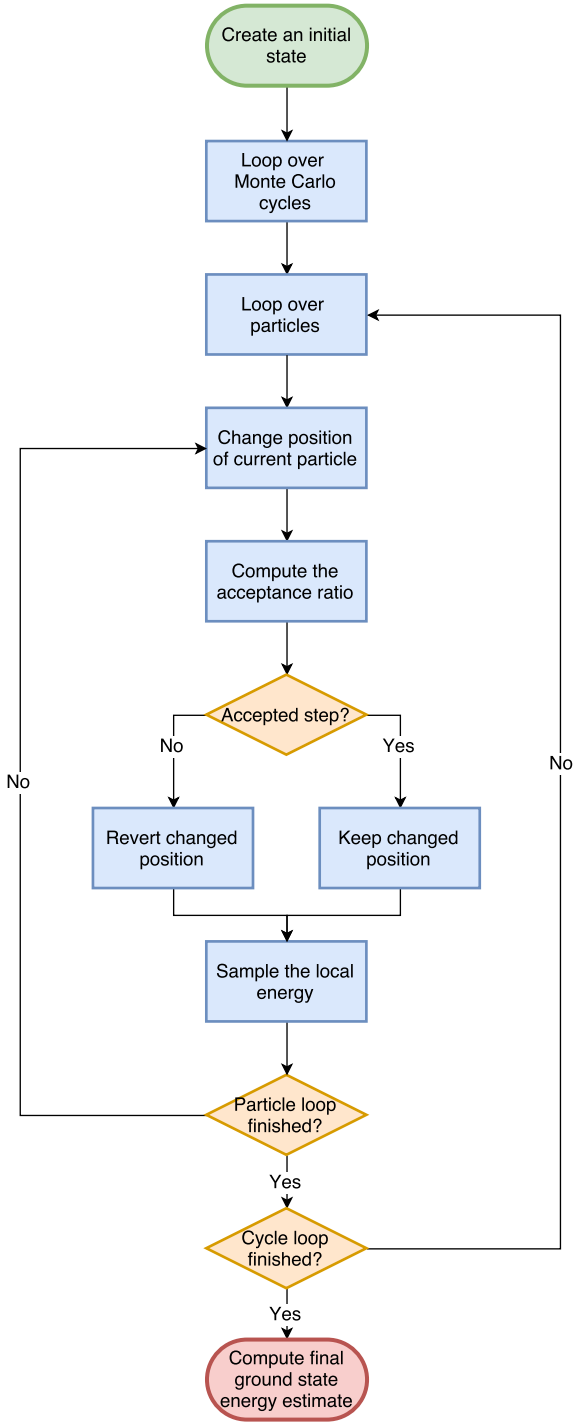


Figure 2.1: Flowchart of the Metropolis algorithm used in a variational Monte Carlo simulation. The variational parameters, number of Monte Carlo cycles, and various parameters (e.g. number of particles) are set before creating the initial state. The position change and acceptance ratio we use depend on whether we use the regular Metropolis algorithm (brute force), or the Metropolis-Hastings algorithm (importance sampling).

Chapter 3

Basis Functions

When we do Quantum Monte Carlo simulations we need a wave function for the system. A part of this wave function is made up by the single particle wave functions, i.e. the wave function we have if the system only has one particle and thus no particle-particle interactions. For the harmonic oscillator potential we have the fairly simple harmonic oscillator wave functions given in Eq. (5.3.2). However, for other potentials the single particle wave functions may not be as simple to implement. Instead we can approximate the single particle wave functions by using a linear combination with the simple harmonic oscillator functions as basis functions. We do this by first diagonalizing the single particle system for the given potential. This provides us with eigenvalues and eigenvectors for the single particle problem. We then take the inner product of the eigenvectors and the harmonic oscillator basis functions to find the overlap coefficients. Finally, in the QMC simulation we use the overlap coefficients and the harmonic oscillator basis functions to approximate the single particle wave functions for each particle.

3.1 Diagonalizing the Single Particle Problem

The potentials we focus on in this thesis are separable in the x, y and z directions. For one spatial dimension the time independent Schrödinger equation is on the form [9]

$$-\frac{\hbar^2}{2m}\frac{\partial^2\psi(x)}{\partial x^2} + V(x)\psi(x) = E\psi(x), \tag{3.1.1}$$

with eigenvectors $\psi_n(x)$ and eigenvalues E_n. The potential is $V(x)$. The reduced Plank constant $\hbar$ and the particle mass m are both set to 1 since we use natural units, which leaves us with

$$-\frac{1}{2}\frac{\partial^2\psi(x)}{\partial x^2} + V(x)\psi(x) = E\psi(x). \tag{3.1.2}$$

To find the eigenvalues and eigenvectors we follow Ref. [10] and start by discretizing this equation. Using the central finite difference scheme [7] for the second derivative we get

$$\frac{\partial^2\psi(x)}{\partial x^2} = \frac{\psi(x+h) - 2\psi(x) + \psi(x-h)}{h^2}, \tag{3.1.3}$$

where h is a chosen discretization step. We choose a min and max value for x and choose

a number of steps N so that

$$h = \frac{x_{\max} - x_{\min}}{N}. \tag{3.1.4}$$

We then have that an arbitrary value of x is

$$x_i = x_{\min} + ih \qquad i = 0, 1, 2, \ldots, N. \tag{3.1.5}$$

Inserting this into Eq.(3.1.2) we get the following Schrödinger equation for x_i

$$-\frac{\psi(x_i + h) - 2\psi(x_i) + \psi(x_i - h)}{2h^2} + V(x_i)\psi(x_i) = E\psi(x_i), \tag{3.1.6}$$

or, using a more compact notation

$$-\frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{2h^2} + V_i\psi_i = E\psi_i. \tag{3.1.7}$$

We define the diagonal matrix elements

$$d_i = \frac{2}{2h^2} + V_i = \frac{1}{h^2} + V_i, \tag{3.1.8}$$

and the off-diagonal matrix elements

$$e_i = -\frac{1}{2h^2}. \tag{3.1.9}$$

All of the off-diagonal matrix elements are equal, while the diagonal ones differ due to the potential. Using these definitions we can write the Schrödinger equation as

$$d_i\psi_i + e_{i-1}\psi_{i-1} + e_{i+1}\psi_{i+1} = E\psi_i, \tag{3.1.10}$$

which we can solve as a tridiagonal matrix eigenvalue problem in order to find Ψ_i and E:

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 & \ldots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \ldots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & d_{N-2} & e_{N-2} \\ 0 & \ldots & \ldots & \ldots & \ldots & e_{N-2} & d_{N-1} \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \ldots \\ \ldots \\ \ldots \\ \psi_{N-1} \end{pmatrix} = E \begin{pmatrix} \psi_1 \\ \psi_2 \\ \ldots \\ \ldots \\ \ldots \\ \psi_{N-1} \end{pmatrix} \tag{3.1.11}$$

These types of equations can easily and efficiently be solved by using an Armadillo function for C++ called `"eig_sym"`.

3.2 Finding the Overlap Coefficients

Now that we have found the eigenvectors $\psi_n(x)$ through diagonalization, we can find the overlap coefficients we need. The overlap coefficients are the inner product of the

eigenvectors and the basis functions we use [21], i.e.

$$C_{n',n} = \langle \psi_{n'} | \phi_n \rangle = \sum_{i=0}^{N-1} \psi_{n'}(x_i)\phi_n(x_i) \qquad n', n = 0, 1, 2, \ldots, \tag{3.2.1}$$

where $\phi_n(x)$ are the basis functions. For the eigenvectors, $\psi_{n'}(x)$, we refer to the quantum number as n', while for the basis functions we use n. Since they do not always have the same value we need to differentiate between them. N is the number of steps we chose when discretizing and x_i are the corresponding discretized positions. We use harmonic oscillator functions as the basis functions, which are on the form

$$\phi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} e^{-\frac{m\omega x^2}{2\hbar}} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right), \tag{3.2.2}$$

which, since we use natural units, we can simplify to

$$\phi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{\omega}{\pi}\right)^{1/4} e^{-\frac{\omega x^2}{2}} H_n\left(\sqrt{\omega}x\right). \tag{3.2.3}$$

ω is the harmonic oscillator frequency, and $H_n(x)$ are the Hermite polynomials. The Hermite polynomials are the solutions of the differential equation

$$\frac{d^2H(x)}{dx^2} - 2x\frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0, \tag{3.2.4}$$

where λ is a constant. These Hermite polynomials fulfill the following recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x), \tag{3.2.5}$$

with

$$H_0(x) = 1, \tag{3.2.6}$$

$$H_1(x) = 2x. \tag{3.2.7}$$

3.3 Approximating the Single Particle Wave Functions

When approximating the single particle wave function during the quantum Monte Carlo simulation, we do the reverse of what we did when finding the coefficients, however we do it for the specific particle whose single particle wave function we are approximating. The approximation to the single particle wave function in one dimension is given by

$$\psi_{n'}(x) = \sum_{n_x=0}^{\Lambda} C_{n',n_x}\phi_{n_x}(x), \tag{3.3.1}$$

where the loop is over eigenstates. Λ is the total number of eigenstates we're using, and the approximation improves with increasing Λ. n are the quantum numbers corresponding to the eigenstate and for one dimension $n = n_x$. However, when we have more dimensions, each eigenstate will have multiple quantum numbers, $n = (n_x, n_y, n_z)$ (one for each dimension). $\phi_{n_x}(x)$ are the basis functions given in Eq. (3.2.3). n' is here the

quantum number of the particle whose single particle wave function we are approximating, and x is the position of the particle. When we look at more than one dimension, each term, T, of the sum in Eq. (3.3.1) is a product on the form

$$T = C_{n',n_x}\phi_{n_x}(x)C_{n',n_y}\phi_{n_y}(y)C_{n',n_z}\phi_{n_z}(z). \tag{3.3.2}$$

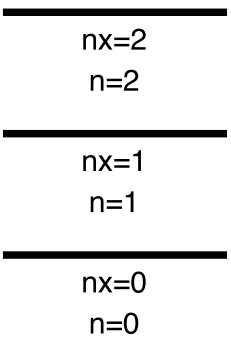


Figure 3.1: This figure shows the eigenstates for the three lowest energy levels in one dimension. Each line is one eigenstate, and a given eigenstate is on a higher energy level than those below it in the figure. The eigenstates are indexed by n, and nx is the quantum number corresponding to the eigenstate. In one dimension we only have one eigenstate per energy level and therefore we always have $nx = n$. However, as seen in Figure 3.2 this is not the case for higher dimensions.

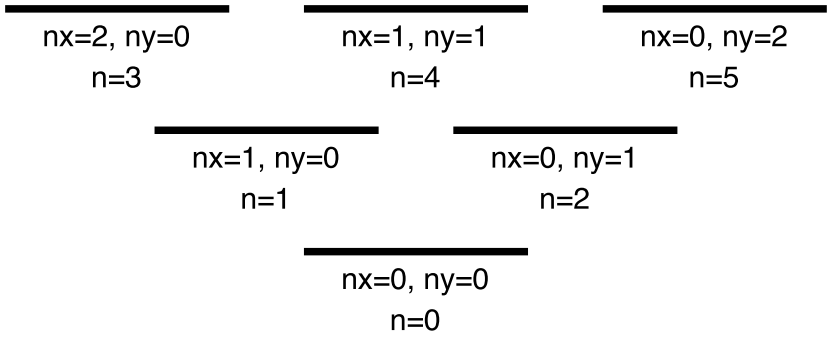


Figure 3.2: This figure shows the eigenstates for the three lowest energy levels in two dimensions. Each line is one eigenstate and a given eigenstate is on a higher energy level than those below it in the figure. The eigenstates are indexed by n. Each eigenstate has two quantum numbers nx and ny. Unlike in one dimension (Figure 3.1), when we have two dimensions we can have multiple eigenstates per energy level (e.g. eigenstates $n = 1$ and $n = 2$).

Chapter 4

Hartree-Fock

Doing Hatree-Fock calculations has not been a part of this thesis. However, it is the next step in improving the single particle wave function approximations the thesis is mainly focused on. This chapter includes an introduction to the Hartree-Fock method, and useful information for doing further work. The Hartree-Fock method is well described in [12], which this chapter is based upon. This method uses an algorithm to find an approximative expression for the ground state of a given Hamiltonian. First we need to define a single particle basis $\{\psi_\alpha\}$ (e.g. a single particle harmonic oscillator basis), so that

$$\hat{h}^{HF}\psi_\alpha = \epsilon_\alpha \psi_\alpha, \tag{4.0.1}$$

where $\hat{h}^{HF}$ is the Hartree-Fock Hamiltonian

$$\hat{h}^{HF} = \hat{t} + \hat{u}_{ext} + \hat{u}^{HF}. \tag{4.0.2}$$

The goal of the Hartree-Fock algorithm is to determine the single particle potential, $\hat{u}^{HF}$, so that we find a local minimum for

$$\langle \hat{H} \rangle = E^{HF} = \langle \Phi_0 | \hat{H} | \Phi_0 \rangle, \tag{4.0.3}$$

where we have a Slater determinant, Φ_0, as the ansatz for the ground state. As with variational Monte Carlo methods, the variational principle ensures that we approach the exact ground state energy, E_0, from above, i.e.

$$E^{HF} \geq E_0. \tag{4.0.4}$$

We use Hartree-Fock to get coefficients for a self-consistent single particle basis, which we can then use in the VMC calculations instead of the coefficients we got from diagonalizing simple potentials (as described in 3).

4.1 Hamiltonian

If we assume that the interacting part of the Hamiltonian can be approximated by a two-body interaction, we can write the Hamiltonian as

$$\hat{H} = \hat{H}_0 + \hat{H}_1 = \sum_{i=1}^{N} \hat{h}_0(x_i) + \sum_{i<j}^{N} V(r_{ij}), \tag{4.1.1}$$

where

$$\hat{H}_0 = \sum_{i=1}^{N} \hat{h}_0(x_i) = \sum_{i=1}^{N} \left(\hat{t}(x_i) + \hat{u}_{\text{ext}}(x_i)\right) \tag{4.1.2}$$

is the non-interacting part, and

$$\hat{H}_1 = \sum_{i<j}^{N} V(r_{ij}) \tag{4.1.3}$$

is the interacting part. $t(\hat{x_i})$ represents the kinetic energy of particle i, while $\hat{u}_{\text{ext}}(x_i)$ represents the one-body part of the potential energy, which can be approximated by a harmonic oscillator potential. $V(r_{ij})$ is the two-body potential between particles i and j.

Our Hamiltonian is invariant under permutation of two particles. If we define $\hat{P}$ as an operator which interchanges two particles, due to symmetries in our Hamiltonian, this operator will commute with the total Hamiltonian,

$$\left[\hat{H}, \hat{P}\right] = 0. \tag{4.1.4}$$

This means that the eigenfunction $\Psi_\lambda(x_1, x_2, \ldots, x_N)$ of our Hamiltonain is also an eigenfunction of $\hat{P}$, so that

$$\hat{P}_{ij}\Psi_\lambda(x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_N) = \beta\Psi_\lambda(x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_N), \tag{4.1.5}$$

where β is the eigenvalue of $\hat{P}$ and the ij suffix indicates that we permute particles i and j. Since we're looking at fermions, the Pauli principle states that the total wave function has to be antisymmetric, which yields the eigenvaule $\beta = -1$.

We approxiamte the exact eigenfunction with a Slater determinant,

$$\Phi(x_1, x_2, \ldots, x_N, \alpha, \beta, \ldots, \nu) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_\alpha(x_1) & \psi_\alpha(x_2) & \ldots & \ldots & \psi_\alpha(x_N) \\ \psi_\beta(x_1) & \psi_\beta(x_2) & \ldots & \ldots & \psi_\beta(x_N) \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \psi_\nu(x_1) & \psi_\nu(x_2) & \ldots & \ldots & \psi_\nu(x_N) \end{vmatrix}, \tag{4.1.6}$$

where x_i represent the coordinates and spin values of particle i, and $\alpha, \beta, \ldots, \nu$ are quantum numbers, while $\psi_\alpha(x_i)$ are eigenfunctions of the one-body Hamiltonian h_i, i.e.

$$\hat{h}_0(x_i) = \hat{t}(x_i) + \hat{u}_{\text{ext}}(x_i), \tag{4.1.7}$$

and

$$\hat{h}_0(x_i)\psi_\alpha(x_i) = \epsilon_\alpha\psi_\alpha(x_i). \tag{4.1.8}$$

The energies ϵ_α are the unperturbed energies, i.e. the non-interacting single particle energies. With no interaction between particles the total energy would be the sum of these unperturbed energies.

For a given $n \times n$ matrix $\mathbf{A}$ the determinant is

$$\det(\mathbf{A}) = |\mathbf{A}| = \begin{vmatrix} a_{11} & a_{12} & \dots & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{vmatrix}, \tag{4.1.9}$$

which can also be written as

$$|\mathbf{A}| = \sum_{i=1}^{n!}(-1)^{p_i}\hat{P}_i a_{11}, a_{22}, \dots, a_{nn}, \tag{4.1.10}$$

where $\hat{P}_i$ is the permutation operator permuting the column indices, and the sum runs over all $n!$ permutations. The quantity p_i is the number of transpositions of column indices needed to bring a given permutation back to its original ordering.

The Slater determinant from Eq.(4.1.6) is the trial function for the Hartree-Fock method, and we can now rewrite the determinant as

$$\Phi(x_1, x_2, \dots, x_N, \alpha, \beta, \dots, \nu) = \frac{1}{\sqrt{N!}}\sum_p (-1)^p \hat{P}\psi_\alpha(x_1)\psi_\beta(x_2)\dots\psi_\nu(x_N) = \sqrt{N!}\hat{A}\Phi_H, \tag{4.1.11}$$

where $\hat{A}$ is the anti-symmetrization operator defined by the summation over all possible permutations of two particles. This operator is given by

$$\hat{A} = \frac{1}{N!}\sum_p (-1)^p \hat{P}, \tag{4.1.12}$$

with p being the number of permutations. We have also introduced the Hartree-function, defined by the product of all possible single particle functions

$$\Phi_H(x_1, x_2, \dots, x_N, \alpha, \beta, \dots, \nu) = \psi_\alpha(x_1)\psi_\beta(x_2)\dots\psi_\nu(x_N). \tag{4.1.13}$$

4.2 Expectation Value of the Hamiltonian

From the variational principle we know that

$$E_0 \leq E[\Phi] = \int \Phi^* \hat{H}\Phi d\tau, \tag{4.2.1}$$

where E_0 is the ground state energy, $d\tau = d\mathbf{r}_1, d\mathbf{r}_2, \ldots, d\mathbf{r}_N$, and Φ is trial function which we assume is normalized, i.e.

$$\int \Phi^* \Phi d\tau = 1. \tag{4.2.2}$$

Both the non-interacting part of the Hamiltonian, $\hat{H}_0$, and the interacting part, $\hat{H}_1$, are invariant under all possible permutations of any two particles, and therefore commute with $\hat{A}$

$$\left[\hat{H}_0, \hat{A}\right] = \left[\hat{H}_1, \hat{A}\right] = 0. \tag{4.2.3}$$

In addition, $\hat{A}$ satisfies

$$\hat{A}^2 = \hat{A}, \tag{4.2.4}$$

since every permutation of the Slater determinant reproduces it.

4.2.1 One-body Hamiltonian

The expectation value of $\hat{H}_0$ is given by

$$\int \Phi^* \hat{H}_0 \Phi d\tau = N! \int \Phi_H^* \hat{A} \hat{H}_0 \hat{A} \Phi_H d\tau, \tag{4.2.5}$$

and using Eq.(4.2.3) and (4.2.4), we can reduce it to

$$\int \Phi^* \hat{H}_0 \Phi d\tau = N! \int \Phi_H^* \hat{H}_0 \hat{A} \Phi_H d\tau. \tag{4.2.6}$$

Furthermore, we replace the anti-symmetrization operator with its definition, and replace $\hat{H}_0$ with the sum of one-body operators, and obtain

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{i=1}^{N} \sum_{p} (-1)^p \int \Phi_H^* \hat{h}_0 \hat{P} \Phi_H d\tau. \tag{4.2.7}$$

If two or more particles are permuted in only one of the Hartree-functions, Φ_H, the integral vanishes since the individual single particle wave functions are orthogonal. We can therefore simplify to

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{i=1}^{N} \int \Phi_H^* \hat{h}_0 \Phi_H d\tau. \tag{4.2.8}$$

The orthogonality of the single particle wave functions allows us to simplify the integral even more, and we end up with the following expressing for the expectation value of the sum of one-body Hamiltonians

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{\mu=1}^{N} \int \psi_\mu^*(\mathbf{r}) \hat{h}_0 \psi_\mu(\mathbf{r}) d\mathbf{r}. \tag{4.2.9}$$

We introduce the following, more compact, notation for the integral

$$\langle\mu|\hat{h}_0|\mu\rangle = \int \psi_\mu^*(\mathbf{r})\hat{h}_0\psi_\mu(\mathbf{r})d\mathbf{r}, \tag{4.2.10}$$

and rewrite Eq.(4.2.9) as

$$\int \Phi^*\hat{H}_0\Phi d\tau = \sum_{\mu=1}^{N}\langle\mu|\hat{h}_0|\mu\rangle \tag{4.2.11}$$

4.2.2 Two-body Hamiltonian

For the two-body part of the Hamiltonian, we can obtain the expectation value in a similar way as for the one-body part. We start with

$$\int \Phi^*\hat{H}_1\Phi d\tau = N!\int \Phi_H^*\hat{A}\hat{H}_1\hat{A}\Phi_H d\tau, \tag{4.2.12}$$

and by the same arguments as for the one-body Hamiltonian, we can reduce this to

$$\int \Phi^*\hat{H}_1\Phi d\tau = \sum_{i\leq j=1}^{N}\sum_p (-1)^p \int \Phi_H^* V(r_{ij})\hat{P}\Phi_H d\tau. \tag{4.2.13}$$

Unlike the one-body case, permutations of any two particles doesn't vanish in the two-body case. This is due to the dependence on the inter-particle distance r_{ij}. For the two-body case we get

$$\int \Phi^*\hat{H}_1\Phi d\tau = \sum_{i\leq j=1}^{N}\int \Phi_H^* V(r_{ij})(1-P_{ij})\Phi_H d\tau, \tag{4.2.14}$$

where P_{ij} is the permutation operator for interchanging particle i and particle j. As for the one-body case, we assume that the single-particle wave functions are orthogonal, and get

$$\begin{aligned}\int \Phi^*\hat{H}_1\Phi d\tau = \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\Big[&\int \psi_\mu^*(x_i)\psi_\nu^*(x_j)V(r_{ij})\psi_\mu(x_i)\psi_\nu(x_j)dx_i dx_j \\ &- \int \psi_\mu^*(x_i)\psi_\nu^*(x_j)V(r_{ij})\psi_\nu(x_i)\psi_\mu(x_j)dx_i dx_j\Big].\end{aligned} \tag{4.2.15}$$

The first term is called the direct term, or the Hartree term, while the second term appears because of the Pauli principle. This term is called the exchange term or the Fock term. Since we now sum over all pairs twice, we need to add a factor 1/2. The single-particle wave functions $\psi_\alpha(x)$ are defined by the quantum numbers α and x as the overlap

$$\psi_\alpha(x) = \langle x|\alpha\rangle. \tag{4.2.16}$$

We introduce the following, more compact, notation for the integrals

$$\langle \mu\nu|\hat{v}|\mu\nu\rangle = \int \psi_\mu^*(x_i)\psi_\nu^*(x_j)V(r_{ij})\psi_\mu(x_i)\psi_\nu(x_j)dx_i dx_j, \tag{4.2.17}$$

and

$$\langle \mu\nu|\hat{v}|\nu\mu\rangle = \int \psi_\mu^*(x_i)\psi_\nu^*(x_j)V(r_{ij})\psi_\nu(x_i)\psi_\mu(x_j)dx_i dx_j. \tag{4.2.18}$$

We can define a combination of the direct matrix element and the exchange matrix element which we call the anti-symmetrization matrix element

$$\langle \mu\nu|\hat{v}|\mu\nu\rangle_{\text{AS}} = \langle \mu\nu|\hat{v}|\mu\nu\rangle - \langle \mu\nu|\hat{v}|\nu\mu\rangle, \tag{4.2.19}$$

which for a general matrix element is

$$\langle \mu\nu|\hat{v}|\sigma\tau\rangle_{\text{AS}} = \langle \mu\nu|\hat{v}|\sigma\tau\rangle - \langle \mu\nu|\hat{v}|\tau\sigma\rangle. \tag{4.2.20}$$

This anti-symmetrization element has the symmetry property

$$\langle \mu\nu|\hat{v}|\sigma\tau\rangle_{\text{AS}} = -\langle \mu\nu|\hat{v}|\tau\sigma\rangle_{\text{AS}} = -\langle \nu\mu|\hat{v}|\sigma\tau\rangle_{\text{AS}}, \tag{4.2.21}$$

and it is hermitian, which implies that

$$\langle \mu\nu|\hat{v}|\sigma\tau\rangle_{\text{AS}} = \langle \sigma\tau|\hat{v}|\mu\nu\rangle_{\text{AS}}. \tag{4.2.22}$$

We can now rewrite Eq.(4.2.15) as

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\langle \mu\nu|\hat{v}|\mu\nu\rangle_{\text{AS}}. \tag{4.2.23}$$

By combining Eq.(4.2.11) and (4.2.23) we end up with the energy functional

$$E\left[\Phi\right] = \sum_{\mu=1}^{N}\langle \mu|\hat{h}_0|\mu\rangle + \frac{1}{2}\sum_{\mu=1}^{N}\sum_{\nu=1}^{N}\langle \mu\nu|\hat{v}|\mu\nu\rangle_{\text{AS}}. \tag{4.2.24}$$

4.3 Polar Coordinates

If our trial function is a harmonic oscillator function, the integral

$$\langle \mu\nu|\hat{v}|\sigma\tau\rangle = \int \psi_\mu^*(x_i)\psi_\nu^*(x_j)V(r_{ij})\psi_\sigma(x_i)\psi_\tau(x_j)dx_i dx_j, \tag{4.3.1}$$

can be calculated in analytical form if we use polar coordinates instead of Cartesian coordinates. In two dimensions, using polar coordinates we have

$$x = r\cos\theta \tag{4.3.2}$$

$$y = r\sin\theta \tag{4.3.3}$$

$$r = \sqrt{x^2 + y^2}, \tag{4.3.4}$$

and the time independent wave function is composed of a radial part and an angular part

$$\psi(r,\theta) = R(r)Y(\theta). \tag{4.3.5}$$

The normalized solution for the angular part in two dimensions is

$$Y(\theta) = \frac{1}{\sqrt{2\pi}} e^{im\theta}. \tag{4.3.6}$$

Since the total wave function must satisfy the physical condition

$$\psi(r,\theta) = \psi(r,\theta + 2\pi), \tag{4.3.7}$$

the quantum number m is restricted to integral values

$$m = 0, \pm 1, \pm 2, \ldots \tag{4.3.8}$$

The solution for the radial part is

$$r_{nm}(r) = \sqrt{\frac{2n!}{(n+|m|)!}} \beta^{\frac{1}{2}(|m|+1)} r^{|m|} e^{-\frac{1}{2}\beta r^2} L_n^{|m|}(\beta r^2), \tag{4.3.9}$$

where n is the principal quantum number

$$n = 0, 1, 2, 3, \ldots \tag{4.3.10}$$

and m is the angular momentum quantum number given in Eq.(4.3.8). β is defined as

$$\beta = \frac{m_e \omega}{\hbar}, \tag{4.3.11}$$

where m_e is the particle mass, and ω is the oscillator frequency. $L_n^{|m|}$ are the associated Laguerre polynomials defined as the solutions to the differential equation

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2} \right) L(x) = 0, \tag{4.3.12}$$

where l is an integer $l \geq 0$ and λ is a constant. The polynomials for the first few n values are

$$L_0(x) = 1, \tag{4.3.13}$$

$$L_1(x) = 1 - x, \tag{4.3.14}$$

$$L_2(x) = 2 - 4x + x^2, \tag{4.3.15}$$

$$L_3(x) = 6 - 18x + 9x^2 - x^3, \tag{4.3.16}$$

and

$$L_4(x) = 24 - 96x + 72x^2 - 16x^3 + x^4. \tag{4.3.17}$$

The Laguerre polynomials fullfill the orthogonality relation

$$\int_0^\infty e^{-x} L_n(x)^2 dx = 1, \tag{4.3.18}$$

and the recursion relation

$$(n+1)L_{n+1}(x) = (2n+1-x)L_n(x) - nL_{n-1}(x). \tag{4.3.19}$$

The eigenfunction for a particle moving in a two dimensional harmonic oscillator is then

$$\psi(r,\theta) = \sqrt{\frac{n!}{\pi(n+|m|)!}} \beta^{\frac{1}{2}(|m|+1)} r^{|m|} e^{-\frac{1}{2}\beta r^2} L_n^{|m|}(\beta r^2) e^{im\theta}, \tag{4.3.20}$$

with eigenvalue

$$E = \hbar\omega(2n + |m| + 1), \tag{4.3.21}$$

which is the non-interacting energy, similarly to the following expression for Cartesian coordinates

$$E = \hbar\omega(n_x + n_y + 1). \tag{4.3.22}$$

4.4 Using Hartree-Fock to Improve the Overlap Coefficients

The use for the Hartree-Fock method in relation to this thesis is to improve the overlap coefficients used in the single particle wave function approximations. Currently creation of the overlap coefficients is based on diagonalizing the single particle problem, and consequently the coefficients don't account for interaction between particles. The goal of using Hartree-Fock calculations to modify the coefficients is to change that. Using coefficients which do account for particle interaction should remove the need for the variational parameter α, and would hopefully reduce the amount of harmonic oscillator basis functions needed.

A simple independent Hartree-Fock program is included in the Github repository [17]. This program can serve as a starting point for implementing Hartree-Fock modifications of the overlap coefficients. The VMC machinery developed in this thesis uses Cartesisan coordinates, but it is more convenient to do the Hartree-Fock calculations in polar coordinates. As a result, the implementation will need a transformation to polar coordinates in order to do the Hartree-Fock calculations, then a transformation back to Cartesian coordinates so the overlap coefficients can be used in the VMC solver. For information about these transformations see Chapter 6.4.4 of Ref. [21].

Chapter 5

Systems

The systems we are looking at contain electrons confined in a harmonic oscillator like potentials, with the following idealized total Hamiltonian

$$H = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + V_c(\mathbf{r}_i) \right) + \sum_{i<j} \frac{1}{r_{ij}}, \tag{5.0.1}$$

where we have used natural units ($\hbar = c = e = m_e = 1$) and all energies are in so-called atomic units a.u. $V_c(\mathbf{r}_i)$ is the external confinement potential at position $\mathbf{r}_i$. Using the above Hamiltonian we will study systems of many electrons N as functions of the oscillator frequency ω. The unperturbed part of the Hamiltonian is

$$H_0 = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + V_c(\mathbf{r}_i) \right), \tag{5.0.2}$$

while the repulsive Coulomb interaction between two electrons is given by

$$H_1 = \sum_{i<j} \frac{1}{r_{ij}}, \tag{5.0.3}$$

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between two electrons. The modulus of the positions of the electrons (for a given electron i) is defined as $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2}$.

5.1 Potentials

5.1.1 Standard Harmonic Oscillator Well

For the standard harmonic oscillator well, the confinement potential is simply [6]

$$\begin{aligned} V_c(\mathbf{r}) &= \frac{1}{2} m_e \omega^2 \mathbf{r}^2 \\ &= \frac{1}{2} \omega^2 \mathbf{r}^2, \end{aligned} \tag{5.1.1}$$

where $\mathbf{r}$ is the distance from the center of the well. The potential is zero at the center of the well, and then increases proportionally to $\mathbf{r}^2$ as we move away from the center. Since we use natural units we have $m_e = 1$. The strength of the potential is also effected by the constant ω, which is the oscillator frequency. The harmonic oscillator well potential is well studied both analytically and numerically and is therefore good for providing well known benchmarks for our results. The shape of this confinement potential is shown in Figure 5.1.

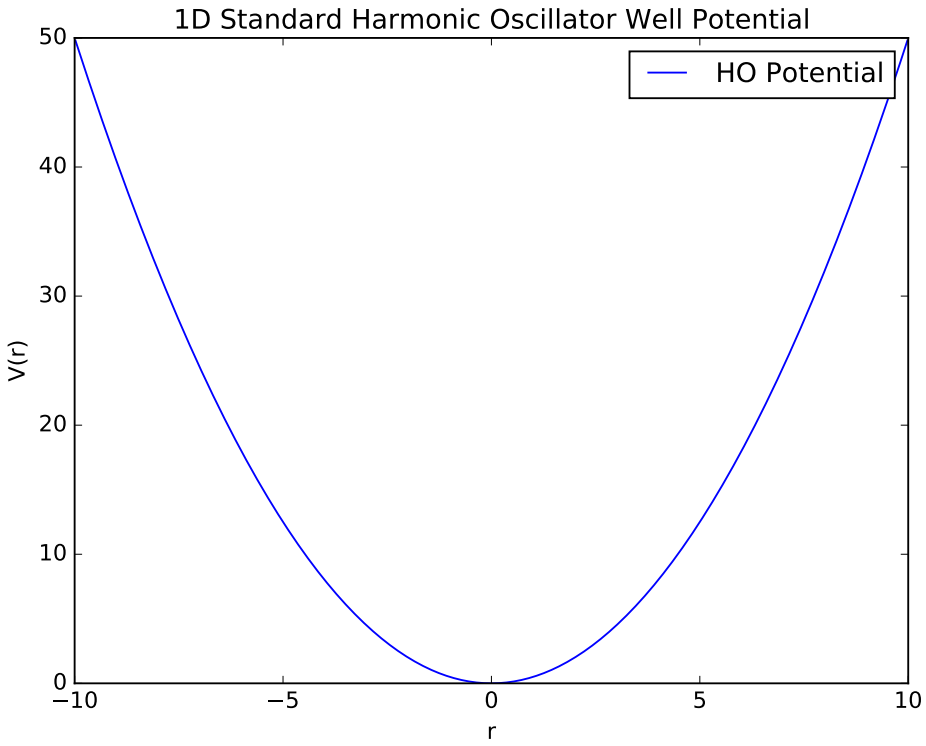


Figure 5.1: Standard Harmonic Oscillator Well Potential as described by Eq.(5.1.1), with $\omega = 1$.

5.1.2 Double Harmonic Oscillator Well

When we have a harmonic oscillator potential with multiple minima, the confinement potential is on the general form [11]

$$V_c(\mathbf{r}) = \frac{1}{2} m_e \omega^2 \min \left[\sum_j^M (\mathbf{r} - \mathbf{L}_j)^2 \right], \tag{5.1.2}$$

where, in two dimensions, we have $\mathbf{r} = (x, y)$, and the multiple $\mathbf{L}_j = (\pm L_x, \pm L_y)$ give the positions of the minima. M is the total number of minima in the potential. When $M = 1$ and $\mathbf{L}_1 = (0, 0)$ we have the standard harmonic oscillator well, while if $M = 2$ and $\mathbf{L}_{1,2} = (\pm L_x, 0)$ we get the double-well potential. We can also write the confinement potential

using the absolute values of the coordinates. For a double-well in two dimensions ($\mathbf{L}_{1,2} = (\pm L_x, 0)$) we then get

$$\begin{aligned} V_c(x,y) &= \frac{1}{2} m_e \omega^2 [\mathbf{r}^2 - 2L_x|x| - 2L_y|y| + L_x^2 + L_y^2] \\ &= \frac{1}{2} m_e \omega^2 [\mathbf{r}^2 - 2L_x|x| + L_x^2]. \end{aligned} \tag{5.1.3}$$

Double-well potentials, and the barrier between the wells, are interesting for quantum computing and information processing. Double-well potentials are investigated as candidates for logic gates in quantum computing [31]. As explained by Ref. [20], in regular computing a logic gate is a device implementing a Boolean function. It takes one or more binary inputs and performs a logical operation one them, producing a single binary output. In quantum computing the difference is that the gates should be able to have more varied output, i.e. it should have more output possibilities than only 0 or 1. Double-well potentials are also a starting point for understanding periodic multiwell potentials. The double harmonic oscillator potential is parabolic with minima in e.g. $x = \pm L_x$, $y = 0$, and the parabolic wells meet at an absolute value barrier at $x = 0$. Figure 5.2 shows an example of a double-well potential, where $L_x = 5$ and $\omega = 1$.

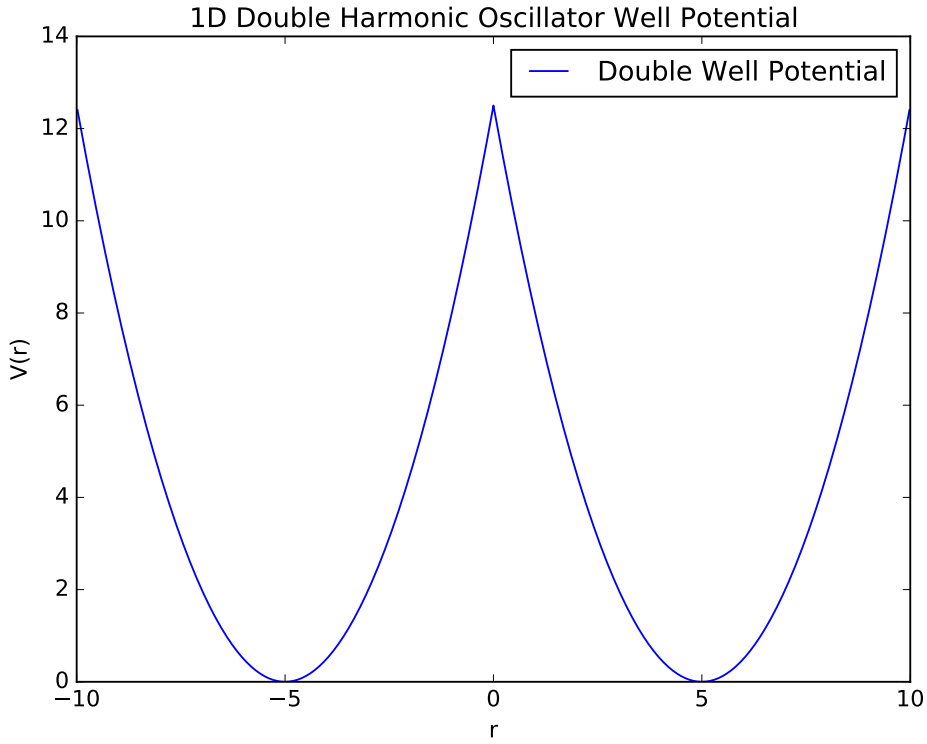


Figure 5.2: Double Harmonic Oscillator Well Potential as described by Eq.(5.1.3), with $L_x = 5$, $L_y = 0$, and $\omega = 1$.

5.1.3 Finite Square Well

Another interesting potential to look at is the square well potential, where the potential will have one constant value within the well, and another constant value outside of it. For a finite square well, the potential in one dimension can be written as [6]

$$V_c(x) = \begin{cases} C_1, & \text{if } -L < x < L \\ C_2, & \text{otherwise} \end{cases}, \tag{5.1.4}$$

where L is the distance from the center of the well to the walls, and we have centered the well at $x = 0$. For simplicity we set $C_1 = 0$, and rename C_2 to V_0 which we can then choose as the height of the well. We then have

$$V_c(x) = \begin{cases} 0, & \text{if } -L < x < L \\ V_0, & \text{otherwise} \end{cases}. \tag{5.1.5}$$

This potential has a shape that is somewhat similar to a harmonic oscillator well, and it is also fairly simple since it's either a set constant value or zero, everywhere. Due to it's simplicity this type of potential can more easily be produced in laboratory experiments. Figure 5.3 shows a finite square well potential with $V_0 = 1$ and $L = 3$.

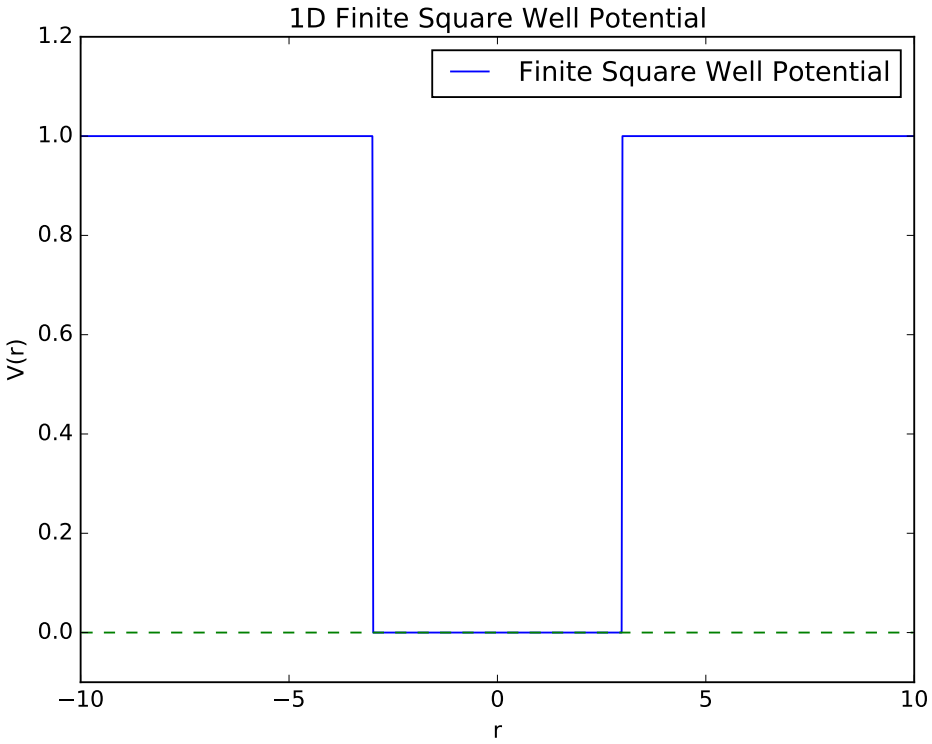


Figure 5.3: Finite Square Well Potential as described by Eq.(5.1.5), with $L = 3$ and $V_0 = 1$.

5.2 Two-body Quantum Dot

For two electrons in a two-dimensional harmonic oscillator potential without interaction between the electrons, the energy of each electron is given by

$$\epsilon_n = \omega(n_x + n_y + 1). \tag{5.2.1}$$

For the ground state we have $n_x = n_y = 0$, and since there are two electrons the total (unperturbed) ground state energy is then $\epsilon_0 = 2\omega$. Since electrons are spin-$\frac{1}{2}$ particles they have two possible spin states, which means that both electrons can be in the ground state ($n_x = n_y = 0$) as long as they have different spin. One electron will have spin $+\frac{1}{2}$ and the other will have spin $-\frac{1}{2}$, so the total spin of the system will be zero. It makes sense for the ground state to have total spin zero, since the total spin contributes to the energy of the state, and the ground state is the state with lowest energy.

The perturbed trial wave function we will use for the two-body quantum dot has the form

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}) = C \exp\left(-\alpha\omega(r_1^2 + r_2^2)/2\right) \exp\left(\frac{ar_{12}}{(1 + \beta r_{12})}\right), \tag{5.2.2}$$

where $a = 1$ when the electrons have anti-parallel spins and $a = 1/3$ when they have parallel spins. For the two-body quantum dot we are interested in the ground state for only two electrons, meaning the electrons will always have anti-parallel spins and $a = 1$ in this case. α and β are the variational parameters.

5.3 Many-body Quantum Dot

A closed shell system is a system where all used energy levels are filled. This is the case when our number of electrons equal a so-called magic number, i.e. $N = 2, 6, 12, 20, \ldots$. For a closed shell system with more than two electrons, we use the trial wave function

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}, \ldots, \mathbf{r_N}) = Det\left(\phi_1(\mathbf{r_1}), \phi_2(\mathbf{r_2}), \ldots, \phi_N(\mathbf{r_N})\right) \prod_{i<j}^{N} \exp\left(\frac{ar_{ij}}{(1 + \beta r_{ij})}\right), \tag{5.3.1}$$

where Det is a Slater determinant. The single particle wave functions are harmonic oscillator wave functions with the following form

$$\phi_{n_x,n_y}(x, y) = AH_{n_x}(\sqrt{\omega}x)H_{n_y}(\sqrt{\omega}y) \exp\left(-\omega(x^2 + y^2)/2.\right. \tag{5.3.2}$$

The functions $H_{n_x}(\sqrt{\omega}x)$ are Hermite polynomials [8], and A is a normalization constant. In this case, two chosen electrons can have either anti-parallel or parallel spins, so the value of a depends on which two electrons we are looking at.

5.4 Closed Form Expressions

We want to find the expectation value of the local energy, and having a closed form expression for this energy is convenient. The local energy is given by

$$E_L(\mathbf{R}) = \frac{1}{\Psi_T(\mathbf{R})} H\Psi_T(\mathbf{R}). \tag{5.4.1}$$

By finding the local energy using two different methods, one using the closed form expression and the other using a purely numerical approach, we can compare the results of the methods to each other to verify that the program works properly. Using the closed form expression is also computationally faster than the numerical approach, which is convenient when doing large simulations. We can also use closed form expressions for the drift term in importance sampling

$$F = \frac{2\nabla\Psi_T}{\Psi_T}. \tag{5.4.2}$$

The necessary closed form expressions for the two-body quantum dot and the many-body quantum dot are calculated in the first appendix, in section A.1 and section A.2 respectively.

Part II

Implementation

Chapter 6

Program Structure

6.1 Variational Monte Carlo Simulations

The variational Monte Carlo code consists primarily of several classes. The Particle class stores information about a single particles position, and is responsible for changing this position when necessary and providing it to other parts of the program. The System class contains information about the system, such what kind of Hamiltonian and wave function we have. It also has a vector containing instances of the Particle class, one instance for each particle in the system. In addition this class is responsible for running the Metropolis algorithm, including checking if the Metropolis step is accepted, and calculating the drift (quantum) force and Greens function when using importance sampling. The Sampler class is responsible for sampling the energy etc. after each step and computing averages at the end of the simulation.

The Hamiltonian class is responsible for calculating the Hamiltonian, i.e. the kinetic part, the non-interacting potential part, and also the interacting potential part if interactions are included. In addition, the Hamiltonian class contains the functions for calculating the single-particle wave functions and its gradient and Laplacian. The last part was initially done in the WaveFunction class, but when including different types of potentials it turned out to be more convenient to evaluate the single-particle wave functions etc. in the Hamiltonian class. The WaveFunction class deals with the more general things concerning the many-body wave function, such as updating the Slater determinant and evaluating the full many-body wave function and its gradient and Laplacian. It receives the single-particle wave function evaluations from the Hamiltonian class when they are needed. The WaveFunction class also contains the function for calculating the Metropolis ratio, which the System class uses to determine if a given step is accepted or not.

The InitialStates class sets up an initial state for the particles, i.e. it creates instances of the Particle class and gives them their initial positions. The VariationMethods class contains methods for variation of the variational parameters. Currently the only method implemented is the steepest descent method, however the program is set up in a way which makes adding other methods, such as the more advanced conjugate gradient method, fairly simple.

6.1.1 Initializing

The program starts in the "main.cpp" file where system settings such as number of dimensions, number of particles and number of Monte Carlo cycles, and also constants such as the harmonic oscillator frequency ω. Certain flags are also set to true or false, such as whether to include interactions or not, and what type of potential is used.

An instance of the InitialStates class is then created which sets up initial positions for each particle where, for each dimension, the particle is given a random value uniformly distributed between 0 and 1 as its position in that dimension. For each particle, an instance of the Particle class is created, which is given the position for that particle. In addition, every other particle has its positions in each dimension multiplied by -1. This is done so that when we have a double well with a barrier between the wells at e.g. $x = 0$, the total amount of particles will be evenly split between the wells. The seed used to generate random positions is dependent on the "m_my_rank" variable, which is decided by which node we're on when running the program in parallel. This is because we need to give the nodes different initial states from each other, to avoid running the exact same simulation on every node.

```
1  void RandomUniform::setupInitialState() {
2
3      long idum = -1-m_my_rank;
4      Random::setSeed(idum);
5
6      // Create random positions for all particles:
7      for (int i=0; i < m_numberOfParticles; i++) {
8          std::vector<double> position = std::vector<double>();
9
10         for (int j=0; j < m_numberOfDimensions; j++) {
11             double pos = pow(-1, i%2)*Random::nextDouble();
12             position.push_back(pos);
13         }
14         // Create particles:
15         m_particles.push_back(new Particle());
16         m_particles.at(i)->setNumberOfDimensions(m_numberOfDimensions);
17         m_particles.at(i)->setPosition(position);
18     }
19 }
```

Setting up the Initial State

Next, in "main.cpp", an instance of the Hamiltonian class is created as well as an instance of the WaveFunction class. All three instances are then stored in an instance of the System class, which proceeds to run the Metropolis algorithm.

6.1.2 Monte Carlo

The System class is responsible for running the Metropolis algorithm. It loops over the number of Metropolis steps and the number of particles and changes the position of the particle. The change in position depends on whether or not we use importance sampling (the Metropolis-Hastings algorithm). If we don't use importance sampling the change in position is chosen by a random uniform distribution independent of the state of the system. Based on the change in position the WaveFunction class calculates the Metropolis ratio, which is used to determine if the position change is accepted or not. If importance sampling is used the change in position depends on the so-called quantum force, and the Metropolis ratio is also dependent on the Greens function which in turn

depends on the quantum force. Both the quantum force and the Greens function are calculated in the System class.

The ratio used to determine whether or not to accept the step is compared to a random uniformly distributed number between 0 and 1. A ratio greater or equal to the random number results the step being accepted. If the position change is accepted it is kept, otherwise it's reverted, and then the entire system with its current particle positions is sampled by the Sampler class.

The Sampler class adds the energy of the system at each step to the cumulative energy, and computes the average after the loop over Metropolis steps has finished. However, not every Metropolis step is sampled. Before we start sampling we let the system run through some steps (e.g. 10% of the total amount) in order to let the system equilibrate first. Below is the code for handling a Metropolis step when we're not using importance sampling. The change in position is first applied to the particle in the "computeMetropolisRatio" function of the WaveFunction class, and then reverted in this listed function if the step is not accepted.

```
1  bool System::metropolisStep(int currentParticle) {
2      /* Perform the actual Metropolis step: Take the current particle and
3       * change it's position by a random amount, and check if the step is
4       * accepted by the Metropolis test (compare the wave function evaluated
5       * at this new position with the one at the old position).
6       */
7
8      // Change position of current particle by a random amount creating a trial
           state
9      setCurrentParticle(currentParticle);
10     std::vector<double> positionChange(m_numberOfDimensions);
11
12     for (int i=0; i<m_numberOfDimensions; i++){
13         positionChange[i] = (Random::nextDouble()*2-1)*m_stepLength;
14     }
15
16     // Metropolis ratio
17     double qratio = m_waveFunction->computeMetropolisRatio(m_particles,
           currentParticle, positionChange);
18
19     // Check if trial state is accepted
20     if (Random::nextDouble() <= qratio){
21         m_waveFunction->updateSlaterDet(currentParticle);
22         return true;
23     }
24
25     for (int i=0; i<m_numberOfDimensions; i++){
26         // If trial state is not accepted, revert to old position for chosen
               particle (revert to old state)
27         m_particles[currentParticle]->adjustPosition(-positionChange[i], i);
28         m_waveFunction->updateDistances(currentParticle);
29         m_waveFunction->updateSPWFMat(currentParticle);
30         m_waveFunction->updateJastrow(currentParticle);
31     }
32
33     return false;
34 }
```

Listing 6.1: Metropolis Step Without Importance Sampling

6.1.3 Virtual Functions

Virtual functions are an essential part of how the Hamiltonian and WaveFunction classes are built up. Normally if a super class has defined a function and a sub class defines an identical function, the super class function overwrites the sub class function. However, if we make the super class function virtual, the sub class function will not be overwritten by the compiler. This is useful because it allows us to have a super class with multiple sub classes where each sub class has its own implementation of a given function which is virtual in the super class. For example, our Hamiltonian class has sub classes for different types of Hamiltonians (i.e. HO well, double HO well, square well, etc.). Depending on the system we are looking at, we create an instance of the sub class corresponding to the Hamiltonian of the system. By doing this we can do a general call to a function which all sub classes of the Hamiltonian class have, but have implemented in different ways. The implementation that is used will then be the one belonging to the sub class we created the instance of.

Let us look at a more concrete example to make it more clear. Each sub class of the Hamiltonian class has an implementation of the "computeLocalEnergy" function. The difference between the function implementation for the double HO well and the regular HO well is how the potential energy is calculated.

```
1  double potentialEnergy = 0;
2      double repulsiveTerm = 0;
3
4      for (int i=0; i < numberOfParticles; i++){
5          double rSquared = 0;
6          std::vector<double> r_i = particles[i]->getPosition();
7          for (int k=0; k < numberOfDimensions; k++){
8              rSquared += r_i[k]*r_i[k];
9          }
10         potentialEnergy += rSquared;
11
12         for (int j=i+1; j < numberOfParticles; j++){
13             double r_ijSquared = 0;
14             std::vector<double> r_j = particles[j]->getPosition();
15             for (int k=0; k < numberOfDimensions; k++){
16                     r_ijSquared += (r_i[k] - r_j[k]) * (r_i[k] - r_j[k]);
17             }
18
19             double r_ij = sqrt(r_ijSquared);
20             repulsiveTerm += 1./r_ij;
21         }
22     }
23     potentialEnergy *= 0.5*m_omega*m_omega;
24     if (m_repulsion) { potentialEnergy += repulsiveTerm; }
```

Listing 6.2: Potential energy calculation for the regular harmonic oscillator well sub class of the Hamiltonian class.

```
1  double potentialEnergy = 0;
2      double repulsiveTerm = 0;
3
4      for (int i=0; i < numberOfParticles; i++){
5          double rSquared = 0;
6          double term2 = 0;
7          double term3 = 0;
8
9          std::vector<double> r_i = particles[i]->getPosition();
10         for (int k=0; k < numberOfDimensions; k++){
11             rSquared += r_i[k]*r_i[k];
```

```
12              term2 += 2.*abs(r_i[k])*m_L(k);
13              term3 += m_L(k)*m_L(k);
14          }
15
16          potentialEnergy += rSquared - term2 + term3;
17
18          for (int j=i+1; j < numberOfParticles; j++){
19              double r_ijSquared = 0;
20              std::vector<double> r_j = particles[j]->getPosition();
21              for (int k=0; k < numberOfDimensions; k++){
22                      r_ijSquared += (r_i[k] - r_j[k]) * (r_i[k] - r_j[k]);
23              }
24
25              double r_ij = sqrt(r_ijSquared);
26              repulsiveTerm += 1./r_ij;
27          }
28      }
29
30      potentialEnergy *= 0.5*m_omega*m_omega;
31
32      if (m_repulsion) { potentialEnergy += repulsiveTerm; }
```

Listing 6.3: Potential energy calculation for the double harmonic oscillator well sub class of the Hamiltonian class.

At the start of the program we can use one of the following lines of code to choose which of the two Hamiltonians we are using for the simulation. (We can of course choose any other implemented Hamiltonian sub class as well.)

```
1 system->setHamiltonian       (new HarmonicOscillatorElectrons(system, omega,
      analyticalKinetic, repulsion));
2 system->setHamiltonian       (new DoubleHarmonicOscillator(system, L, omega,
      analyticalKinetic, repulsion));
```

Listing 6.4: Setting the Hamiltonian of the system.

This will store the instance we chose in the "m_hamiltonian" object of the System class. Now we can call the "computeLocalEnergy" function from the Sampler class with the following line.

```
1 std::vector<double> energies = m_system->getHamiltonian()->computeLocalEnergy(
      m_system->getParticles());
```

Listing 6.5: Calling the "computeLocalEnergy" function from the Sampler class. The function implementation used will be the one belonging to the Hamiltonian sub class we have chosen as the Hamiltonian of the system.

Here the "m_system->getHamiltonian()" call returns the "m_hamiltonian" object, and the "m_system->getParticles()" call returns the "m_particles" vector which is used as argument for the "computeLocalEnergy" function, so the call is essentially

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

The program will then call the implementation of the "computeLocalEnergy" function which corresponds to the sub class instance we have stored in "m_hamiltonian". If we chose line 1 in Listing 6.4, then the code in Listing 6.2 will be used, while if we chose line 2, then the code in Listing 6.3 will be used. This saves us from having to use an if-test every time we want to call the "computeLocalEnergy" function. Instead we just need one if-test at the beginning to choose between line 1 and 2 in Listing 6.4.

This is significant for optimizing the program when we might have millions of Monte Carlo cycles, since we avoid millions of if-test, and it also makes the code a lot cleaner. Using virtual functions in this way is a form of polymorphism, which is the concept of "providing a single interface to entities of different types." [15]

There are two types of virtual functions we can use in C++; the regular virtual functions and pure virtual functions. When a virtual function is defined in a super class, the compiler requires that function to always have a valid implementation. This means that if the super class only defines the function, but doesn't provide a proper implementation of the function, then every sub class of the super class has to provide an implementation. This is what is called a pure virtual function. The alternative is a regular virtual function, which is a function which is defined as virtual in the super class, but also has an implementation in the super class.

The implementation in the super class then acts as a default implementation, which will be used if the instanced sub class doesn't have its own implementation of said function. If the sub class does have its own implementation this will be used instead of the default. This can be useful if some of the sub classes have the same implementation of a given function while other sub classes have unique implementations. The sub classes who share an implementation can use the default implemented by the super class, while the other sub classes can use their own unique implementations. This way we don't have to implement the exact same function several times, and the amount of code is reduced. Another use for the default implementation is if some sub classes have unique implementations of a given function, but other sub classes doesn't need to have the function at all. If a pure virtual function was used, then every sub class that didn't need the function would still need a dummy version of the function. This dummy version wouldn't do anything and would never be called, but would still be necessary for compiling the program. Instead we can use a regular virtual function, where the default version implemented in the super class is a dummy version, which then allows the sub classes that don't need the function to skip implementing it.

```
1 virtual std::vector<double> computeLocalEnergy(std::vector<class Particle*>
      particles) = 0;
2 virtual std::vector<double> computeLocalEnergy(std::vector<class Particle*>
      particles) { particles = particles; }
```

Listing 6.6: Example of definitions of a pure virtual function (line 1) and a regular virtual function with a dummy implementation (line 2).

The pure virtual functions can also be useful. For example, if you know that every sub class should have its own unique implementation of a function, then the lack of a default implementation can act as a test or safeguard for the code when implementing new sub classes. If you forget to implement the function, the compiler will abort and give an error. You will then be made aware that something is wrong, instead of potential using a (wrong) default implementation, which makes it seem like the program is running fine, while it is actually providing erroneous results. The "computeLocalEnergy" function in the VMC program is an example of a pure virtual function, so if someone wanted to add a new Hamiltonian to the program they would have to implement this function for the new Hamiltonian.

6.1.4 Hamiltonians

The main job for the Hamiltonian class is to compute the local energy. This means that it has to calculate the energy from the external potential (e.g. harmonic oscillator well), the kinetic energy of the particles, and, if included, the potential energy from particle-particle interactions (e.g. Coulomb repulsion). In this thesis we look at electrons in different types of external potentials, and therefore we have Coulomb repulsion as the particle-particle interaction. The kinetic energy can be computed with numerical differentiation or using analytical expressions for the Laplacian of the wave function. The numerical differentiation can be implemented generally for any Hamiltonian, so we can implement a function for it once, in the Hamiltonian super class, and then call it from the sub class we're using. This is done by the following function.

```
1  double Hamiltonian::computeKineticEnergy(std::vector<Particle*> particles){
2      // Compute the kinetic energy using numerical differentiation.
3
4      double numberOfParticles = m_system->getNumberOfParticles();
5      double numberOfDimensions = m_system->getNumberOfDimensions();
6      double h = 1e-4;
7
8      // Evaluate wave function at current step
9      double waveFunctionCurrent = m_system->getWaveFunction()->evaluate(particles)
           ;
10     double kineticEnergy = 0;
11
12     for (int i=0; i < numberOfParticles; i++){
13         for (int j=0; j < numberOfDimensions; j++){
14
15             // Evaluate wave function at forward step
16             particles[i]->adjustPosition(h, j);
17             m_system->getWaveFunction()->updateDistances(i);
18             m_system->getWaveFunction()->updateSPWFMat(i);
19             m_system->getWaveFunction()->updateJastrow(i);
20             double waveFunctionPlus = m_system->getWaveFunction()->evaluate(
                   particles);
21
22             // Evaluate wave function at backward step
23             particles[i]->adjustPosition(-2*h, j);
24             m_system->getWaveFunction()->updateDistances(i);
25             m_system->getWaveFunction()->updateSPWFMat(i);
26             m_system->getWaveFunction()->updateJastrow(i);
27             double waveFunctionMinus = m_system->getWaveFunction()->evaluate(
                   particles);
28
29             // Part of numerical diff
30             kineticEnergy -= (waveFunctionPlus - 2*waveFunctionCurrent +
                   waveFunctionMinus);
31
32             // Move particles back to original position
33             particles[i]->adjustPosition(h, j);
34             m_system->getWaveFunction()->updateDistances(i);
35             m_system->getWaveFunction()->updateSPWFMat(i);
36             m_system->getWaveFunction()->updateJastrow(i);
37         }
38     }
39     // Other part of numerical diff. Also divide by evaluation of current wave
           function
40     // and multiply by 0.5 to get the actual kinetic energy.
41     kineticEnergy = 0.5*kineticEnergy / (waveFunctionCurrent*h*h);
42     return kineticEnergy;
43 }
```

Listing 6.7: Function for calculating the kinetic energy by numerical differentiation. The function is general for all the sub classes of the Hamiltonian class, and is therefore implemented once, in the super class "hamiltonian.cpp".

If we use analytical expressions for the Laplacian when calculating the kinetic energy, then we call the "computeDoubleDerivative" function from the WaveFunction class. When calculating the local energy, the difference from sub class to sub class will be the calculation of the external potential energy as shown in Listing 6.2 and Listing 6.3. In addition to calculating the local energy, the Hamiltonian sub classes also contain functions for evaluating the single particle wave function and its gradient and Laplacian. Since the single particle wave function contains a Hermite polynomial we also calculate these in the Hamiltonian sub classes. The Hermite polynomials are analytical expressions, but they also have a recursive relation. Therefore we can either implement them by writing out the analytical expression for every Hermite polynomial we might need and then choose the one we need at a given moment, or we can calculate them recursively. There are advantages to each of these implementations. The recursive method requires less code and is therefore quicker to implement and is general, so that it will work regardless of how many particles we put in the system. However, since it needs to iterate from the lowest polynomial every time, it can become very computationally expensive when we have a lot of particles.

Using the analytical expressions we would have to add enough polynomials to support the amount of particles we want in the system, and if we wanted to increase the number of particles further, we would have to implement more analytical expressions. There are two ways we could use for choosing among the analytical expressions. The simplest is to just have an if-test for every expression and then start at the beginning and check if we wanted to use H_0 (the lowest Hermite polynomial). If not we would move on to the if-test for H_1 and then continue like that until we found the one we wanted to use and return that. This would be computationally faster than the recursive method, but it would still potentially need to go through a lot of if-tests, especially when the number of particles in the system is high. There is a more optimized alternative, but it is more difficult to implement. Instead of of using if-tests to check every expression from the start until we find the one we want, we can instead store all the expressions in a list, and then use the index for the expression we want to get it from the list without having to touch any of the other expressions at all. In this thesis the recursive method was used originally, however a switch to the last method described was made eventually, which is further discussed in section 7.2.

```
1  double HarmonicOscillatorElectrons::computeHermitePolynomial(int nValue, double
       position) {
2      // Computes Hermite polynomials.
3      double alphaSqrt = sqrt(m_alpha);
4      double omegaSqrt = sqrt(m_omega);
5      double factor = 2*alphaSqrt*omegaSqrt*position;
6
7      double HermitePolynomialPP = 0;                    // H_{n-2}
8      double HermitePolynomialP = 1;                     // H_{n-1}
9      double HermitePolynomial = HermitePolynomialP;     // H_n
10
11     for (int n=1; n <= nValue; n++) {
12         HermitePolynomial = factor*HermitePolynomialP - 2*(n-1)*
               HermitePolynomialPP;
```

```
13            HermitePolynomialPP = HermitePolynomialP;
14            HermitePolynomialP = HermitePolynomial;
15        }
16
17        return HermitePolynomial;
18    }
```

Listing 6.8: Recursive method for computing Hermite polynomials. This method requires the least amount of code and will work for any amount of particles in the system. However, it is also the least optimized method in terms of computation time, and it can be very time consuming when the amount of particles is high.

6.1.5 Wave Functions

The WaveFunction class is responsible for maintaining and evaluating the full wave function of the system. The systems we're looking at in this thesis are systems where a chosen number of electrons are confined in various external potentials. The wave function then approximated by a Slater determinant of the single particle wave functions, and a Jastrow factor (if interactions are included). Since the single particle functions are implemented in the Hamiltonian sub classes as discussed above, we don't need different sub classes of the WaveFunction class for different external potentials. Two sub classes of the WaveFunction class are used here; The ManyElectrons sub class and the ManyElectronsCoefficients sub class. The first one is used for regular variational Monte Carlo simulations of the systems we're looking at, while the second uses diagonalization of a general potential well to expand its solutions in terms of harmonic oscillator functions. For the most part these two sub classes have the implementation, with the difference between them being how the single particle wave functions in the Slater matrix are approximated.

Before the Monte Carlo simulation starts, the WaveFunction sub classes set up several matrices. The "setUpSlaterDet" function sets up the Slater matrix of single particle wave functions, and matrices for its gradient and Laplacian. We split the Slater matrix in one spin up part and one spin down part, both of which are square matrices. However, even though we treat them as two different matrices we store them together in the matrix called "m_SPWFMat". The first half of the particles are considered spin up, while the other half are considered spin down. We also store the inverse of the spin up part and the spin down part separately in their own matrices. The "setUpDistances" function sets up a matrix containing the distances between particle pairs, i.e. element ij is the distance between particle i and particle j. The "setUpJastrowMat" function sets up matrices for the Jastrow factor and its gradient. The latter two functions are used to optimize performance and will be discussed more later. The sub classes of the WaveFunction class also contains functions for updating the matrices at every Metropolis step.

The "evaluate" function evaluates the total wave function which is used when computing the kinetic energy using numerical differentiation in the Hamiltonian class. Since we split the Slater determinant into a spin up part and a spin down part, we need to implement a special case for when we only have one particle in the system. Since the one particle can't have both spin up and spin down, one of the Slater matrices will be empty, and since we use the product of the spin up determinant and the spin down determinant normally, we have to choose one of them and exclude the other. Here we have chosen to

give the single particle spin up in the one particle case.

```
1  double ManyElectrons::evaluate(std::vector<class Particle*> particles) {
2      // Evaluates the wave function using brute force.
3
4      mat spinUpSlater;
5      mat spinDownSlater;
6      if (m_numberOfParticles == 1) {
7          spinUpSlater = zeros<mat>(m_numberOfParticles, m_numberOfParticles);
8          spinDownSlater = zeros<mat>(m_numberOfParticles, m_numberOfParticles);
9          spinUpSlater(0,0) = m_SPWFMat(0,0);
10     }
11     else {
12         spinUpSlater = zeros<mat>(m_halfNumberOfParticles,
               m_halfNumberOfParticles);
13         spinDownSlater = zeros<mat>(m_halfNumberOfParticles,
               m_halfNumberOfParticles);
14
15         for (int i=0; i < m_halfNumberOfParticles; i++) {
16             for (int j=0; j < m_halfNumberOfParticles; j++) {
17                 spinUpSlater(i,j) = m_SPWFMat(i,j);
18                 spinDownSlater(i,j) = m_SPWFMat(i+m_halfNumberOfParticles,j);
19             }
20         }
21     }
22
23     double beta = m_parameters[1];
24     double exponent = 0;
25     if (m_Jastrow) {
26         for (int i=0; i < m_numberOfParticles; i++) {
27             for (int j=i+1; j < m_numberOfParticles; j++) {
28                 exponent += m_JastrowMat(i,j);
29             }
30         }
31     }
32
33     double waveFunction;
34     if (m_numberOfParticles == 1) {
35         waveFunction = det(spinUpSlater)*exp(exponent);
36     }
37     else {
38         waveFunction = det(spinDownSlater)*det(spinUpSlater)*exp(exponent);
39     }
40
41     return waveFunction;
42 }
```

Similarly, there are functions for evaluating the gradient and Laplacian of the full wave function. The gradient for the Slater part and Jastrow (correlation) part of the wave function are computed separately and then combined afterwards, which is also the case for the Laplacian. The full gradient is simply the two parts added together, but for the Laplacian we get an additional cross term equal to two times the gradient parts multiplied together. This is shown in Eq. (A.2.17). There is also a function for calculating the derivative of the wave function with respect to the variational parameters, which is needed when doing the variation of parameters.

The Metropolis ratio is also calculated in the sub classes of the WaveFunction class, and in our case, since the full wave function is made up of a Slater determinant and a Jastrow factor, we can write the Metropolis ratio as a product of a Slater part and a Jastrow part. From Eq. (2.1.16) we know that the expression for the Slater part of the

ratio is

$$R_{SD} = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}), \tag{6.1.1}$$

where ϕ are the single particle wave functions, and d_{ji}^{-1} is element ji of the inverse Slater matrix. From Eq. (2.1.20) we know the expression for the Jastrow part

$$R_C = \exp\left(\sum_{i=1, i\neq k}^{N} f_{ik}^{\text{new}} - f_{ik}^{\text{old}}\right), \tag{6.1.2}$$

where N is the number of particles and

$$f_{ij} = \frac{a r_{ij}}{(1+\beta r_{ij})}. \tag{6.1.3}$$

In the code we tabulate the necessary values in matrices that are maintained in other functions, which makes the code for the Metropolis ratio very clean with just a few simple loops.

```
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;
```

Listing 6.9: The computation of the Metropolis ratio when the full wave function consists of a Slater determinant and a Jastrow factor. The ratio can be split into a Slater part and a Jastrow part which are multiplied together to form the full ratio. The "m_spinUpSlaterInverse" matrix is the inverse of the spin up Slater matrix, and similarly for the spin down matrix, while "m_SPWFMat" is a matrix containing the single particle wave functions, i.e. it is the spin up and spin down Slater matrices stored together in one matrix. The "m_JastrowMat" matrix contains the exponent of the Jastrow factor for all particle pairs for the trial state, while the "m_JastrowMat" is the same matrix, but for the last accepted state (i.e. the state before the trial state).

Since our Slater matrix is split into a spin up and a spin down part, the sum for the Slater part of the ratio only needs to run over the particles with the same spin as the moved particle. The first if-test checks the spin of the moved particle, since the spin up particles make up the first half of the particles (i.e. the particles with index $0 \leq i < N/2$, N being the total amount of particles), while the spin down particles make up the other half.

The main difference between the two WaveFunction sub classes we're using is how the Slater matrix is set up and maintained. The Slater determinant is made up of the single particle wave functions, so for the "ManyElectrons" sub class we just make calls to the "evaluateSingleParticleWF" function in the sub classes of the Hamiltonian class, which uses a closed form expression for the wave function, and use the returned values as elements in the Slater matrices. For the "ManyElectronsCoefficents" sub class however, it is not quite as simple. The method used in this sub class aims to use the harmonic oscillator single particle wave functions as basis functions to approximate the single particle wave functions of more complicated external potentials. We need to implement Eq. (3.3.1), and use the resulting single particle wave functions to fill the Slater matrices. We restate the equation here for convenience

$$\psi_{n'}(x) = \sum_{n_x=0}^{\Lambda} C_{n',n_x}\phi_{n_x}(x). \tag{6.1.4}$$

The overlap coefficients C_{n',n_x} are provided by the diagonalization program, while $\psi_{n_x}(x)$ are the harmonic oscillator single particle wave functions we use as basis functions. We see from the equation that every element in the Slater matrices now has to be a sum over overlap coefficients and basis functions.

```
1  for (int j=0; j<m_halfNumberOfParticles; j++) {
2      vec n(m_numberOfDimensions);
3      for (int d = 0; d < m_numberOfDimensions; d++) {
4          n[d] = m_quantumNumbers(j, d);
5      }
6
7      m_SPWFMat(i,j) = m_system->getHamiltonian()->evaluateSingleParticleWF(n, r_i,
           j);
8
9      m_SPWFDMat(i,j) = m_system->getHamiltonian()->computeSPWFDerivative(n, r_i, j
           );
10
11     m_SPWFDDMat(i,j) = m_system->getHamiltonian()->computeSPWFDoubleDerivative(n,
           r_i, j);
12 }
```

Listing 6.10: Loop updating the Slater matrices when closed form expressions for the single particle wave functions are used. i is the index of the particle that was moved in the current Metropolis step, while r_i is the position of that particle. For each changed element we only need to call functions which evaluate closed form expressions for the single particle wave function and its gradient and Laplacian.

```
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) = 0;

    m_SPWFDMat(i,j) = zeros(m_SPWFDMat(i,j).size());

    m_SPWFDDMat(i,j) = 0;

    for (int eig = 0; eig < m_numberOfEigstates; 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 6.11: Loop updating the Slater matrices when the single particle wave functions are expanded in a basis of harmonic oscillator functions. i is the index of the particle that was moved in the current Metropolis step, while r_i is the position of that particle. We need an additional loop over basis functions compared to if we use closed form expressions for the single particle wave functions. We need two quantum number variables, one for the basis functions, and one which corresponds to the single particle wave function we're trying to approximate.

For the inverse Slater matrices (spin up and spin down) we use the updating algorithm explained in section A.2. Since the single particle wave functions are stored in the Slater matrices discussed above, the updating algorithm for the inverse Slater matrices only needs the elements of those Slater matrices, a copy of the inverse matrices from before the update, and the Slater part of the Metropolis ratio. The code for the updating algorithm for the spin up inverse Slater determinant is listed below. The if-test checks if the particle i moved for the current Metropolis step is a spin up particle or a spin down particle. There are then two loops over j which are essentially two parts of a sum over all particles $j \neq i$, calculating the elements which do not correspond to the moved particle i. Finally there is a loop for calculating the elements which do correspond to the moved particle.

```
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 6.12: Updating algorithm for the spin up inverse Slater matrix. i is the particle moved at the current Metropolis step, and the if-test checks if that particle has spin up. There are three loops for updating all of the elements, two of which update the $j \neq i$ elements, while the final loop updates the elements where $j = i$. We need at least two loops here since the elements are updated differently depending on whether $j = i$ or not. We split the $j \neq i$ loop into two loops, as seen here, in order to avoid needing an "if $j \neq i$"-test inside the loop (which would slow down the program). Instead we have one loop for all $j < i$ and one for all $j > i$.

6.1.6 Variation of Parameters

The variation of parameters is done by the loop listed below using the steepest descent method. At the start of each iteration we set up an initial state which uses the updated variational parameters from the previous iteration. We run a Monte Carlo simulation with few cycles and find the expectation values discussed in section 2.3. We use these expectation values to find the derivative of the local energy with respect to each of the variational parameters and then update the parameters according to Eq. (2.3.1). When we run the program in parallel the Monte Carlo simulation varies somewhat from node to node, so each node finds its own value for the derivative of the local energy. However, we want the changes to the variational parameters to be the same for all of the nodes, so we take the average of the derivatives found and use that to calculate the new parameters.

Then the new parameters are broadcast to all of the nodes. The loop continues until the sum of the change to the parameters is below a chosen tolerance.

```
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]);

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

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

    //parametersNew = parameters - derivative*m_stepLengthSD;
    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;              //"\033[F";
    }


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

Listing 6.13: Loop for variation of the variational parameters using the steepest descent method.

6.1.7 Testing the Code

To test the code we can compare the results for systems without interaction to known benchmarks (either analytical results, or results from other types of simulation). We also want to see how many basis functions we need in order to get good results when creating the trial wave function from the basis functions. The number of Monte Carlo cycles used is $1e4$ unless otherwise is stated. Since we're using harmonic oscillator wave functions as basis function we should only need a few basis functions when the external potential of the system is also a harmonic oscillator. It turns out that we need one basis function for every eigenstate in the ground state of the system. The number of eigenstates in the ground state depends on how many particles we have and two particles can share an eigenstate if one of them is spin up and the other is spin down. Therefore the number of eigenstates in the ground state is equal to half of the total number of particles. If we try to use fewer basis functions when creating the trial wave function, the Slater matrix will be singular. Instead of setting the number of basis functions directly, we set the number of energy levels the basis functions fill up. If we set the number of energy levels to n, the total amount of basis functions is

$$N_B = \frac{n(n+1)}{2}, \tag{6.1.5}$$

in the two dimensional case, or

$$N_B = \frac{n(n+1)(n+2)}{6}, \tag{6.1.6}$$

in the three dimensional case.

6.1.7.1 Benchmarks for Verifying the Implementation

In order to verify that the implementation works correctly we should compare our results with known benchmarks. In the unperturbed case the exact ground state energies are analytically known. For a two-dimensional single harmonic oscillator we have that the energy for a given electron is given by Eq.(5.2.1). Table 6.1 contains the total energy of unperturbed closed shell systems up to $N = 12$ electrons in two dimensions, as well as the energy for the one particle case. We similarly have analytically known benchmarks for the three dimensional case as well, which are listed with the corresponding results. For double harmonic oscillator well we treat the two wells as independent of each other in the non-interacting case, and assume that half of the particles are in each well. Benchmarks for the double well can then be derived from the single well benchmarks. For the finite square well we don't have analytically known benchmarks, but we use benchmarks from the diagonalization of the single particle problem. This is explained in detail in Section 6.1.7.4.

N	E
1	1ω
2	2ω
6	10ω
12	28ω

Table 6.1: Benchmarks of the energy E for unperturbed closed shell systems in two dimensions with N electrons.

6.1.7.2 Single Harmonic Oscillator Well

For the single harmonic oscillator well we should be able to get good results with very few basis functions. From Table 6.2 and 6.3 we see that in order to get results which are consistent with the benchmarks we only need the minimum amount of basis functions.

N	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
1	1	1ω	1	1
2	2	2ω	1	1
6	10	10ω	2	3
12	28	28ω	3	6

Table 6.2: Results for a system with a harmonic oscillator external potential in two dimensions, with oscillator frequency $\omega = 1$ and N particles. The results are consistent with the benchmarks when using the minimum amount of basis functions to create the trial wave function. The basis functions column shows the number of basis functions we loop over when creating the trial wave function. The energy levels column is the amount of energy levels those basis functions fill up. For each spin up and spin down particle pair we need one basis function, otherwise the Slater determinant becomes singular. So for 1 or 2 particles we only need 1 basis function, while for 6 particles we need 3 basis functions. In general the number of basis functions has to be greater or equal to half of the total amount of particles.

6.1.7.3 Double Harmonic Oscillator Well

For a system with a double harmonic oscillator as the external potential, the amount of harmonic oscillator basis functions needed to create a good trial wave function is much higher than for the regular harmonic oscillator potential. Nevertheless, the code should be able to reproduce benchmark results if enough basis functions are used. So long as the number of particles is small enough to not "spill" over between the wells, the wells can be considered as two separate single harmonic oscillator wells in the non-interacting case. The potential barrier between the wells increases with the distance between the well centers. This is because an increased distance between the wells also means an increased distance between the well centers and the potential barrier. Since the well potentials increase when moving away from the center, the potentials will be greater when they meet to form the barrier. An increase in the potential barrier also means an increase in the amount of particles we can put in the wells before they start "spilling"

N	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
1	1.5	1.5ω	1	1
2	3	3ω	1	1
8	18	18ω	2	4
20	60	60ω	3	10

Table 6.3: Results for a system with a harmonic oscillator external potential in three dimensions, with oscillator frequency $\omega = 1$ and N particles. Similarly to the two dimensional case (6.2), we only need the minimum amount of basis functions to get benchmark consistent results. The basis functions column shows the number of basis functions we loop over when creating the trial wave function and is equal to the amount of basis functions we use. The energy levels column is the amount of energy levels those basis functions fill up. For each spin up and spin down particle pair we need one basis function, otherwise the Slater determinant becomes singular. So for 1 or 2 particles we only need 1 basis function, while for 8 particles we need 4 basis functions. In general the number of basis functions has to be greater or equal to half of the total amount of particles.

over the potential barrier. The benchmarks in Table 6.6 and 6.8 assume that the wells can be treated as separate with no "spilling" between them and with the same amount of particles in each well.

The benchmarks are found in two separate ways. The first one is to run VMC simulations of a double well, but by using a super position of two harmonic oscillator functions to create the single particle wave functions, instead of expanding the diagonalization result in a basis of harmonic oscillator functions. This way of simulating the double well is the same as Jørgen Høgberget used in his master thesis [22]. The other way to find the benchmarks is to look at the single particle energies we get from diagonalizing the single particle problem. For a two-dimensional well which is double in the x direction ($L_x = 4$) and single in the y direction ($L_y = 0$) the 5 lowest eigenvalues we get, using $N = 10000$ grid points (steps), are listed in Table 6.4. From the table we see that for the y dimension the values are just about what we would expect from a regular harmonic oscillator well, i.e. $0.5, 1.5, 2.5, \ldots$, but with some minor errors due to the limited number of steps used in the diagonalization. For the x dimension we get the same values, however each value is listed twice due to the double well, so we get pairs of identical eigenvalues. If we included higher eigenvalues we would eventually see eigenvalues which stray from the single well pattern, because the eigenvalues would lie above the potential barrier between the wells. If we had an infinite amount of steps N, and a high enough potential barrier, the eigenvalues would be as listed in Table 6.5. Using these eigenvalues we can find the single particle energies for various eigenstates. We see that the eigenstates with lowest energy would be ones with $E_x = E_y = 0.5$, and to achieve this we have two options. We can use the eigenvalue number 0 for both x and y, or we can use number 1 for x and number 0 for y. As a result we have two eigenstates on the lowest energy level. For fermions we can have two particles in each eigenstate (one spin-up and one spin-down), so the lowest energy level can fit up to 4 particles. A system with 4 particles filling up the lowest energy level would then be a closed shell system with one shell, and therefore 4 is the first so-called magic number for this double well. This system would have a

non-interacting energy

$$E = (0.5 + 0.5) \times 2 + (0.5 + 0.5) \times 2 = 4. \tag{6.1.7}$$

To find out how many particles fit in the second shell we need to find all the combinations of E_x and E_y which give the second lowest energy. Since the second lowest eigenvalues have a value of 1.5 the second lowest energy is achieved when either $E_x = 0.5$ and $E_y = 1.5$ or when $E_x = 1.5$ and $E_y = 0.5$. From Table 6.5 we see that there are two combinations which give $E_x = 0.5$ and $E_y = 1.5$, and two which give $E_x = 1.5$ and $E_y = 0.5$, so in total there are 4 eigenstates on the second lowest energy level. Again we can have two particles in each eigenstate, so the second shell can fit a total of 8 fermions, and the second magic number is then $4 + 8 = 12$. The non-interacting energy of a full second shell would be

$$E = (0.5 + 1.5) \times 2 + (0.5 + 1.5) \times 2 + (1.5 + 0.5) \times 2 + (1.5 + 0.5) \times 2 = 16, \tag{6.1.8}$$

and the energy for a system with 12 particles filling up the two lowest shells would be

$$E = 4 + 16 = 20. \tag{6.1.9}$$

#	E_x	E_y
0	0.50000	0.50000
1	0.50000	1.50000
2	1.49999	2.50000
3	1.50001	3.50000
4	2.49989	4.49999

Table 6.4: First few eigenvalues from diagonalizing the single particle problem with a double harmonic oscillator well potential in the x dimension. The distance between a well center and the potential barrier is $L_x = 4$, and the number of steps used when diagonalizing is $N = 10000$.

#	E_x	E_y
0	0.5	0.5
1	0.5	1.5
2	1.5	2.5
3	1.5	3.5
4	2.5	4.5

Table 6.5: The first few ideal eigenvalues we would get from diagonalizing the single particle problem with a double harmonic oscillator well potential in the x dimension if we had an infinite amount of steps N.

From Table 6.6 we see that we need more basis functions to get a good result for 1 particle than we do for 2. The program is set up to handle two or more particles (for splitting the Slater determinant etc.), so the one particle case had to be implemented separately. Therefore there is a possibility that there may be some fault in the code for

the one particle case. Another possibility is that the one particle case could simply be more complicated to simulate with a double well external potential than the two particle case. In the two particle case the distribution of the particles would simply be one in each well, but for the one particle case, the particle could be in either well and still give the same ground state energy, essentially giving us two ground states. Either way the result is consistent with the benchmark given enough basis functions. From the table we see that increasing the number of basis functions improves the results as it should.

We also see that for 12 particles the result is not consistent with the benchmark, but greatly increasing the number of basis functions doesn't provide a significantly better result. In this case the energy converges towards a value different from the benchmark value. There seems to be some issue with the code which causes some particles to end up on a higher energy level than they are supposed to in this double well case. If we look at the results for more numbers of particles around 12 we see that, we get proper results for 8, 10, 14, 16, and 18 particles, but the problem reoccurs for 20 particles. These results are listed in Table 6.7.

N	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
1	0.993374	1ω	40	820
2	1.9996	2ω	23	276
2	2.00002	2ω	40	820
4	4.00034	4ω	25	325
12	22.0009	20ω	27	378
12	22.0001	20ω	35	630

Table 6.6: Results for a system with a double harmonic oscillator external potential in two dimensions, with oscillator frequency $\omega = 1$, N particles, and $L_x = 4$ being the distance between the center of each well and the potential barrier between them. The results are consistent with the benchmarks for 1, 2 and 4 particles, but we need a large amount of basis functions to get good results. For the 12 particle case the result is not consistent with the benchmarks, indicating that there is some fault in the code which affects this specific case, but not the lower particle cases. The basis functions column shows the number of basis functions we loop over when creating the trial wave function. The energy levels column is the amount of energy levels those basis functions fill up.

N	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
8	12	12ω	40	820
10	16	16ω	40	820
14	26	26ω	40	820
16	32	32ω	40	820
18	38	38ω	40	820
20	46	44ω	40	820

Table 6.7: More results for the same system as in Table 6.6. Here we check if the issue for 12 particles in Table 6.6 reoccurs for other numbers of particles close to 12. We see that for 8, 10, 14, 16 and 18 particles the results are good, but for 20 particles the problem reoccurs.

In the three dimensional case we see from Table 6.8 that the same problem occurs, but here it occurs already for 4 particles. For the one particle and two particles cases we're able to reproduce the benchmarks, but not for 4 and 16 particles.

N	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
1	1.4925	1.5ω	40	11480
2	3.00192	3ω	23	2300
4	8.00019	6ω	25	2925
16	42	36ω	30	4960

Table 6.8: Results for a system with a double harmonic oscillator external potential in three dimensions, with oscillator frequency $\omega = 1$, N particles, and $L_x = 4$ being the distance between the center of each well and the potential barrier between them. The results are consistent with the benchmarks for 1 and 2 particles, but we need a large amount of basis functions to get good results. For 4 and 16 particles the result is not consistent with the benchmarks, so the issue for 12 particles in two dimensions reoccurs already for 4 particles in the three dimensional case. The basis functions column shows the number of basis functions we loop over when creating the trial wave function. The energy levels column is the amount of energy levels those basis functions fill up.

The issues with the double well occurred because when the VMC solver used the coefficients corresponding to the eigenvalues listed in Table 6.4 it used the indices in Table 6.9 to index the coefficients matrix. One pair of x and y indices would cover one eigenstate and two particles (due to spin).

#	I_x	I_y
1	0	0
2	1	0
3	0	1
4	2	0
5	1	1
6	0	2
7	3	0

Table 6.9: The indices previously used to index the coefficient matrix for a double well potential. I_x are the indices for the x dimension and I_y for the y dimension.

When we compare Table 6.5 and Table 6.9 we see that the indices in the later table are ordered wrong. The 6'th set of indices in Table 6.9 would give the eigenvalues $E_x = 0.5$ and $E_y = 2.5$ with the energy sum

$$E = 0.5 + 2.5 = 3, \tag{6.1.10}$$

however the 7'th set would give $E_x = 1.5$, $E_y = 0.5$ and

$$E = 1.5 + 0.5 = 2. \tag{6.1.11}$$

So the $7'$th set of indices corresponds to an eigenstate with lower energy than the 6'th

set does. This wasn't a problem up to and including 10 particles since then only the first 5 sets of indices was used. For 14 particles we would include both the 6'th and 7'th set of indices, but in wrong order, though that doesn't change the outcome in the non-interacting case. In the 12 particle case however, we would include the 6'th set when we were supposed to have the 7'th set instead. The result was that two particles ended up one energy level higher than they should, which resulted in the total energy being 2 units higher than the benchmark (for $\omega = 1$). If we expanded Table 6.4 and Table 6.9 further we would see that (as for the 14 particle case) the wrong ordering of set 6 and 7 would no longer matter, but the same issue arises for the 10'th set, which is why the 20 particle case was wrong as well. This was only an issue for the double well, since we then have pairs of similar eigenvalues for the dimension the double well is in (the x dimension in this case). For a single well the indices listed in Table 6.9 are correct. For the double well we need to reorder the indices so we get the correct order, which for the limited span of Table 6.9 would be to swap places of the 6'th and 7'th set. In general the reordering is done by finding the sum of I_x and I_y, but with the indices halved if we have a double well in that dimension, so in this case every I_x would be halved when added to the sum. Then we order by sum, from lowest to highest. Note that we keep the original values of the indices, the halving of I_x is only in the scope of the sum in order to find the correct order. The 7'th set in Table 6.9 then has an ordering sum of 1.5 and comes before the 6'th set which has a sum of 2. The new and correct results for the previously wrong cases, both in 2 and 3 dimensions, are listed in Table 6.10.

Dimensions	N	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
2	12	19.9998	20ω	27	378
2	20	44.0003	44ω	35	630
3	4	5.99929	6ω	25	2925
3	16	36.0002	36ω	30	4960

Table 6.10

6.1.7.4 Finite Square Well

When it comes to the finite square well potential, we don't have any exact benchmarks to compare our results to. However, when diagonalizing the one particle problem, in addition to the eigenvectors we need to find the overlap coefficients, we also get eigenvalues which correspond to the single particle energies at various energy levels. Since we diagonalize separately for each dimension we get two (or three) identical sets of eigenvalues as shown in Table 6.11. To find the total energy value of a given eigenstate we need to add together one eigenvalue from each dimension. The lowest eigenstate is of course the state whose energy is the sum of the lowest eigenvalues for each dimension, and since we're looking at fermions, two particles can occupy this state. The next eigenstates are those which have the second lowest eigenvalue for one of the dimensions and the lowest eigenvalue for the rest of the dimensions. This is equivalent to the n_x, n_y and n_z quantum numbers we use with the harmonic oscillator, so the lowest state is the one where $n = n_x + n_y + n_z = 0$ and $n_x = n_y = n_z = 0$, while the next states are the ones where $n = 1$, which are the states $(n_x, n_y, n_z) = (1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$.

#	E_x	E_y	E_z
0	0.165983	0.165983	0.165983
1	0.623589	0.623589	0.623589
2	1.046890	1.046890	1.046890
3	1.079870	1.079870	1.079870
4	1.185600	1.185600	1.185600

Table 6.11: This table lists some of the eigenvalues we get when diagonalizing the single particle problem for a finite square well external potential. Each dimension is diagonalized separately. The oscillator frequency is $\omega = 1$, the distance from the center of the well to each wall is 2 and consequently the width of the well is 4. Everywhere inside the well the potential is zero, and outside the well it is $V_0 = 1$. The number of grid points (or steps) used is $N = 5000$. The eigenstate with lowest energy would be the one with the lowest eigenvalue for every dimension. The three eigenstates with second lowest energy would be those which have the second lowest eigenvalue for one of the dimensions and the lowest eigenvalue for the other dimensions, and so on for the eigenstates with higher energies.

We can use these eigenvalues we find as benchmarks when excluding interaction since the non-interacting energy is simply the sum of single particle energies. We have to be aware though, that since the benchmarks come from a simulation, they are dependent on the precision we use in the simulation. In this case the precision is dependent on how many grid points (or steps) we use when diagonalizing. In Table 6.12 we've listed some resulting eigenvalues for various number of steps N. From the table we see that if we increase the number of steps past 5000, the difference in the sum of the lowest eigenvalues is relatively small. We also see that for higher eigenvalues the difference between 5000 and 10000 steps is even smaller than it is for the lowest eigenvalues. As a result, in order to limit computational cost we will be using 5000 steps unless otherwise is stated.

Due to the benchmarks not being exact, we won't know for sure how close we are to the exact solutions, but getting results which are reasonably consistent with the benchmarks should indicate that the program is working properly. Another thing to note is that not only the benchmarks, but also the results will be dependent on the number of steps as shown in Table 6.13. In addition the results will also depend on the number of basis functions used and the number of Monte Carlo cycles used (shown in Table 6.14). It appears that the number of basis function used to expand the single particle wave functions is more important than the number of Monte Carlo cycles used. For the following test we will keep the number of Monte Carlo cycles constant at $1e4$ cycles to avoid long computation times.

In Table 6.15 we've listed the results for the square well potential in two dimensions. Just as for the double well potential, the one particle case requires significantly more basis functions than the two particle case in order to give good results. For some of the listed cases it seems like the results get worse (compared to the benchmarks) when we increase the number basis functions used. However, as discussed earlier the benchmarks here aren't the exact solutions, so it's entirely possible that the results in some cases are closer to the exact solution than the benchmarks themselves. The important thing to note is that the results are reasonably consistent with the benchmarks for all cases,

N (steps)	Eigenvalue number	E_x	E_y	Sum
100	0	0.177361	0.177361	0.354722
1000	0	0.166936	0.166936	0.333872
2000	0	0.166341	0.166341	0.332682
3000	0	0.166142	0.166142	0.332284
4000	0	0.166043	0.166043	0.332086
5000	0	0.165983	0.165983	0.331966
10000	0	0.165864	0.165864	0.331728
100	14	3.56071	3.56071	7.12142
1000	14	3.60699	3.60699	7.21398
5000	14	3.60700	3.60700	7.21400
10000	14	3.60695	3.60695	7.21390

Table 6.12: This table shows how the eigenvalues we get from diagonalizing the single particle problem for a finite square well external potential varies depending on the number of grid points (or steps) we use when diagonalizing. For all the listed cases the oscillator frequency is $\omega = 1$, the distance from the center of the well to each wall is 2 and consequently the width of the well is 4. The potential is zero inside the well and $V_0 = 1$ outside the well. We see that the eigenvalues we get become smaller and smaller as we increase the number of steps N. However, the difference becomes small as N gets large, so the difference between the eigenvalues for $N = 100$ and $N = 5000$ is much greater than the difference for $N = 5000$ and $N = 10000$. We also see that the differences are somewhat large for small eigenvalues, and become less significant as the eigenvalues increase.

N (steps)	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
5000	0.663081	0.663932	10	55
5000	0.661318	0.663932	20	210
10000	0.661050	0.663456	20	210
5000	0.662659	0.663932	30	465

Table 6.13: Here we see how the results for two particles in a finite square well potential depends on the number of basis functions used, and how both the results and the benchmark varies with the number of steps N used when diagonalizing the single particle problem. Since not only the results, but also the benchmarks vary, it's hard to say how close to the exact solution the results are. However, the VMC simulation reproduces the benchmarks well, which indicates that the program is working as intended. All results listed in this table are for a two dimensional system with oscillator frequency $\omega = 1$. The distance from the center of the well to each wall is 2 and consequently the width of the well is 4. The potential is zero inside the well and $V_0 = 1$ elsewhere.

which is a good indicator that the program is working correctly.

From Table 6.16 we see that for the three dimensional case, the results are also fairly consistent with the benchmarks. For the 20 particle case the result is not quite as good as for lower number of particles, but this is due to an insufficient number of basis functions

N (steps)	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions	MC cycles
5000	0.659647	0.663932	5	15	$1e4$
5000	0.665143	0.663932	5	15	$1e5$
5000	0.663820	0.663932	5	15	$1e6$
5000	0.665083	0.663932	5	15	$1e7$

Table 6.14: Here we see how the results for two particles in a finite square well potential depends on the number of Monte Carlo cycles used in the VMC simulation. We see that there is little change in the result when significantly increasing the number of Monte Carlo cycles. However, for simulations with more particles and with interactions included, there is a possibility that more than $1e4$ Monte Carlo cycles are needed. All results listed in this table are for a two dimensional system with oscillator frequency $\omega = 1$. The distance from the center of the well to each wall is 2 and consequently the width of the well is 4. The potential is zero inside the well and $V_0 = 1$ elsewhere.

N (particles)	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
1	0.333050	0.331966	40	820
2	0.663081	0.663932	10	55
2	0.661177	0.663932	23	276
6	3.828040	3.822220	10	55
6	3.808650	3.822220	25	325
12	11.285200	11.168068	40	820
12	11.020500	11.168068	50	1275

Table 6.15: Results for a two dimensional system with a finite square well as the external potential. The oscillator frequency is $\omega = 1$, and the number of particles is N. The distance from the center of the well to each wall is 2 and consequently the width of the well is 4. The potential is zero inside the well and $V_0 = 1$ elsewhere. The basis functions column shows the number of basis functions we loop over when creating the trial wave function. The energy levels column is the amount of energy levels those basis functions fill up. The results are consistent with the benchmarks, however the benchmarks are not the exact energies, but an approximation provided by diagonalizing the single particle problem. Just as for the double harmonic oscillator potential we see that the one particle case requires a lot more basis functions to get good results, than the two particle case does. For 2 and 6 particles we can get results which are reasonably consistent with the benchmarks by using as little as 55 basis functions. We also see that increasing the number of basis functions might increase the difference between the result and the benchmark. This could be due to the benchmarks not being exact, and in this case the results might actually be closer to the exact energies than the benchmarks are.

used. Using even more basis functions than we did would be very computationally expensive, and the result is still decently close to the benchmark, which is why we've chosen to limit the number of basis functions used.

From Table 6.15 and 6.16 we see that we can get reasonable results with as little as 55 (2D) and 220 (3D) as long as there are few enough particles to fill into two energy levels ($N \leq 6$ for 2D and $N \leq 8$ for 3D). However, once we get more particles the amount

N (particles)	steps	E (VMC)	E (Benchmark)	Energy Levels	Basis Functions
1	5000	0.502191	0.497949	30	4960
1	11500	0.499162	0.497544	40	11480
2	5000	1.008970	0.995898	10	220
2	5000	0.996780	0.995898	23	2300
8	5000	6.748820	6.729228	10	220
8	5000	6.677380	6.729228	30	4960
20	11500	23.706100	23.481330	40	11480

Table 6.16: Results for a three dimensional system with a finite square well as the external potential. The oscillator frequency is $\omega = 1$, and the number of particles is N. The distance from the center of the well to each wall is 2 and consequently the width of the well is 4. The potential is zero inside the well and $V_0 = 1$ elsewhere. The basis functions column shows the number of basis functions we loop over when creating the trial wave function. The energy levels column is the amount of energy levels those basis functions fill up. The results are consistent with the benchmarks, however the benchmarks are not the exact energies, but an approximation provided by diagonalizing the single particle problem. Just as for the double harmonic oscillator potential we see that the one particle case requires a lot more basis functions to get good results, than the two particle case does. For 2 and 8 particles we can get results which are reasonably consistent with the benchmarks by using as little as 220 basis functions. We also see that increasing the number of basis functions might increase the difference between the result and the benchmark. This could be due to the benchmarks not being exact, and in this case the results might actually be closer to the exact energies than the benchmarks are.

of basis functions needed is drastically increased. We could predict this by looking at Table 6.11. The number of states which can be made while keeping the energy under $E = 1$ is 2 for 1D, 3 for 2D and 4 for 3D, and these numbers of states are as many as can fit into the first to energy levels for the corresponding number of dimensions. Our finite square well potential has the value 0 inside the well and 1 elsewhere, so any state which would correspond to an energy $E > 1$ would be outside the well, and the particles in that state would thus be unbound.

Using the two-dimensional case as an example, three states would be inside the well, and each state can hold two particles, so we can have a total of 6 bound particles. As long as all particles in the system are bound the harmonic oscillator basis functions are a reasonably good fit to the finite square well potential. However, once we get unbound particles in the system the part of the potential outside the well also comes into use, and the basis functions are not a good fit for the shape of the potential outside the well. As a result we need a lot more basis functions to get a good approximation when there are unbound particles in the system. A visual representation of the two-dimensional case is given in Figure 6.1. Alternative basis functions, e.g. Hydrogen-like basis functions, might fit better with the finite square well potential, and should be considered when studying systems with both bound and unbound particles.

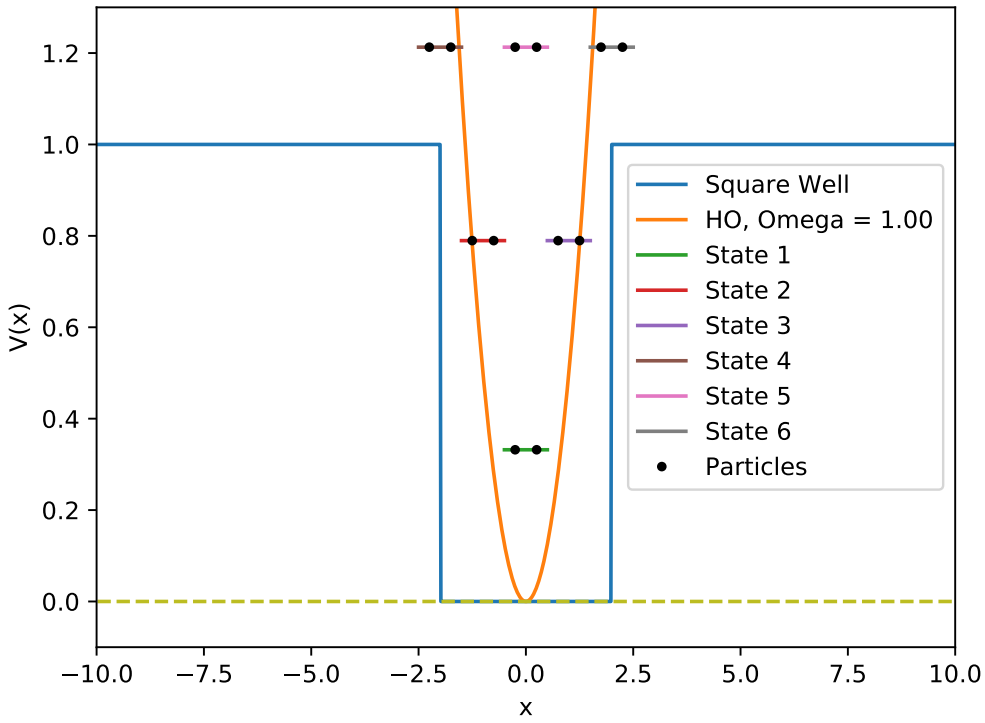


Figure 6.1: The figure shows that the lowest three particle states for our specific finite square well potential in two dimensions are bound states, while the higher states are unbound. If only the bound states are occupied then harmonic oscillator basis functions are a reasonable fit to the potential, but not if unbound states are occupied.

6.2 Diagonalization (Overlap Coefficients)

The diagonalization program is tasked with diagonalizing the one particle problem in a given potential well, and to use the resulting eigenvectors to find the overlap coefficients required to expand the solutions of the given potential well in terms of harmonic oscillator basis functions. The potential is first discretized and set up as a tridiagonal matrix. The one particle problem is then solved as an eigenvalue problem by using the Armadillo [26] function, "eig_sym", on the tridiagonal matrix in order to find the eigenvalues and eigenvectors. The eigenvectors correspond to the single particle wave functions for a particle in a corresponding eigenstate confined in the potential well. The eigenvalues are the single particle energies for the eigenstates. The program then finds the coefficients for the overlap between the eigenvectors and the harmonic oscillator basis functions. The program also expands the solutions of the potential well in the harmonic oscillator basis in order to verify the code.

6.2.1 Diagonalizing

The first step is to set up the tridiagonal matrix. See section 3.1 for an explaination of the elements in the tridiagonal matrix. In the code the tridiagonal matrix is filled by the following simple function.

```
1  void System::diagonalizeMatrix(mat r, vec L, int N, cube &diagMat, mat &
       savePotential) {
2      double Constant = 1./(2*m_h*m_h);
3      mat V(N+1, m_numberOfDimensions);
4      for (int d = 0; d < m_numberOfDimensions; d++) {
5          V.col(d) = m_waveFunction->potential(r.col(d), L(d));
6          diagMat.slice(d).diag(0)  =  2.*Constant + V.col(d).subvec(1,N-1);      //
               Set d_i elements in A
7          diagMat.slice(d).diag(1)  = -1.*Constant*ones(N-2);                    //Set
               e_i elements in A
8          diagMat.slice(d).diag(-1) = diagMat.slice(d).diag(1);
                                       //Set e_i elements in A
9      }
10
11     savePotential = V;
12     return;
13 }
```

The only non-constant elements in the matrix are the diagonal ones which depend on the external potential. The WaveFunction class of the diagoanlization program (not the same as the class from the VMC program) has sub classes for the different external potentials each containing a simple function for the potential. These functions are used to fill in the diagonal elements of the matrix. The tridiagonal matrix eigenvalue problem is solved for each dimension separately, so one tridiagonal matrix is made for each dimension, then the "eig_sym" function is used on each of the matrices. This is done in the following loop.

```
1  // Finding eigenvalues and eigenvectors using armadillo:
2  for (int d = 0; d < m_numberOfDimensions; d++) {
3      vec eigvalsTemp = eigvals.col(d);
4      eig_sym(eigvalsTemp, eigvecs.slice(d), diagMat.slice(d));
5      eigvals.col(d) = eigvalsTemp;
6  }
```

6.2.2 Finding the Overlap Coefficients

We know from section 3.2 that the overlap coefficients are given by Eq. (3.2.1), which we restate here for convenience.

$$C_{n',n} = \langle \psi_{n'} | \phi_n \rangle = \sum_{i=0}^{N-1} \psi_{n'}(x_i)\phi_n(x_i) \qquad n', n = 0, 1, 2, \ldots, \tag{6.2.1}$$

where the sum goes over the discretization steps used when diagonalizing the single particle problem. This equation is used to find a single general overlap coefficient. However, we need a matrix containing all of the overlap coefficients. We also want to find the coefficients separately for each dimension, so we want one such matrix for each dimension. Therefore, in addition to a loop for the sum in Eq. (6.2.1), we need three more loops; one for n', one for n and one for dimensions. The following function finds all the coefficients for a given dimension.

```
1  void System::findCoefficients(int nMax, int nPrimeMax, vec x, mat &C, int
       currentDim){
2      cout << "Finding coefficients for dimension " << currentDim+1 << " of " <<
           m_numberOfDimensions << endl;
3      cout.flush();
4      std::string upLine = "\033[F";
5      for (int nPrime = 0; nPrime < nPrimeMax; nPrime++) {
6          cout << "nPrime = " << nPrime << " of " << nPrimeMax-1 << endl;
7          for (int nx = 0; nx < nMax; nx++) {
8              cout << "[" << int(double(nx)/nMax * 100.0) << " %]\r";
9              cout.flush();
10             double innerprod = 0;
11             for (int i = 0; i < m_N-1; i++) {
12                 innerprod += m_psi.slice(currentDim).col(nPrime)(i)*
                       m_waveFunction->harmonicOscillatorBasis(x, nx)(i);
13             }
14             C(nx, nPrime) = innerprod;
15         }
16         cout << upLine;
17     }
18     cout << upLine;
19     C *= m_h;
20 }
```

This function is called once for each dimension and the three resulting matrices are stored together in a three-dimensional matrix, which is then saved to a file. This file is the loaded in the VMC program. There the coefficients are used with harmonic oscillator basis functions to create approximate single particle wave functions which are used in the Slater determinant.

6.2.3 Expanding the Solutions

The main goal of this program is to find the coefficients and store them so that they can be used in the simulations in the VMC program. However, we can use the coefficients together with harmonic oscillator basis functions to recreate the eigenvectors we got from diagonalizing the single particle problem, in order to test that the coefficients are correct.

We use Eq. (3.3.1) from section 3.3, which is

$$\psi_{n'}(x) = \sum_{n_x=0}^{\Lambda} C_{n',n_x}\phi_{n_x}(x). \tag{6.2.2}$$

We store the ψ's and use the Python program "plot_data.py" to compare them to the eigenvectors from the diagonalizing. For two dimensions we have the following loop.

```
1  for (int nPrime = 0; nPrime < nPrimeMax; nPrime++) {
2      for (int i = 0; i < m_numberOfEigstates; i++) {
3          int nx = m_qNumbers(i, 0);
4          int ny = m_qNumbers(i, 1);
5
6          vec plusTermX = C(nx, nPrime, 0)*m_waveFunction->harmonicOscillatorBasis(
               rCut.col(0), nx);
7          vec plusTermY = C(ny, nPrime, 1)*m_waveFunction->harmonicOscillatorBasis(
               rCut.col(1), ny);
8
9          supPos.col(nPrime) += plusTermX%plusTermY;
10         supPosSep.slice(0).col(nPrime) += plusTermX;
11         supPosSep.slice(1).col(nPrime) += plusTermY;
12     }
13 }
```

The "supPosSep" three-dimensional matrix stores all the ψ'_n for each dimension separately, while the "supPos" matrix stores the product for all dimensions.

6.2.4 Testing the Code

6.2.4.1 Double Harmonic Oscillator Well

There are several test we can use to validate the coefficients we've found. One thing we can check is that the L_2-norm is a small number:

$$||\psi_{n'}^{\text{diag}} - \psi_{n'}^{\text{exp}}||_2 = \sqrt{\sum_{i}^{N} |\psi_{n'}^{\text{diag}}(x_i) - \psi_{n'}^{\text{exp}}(x_i)|^2} < \epsilon. \tag{6.2.3}$$

Here $\psi_{n'}^{\text{diag}}$ are the solutions (eigenvectors) we got from diagonalizing the single particle problem, while $\psi_{n'}^{\text{exp}}$ are the approximations from the linear expansion in Eq. (6.2.2). The L_2-norm being less than a small number ϵ would mean that the linear expansion approximations are reasonably similar to the solutions from diagonalizing. We can also do a similar test by plotting $\psi_{n'}^{\text{diag}}$ and $\psi_{n'}^{\text{exp}}$ together for various n' and checking that the two curves are (to some precision) on top of each other. We expect the result of both of these test to be best for low n' values and then get worse and worse as n' increases. We also expect that increasing the number of basis functions used in the linear expansion should improve the results and consequently give good results for a larger number of n'.

In Table 6.17 we have listed the $L2$-norm for $n' = 0$ and $n' = 9$ with 210 and 378 basis functions used for the linear expansion. The values in the table are for a double well potential in two dimensions. We see that the $L2$-norm is generally smaller when we're using more basis function as expected, and as n' increases we need more basis functions to get a good $L2$-norm. When using 378 basis functions for the $n' = 0$ case we get a reasonably good $L2$-norm, while for the other cases we should use more basis functions.

n'	Energy Levels	Basis Functions	L_2-norm
0	20	210	0.246582153723
0	27	378	0.00288150262617
9	20	210	0.402442469552
9	27	378	0.175246533084

Table 6.17: $L2$-norm values for a double well potential in two dimensions with $\omega = 1$, $N = 1000$, $L_x = 4.0$ and $L_y = 0.0$. The basis functions column lists the number of basis functions used in the linear expansion. We see that for a given number of basis functions the $L2$-norm is better (smaller) for $n' = 0$ than for $n' = 9$, indicating that for larger n' we need a higher number of basis functions to get good results. We also see that for a given n', using more basis functions gives a better $L2$-norm.

We need one value of n' for every two particles in the system we want to study, so for a system with only two particles we only need $n' = 0$, so 378 basis functions should be sufficient. For a system with 20 particles we would need $n' = 0, 1, \ldots, 9$, so based on the $L2$-norm for $n' = 9$, using 378 basis functions probably won't be enough for this system (in Table 6.10 we used 630 basis functions to get a good result for this system).

In Figure 6.2 we see the plots of $\psi_{n'}^{\text{diag}}$ and $\psi_{n'}^{\text{exp}}$ together for $n' = 0$. Again we have used a two dimensional double well potential. For Figure 6.2a we used 210 basis functions for $\psi_{n'}^{\text{exp}}$. We see $\psi_{n'}^{\text{exp}}$ has a somewhat similar shape as $\psi_{n'}^{\text{diag}}$, but the error is significant close to $x = 0$. In Figure 6.2b we have increased the number of basis functions to 378, and as expected $\psi_{n'}^{\text{exp}}$ matches $\psi_{n'}^{\text{diag}}$ much better. Figure 6.3 is equivalent to Figure 6.2, but for $n' = 9$. Here as well we see that with 210 basis functions we get significant errors, while with 378 basis functions the match between $\psi_{n'}^{\text{exp}}$ and $\psi_{n'}^{\text{diag}}$ is much better. However, for $n' = 9$, even with 378 basis function we still don't get a great match. Just as for with the $L2$-norm, this shows that increasing the number of basis functions improves the results, and that greater n' values require a greater number of basis functions to yield good results.

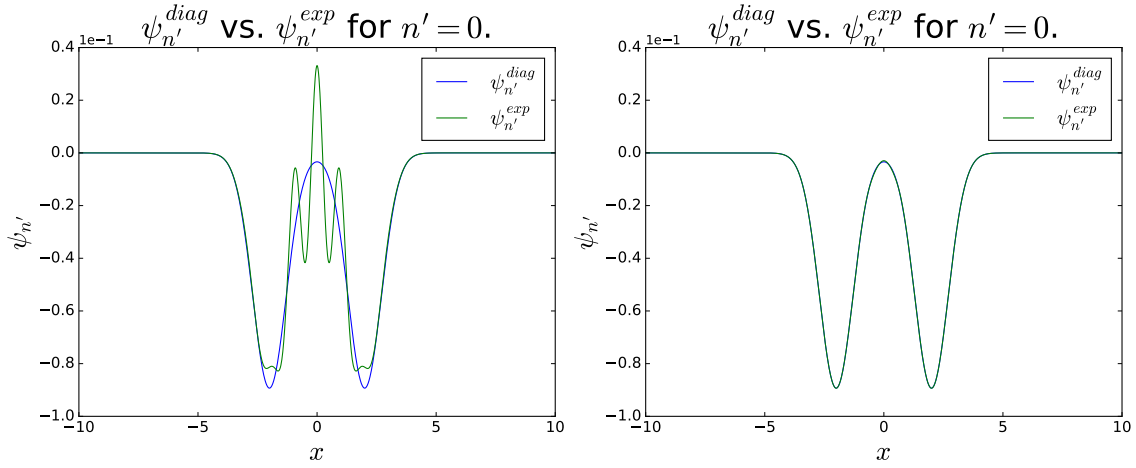
Another test we can do is to check that the norm of ψ^{exp} for each dimension separately is approximately equal to 1 and that the accuracy increases with increasing number of basis function like in the previous tests. The norm is given by

$$||\psi|| = \sqrt{\langle\psi|\psi\rangle}, \tag{6.2.4}$$

and we want the following to be true

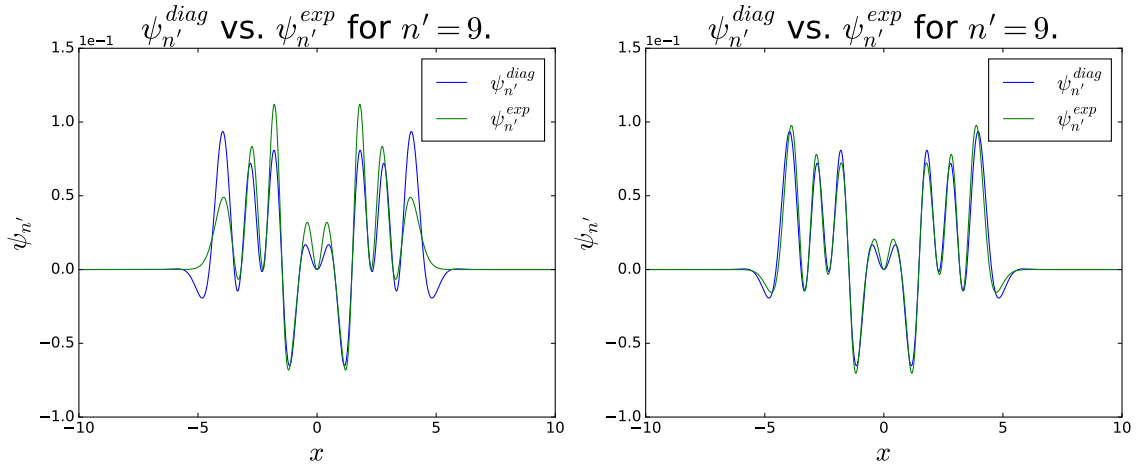
$$||\psi_{n'}^{\text{exp}}(x)|| \approx ||\psi_{n'}^{\text{exp}}(y)|| \approx ||\psi_{n'}^{\text{exp}}(z)|| \approx 1, \tag{6.2.5}$$

in the three dimensional case. For this test we will use the same system as for the two previous tests, i.e. a two dimensional system, where the potential is a double harmonic oscillator well in the x-dimension and a single harmonic oscillator well in the y-dimension. Table 6.18a and Table 6.18b lists $||\psi_{n'}^{\text{exp}}(x)||$ and $||\psi_{n'}^{\text{exp}}(y)||$ for 20 energy levels and 27 energy levels respectively. Note that since we here look at each dimension separately the number of energy levels and the number basis functions are the same. The tables list the results for $n' = 0, 1, \ldots, 9$. In both tables $||\psi_{n'}^{\text{exp}}(y)||$ is exactly 1 to machine precision for all n'. This makes sense since the potential for the y-dimension is a single harmonic



(a) 210 Basis Functions (20 Energy Levels). (b) 378 Basis Functions (27 Energy Levels).

Figure 6.2: $\psi_{n'}^{\text{diag}}$ vs. $\psi_{n'}^{\text{exp}}$ for a two-dimensional double well potential with $n' = 0$, $N = 1000$, $L_x = 4.0$, $L_y = 0$ and $\omega = 1$. We see that when we use only 210 basis functions the linear expansion approximation is a fairly bad match to the solution we got from diagonalizing. However, the shape is still somewhat similar. When we increase the number of basis functions to 378 we get a pretty good match.



(a) 210 Basis Functions (20 Energy Levels). (b) 378 Basis Functions (27 Energy Levels).

Figure 6.3: $\psi_{n'}^{\text{diag}}$ vs. $\psi_{n'}^{\text{exp}}$ for a two-dimensional double well potential with $n' = 9$, $N = 1000$, $L_x = 4.0$, $L_y = 0$ and $\omega = 1$. As for the $n' = 0$ case we get a bad match when using only 210 basis functions. Still, there are some similarities between the linear expansion approximation and the solution. When we increase the number of basis functions to 378 we get a significantly better match, but it's not quite as good as in the $n' = 0$ case.

oscillator well and we use single harmonic oscillator functions as basis functions. For the x-dimension on the other hand we have a double well potential and should therefore expect some deviation from 1. As with the previous tests we expect that the deviation should increase as we increase n'. From Table 6.18a we see that the deviation doesn't strictly increase with increasing n', but that significantly increasing n', significantly increases the deviation (e.g. going from $n' = 0$ to $n' = 9$). If we compare Table 6.18a with Table 6.18b we again see that increasing the number of basis functions improves the results for all n'.

n'	$\|\|\psi^{\text{exp}}_{n'}(x)\|\|$	$\|\|\psi^{\text{exp}}_{n'}(y)\|\|$
0	0.999816193222	1.0
1	0.999930888372	1.0
2	0.996501279797	1.0
3	0.998471329039	1.0
4	0.974110444039	1.0
5	0.986589494196	1.0
6	0.904585077720	1.0
7	0.939665912097	1.0
8	0.817567381049	1.0
9	0.850878481823	1.0

(a) 20 Basis Functions (20 Energy Levels)

n'	$\|\|\psi^{\text{exp}}_{n'}(x)\|\|$	$\|\|\psi^{\text{exp}}_{n'}(y)\|\|$
0	0.999999977026	1.0
1	0.999999918994	1.0
2	0.999998831156	1.0
3	0.999996288163	1.0
4	0.999974270670	1.0
5	0.999926388426	1.0
6	0.999675990997	1.0
7	0.999175833447	1.0
8	0.997471707200	1.0
9	0.994189907404	1.0

(b) 27 Basis Functions (27 Energy Levels)

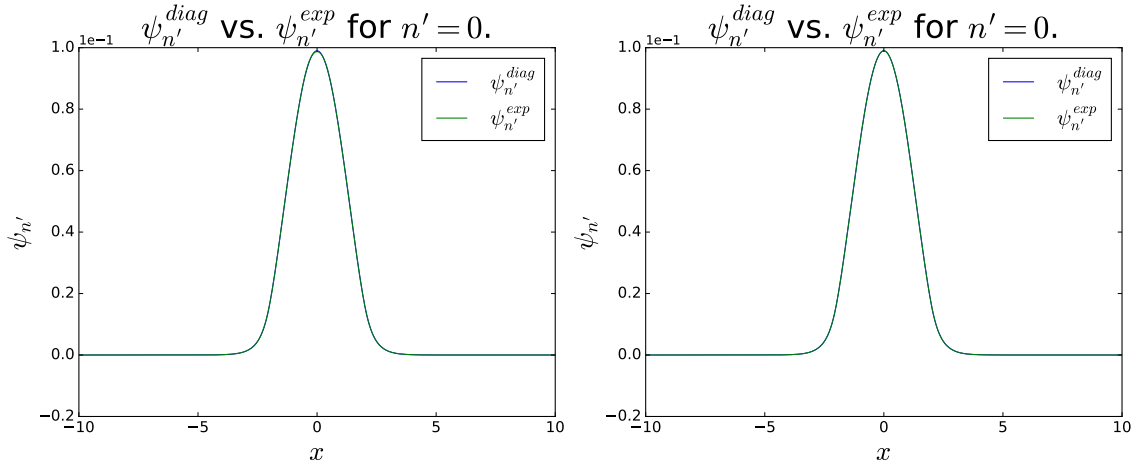
Table 6.18: $||\psi^{\text{exp}}_{n'}(x)||$ and $||\psi^{\text{exp}}_{n'}(y)||$ values for a two-dimensional double well potential with $\omega = 1$, $N = 1000$, $L_x = 4.0$ and $L_y = 0.0$. Since we're looking at each dimension separately the number of basis functions is the same as the number of energy levels. In both tables $||\psi^{\text{exp}}_{n'}(y)||$ is equal to 1 to machine precision, while $||\psi^{\text{exp}}_{n'}(x)||$ has some deviation from 1. This is due to the double well being in the x-dimension, while the potential in the y-dimension is a single harmonic oscillator well. Since the basis functions we use are single harmonic oscillator functions, the results for the y-dimension should be exact. For $||\psi^{\text{exp}}_{n'}(x)||$ we also see that the deviation from 1 is smaller when we have more basis functions and that the deviation typically increases when n' increases (but not always).

6.2.4.2 Finite Square Well

We now redo the above test, but for a finite square well potential instead of a double harmonic oscillator well potential. With the distance between the well center and each wall being 2 and $V_0 = 1$, we get the results for the $L2$'norm listed in Table 6.19. From the table we see that the $L2$'norm is better for $n' = 0$ and worse for $n' = 9$ than when we used a double well potential (Table 6.17). The $L2$-norm for $n' = 0$ is really good for both 210 and 378 basis functions, while for $n' = 9$ it's not quite as good.

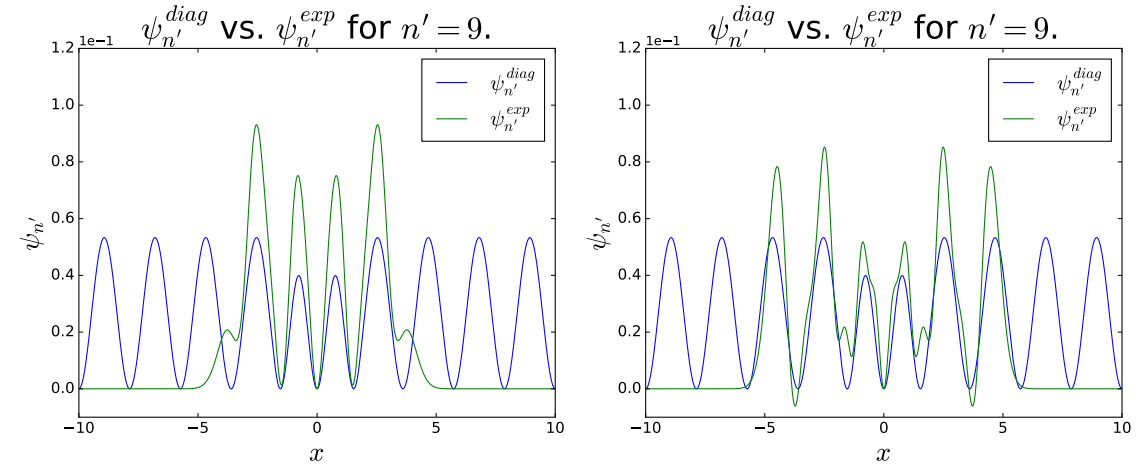
When looking at Figure 6.4 and 6.5, we see the same situation. $\psi^{\text{exp}}_{n'}$ matches $\psi^{\text{diag}}_{n'}$ really well for $n' = 0$ even for 210 basis functions, but for $n' = 9$, even with 378 basis functions, $\psi^{\text{exp}}_{n'}$ is a bad approximation to $\psi^{\text{diag}}_{n'}$.

The results for the norm $||\psi^{\text{exp}}_{n'}||$ are listed in Table 6.20a and 6.20b. Here we see that



(a) 210 Basis Functions (20 Energy Levels). (b) 378 Basis Functions (27 Energy Levels).

Figure 6.4: $\psi_{n'}^{\text{diag}}$ vs. $\psi_{n'}^{\text{exp}}$ for a two-dimensional finite square well potential with $n' = 0$, $N = 1000$, $\omega = 1$, $V_0 = 1$ and the distance between the center and each wall is 2. We see that the curves are indistinguishable with both 210 and 378 basis functions.



(a) 210 Basis Functions (20 Energy Levels). (b) 378 Basis Functions (27 Energy Levels).

Figure 6.5: $\psi_{n'}^{\text{diag}}$ vs. $\psi_{n'}^{\text{exp}}$ for a two-dimensional finite square well potential with $n' = 0$, $N = 1000$, $\omega = 1$, $V_0 = 1$ and the distance between the center and each wall is 2. We see that there's quite a big difference between the curves, but it's somewhat improved when increasing the number of basis functions from 210 to 378.

n'	Energy Levels	Basis Functions	L_2-norm
0	20	210	0.00253244921988
0	27	378	0.00119302677155
9	20	210	0.91578729592
9	27	378	0.75248968038

Table 6.19: $L2$-norm values for a finite square well potential in two dimensions with $\omega = 1$, $N = 1000$, $V_0 = 1$ and the distance between the center and each wall is 2. The basis functions column lists the number of basis functions used in the linear expansion. For $n' = 0$ the $L2$-norm is better than it was for the double well, but for $n' = 9$ it's worse.

for 210 basis functions the results are better for low n' than they were with a double well potential, but they also fall off much quicker as n' increases. We see that specifically we get good results for the first two n', which fits with what we saw about bound and unbound states in Section 6.1.7.4. Here we look at each dimension separately and in 1D two states are bound and the rest are unbound. Thus $n' = 0, 1$ correspond to bound states while $n' = 2, \ldots, 9$ correspond to unbound states. When increasing the number of basis functions from 210 to 378 the norms for all n' get closer to the target value 1. However, for 378 basis functions the results are worse than they were with a double well potential for all n'.

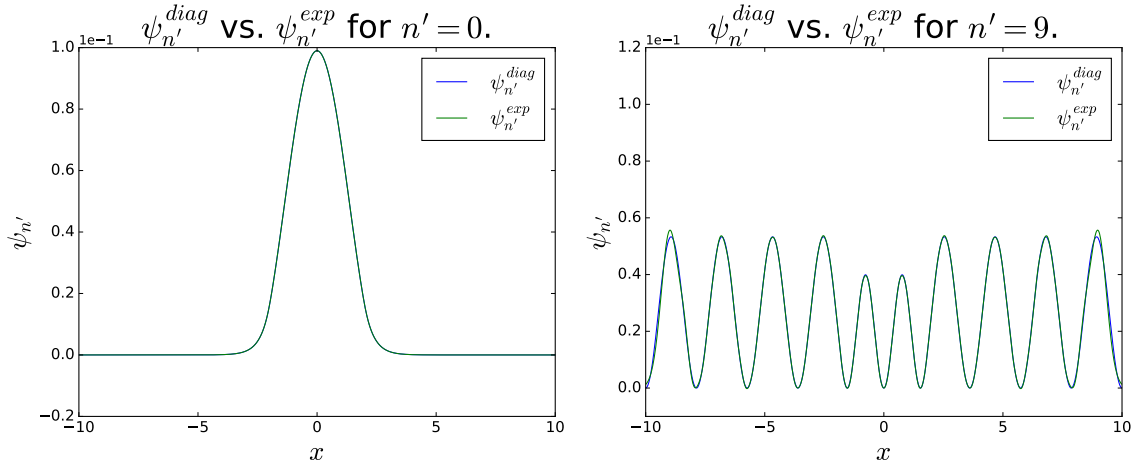
n'	$\|\|\psi^{\text{exp}}_{n'}(x)\|\| = \|\|\psi^{\text{exp}}_{n'}(y)\|\|$
0	0.999998242918
1	0.999951314496
2	0.867214266147
3	0.768857257126
4	0.740421802754
5	0.725041681081
6	0.798534488615
7	0.798512568557
8	0.753433100459
9	0.757833831802

(a) 20 Basis Functions (20 Energy Levels)

n'	$\|\|\psi^{\text{exp}}_{n'}(x)\|\| = \|\|\psi^{\text{exp}}_{n'}(y)\|\|$
0	0.9999991691
1	0.999986541
2	0.952180366269
3	0.887886200997
4	0.86139126026
5	0.779018497771
6	0.815843172148
7	0.829833024963
8	0.84973902467
9	0.859054357563

(b) 27 Basis Functions (27 Energy Levels)

Table 6.20: $||\psi^{\text{exp}}_{n'}(x)||$ and $||\psi^{\text{exp}}_{n'}(y)||$ values for a two-dimensional finite square well potential with $n' = 0$, $N = 1000$, $\omega = 1$, $V_0 = 1$ and the distance between the center and each wall is 2. Since we're looking at each dimension separately the number of basis functions is the same as the number of energy levels. Unlike for the double well, here $||\psi^{\text{exp}}_{n'}(x)||$ and $||\psi^{\text{exp}}_{n'}(y)||$ are equal since the potential looks the same in both dimensions. Since the basis functions we use are single harmonic oscillator functions, and the potential is a finite square well, the norms are not exactly equal to the target value 1. Just as for the double well potential, we see that the deviation from 1 is smaller when we have more basis functions and that the deviation typically increases when n' increases (but not always).



(a) 5050 Basis Functions (100 Energy Levels). (b) 5050 Basis Functions (100 Energy Levels).

Figure 6.6: $\psi_{n'}^{\text{diag}}$ vs. $\psi_{n'}^{\text{exp}}$ for a two-dimensional finite square well potential with $N = 1000$, $\omega = 1$, $V_0 = 1$ and the distance between the center and each wall is 2. We see that when using 5050 basis functions, not only for $n' = 0$, but also for $n' = 9$ we have a really good match between the curves of $\psi_{n'}^{\text{diag}}$ and $\psi_{n'}^{\text{exp}}$. For $n' = 9$ the match is vastly better than when using 210 and 378 basis functions.

We redo the tests once more with a huge number of basis functions (5050), to see if we can get good results for $n' = 9$. From Table 6.21 we see that the $L2$-norm for $n' = 0$ has improved even further, but now the $L2$-norm for $n' = 9$ has also reached a reasonably good level. From Figure 6.6 we see the same thing. For $n' = 0$ the curves in Figure 6.6a are still indistinguishable, like they were when using 210 and 378 basis functions. However, now for $n' = 9$ as well, the curves in Figure 6.6b are also nearly indistinguishable. There is only a slight visible difference on the left-most and right-most maxima. Also the results from Table 6.22 confirms that $\psi_{n'}^{\text{exp}}$ is a good approximation to $\psi_{n'}^{\text{diag}}$ for $n' = 9$ when we're using 5050 basis functions. From the table we see that now the norm is really close to 1 for all $n' = 0, 1, \ldots, 9$.

n'	Energy Levels	Basis Functions	L_2-norm
0	100	5050	0.000265421396605
9	100	5050	0.0281926267177

Table 6.21: $L2$-norm values for a finite square well potential in two dimensions with $\omega = 1$, $N = 1000$, $V_0 = 1$ and the distance between the center and each wall is 2. The basis functions column lists the number of basis functions used in the linear expansion. The $L2$-norm for $n' = 0$ continues to be really good, but now the $L2$-norm for $n' = 9$ has also reached a reasonably good level.

We've seen that the results for a square well are better than those for a double harmonic oscillator well when n' is small, but then much worse for large n'. A possible explanation for this is that the harmonic oscillator basis functions are a good fit for the square well as long as the energy is small enough that the finite nature of the well isn't significant. Higher n' corresponds to higher energy, and when the energy is high enough

n'	$\|\|\psi^{\text{exp}}_{n'}(x)\|\| = \|\|\psi^{\text{exp}}_{n'}(y)\|\|$
0	0.999999972833
1	0.999999902664
2	0.999998457484
3	0.999996892487
4	0.999993799793
5	0.999988771956
6	0.999984406264
7	0.999977312106
8	0.99996981249
9	0.999964256428

Table 6.22: $||\psi^{\text{exp}}_{n'}(x)||$ and $||\psi^{\text{exp}}_{n'}(y)||$ values for a two-dimensional finite square well potential with $n' = 0$, $N = 1000$, $\omega = 1$, $V_0 = 1$ and the distance between the center and each wall is 2. The number of energy levels is 100, and since we're looking at each dimension separately the number of basis functions is the same as the number of energy levels. Unlike for the double well, here $||\psi^{\text{exp}}_{n'}(x)||$ and $||\psi^{\text{exp}}_{n'}(y)||$ are equal since the potential looks the same in both dimensions. Since the basis functions we use are single harmonic oscillator functions, and the potential is a finite square well, the norms are not exactly equal to the target value 1. Now that we have significantly increased the number of basis functions compared to Table 6.20, we see that the norm is reasonably close to 1 for all $n' = 0, 1, \ldots, 9$.

that we're above the finite well, the potential is simply a straight line. At this point the harmonic oscillator basis function are probably no longer a good fit, so we need a much greater amount of them to be able to properly approximate $\psi^{\text{diag}}_{n'}$.

Chapter 7

Optimizing Performance

A VMC simulation can quickly become very time consuming as we increase the number of particles in our system. In order to maintain efficiency of computations as the number of particles increases, it is important to optimize the simulation. One fairly common trait of suboptimal code is recalculation of identical values. An expression giving a value which needs to be used multiple times, should only be calculated once and the resulting values stored for later use. Doing this for various expressions was a big part of optimizing the VMC simulation. Another important optimization of the VMC simulation is the use of polymorphism. Most of the polymorphism used was included in the code from the beginning, but some were added later when optimizing the finished code. The third major optimization used, was to make the simulation parallelized using Open MPI [27].

7.1 Storing Reused Data

Storing various values which are used several times, instead of recalculating them every time, saves a lot of CPU time. This is particularly true as the number of particles increases to large numbers. By storing the values in matrices, we can simply access the matrix element of the value we need when we need it, rather than recalculating it every time that value is needed. This section loosely follows Chapter 4.6 of Ref. [22].

7.1.1 Relative Distances

The simplest reusable values to store for the VMC simulation are the relative distances between the positions of particles. Whenever a relative distance is needed, for example to calculate the Jastrow factor, we can access the corresponding matrix element instead of recalculating the distance. All the relative distances are calculated once when we initiate the simulation and stored in a symmetric matrix on the form

$$\mathbf{r} = \mathbf{r}^T = \begin{pmatrix} 0 & r_{12} & \dots & \dots & r_{1N} \\ r_{21} & 0 & \ddots & & r_{2N} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & r_{nN} \\ r_{N1} & r_{N2} & \dots & r_{Nn} & 0 \end{pmatrix}, \tag{7.1.1}$$

where N is the number of particles and $n = N - 1$. Naturally $r_{12} = r_{21}$ since both are the relative distance between particle 1 and 2, so the matrix is symmetric. This saves us some additional computation time when creating and updating the matrix since we only have to calculate the values in either the upper or lower triangular part of the matrix, and mirror the values onto the other triangular part. Since we only move one particle at a time in the simulation, we only need to update N relative distances when a particle is moved, since moving particle 1 won't change e.g. r_{23}. The relative distance matrix is updated by the "updateDistances" function whenever a particle is moved.

```
1  void ManyElectronsCoefficients::updateDistances(int currentParticle) {
2      // Function for updating the distances between particles.
3      int i = currentParticle;
4      std::vector<double> r_i = m_system->getParticles()[i]->getPosition();
5
6      for (int j=0; j<i; j++) {
7          std::vector<double> r_j = m_system->getParticles()[j]->getPosition();
8          double r_ij = 0;
9
10         for (int d = 0; d < m_numberOfDimensions; d++) {
11             r_ij += (r_i[d]-r_j[d])*(r_i[d]-r_j[d]);
12         }
13         m_distances(i,j) = m_distances(j,i) = sqrt(r_ij);
14     }
15
16     for (int j=i+1; j<m_numberOfParticles; j++) {
17         std::vector<double> r_j = m_system->getParticles()[j]->getPosition();
18         double r_ij = 0;
19
20         for (int d = 0; d < m_numberOfDimensions; d++) {
21             r_ij += (r_i[d]-r_j[d])*(r_i[d]-r_j[d]);
22         }
23         m_distances(i,j) = m_distances(j,i) = sqrt(r_ij);
24     }
25
26 }
```

Listing 7.1: Function for updating the relative distances between particles whenever a particle is moved. i is the moved particle so only the matrix elements where one of the indices is i are changed when particle i moves. Since no particle has any distance to itself, we split the loop in two so that we can exclude the $j = i$ case without using an if-test. Since the matrix is symmetric, $r_{ij} = r_{ji}$, and thus we can halve the number of distance calculations.

7.1.2 Slater Matrices

The Slater matrices consists of single particle wave functions $\phi_j(\mathbf{r}_i)$, and these single particle wave functions are used repeatedly in the VMC simulations, to for example calculate the Metropolis ratio, update the inverse Slater matrices, etc. However, a given single particle wave functions only changes when the corresponding particles position changes. Consequently we can avoid having to recalculate these single particle wave functions by storing them in a matrix, and update the relevant elements whenever a particle is moved, just as we did for the relative distances. Whenever we need a given single particle wave function, we then simply access the corresponding element of the matrix. In the code we have named this matrix "m_SPWFmat", and it is a concatenation

of the spin-up and spin-down Slater matrices

$$\mathbf{S} = \text{join}(\mathbf{S}^{\uparrow}, \mathbf{S}^{\downarrow}) = \begin{pmatrix} \phi_1(\mathbf{r}_1) & \phi_1(\mathbf{r}_2) & \dots & \phi_1(\mathbf{r}_N) \\ \phi_2(\mathbf{r}_1) & \phi_2(\mathbf{r}_2) & \dots & \phi_2(\mathbf{r}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{N/2}(\mathbf{r}_1) & \phi_{N/2}(\mathbf{r}_2) & \dots & \phi_{N/2}(\mathbf{r}_N) \end{pmatrix}, \tag{7.1.2}$$

where $\text{join}(\mathbf{S}^{\uparrow}, \mathbf{S}^{\downarrow})$ means that we are joining the columns of the spin-up Slater matrix $\mathbf{S}^{\uparrow}$ and the spin-down matrix $\mathbf{S}^{\downarrow}$.

Not only the single particle wave functions, but also their gradients and Laplacians are used repeatedly in the simulation. Therefore to optimize further we create corresponding matrices containing these gradients and Laplacians. In the code the gradient matrix is called "m_SPWFDMat" and is on the form

$$\begin{pmatrix} \nabla\phi_1(\mathbf{r}_1) & \nabla\phi_1(\mathbf{r}_2) & \dots & \nabla\phi_1(\mathbf{r}_N) \\ \nabla\phi_2(\mathbf{r}_1) & \nabla\phi_2(\mathbf{r}_2) & \dots & \nabla\phi_2(\mathbf{r}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \nabla\phi_{N/2}(\mathbf{r}_1) & \nabla\phi_{N/2}(\mathbf{r}_2) & \dots & \nabla\phi_{N/2}(\mathbf{r}_N) \end{pmatrix}, \tag{7.1.3}$$

while the Laplacian matrix is called "m_SPWFDDMat" and is on the form

$$\begin{pmatrix} \nabla^2\phi_1(\mathbf{r}_1) & \nabla^2\phi_1(\mathbf{r}_2) & \dots & \nabla^2\phi_1(\mathbf{r}_N) \\ \nabla^2\phi_2(\mathbf{r}_1) & \nabla^2\phi_2(\mathbf{r}_2) & \dots & \nabla^2\phi_2(\mathbf{r}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \nabla^2\phi_{N/2}(\mathbf{r}_1) & \nabla^2\phi_{N/2}(\mathbf{r}_2) & \dots & \nabla^2\phi_{N/2}(\mathbf{r}_N) \end{pmatrix}. \tag{7.1.4}$$

7.1.3 Jastrow Matrices

The Jastrow factor we're using for our simulations is on the form

$$\prod_{i<j}^{N} \exp\left(\frac{ar_{ij}}{1+\beta r_{ij}}\right) = \exp\left(\sum_{i<j}^{N} \frac{ar_{ij}}{1+\beta r_{ij}}\right), \tag{7.1.5}$$

where r_{ij} are the relative distances we have already stored in a matrix. From this we see that the distances matrix already optimizes the calculation of the Jastrow factor. However, we can further optimize this calculation by storing the full fraction

$$\frac{ar_{ij}}{1+\beta r_{ij}} \tag{7.1.6}$$

in it's own matrix. Element ij of this new matrix would then be the value we get from the fraction for r_{ij}, and so for every relative distance we have a corresponding fraction value. This saves us from having to compute all of those fraction every time the Jastrow factor needs to be computed. Similarly to the relative distances matrix, only the relevant elements need to be recalculated whenever a particle is moved.

The gradient of the Jastrow factor with respect to a given particle is

$$\frac{\nabla_k J}{J} = \sum_{j \neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{a}{(1+\beta r_{kj})^2}, \tag{7.1.7}$$

and we define

$$d\mathbf{J}_{kj} \equiv \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{a}{(1+\beta r_{kj})^2}, \tag{7.1.8}$$

which is antisymmetric, i.e.

$$d\mathbf{J}_{kj} = -d\mathbf{J}_{jk}. \tag{7.1.9}$$

The fractions $d\mathbf{J}_{kj}$ are used both for calculating the Jastrow gradients and for the Jastrow Laplacian. Storing $d\mathbf{J}_{kj}$ for all pairs of particles can therefore save us from recalculating it over and over. The matrix used for storing these fractions is fairly similar to the relative distances matrix, but with an antisymmetry instead of a symmetry. It's on the form

$$d\mathbf{J} \equiv \begin{pmatrix} 0 & d\mathbf{J}_{12} & d\mathbf{J}_{13} & \dots & d\mathbf{J}_{1N} \\ -d\mathbf{J}_{12} & 0 & d\mathbf{J}_{23} & \dots & d\mathbf{J}_{2N} \\ -d\mathbf{J}_{13} & -d\mathbf{J}_{23} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & d\mathbf{J}_{nN} \\ -d\mathbf{J}_{1N} & -d\mathbf{J}_{2N} & \dots & -d\mathbf{J}_{nN} & 0 \end{pmatrix}, \tag{7.1.10}$$

with

$$d\mathbf{J} = -d\mathbf{J}^T. \tag{7.1.11}$$

When a particle is moved we only need to update the relevant elements, which amount to one row and one column. We also store the gradients with respect to the different particles together in a matrix, since these gradients are used multiple times in the simulation. This matrix has dimensionality $(N \times d)$, where d is the number of dimensions. Updating the gradient with respect to a given particle can now be simplified to

$$\frac{\nabla_k J^{\text{new}}}{J^{\text{new}}} = \sum_{j \neq k} d\mathbf{J}^{\text{new}}_{kj}, \tag{7.1.12}$$

but we can also further optimize the updating process by utilizing the "old" versions of the matrices, i.e. the matrices as they were before the most recent particle move:

$$\frac{\nabla_k J^{\text{old}}}{J^{\text{old}}} = \sum_{j \neq k} d\mathbf{J}^{\text{old}}_{kj}. \tag{7.1.13}$$

We know that moving a particle p only changes one row and one column in $d\mathbf{J}$. This means that for a particle $k \neq p$ only one term, $d\mathbf{J}_{kp}$, in the gradient sum has changed

due to the move

$$\begin{aligned}\frac{\nabla_{k\neq p}J^{\rm new}}{J^{\rm new}} &= \sum_{j\neq k} d\mathbf{J}^{\rm new}_{kj} \\ &= \sum_{j\neq k,p} d\mathbf{J}^{\rm old}_{kj} + d\mathbf{J}^{\rm new}_{kp} \\ &= \sum_{j\neq k,p} d\mathbf{J}^{\rm old}_{kj} + d\mathbf{J}^{\rm old}_{kp} - d\mathbf{J}^{\rm old}_{kp} + d\mathbf{J}^{\rm new}_{kp} \\ &= \frac{\nabla_{k\neq p}J^{\rm old}}{J^{\rm old}} - d\mathbf{J}^{\rm old}_{kp} + d\mathbf{J}^{\rm new}_{kp},\end{aligned} \tag{7.1.14}$$

where we have used

$$\frac{\nabla_{k\neq p}J^{\rm old}}{J^{\rm old}} = \sum_{j\neq k,p} d\mathbf{J}^{\rm old}_{kj} + d\mathbf{J}^{\rm old}_{kp}. \tag{7.1.15}$$

We see then from Eq. (7.1.14) that the gradient with respect to a particle $k \neq p$ can be updated by simply subtracting $d\mathbf{J}^{\rm old}_{kj}$ and adding $d\mathbf{J}^{\rm new}_{kj}$. For $k = p$ we still need to calculate the full sum in Eq. (7.1.12). The function for updating the Jastrow matrices is listed below.

```
1  void ManyElectronsCoefficients::updateJastrow(int currentParticle) {
2
3      int p = currentParticle;
4      std::vector<double> r_p = m_system->getParticles()[p]->getPosition();
5      double beta = m_parameters[1];
6      m_dJastrowMatOld = m_dJastrowMat;
7
8      for (int j=0; j<p; j++) {
9          std::vector<double> r_j = m_system->getParticles()[j]->getPosition();
10         double r_pj = m_distances(p,j);
11         double denom = 1 + beta*r_pj;
12
13         m_JastrowMat(p,j) = m_a(p,j)*r_pj / denom;
14         m_JastrowMat(j,p) = m_JastrowMat(p,j);
15
16         for (int d = 0; d < m_numberOfDimensions; d++) {
17             m_dJastrowMat(p,j,d) = (r_p[d]-r_j[d])/r_pj * m_a(p, j)/(denom*denom)
                   ;
18             m_dJastrowMat(j,p,d) = -m_dJastrowMat(p,j,d);
19         }
20     }
21     for (int j=p+1; j<m_numberOfParticles; j++) {
22         std::vector<double> r_j = m_system->getParticles()[j]->getPosition();
23         double r_pj = m_distances(p,j);
24         double denom = 1 + beta*r_pj;
25
26         m_JastrowMat(p,j) = m_a(p,j)*r_pj / denom;
27         m_JastrowMat(j,p) = m_JastrowMat(p,j);
28
29         for (int d = 0; d < m_numberOfDimensions; d++) {
30             m_dJastrowMat(p,j,d) = (r_p[d]-r_j[d])/r_pj * m_a(p, j)/(denom*denom)
                   ;
31             m_dJastrowMat(j,p,d) = -m_dJastrowMat(p,j,d);
32         }
33     }
34
35     m_JastrowGradOld = m_JastrowGrad;
36
37     for (int d = 0; d < m_numberOfDimensions; d++) {
```

```
38          m_JastrowGrad(p, d) = 0;
39
40          for (int j=0; j<p; j++) {
41              m_JastrowGrad(p, d) += m_dJastrowMat(p,j,d);
42          }
43          for (int j=p+1; j<m_numberOfParticles; j++) {
44              m_JastrowGrad(p, d) += m_dJastrowMat(p,j,d);
45          }
46          for (int i=0; i<p; i++) {
47              m_JastrowGrad(i, d) = m_JastrowGradOld(i,d) - m_dJastrowMatOld(i,p,d)
                    + m_dJastrowMat(i,p,d);
48          }
49          for (int i=p+1; i<m_numberOfParticles; i++) {
50              m_JastrowGrad(i, d) = m_JastrowGradOld(i,d) - m_dJastrowMatOld(i,p,d)
                    + m_dJastrowMat(i,p,d);
51          }
52      }
53  }
```

Listing 7.2: Function for updating the Jastrow related matrices whenever a particle is moved. p is the moved particle so only the matrix elements where one of the indices is p are changed in "m_JastrowMat" and "m_dJastrowMat" when particle p moves. In "m_JastrowGrad" all elements need to be changed, however for $i \neq p$ the updating is simple and efficient. Since no particle has any distance to itself, we split the loop in two so that we can exclude the $j = p$ case without using an if-test. "m_JastrowMat" is symmetric and "m_dJastrowMat" is antisymmetric so for these we can halve the number of calculations.

7.2 Optimizing Hermite Polynomial Calculation

As discussed in Section 6.1.4, there are several ways we can calculate the Hermite polynomials we need for the single particle wave functions. In this section we will look at how virtual functions and polymorphism can be used to optimize these calculations. We create a super class "HermitePolynomials", which has sub classes for the Hermite polynomials, their derivatives and their double derivatives. So if we implement the first 20 polynomials, we get a total of 60 sub classes. Each of these sub classes have a virtual function called "eval" which takes a one-dimensional position (e.g. x) as argument and calculates the value corresponding to the sub class. For example the "eval" function of sub class "HermitePolynomial_0" will return the value of the first Hermite polynomial for the given argument x, while the "eval" function of the sub class "dell_HermitePolynomial_0" will instead return the derivative of the first polynomial. In the VMC simulation the quantum numbers nx, ny and nz decide which polynomial we want the value, derivative, or double derivative of. By using the aforementioned sub classes, we can represent the Hermite polynomials as "HermitePolynomials" objects, which each holds one "eval" function. These objects can then be loaded into an array "m_hermitePolynomials" in such a way that the first element of the array is the "HermitePolynomials" object representing the first polynomial, and so on for the rest of the elements. The way to calculate the Hermite polynomial for an arbitrary quantum number nx is then simply

```
1  double HP = m\_hermitePolynomials[nx]->eval(x);
```

Thus we can calculate a Hermite polynomial for a given nx without any consideration for any other Hermite polynomial. If we used if-tests instead we would have to test nx against all values from 0 to nx, which could be a lot of if-tests depending on how many Hermite polynomial we need for the full simulation. If we used the recursive method we would have to go through all Hermite polynomials up to the nx'th one. With our method we can also create similar arrays for the derivatives and the double derivatives. The downside to using this method is that it involves implementing a lot of sub classes. In our code we've included 50 Hermite polynomials, which means 150 total sub classes. It then greatly helps to use automatic code generation through e.g. SymPy. We can then generate the sub classes automatically and copy them into the VMC code. The use of a find/replace function in a text editor quickly solves any general issue from the code generation.

One thing to note is that this method of calculating Hermite polynomials is not necessarily faster than the recursive method. Using the Hermite polynomial expressions explicitly, which the polymorphism method does, involves a lot of power expressions. The recursive method becomes slower more quickly as the number of Hermite polynomials increases than the polymorphism method does, however even for quite a large number of Hermite polynomials the recursive method is faster if the power expressions are not calculated optimally. With sub optimal calculation of power expressions the polymorphism method didn't break even with the recursive method until around 50 Hermite polynomials were used in a timing test with two particles. An example of how power expressions can be optimized is listed below.

```
1 x8 = x*x*x*x*x*x*x*x;
2
3 x2 = x*x;
4 x4 = x2*x2;
5 x8 = x4*x4;
```

Listing 7.3: Two methods for calculating x^8. The first method uses 7 FLOPS, while the second method uses only 3 FLOPS.

The listing shows two methods of calculating x^8, with the second method using less than half as many FLOPS as the first method. In our VMC simulation where we need to calculate a lot of Hermite polynomials with large power expressions, this optimization can save a lot of FLOPS. Our automatically generated code for the Hermite polynomials uses the "pow" function for calculating power expression. This function normally does not optimize the calculation of power expressions, because the optimizations in some cases can result in round-off errors. We use the compiler flag "ffast-math" to apply optimizations to mathematical functions such as "pow". Mathematically the functions are the same after the optimizations, but they are not exactly the same in floating-point arithmetic. Therefore, using the flag "can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions" [16]. Since using the flag can produce errors, we need to make sure that our results are consistent with and without using the flag. With the optimizations from the flag the polymorphism method becomes a lot more efficient than the recursive method even for few Hermite polynomials, and the benefit becomes greater and greater as the number of Hermite polynomials used increases. This is perfect for our simulation, since the number of Hermite polynomials needed increases as the number of basis functions we use increases. As mentioned earlier, increasing the number of basis functions improves

the results, so this optimization allows us to get good results for significantly lower computational cost.

7.3 Parallelization

The final optimization to discuss is the parallelization of the VMC simulation. The entire simulation is parallelized with Open MPI by dividing the number of Monte Carlo cycles among multiple nodes (processors). Each node gets its own seed for generating the random initial state of the system, and then runs its own VMC simulation with a fraction of the total amount of Monte Carlo cycles. After the nodes have finished their individual simulations, the results are brought together and averaged. When running a VMC simulation in parallel the individual simulations of nodes are completely independent of each other. This makes parallelization of VMC simulations very efficient, because the nodes only need to communicate at the very start and the very end. Thus we avoid race conditions[1] in the middle of the simulation, where one node has to wait for other nodes to catch up, before it can continue its own simulation. The only mid-simulation communication between nodes is when we're varying the variational parameters in order to find optimal parameters. However, this is typically done with small simulations where optimization is less important. After the optimal parameters have been found, a simulation with more MC cycles is ran using those parameters, so then there's no mid-simulation communication.

With parallelization we can run the simulation on several nodes on a single computer. Modern computers typically have 4 or 8 processors, which can give a good amount of speed-up. We can also run the simulation on a super computer cluster in order to utilize even more processors. Since we don't need any mid-simulation communication between the nodes, we can expect a scaling which is approximately linear, i.e. if we double the amount of nodes we halve the computation time. When doing VMC calculations in parallel we have to be careful not to use too many processors for a small amount of MC cycles. If we have too many processors the amount of MC cycles for each processor might be too low to give good results. In addition the amount of MC cycles could end up being a non-integer number, which wouldn't make sense.

[1] A race condition or race hazard is the behavior of an electronic, software, or other system where the output is dependent on the sequence or timing of other uncontrollable events. [19]

Part III

Results

Chapter 8

Results

Natural units ($\hbar = c = e = m_e = 1$) were used to obtain all the results listed in this chapter.

8.1 Optimization

Here we look at the effect that various optimizations had on the computation time of our program. The optimization to the simulation itself gave a total speed-up factor of 31.246 for the tested system, while running the program in parallel on 4 processors gave an additional speed-up factor of around 3.6.

8.1.1 Storing Reused Data and Optmizing Hermite Polynomials

For the following optimization results we used a system of two particles in a two-dimensional double harmonic oscillator well as reference system. For the results in Table 8.1 we used 465 basis functions. From the table we see that including the "-ffast-math" compiler flag gives a decent speed up, but the storing of the Slater related matrices and the efficient computation of Hermite polynomials are the most important optimizations in this case. We also see that the "-ffast-math" flag is much more important when we include the Hermite optimization described in Section 7.2. In this case the storing of the relative distances and the Jastrow related matrices, provide seemingly no speed up. This makes sense since these optimizations scale well with the number of particles used, and therefore become important for large numbers of particles. In our case we only have two particles so the effect of these optimizations is negligible. The optimization of Hermite polynomial calculation obviously scales with the number of Hermite polynomials we need. The number of Hermite polynomials we need scales with the number of basis functions we use, and in our case we use 465 basis functions, which means we need the first 30 Hermite polynomials, each of which has to be calculated many times for various positions of particles. The efficiency gain from storing the Slater related matrices also scales with the number of basis functions. This is because these matrices store the single particle wave functions and their derivatives. These single particle wave functions are calculated by a loop over the number of basis functions, so if we avoid recalculating them unnecessarily, we reduce the number of basis function loops, which of course is more important if we use a lot of basis functions.

Optmizations	Avg. Time [s]	Speed Up	Total Speed Up
B	122.173	1.000	1.000
B F	104.244	1.172	1.172
B F D	104.296	1.000	1.171
B F D S	16.8304	6.197	7.259
B F D S J	16.8524	0.999	7.250
B F D S J H	3.91007	4.310	31.246
B D S J H	30.7192	0.127	4.017

Table 8.1: Optimization results for two particles in a two-dimensional double harmonic oscillator well. The number of basis functions used was 465. B stands for base i.e. the program before doing optimizations. F is for using the "-ffast-math" compiler flag, D is for storing the distance matrix, S is for storing the Slater related matrices, J is for storing the Jastrow related matrices, and H is for optimal computation of Hermite polynomials. The "Avg. Time" column is the average time over five runs, the "Speed Up" column lists the speed up factor relative to the row above, and the "Total Speed Up" column lists the speed up factor relative to the first row. Optimizations D and J have seemingly no effect on the run-time, because they scale with the number of particles and we only use two particles in this case. The gain from using the F optimization is much greater when we also use the H optimization. This is apparent by comparing the first two rows with the last two rows. The S and H optimizations provide the most speed up for this system.

In order to see that storing the distances and the Jastrow related matrices do actual contribute to the optimization, we will look at a different system. We now look at a system of 24 particles in a two-dimensional double harmonic oscillator well. However, we now approximate the single particle wave functions with super positions of two harmonic oscillator functions, similarly to what was done in Chapter 5.1.3 and 5.2.3 of Ref. [22]. This means that the number of Hermite calculations needed is significantly reduced, and the effect of storing the distances and the Jastrow related matrices should stand out due to the increased number of particles. Note that this way of calculating the single particle wave functions is not the focus of this thesis, but it has been implemented in order to compare the results of the two methods. In this case we use this method in order to simulate 24 particles with reasonably short run times, as this method is generally faster especially for large number of particles. The benefit of storing the distances and the Jastrow related matrices should be similar for both methods, since the difference between them is how the single particle wave functions are calculated, and these single particle wave functions do not depend on the Jastrow factor or the distances between particles. The optimization results for this system are listed in Table 8.2. As expected the importance of storing the distances and the Jastrow related matrices has gone up. The storing of distances provide a decent speed up, while the storing of the Jastrow related matrices is in this case the most important optimization. The storing of the Slater related matrices is not quite as important as it was in Table 8.1, but it still provides a pretty good speed up. The effect of using the "-ffast-math" flag is much smaller in this case, and the effect of calculating Hermite polynomials is negligible. This method needs a lot less Hermite polynomial calculations, since there's no loop over basis

functions when calculating the single particle wave functions. However, the number of Hermite polynomial calculations does scale with the number of particles, and the recursive method becomes slow for large numbers of particles, so for systems with even more particles the Hermite optimization should become significant for this method as well.

Optmizations	Avg. Time [s]	Speed Up	Total Speed Up
B	103.576	1.000	1.000
B F	101.766	1.018	1.018
B F D	92.2338	1.103	1.123
B F D S	54.7138	1.686	1.893
B F D S J	10.7135	5.107	9.668
B F D S J H	10.6977	1.001	9.682
B D S J H	10.8811	0.983	9.519

Table 8.2: Optimization results for 24 particles in a two-dimensional double harmonic oscillator well. B stands for base i.e. the program before doing optimizations. F is for using the "-ffast-math" compiler flag, D is for storing the distance matrix, S is for storing the Slater related matrices, J is for storing the Jastrow related matrices, and H is for optimal computation of Hermite polynomials. The "Avg. Time" column is the average time over five runs, the "Speed Up" column lists the speed up factor relative to the row above, and the "Total Speed Up" column lists the speed up factor relative to the first row. The single particle wave functions were approximated by a super position of two harmonic oscillator functions instead of the method regularly used in this thesis. The method used here involves less Hermite polynomial calculations, and as a result, optimizing the calculation of Hermite polynomials is less important. The benefit of optimizations D and J should be similar between the methods, and now that the number of particles is increased the effect of these optimizations is also increased compared to Table 8.1. Optimazation D now provides a decent speed up, while optimization J provides the majority of the total speed up. Optimization S also provides a good speed up, but it is not as significant as it was in Table 8.1.

8.1.2 Parallelization

As discussed in Section 7.3, the VMC simulation can be parallelized without mid-simulation communication between the processors. Therefore we can expect near linear scaling, so doubling the number of processors should about halve the run time. In Table 8.3 we have listed average run times for a system of two particles in a two-dimensional double harmonic oscillator well, when using 1275 basis functions and 1, 2 and 4 processors. From the table we see that the speed up is close to what we expect, but not quite.

A possible reason for why the results in Table 8.3 were somewhat different than expected could be that the run times were so short that the overhead[1] was responsible for a significant fraction of the run time. If this is the case, then doing the same test for

[1] "In computer science, overhead is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task." [18]

Processors	Avg. Time [s]	Speed Up	Total Speed Up
1	36.9145	1.0000	1.0000
2	19.1713	1.9255	1.9255
4	10.3109	1.8593	3.5801

Table 8.3: Parallelization results for two particles in a two-dimensional double harmonic oscillator well. The number of basis functions used was 1275. The "Avg. Time" column is the average time over five runs, the "Speed Up" column lists the speed up factor relative to the row above, and the "Total Speed Up" column lists the speed up factor relative to the first row. We see that doubling the number of processors gives a speed up factor close to 2, but it's still somewhat off, especially when going from 2 to 4 processors.

a system that requires more CPU time, should provide results closer to the near linear scaling we expect. We increase the number of particles to 4, and the run times for this system are listed in Table 8.4. We end up with run times which are about three times as long as for the previous system, and as expected the speed up from parallelization is indeed somewhat greater for this system than for the two-particle system.

Processors	Avg. Time [s]	Speed Up	Total Speed Up
1	107.468	1.0000	1.0000
2	55.4418	1.9384	1.9384
4	29.4316	1.8838	3.6514

Table 8.4: Parallelization results for 4 particles in a two-dimensional double harmonic oscillator well. The number of basis functions used was 1275. The "Avg. Time" column is the average time over five runs, the "Speed Up" column lists the speed up factor relative to the row above, and the "Total Speed Up" column lists the speed up factor relative to the first row. We see that doubling the number of processors gives a speed up factor close to 2, and the speed up factor are closer to 2 in this case than they were in Table 8.3. This is likely due to the generally longer run times for this system compared to the two-particle system. Since the run time is longer, the overhead is a smaller percentage of the total run time, which means it has less of an effect on the speed up factors.

8.2 Single Harmonic Oscillator Well

In this section we look at the energy and one-body density of systems of particles in a single harmonic oscillator potential well, both in two and three dimensions.

8.2.1 Ground State Energies

Here we look at the ground state energies for systems with various ω and number of particles. The ground state energies are calculated with the single particle wave functions being approximated by expansion in a single harmonic oscillator basis, and also by using harmonic oscillator single particle wave functions directly. Since the wave functions we're trying to approximate and the basis functions we use are of the same type (single harmonic oscillator) we should expect the two methods to yield the exact same results

using a small amount of basis functions, just as we saw for the non-interacting case in Section 6.1.7.2. It turns out that we don't get the exact same results for the interacting case, however the same results are achieved if the variational parameter α is kept constant at $\alpha = 1$, while the other variational parameter β is varied as usual. This indicates that there is an α dependence in the coefficients used in the basis function expansion, which our method fails to include. This α dependence could be included by doing a Hartree-Fock calculation on the coefficients, and this would be the next step in improving the method, but has not been done in this thesis.

8.2.1.1 Two Dimensions

The ground state energies for the two-dimensional case are listed in Table 8.5. E_{coeff} are the results when the single particle wave functions are approximated by expansion in a single harmonic oscillator basis, while E_{reg} are the results when using harmonic oscillator single particle wave functions directly. We see from the table that E_{coeff} and E_{reg} are similar, especially for small numbers of particles N, but they are not exactly equal. Some further calculations not included here, revealed that the exact same energies where achieved if α was held constant at $\alpha = 1$, and β was treated as the only variational parameter. This indicates that there is an α dependence in the coefficients used in the basis function expansion, which is not included when creating the coefficients. The coefficients could be modified to include this α dependence by doing a Hartree-Fock calculation, and if this is done E_{coeff} and E_{reg} should be exactly equal.

The results E_{coeff} and E_{reg} are also bench marked against results from various other methods such as Coupled Cluster and Full Configuration Interaction, as well as other VMC results. From Ref. [2] we also have an analytical result for the ground state energy, $E = 3$, for the two-body case with $\omega = 1$. The results are reasonably consistent with the bench marks for all calculated systems. Our E_{reg} results and the $E_{\text{ref}}^{(\text{a})}$ bench marks use the same VMC method, and as such should give fairly similar results. However, the optimization of parameters is done using different methods, which can result in slightly different parameter values being used, which in turn can result in differences larger than the statistical error. If the parameters used were exactly the same there could still be differences due to different amount of Monte Carlo cycles used, but in that case the statistical error would cover the difference. Table 8.6 has additional bench marks from using Diffusion Monte Carlo, which is an improved version of VMC. Expanding our program to include DMC calculation as well as using Hartree-Fock to improve the coefficients is a possibility for further work.

8.2.1.2 Three Dimensions

For the three-dimensional case the ground state energy results are listed in Table 8.7. Here we only have one source of bench marks, but with both VMC and DMC bench marks. From the table we mainly see the same things as for the two-dimensional case. The results are reasonably consistent with the bench marks and with each other. By looking at the two particle case we can see that in general the energy is larger in three dimensions than in two dimensions for the same system, but the difference is smaller for smaller ω. For E_{reg} and $\omega = 0.01$ the ratio is $0.0790/0.0745 \approx 1.0604$, while for $\omega = 1$ it is $3.7242/2.9984 \approx 1.2421$.

N	ω	E_{coeff}	E_{reg}	$E_{\text{ref}}^{(\text{a})}$	$E_{\text{ref}}^{(\text{b})}$	$E_{\text{ref}}^{(\text{c})}$	$E_{\text{ref}}^{(\text{d})}$
2	0.01	0.0754(3)	0.0745(2)	0.07406(5)	-	0.0738	0.07383505
						{23}	{19}
	0.10	0.4460(3)	0.4428(2)	0.44130(5)	-	0.4408	0.44079191
						{23}	{19}
	0.28	1.0283(3)	1.0245(3)	1.02215(5)	-	1.0217	1.0216441
						{23}	{19}
	0.50	1.6658(3)	1.6614(4)	1.66021(5)	-	1.6599	1.6597723
						{23}	{19}
	1.00	3.0034(4)	2.9984(4)	3.00030(5)	-	3.0002	3.0000001
						{23}	{19}
6	0.10	3.602(3)	3.565(2)	3.5690(3)	3.49991	3.5805	3.551776
					{18}	{22}	{9}
	0.28	7.658(3)	7.609(2)	7.6216(4)	7.56972	7.6254	7.599579
					{18}	{22}	{6}
	0.50	11.853(3)	11.781(2)	11.8103(4)	11.76228	11.8055	11.785915
					{18}	{22}	{6}
	1.00	20.243(3)	20.143(3)	20.1902(4)	20.14393	20.1734	20.160472
					{18}	{22}	{8}
12	0.10	12.568(7)	12.282(3)	12.3162(5)	12.22533	12.3497	12.850344
					{17}	{21}	{3}
	0.28	25.866(6)	25.642(4)	25.7015(6)	25.61084	25.7095	26.482570
					{17}	{21}	{2}
	0.50	39.406(5)	39.152(4)	39.2343(6)	39.13899	39.2194	39.922693
					{17}	{21}	{2}
	1.00	65.952(5)	65.666(4)	65.7905(7)	65.68304	65.7399	66.076116
					{17}	{21}	{3}

Table 8.5: The table lists ground state energy results for various single harmonic oscillator systems in two dimensions, and corresponding bench marks. N is the number of particles and ω is the harmonic oscillator frequency. E_{coeff} are the energies obtained when the single particle wave functions are approximated by expansion in a single harmonic oscillator basis. For this the number of basis functions used was $(N/2)(N/2+1)/2$. E_{reg} are the energies obtained when using harmonic oscillator single particle wave functions directly. The bench marks are from the following: (a) J. Høgberget [22] (VMC), (b) S. Reimann [23] (Similarity Renormalization Group theory), (c): C. Hirth [24] (Coupled Cluster Singles and Doubles), (d): V. K. B. Olsen [25] (Full Configuration Interaction). The numbers in parenthesis are the statistical errors found using blocking. In the curly brackets are the numbers of shells used above the last filled shell to construct the basis for the corresponding methods [22].

8.2.2 One-Body Densities

The one-body densities show us how the particles are likely to be distributed in the system. The main thing we look at is how far away from the center of the well a particle

N	ω	$E_{\text{ref}}^{(\text{a})}$	$E_{\text{ref}}^{(\text{b})}$
2	0.01	-	0.073839(2)
	0.10	-	0.44079(1)
	0.28	-	1.02164(1)
	0.50	1.65975(2)	1.65977(1)
	1.00	3.00000(3)	3.00000(3)
6	0.10	-	3.55385(5)
	0.28	7.6001(1)	7.60019(6)
	0.50	11.7888(2)	11.78484(6)
	1.00	20.1597(2)	20.15932(8)
12	0.10	-	12.26984(8)
	0.28	25.6356(1)	25.63577(9)
	0.50	39.159(1)	39.1596(1)
	1.00	65.700(1)	65.7001(1)

Table 8.6: The table lists additional bench marks for Table 8.5. N is the number of particles and ω is the harmonic oscillator frequency. The bench marks are from the following: (a) M. L. Pedersen et al. [3] (DMC), (b) J. Høgberget [22] (DMC).

N	ω	E_{coeff}	E_{reg}	$E_{\text{ref}}^{(\text{a})}$	$E_{\text{ref}}^{(\text{b})}$
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 8.7: The table lists ground state energy results for various single harmonic oscillator systems in three dimensions, and corresponding bench marks. N is the number of particles and ω is the harmonic oscillator frequency. E_{coeff} are the energies obtained when the single particle wave functions are approximated by expansion in a single harmonic oscillator basis. For this the number of basis functions used was $(N/2)(N/2+1)(N/2+2)/6$. E_{reg} are the energies obtained when using harmonic oscillator single particle wave functions directly. The bench marks are from the following: (a) J. Høgberget [22] (VMC), (b) J. Høgberget [22] (DMC). The numbers in parenthesis are the statistical errors found using blocking.

is most likely to be. However, it is also of interest to look a what (in the two-dimensional case) x and y positions are most likely independently of each other. In the first case we take the actual sampled positions of the particles and find the distance from the center of the well. In the second case we look at only the x positions by themselves, and only

the y positions by themselves. Since the single harmonic oscillator well is symmetric, the x distribution and the y distribution should ideally be the same.

We know that the harmonic oscillator external potential has its minimum in the center, i.e. at $r = 0$. However, we also know that the particles have a Coulomb repulsion pushing the away from each other. So instead of the particles clumping together in the center of the well, they would be pushed away from the center by each other. At the same time they are confined by the external potential which pushes them back towards the center. Therefore the particles most likely position should be where the force from the other particles and the force from the external potential cancels each other out. We expect r to be greater than zero, but also limited by the external potential. As the number of particles increases the force between the particles will increase, which will push the particles further out until the force from the external potential is large enough to counteract the increased Coulomb repulsion. We can see this from the right-hand side of Figure 8.2. From the figure we see that for all numbers of particles N the distribution has a top at $r > 0$, and as N increases this top is pushed further and further to the right, to greater and greater r values.

The $N = 12$ case also shows a small change in the shape of the distribution indicating that some particles are closer to the center while others are further out, which corresponds to the particles being on different energy levels. This can be seen more clearly when looking at the x and y positions separately in Figure 8.1, or at the x and y positions as a mesh-grid as seen on the left-hand side of Figure 8.2. The individual x and y distributions are almost identical as expected for a symmetric harmonic oscillator well.

8.3 Double Harmonic Oscillator Well

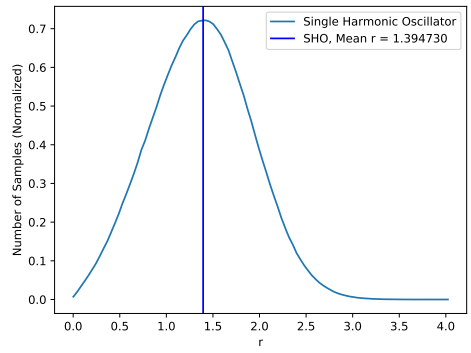
In this section we look systems of particles in a double harmonic oscillator potential well with a distance of $L_x = 1$ between the barrier between the wells and the center of each well. We look at energies and one-body densities in both two and three dimensions.

8.3.1 Ground State Energies

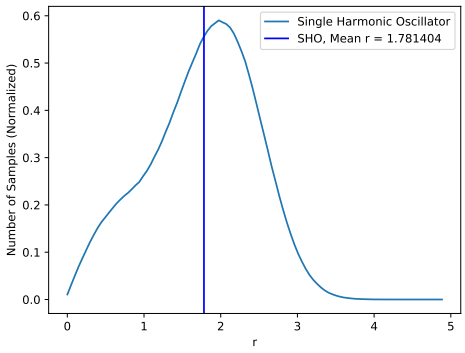
Unlike for the single harmonic oscillator well, for the double harmonic oscillator well we have no reason to expect the results we get when the single particle wave functions are approximated by expansion in a single harmonic oscillator basis (E_{coeff}), to be exactly equal to the result from using harmonic oscillator single particle wave functions directly (E_{reg}). This is because the basis functions are still single harmonic oscillator functions, but the single particle wave functions we're trying to approximate are for a double harmonic oscillator well, so they're not of the same type as the basis functions. Because of this we should also expect to need more basis functions to get good results. Of course the α dependence discussed in Section 8.2.1 is still missing from the coefficients. We compare E_{coeff} with E_{reg}, and see that they are reasonably consistent with each other provided we use enough basis functions when calculating E_{coeff}.

8.3.1.1 Two Dimensions

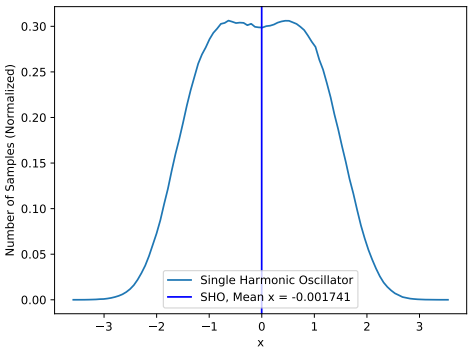
The ground state energies of systems with various numbers of particles N and harmonic oscillator frequencies ω are listed in Table 8.8. In most cases we need more basis functions



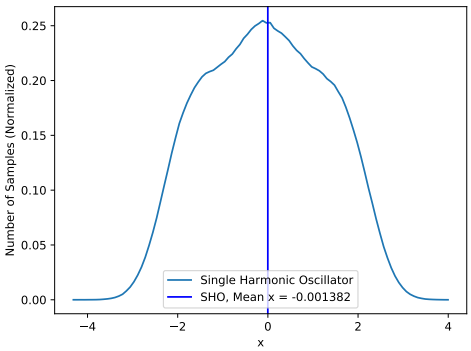
(a) $N = 6$



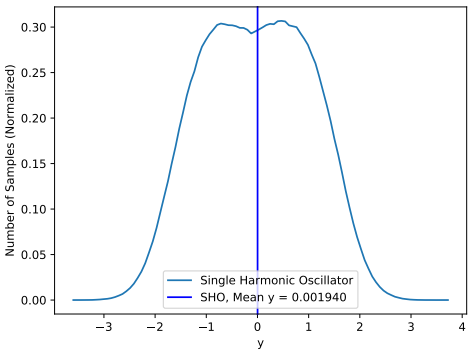
(b) $N = 12$



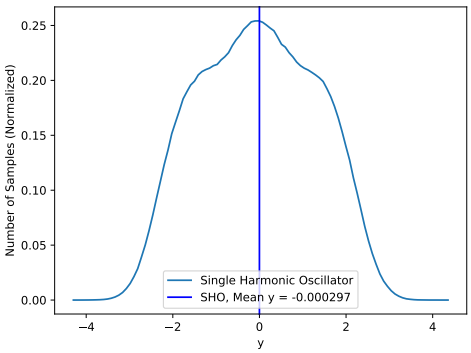
(c) $N = 6$



(d) $N = 12$

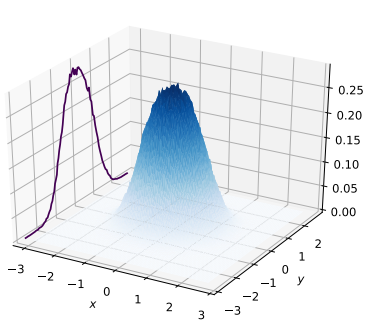


(e) $N = 6$

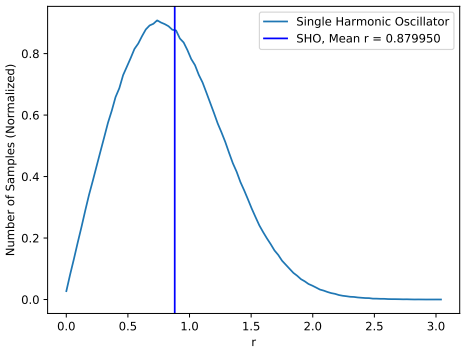


(f) $N = 12$

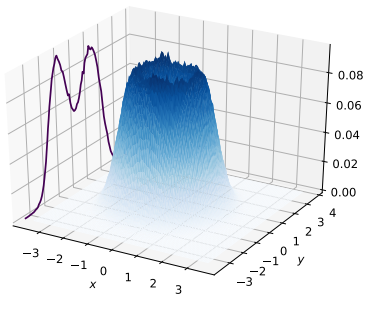
Figure 8.1: Figure 8.1a and 8.1b show the one-body densities of a two-dimensional single harmonic oscillator with N particles and $\omega = 1$. The other figures show the corresponding distributions of x and y positions individually. The x and y distributions are almost identical due to the harmonic oscillator potential being symmetric.



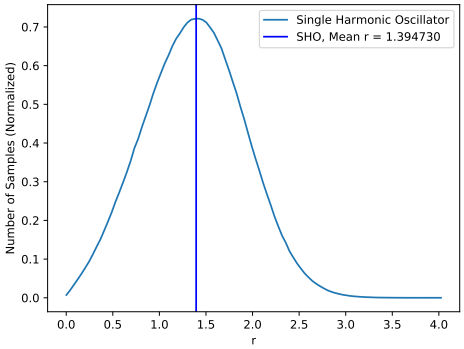
(a) $N = 2$



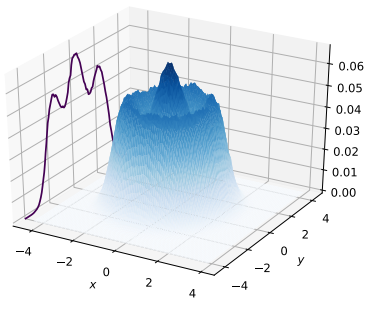
(b) $N = 2$



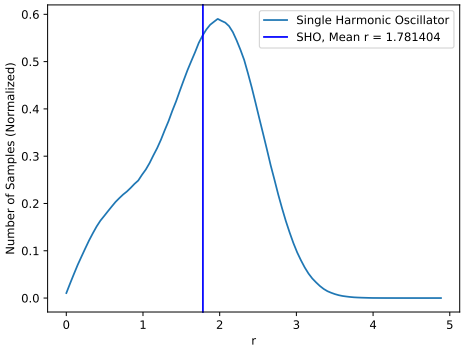
(c) $N = 6$



(d) $N = 6$



(e) $N = 12$



(f) $N = 12$

Figure 8.2: The right-hand side shows the one-body densities of a two-dimensional single harmonic oscillator with N particles and $\omega = 1$. The left-hand side shows the distribution of positions for a mesh-grid of the x and y positions. From the left-hand side we can clearly see that the particles are divided into more energy levels as N increases.

to get good results for the double harmonic oscillator well, than we did for the single well, which is to be expected since the basis functions are single well functions. From the table we see that with a reasonable amount of basis functions we can achieve consistency between E_{coeff} and E_{reg}. Unlike for the single well, here we only have one bench mark to compare our results to. In order to compare with the bench mark we have used the same distance between the wells as was used for the bench mark, i.e. $R = 2$ in the x-direction, which corresponds to $L_x = 1$ in our case. From the table we see that our results are fairly consistent with the bench mark, but just as for the single well case, there are some difference likely due to some difference in the variational parameters.

For $N = 2$ and $N = 12$ we can directly compare the results of Table 8.8 with those from Table 8.5. We see that given the same number of particles and the same ω the double well results are always smaller than their single well counterparts. This is expected since the particles are split between two wells in the double well. For example let's look at the $N = 2$, $\omega = 1$ case. If L_x was equal to 0 the wells would be on top of each other, which means we essentially would have a single well potential, and the energy for a interacting system would then be $E = 3$. If on the other hand L_x was approaching infinity, then (assuming one particle in each well) the system could be considered as two independent single wells with one particle and consequently no interaction, and the energy would then be $E = 2$. So for our case with $L_x = 1$ we should expect $2 < E < 3$, and indeed we get $E_{\text{coeff}} = 2.326$. If we imagine the 12 particle case with L_x approaching infinity there would still be interaction in each well, since there would be 6 particles in each. However, for $N = 6$, $\omega = 1$ in Table 8.5, we have that $E_{\text{coeff}} = 20.243$, so for the double well with 12 particles and $L_x \to \infty$ the energy would be $E \approx 2 \times 20.243 = 40.486$. Again the $L_x = 0$ case corresponds to a single well with 12 particles and for that we have the result $E_{\text{coeff}} = 65.952$ from Table 8.5. So for the double well with $N = 12$, $\omega = 1$ we should expect $40.486 < E < 65.952$, and indeed we get $E_{\text{coeff}} = 55.165$.

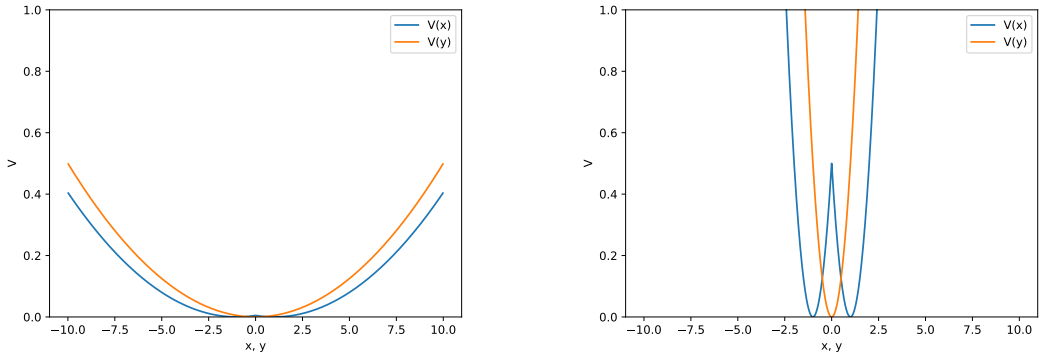
One thing to note about this is that while the double well results are smaller than the single well results in all cases, the difference between the two becomes very small as ω decreases. If we look at the $N = 2$, $\omega = 0.01$ case we see that for the single well we have $E_{\text{coeff}} = 0.0754$ while for the double well we have $E_{\text{coeff}} = 0.0735$. The reason for this is that as ω decreases the barrier between the wells in the double well becomes smaller and smaller, due to the widening of the wells. When this happens the double well starts to look more and more like a single well (though with an extended bottom). This is shown in Figure 8.3.

8.3.1.2 Three Dimensions

The ground state energies for the three-dimensional case are listed in Table 8.9. For the three-dimensional case we don't have any bench marks to compare our results with. However, we can still compare the results with each other, and from the table we see that E_{coeff} and E_{reg} are reasonably consistent with each other. In general we need more basis function to achieve good results for E_{coeff}, than we did in two dimensions. We also see that the similarities between the single well and the double well we observed in the two-dimensional case when ω was small, is also present in the three-dimensional case. As expected the energy is also generally larger for the three-dimensional case than for the two-dimensional case.

N	ω	Basis Functions	E_{coeff}	E_{reg}	E_{ref}
2	0.01	1	0.0735(3)	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)	3.222(1)	-
	0.50	55	4.958(2)	4.831(2)	-
	1.00	55	7.850(2)	7.848(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 8.8: The table lists ground state energy results for various double harmonic oscillator systems in two dimensions, and corresponding bench marks. N is the number of particles and ω is the harmonic oscillator frequency. E_{coeff} are the energies obtained when the single particle wave functions are approximated by expansion in a single harmonic oscillator basis. E_{reg} are the energies obtained when using double harmonic oscillator single particle wave functions directly. The bench mark is from J. Høgberget [22] (DMC). The numbers in parenthesis are the statistical errors found using blocking.



(a) $\omega = 0.1$ (b) $\omega = 1$

Figure 8.3: The shape of a double well potential with centers at $x = \pm 1$ and $y = 0$, for $\omega = 0.1$ and $\omega = 1$ with constant axes so they can be compared. $V(x)$ is the potential in the x-direction (double well) and $V(y)$ in the y-direction (single well). We see that for lower ω, $V(x)$ is more similar to $V(y)$, indicating that the double well potential becomes more similar to a single well potential as ω decreases.

N	ω	Basis Functions	E_{coeff}	E_{reg}
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)
4	0.10	10	1.640(1)	1.642(1)
	0.28	56	3.518(1)	3.483(2)
	0.50	56	5.3858(9)	5.360(2)
	1.00	56	9.153(2)	9.106(2)

Table 8.9: The table lists ground state energy results for various double harmonic oscillator systems in three dimensions. N is the number of particles and ω is the harmonic oscillator frequency. E_{coeff} are the energies obtained when the single particle wave functions are approximated by expansion in a single harmonic oscillator basis. E_{reg} are the energies obtained when using double harmonic oscillator single particle wave functions directly. The numbers in parenthesis are the statistical errors found using blocking.

8.3.2 One-Body Densities

We are interested in looking at how particles distribute in a double harmonic oscillator well. Just as for the single well, the particle will push each other away from the center, but here we don't just have a center for each well, we have a center between them at the potential barrier. We have placed this center at $r = 0$, and each well center is a distance $r = 1$ away. Going forward we'll refer to this center as the system center. If there was no interaction between particles we would expect the particles to gather at the bottom of each well where the external potential is smallest. Since we have a repulsive interaction between the particles we expect the particles to push each other away from the well centers, just as we saw for the single well. Unlike for the single well, the well centers and the system center do not overlap for the double well so the mean distance between particles and the system center should be greater than an equivalent single well system. By comparing Figure 8.4b and 8.1b we see that for $N = 12$ the mean value of r is greater for a double well than for a single well. Just as for the single well, we also see that when going from 4 particles to 12 particles the mean value of r increases.

In addition if we look at all particles in one well as a group, then that group should push the particles in the other well away from the system center and vice versa. This can be seen clearly from Figure 8.5a and 8.5b. In Figure 8.5a we have $N = 2$ and one particle in each well and in Figure 8.5b we have $N = 4$ and two particles in each well. In both cases the distribution ends up with one top for each well, however for the $N = 4$ case the gap between the tops is larger and the distribution at the system center is smaller (as seen from the graph behind the surface plot). This is because when $N = 4$ we have two groups of two particles pushing each other away from the system center, while in the $N = 2$ case each group only has one particle and consequently the force on one group from the other is smaller than in the $N = 4$ case.

When we increase the number of particle to 12 we see from Figure 8.5c and 8.5d that the distribution gets a more interesting shape. For a well with 6 particles in the

single well case we saw from Figure 8.2c that the distribution got a volcano like shape. However, when we have a double well with 6 particles in each well we see from Figure 8.5d that the interaction force from one well on the other causes the volcano like shape to change into three tops surrounding the well center. The three tops are positioned as far away from each other as they can, and between the tops the distribution is somewhat smaller, but still fairly large.

In Figure 8.4 we have plots of the x distribution and y distribution separately for $N = 4$ and $N = 12$. The single well was symmetric so the distributions were identical in that case, but for the double well, we have a double well potential in the x-direction and a single well potential in the y-direction. The y distributions get the same shape as they did for half as many particles in a single well (since each well has $N/2$ particles). The x distributions however, get new shapes which clearly show that we have two identical wells separated by a barrier at $x = 0$.

8.4 Finite Square Well

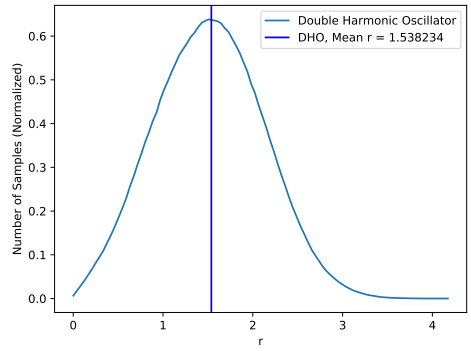
In this section we look systems of particles in a finite square well potential with a distance of 2 between the center and each wall. The value of the potential is 0 inside the well and 1 elsewhere. We look at energies and one-body densities in both two and three dimensions.

8.4.1 Ground State Energies

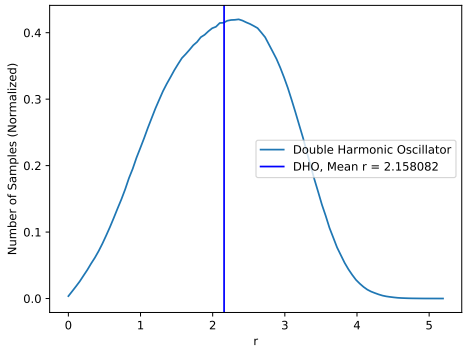
For the finite square well potential we don't have single particle wave functions we can use directly, so we can only approximate the single particle wave functions by expansion in a single harmonic oscillator basis. We also don't have any bench marks, so we can't compare our results to anything. Furthermore, as explained in Section 8.2.1, there is an α dependence missing from our overlap coefficients which means the optimization of α won't be accurate. In the harmonic oscillator systems this wasn't a problem since we had single particle wave functions we could use directly, which we then could use to find optimal parameters and use those parameters for both E_{coeff} with E_{reg}. Due to this not being a possibility for the finite square well, we will here leave α constant at $\alpha = 1$ and just vary β. Of course this won't guarantee that our results come close to the true ground states, but we can still use the results to get an idea of how many basis functions are needed for good results, and whether or not using harmonic oscillator basis functions is practical for a finite square well potential.

8.4.1.1 Two Dimensions

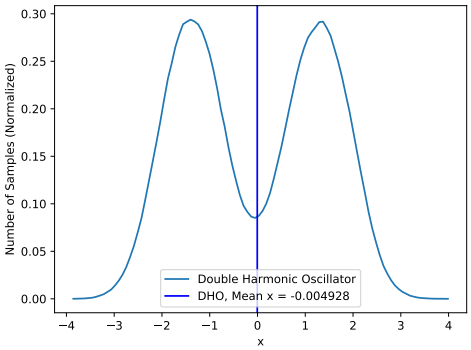
Ground state energy results for a two-dimensional finite square well with N particles are listed in Table 8.10. The finite square well potential does not have a harmonic oscillator frequency ω, however the harmonic oscillator basis functions we use still do. Because of this we should expect similar results regardless of the value of ω as long as we use a sufficient amount of basis functions. Still, what amount of basis functions that is sufficient will depend on ω, since some ω values will yield basis functions which fit better with the finite square well than others. This is illustrated in Figure 8.6. From the figure we see that the most fitting ω value would be somewhere between $\omega = 0.1$ and $\omega = 1$.



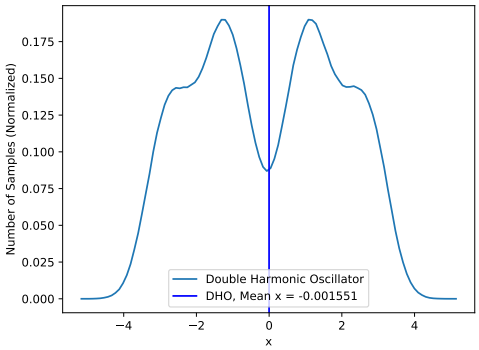
(a) $N = 4$



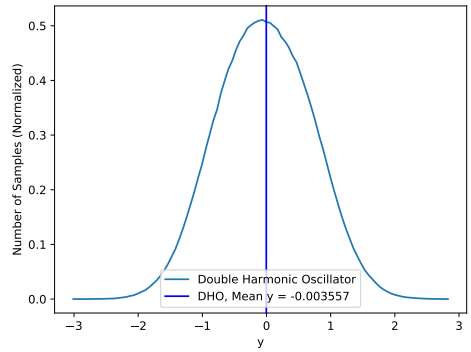
(b) $N = 12$



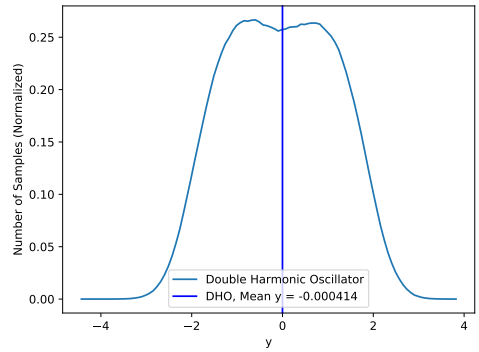
(c) $N = 4$



(d) $N = 12$

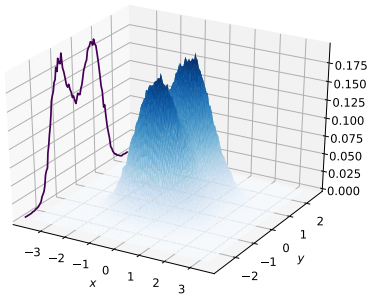


(e) $N = 4$

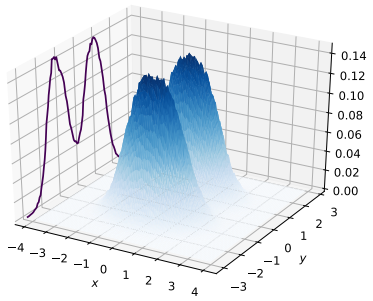


(f) $N = 12$

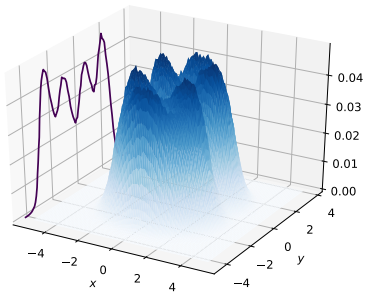
Figure 8.4: Figure 8.4a and 8.4b show the one-body densities of a two-dimensional double harmonic oscillator with with centers at $x = \pm 1$ and $y = 0$, N particles and $\omega = 1$. The other figures show the corresponding distributions of x and y positions individually. Unlike the single well, the double well is not symmetric, so the x and y distributions are not identical.



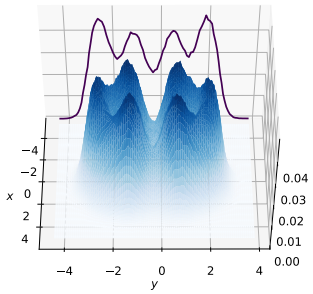
(a) $N = 2$



(b) $N = 4$



(c) $N = 12$



(d) $N = 12$

Figure 8.5: The figures show the distributions of positions for a mesh-grid of the x and y positions for a double harmonic oscillator well with centers at $x = \pm 1$ and $y = 0$. The number of particles is N and the harmonic oscillator frequency is $\omega = 1$. For $N = 2$ and $N = 4$ the distribution is fairly similar to two independent single wells, but for $N = 12$ we get a more exiting shape exclusive to the double well. Figure 8.5c and 8.5d are the same, but from different angles.

From Table 8.10 we see that for $N = 2$, both for $\omega = 0.1$ and $\omega = 1$, the energy converges as the number of basis functions becomes sufficient. The energy values we end up with are not exactly the same for both ω values, but they are reasonably close to each other. The discrepancy could be due to keeping α constant at 1. From the table it also seems like the energy converges faster with increasing number of basis functions for $\omega = 0.1$ than for $\omega = 1$. However, the optimization of β is also dependant on how many basis functions are used, and from Table 8.11 we see that if we use 120 basis functions for optimizing β, but 15 basis functions to do the actual calculation with the optimized β, then we get better results for $\omega = 1$ than for $\omega = 0.1$. So it would seem that with $\omega = 0.1$ we need fewer basis functions to find an optimal β, but given an optimal β we need fewer basis functions to get good results with $\omega = 1$ than with $\omega = 0.1$.

Overall, it seems that for the finite square well we need more harmonic oscillator basis functions than we do for a double harmonic oscillator well, and as N increases the

amount of basis functions needed may be impractical computational cost wise. It may be better to use other types of basis functions, e.g. hydrogen-like wave functions. It would be a good idea though, to do a Hartree-Fock calculation on the overlap coefficients and then properly optimizing α as well as β, to see how many basis functions are needed in that case.

N	ω	Basis Functions	E_{coeff}
2	0.10	15	1.33(2)
	0.10	45	1.271(5)
	0.10	78	1.266(5)
	0.10	120	1.265(5)
	1.00	15	1.525(6)
	1.00	45	1.450(5)
	1.00	78	1.281(2)
	1.00	120	1.281(2)
6	0.10	45	9.38(5)
	1.00	45	10.689(5)

Table 8.10: The table lists ground state energy results for various finite square well systems in two dimensions. N is the number of particles. E_{coeff} are the energies obtained when the single particle wave functions are approximated by expansion in a single harmonic oscillator basis, and ω is the harmonic oscillator frequency used for the harmonic oscillator basis functions. The numbers in parenthesis are the statistical errors found using blocking. For $N = 2$ we see that the energies converge as long as enough basis functions are used, but the somewhat high number of basis functions needed may be impractical with regards to computational cost, especially for higher N.

N	ω	Basis Functions	E_{coeff}
2	0.10	15{120}	1.32(2)
	1.00	15{120}	1.296(2)

Table 8.11: Additional results for Table 8.10. However, here the amount of basis functions used to optimize β (curly brackets), and the amount of basis function used to do the actual energy calculation are not the same. Based on this table and Table 8.10, it would seem that for $\omega = 0.1$ we need fewer basis functions to properly optimize β, but given an optimal β value, we need fewer basis functions to get good results for $\omega = 1$ than for $\omega = 0.1$.

8.4.1.2 Three Dimensions

For the three-dimensional case we see from Table 8.12 that we need a lot more basis function for the energies to converge than we did for the two-dimensional case. As always we expect the energies to be greater for three-dimensions than for two-dimensions, and from the table we see that that holds true here as well.

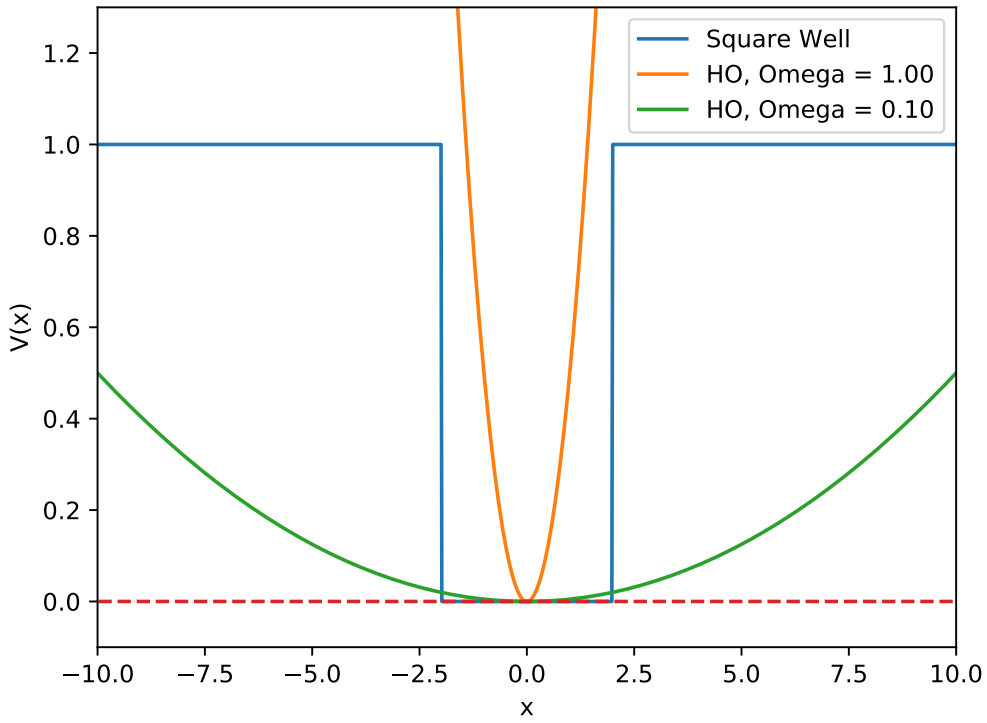


Figure 8.6: Comparison between the finite square well potential and harmonic oscillators with different ω values. From the figure we can see the effect ω has on how good a fit harmonic oscillator basis functions are to the finite square well potential. It appears that the optimal ω value would be somewhere between $\omega = 0.1$ and $\omega = 1$.

N	ω	Basis Functions	E_{coeff}
2	0.10	165	1.412(5)
	0.10	364	1.411(5)
	1.00	165	1.465(3)
	1.00	364	1.463(3)

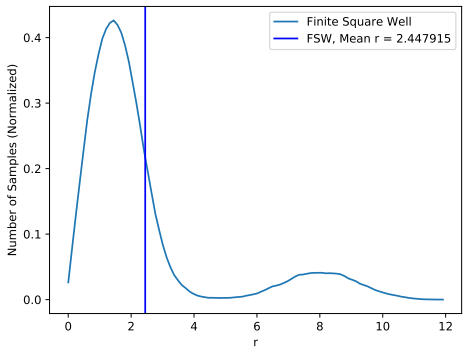
Table 8.12: The table lists ground state energy results for various finite square well systems in three dimensions. N is the number of particles. E_{coeff} are the energies obtained when the single particle wave functions are approximated by expansion in a single harmonic oscillator basis, and ω is the harmonic oscillator frequency used for the harmonic oscillator basis functions. The numbers in parenthesis are the statistical errors found using blocking. The amount of basis functions needed for the energies to converge is a lot greater for three dimensions than for two.

8.4.2 One-Body Densities

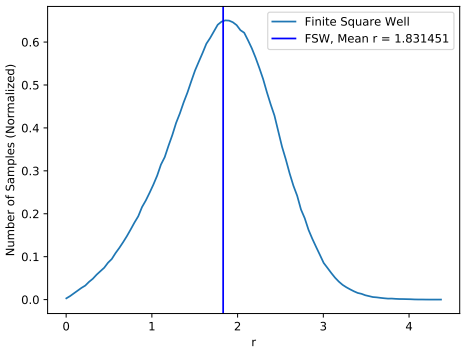
For the specific finite square well we have looked at in this thesis we expect all particles to be withing the well, so-called bound particles, if the number of particles N is 6 or less. For our well being inside the well corresponds to having $r < \sqrt{8}$. Since the walls are $x, y = 2$ away from the center, the furthest away from the center a particle should be is in one of the corners, i.e. the particle would have x and y positions close to ± 2, which would give an r value close to $\sqrt{2^2 + 2^2}$. So for the one-body density we expect the distribution of r to be 0 at $r > \sqrt{8}$.

From Figure 8.7 we see that for $N = 2$, $\omega = 0.1$ and 15 basis functions the particles are not completely bound, but for $\omega = 1$ they are (mostly). However, as the number of basis functions increase the distribution for $\omega = 0.1$ converges towards a final shape quicker than for $\omega = 1$. This fits with the energy results, where we saw that for $\omega = 0.1$ we need more basis functions to get good results, but for $\omega = 0.1$ we also find the optimal β value with fewer basis functions. The shape for $\omega = 0.1$ is worse with 15 basis functions because, even with an optimal β, $\omega = 0.1$ yields bad results for that few basis functions. However, the shape for $\omega = 0.1$ improves quicker as the number of basis functions increases because the optimal β is reached with fewer basis functions. From Figure 8.8 we see that for $N = 6$ with $\omega = 0.1$, the issue from $N = 2$ with 15 basis functions occurs even if we use 45 basis functions. This shows the need for more basis functions as the number of particles increases.

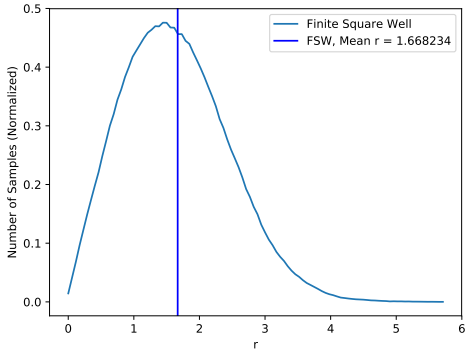
Unlike for harmonic oscillator potential, for the square well potential the strength of the potential is constant until we hit one of the walls, at which point the potential skyrockets. For $N = 2$ we should then expect the most likely position of the particle to be one that minimizes the repulsive forces between them while still keeping both particles within the well. In order to achieve this the particles should be in corners diagonally opposite of each other. This would mean that the distribution of r should have a top at $r \approx \sqrt{8}$ in the two-dimensional case. However, when we look at our distributions in Figure 8.7b we see that the top of the distributions are typically between $r = 1$ and $r = 2$. This is likely due to the fact that we keep α constant at 1, and doing a Hartree-Fock calculation on the overlap coefficient and properly optimizing α should solve this issue.



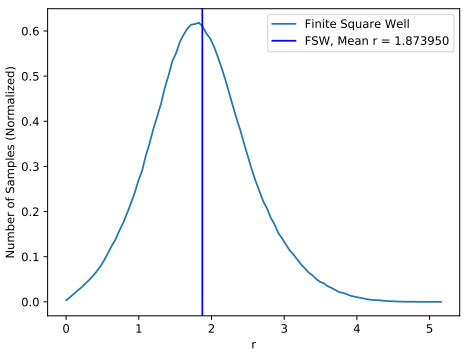
(a) $\omega = 0.1$, 15 Basis Functions



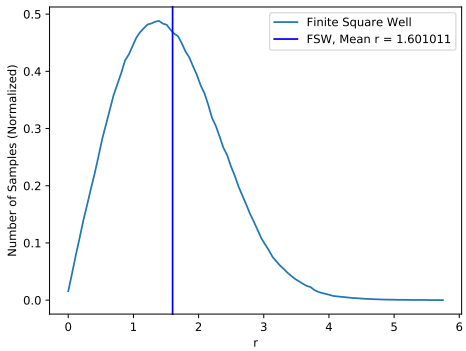
(b) $\omega = 1$, 15 Basis Functions



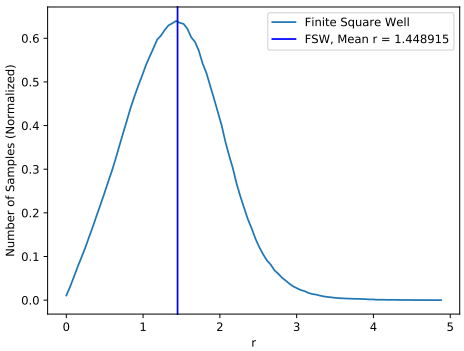
(c) $\omega = 0.1$, 45 Basis Functions



(d) $\omega = 1$, 45 Basis Functions

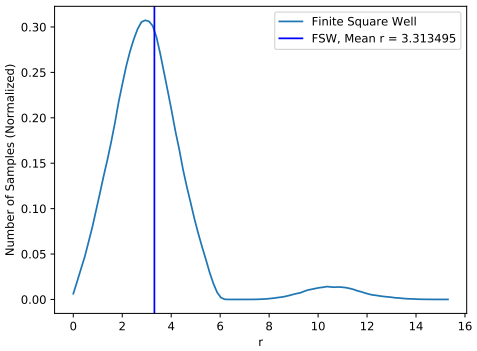


(e) $\omega = 0.1$, 120 Basis Functions

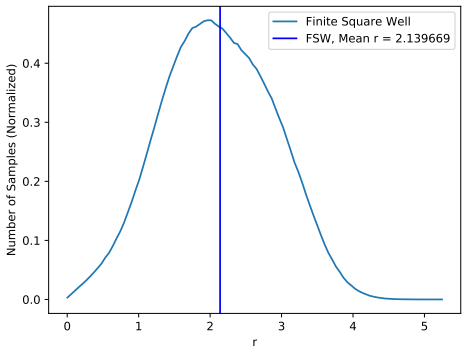


(f) $\omega = 1$, 120 Basis Functions

Figure 8.7: One-body densities for a two-dimensional finite square well potential with $N = 2$ particles. The distance between the well center and each wall is 2, and the potential is 0 inside the well and 1 outside.

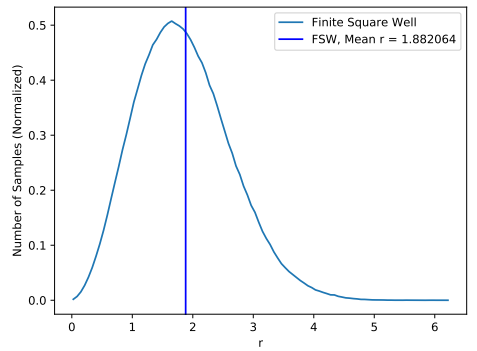


(a) $\omega = 0.1$, 45 Basis Functions

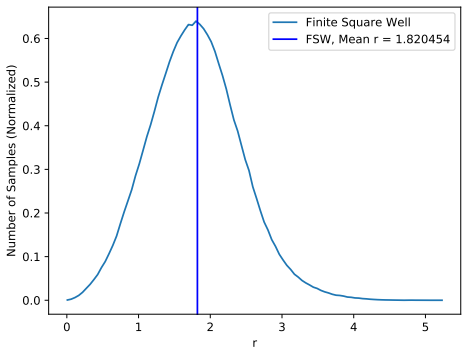


(b) $\omega = 1$, 45 Basis Functions

Figure 8.8: One-body densities for a two-dimensional finite square well potential with $N = 6$ particles. The distance between the well center and each wall is 2, and the potential is 0 inside the well and 1 outside.



(a) $\omega = 0.1$, 364 Basis Functions



(b) $\omega = 1$, 364 Basis Functions

Figure 8.9: One-body densities for a three-dimensional finite square well potential with $N = 2$ particles. The distance between the well center and each wall is 2, and the potential is 0 inside the well and 1 outside.

Chapter 9

Conclusion

Hello world!

Hello, here is some text without a meaning. This...

Appendices

Appendix A

Calculations of Closed Form Expressions

A.1 Two-body quantum dots

To find an expression for the local energy we need to find the Laplacian of the trial wave function

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}) = C \exp\left(-\alpha\omega(r_1^2 + r_2^2)/2\right) \exp\left(\frac{ar_{12}}{(1+\beta r_{12})}\right). \tag{A.1.1}$$

We define

$$u = -\alpha\omega(r_1^2 + r_2^2)/2 \tag{A.1.2}$$

$$v = \frac{ar_{12}}{(1+\beta r_{12})} \tag{A.1.3}$$

Then we have

$$\nabla_1 \exp(u) = -\alpha\omega\mathbf{r}_1 \exp(u). \tag{A.1.4}$$

For $\nabla_1 \exp(v)$ we look at the first component of the gradient

$$\begin{aligned}\frac{\partial}{\partial x_1}\exp(v(r_{12})) =& \frac{\partial r_{12}}{\partial x_1}\frac{\partial}{\partial r_{12}}\exp(v(r_{12})) \\ =& \frac{(x_1 - x_2)}{r_{12}}\frac{\partial}{\partial r_{12}}\exp(v(r_{12})) \\ =& \frac{(x_1 - x_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\exp(v(r_{12})),\end{aligned} \tag{A.1.5}$$

which gives

$$\nabla_1 \exp(v) = \frac{(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\exp(v). \tag{A.1.6}$$

For the second particle we have

$$\nabla_2 \exp(u) = -\alpha\omega\mathbf{r}_2 \exp(u) \tag{A.1.7}$$
$$\nabla_2 \exp(v) = -\nabla_1 \exp(v). \tag{A.1.8}$$

The gradients for the full wave function are then

$$\nabla_1\psi_T = \left(-\alpha\omega\mathbf{r}_1 + \frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\right) C\exp(u)\exp(v) \tag{A.1.9}$$
$$\nabla_2\psi_T = \left(-\alpha\omega\mathbf{r}_2 - \frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\right) C\exp(u)\exp(v). \tag{A.1.10}$$

To find the Laplacian we need

$$\nabla_1(-\alpha\omega\mathbf{r}_1) = -\alpha\omega d = -2\alpha\omega, \tag{A.1.11}$$

where d is the number of dimensions, in our case $d = 2$. We also need

$$\begin{aligned}\nabla_1\left(\frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\right) =& \frac{a}{(1+\beta r_{12})^2}\nabla_1\frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}} + \frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\nabla_1\frac{a}{(1+\beta r_{12})^2}\\ =& \frac{d-1}{r_{12}}\frac{a}{(1+\beta r_{12})^2} + \frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{\partial}{\partial r_{12}}\frac{a}{(1+\beta r_{12})^2}\\ =& \frac{1}{r_{12}}\frac{a}{(1+\beta r_{12})^2} - \frac{2a\beta}{(1+\beta r_{12})^3}.\end{aligned} \tag{A.1.12}$$

We also have

$$\nabla_2(-\alpha\omega\mathbf{r}_2) = \nabla_1(-\alpha\omega\mathbf{r}_1), \tag{A.1.13}$$

and

$$\nabla_2\left(-\frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\right) = \nabla_1\left(\frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\right) \tag{A.1.14}$$

We end up with the Laplacians

$$\frac{\nabla_1^2\psi_T}{\psi_T} = \left(-2\alpha\omega + \frac{1}{r_{12}}\frac{a}{(1+\beta r_{12})^2} - \frac{2a\beta}{(1+\beta r_{12})^3} + \left(-\alpha\omega\mathbf{r}_1 + \frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\right)^2\right) \tag{A.1.15}$$

$$\frac{\nabla_2^2\psi_T}{\psi_T} = \left(-2\alpha\omega + \frac{1}{r_{12}}\frac{a}{(1+\beta r_{12})^2} - \frac{2a\beta}{(1+\beta r_{12})^3} + \left(-\alpha\omega\mathbf{r}_2 - \frac{(\mathbf{r}_1-\mathbf{r}_2)}{r_{12}}\frac{a}{(1+\beta r_{12})^2}\right)^2\right). \tag{A.1.16}$$

The sum of the Laplacians in the Hamiltonian is then

$$\begin{aligned}\frac{\nabla_1^2\psi_T}{\psi_T}+\frac{\nabla_2^2\psi_T}{\psi_T} =& -4\alpha\omega+\frac{1}{r_{12}}\frac{a}{(1+\beta r_{12})^2}-\frac{2a\beta}{(1+\beta r_12)^3}\\ &+\alpha^2\omega^2(r_1^2+r_2^2)+\frac{2a^2}{(1+\beta r_{12})^4}-\alpha\omega r_{12}\frac{2a}{(1+\beta r_{12})^2}.\end{aligned} \tag{A.1.17}$$

The complete expression for the local energy is then

$$\begin{aligned}E_L =&\frac{1}{\psi_T}H\psi_T\\ =&2\alpha\omega+\frac{1}{r_{12}}\frac{a}{(1+\beta r_{12})^2}-\frac{2a\beta}{(1+\beta r_12)^3}-\frac{1}{2}\alpha^2\omega^2(r_1^2+r_2^2)\\ &-\frac{a^2}{(1+\beta r_{12})^4}+\alpha\omega r_{12}\frac{a}{(1+\beta r_{12})^2}+\frac{1}{2}\omega^2(r_1^2+r_2^2)+\frac{1}{r_{12}}\\ =&2\alpha\omega+\frac{1}{2}\omega^2(r_1^2+r_2^2)(1-\alpha^2)-\frac{2a\beta}{(1+\beta r_12)^3}\\ &-\frac{a^2}{(1+\beta r_{12})^4}+\left(\alpha\omega r_{12}+\frac{1}{r_{12}}\right)\frac{a}{(1+\beta r_{12})^2}+\frac{1}{r_{12}}.\end{aligned} \tag{A.1.18}$$

A.2 Many-body quantum dots

These calculations follow Ref. [4]. In the many-body case we have the trial wave function

$$\psi_T(\mathbf{r_1},\mathbf{r_2},\dots,\mathbf{r_N}) = Det\left(\phi_1(\mathbf{r_1}),\phi_2(\mathbf{r_2}),\dots,\phi_N(\mathbf{r_N})\right)\prod_{i<j}^{N}\exp\left(\frac{ar_{ij}}{(1+\beta r_{ij})}\right), \tag{A.2.1}$$

where *Det* is a Slater determinant, and the single-particle wave functions are the harmonic oscillator wave functions given by

$$\phi_{n_x,n_y}(x,y) = AH_{n_x}(\sqrt{\omega}x)H_{n_y}(\sqrt{\omega}y)\exp\left(-\omega(x^2+y^2)/2.\right) \tag{A.2.2}$$

A is a normalization constant, while the functions $H_{n_x}(\sqrt{\omega}x)$ are Hermite polynomials. For $N=6$ electrons we need the Hermite polynomials for $n_x=0,1$ and $n_y=0,1$, for $N=12$ we need to include the $n_x,n_y=2$ Hermite polynomials, and for $N=20$ we also need the Hermite polynomials for $n_x,n_y=3$. When evaluating the trial wave function, the calculation of the gradient and the Laplacian of an N-particle Slater determinant is likely to be most time-consuming. This is because we have to differentiate with respect to all spatial coordinates of all electrons. We can improve the efficiency of the calculation by moving only one electron at the time. When we then differentiate the Slater determinant with respect to a given coordinate of that electron, only one row in the corresponding Slater matrix is changed. This means that we don't have to recalculate the entire determinant at every Metropolis step. Instead we use an algorithm which requires us to keep track of the inverse of the Slater matrix.

The matrix elements of the Slater matrix $\hat{D}$ are given by

$$d_{ij} = \phi_j(x_i), \tag{A.2.3}$$

where $\phi_j(\mathbf{r_i})$ is a single particle wave function. x_i is one of the spatial coordinates of the given particle, while j indicates the quantum numbers (n_x and n_y in our case). The inverse of $\hat{D}$ can be expressed by its determinant $|\hat{D}|$, and its cofactors C_{ij} as follows

$$d_{ij}^{-1} = \frac{C_{ji}}{|\hat{D}|}, \tag{A.2.4}$$

where the interchanged indices of C_{ji} means that the cofactor matrix should be transposed. Assuming $\hat{D}$ is invertible we have

$$\sum_{k=1}^{N} d_{ik} d_{kj}^{-1} = \delta_{ij}. \tag{A.2.5}$$

We define the ratio R, between $|\hat{D}(\mathbf{r}^{\text{new}})|$ and $|\hat{D}(\mathbf{r}^{\text{old}})|$, which by definition can be expressed as

$$R \equiv \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) C_{ij}(\mathbf{r}^{\text{new}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) C_{ij}(\mathbf{r}^{\text{old}})}. \tag{A.2.6}$$

If we move only one electron at a time, $\mathbf{r}^{\text{new}}$ and $\mathbf{r}^{\text{old}}$ differ only by the position of that one, i-th, electron, which means $\hat{D}(\mathbf{r}^{\text{new}})$ and $\hat{D}(\mathbf{r}^{\text{old}})$ differ only by the entries of the i-th row. The i-th row of a cofactor matrix $\hat{C}$ is independent of the entries in the i-th row of the corresponding matrix $\hat{D}$. In our case this means that the i-th row of $\hat{C}(\mathbf{r}^{\text{new}})$ and $\hat{C}(\mathbf{r}^{\text{old}})$ must be equal, so we have

$$C_{ij}(\mathbf{r}^{\text{new}}) = C_{ij}(\mathbf{r}^{\text{old}}) \quad \forall\ j \in \{1, \ldots, N\}. \tag{A.2.7}$$

We use eq. (A.2.4) and eq. (A.2.7) with eq. (A.2.6) in order to obtain

$$R = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) C_{ij}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) C_{ij}(\mathbf{r}^{\text{old}})} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}})}, \tag{A.2.8}$$

where, by eq. (A.2.5), the denominator of the rightmost expression is unity, and we end up with

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}). \tag{A.2.9}$$

This means that if we only move the i-th electron, the ratio R is given by the dot product between the vector, $(\phi_1(\mathbf{r}_i^{\text{new}}), \ldots, \phi_N(\mathbf{r}_i^{\text{new}}))$, of single particle wave functions evaluated at the new position, and the i-th column of the inverse matrix $\hat{D}^{-1}$ evaluated at the original position.

We need to maintain the inverse matrix, so if the new position $\mathbf{r}^{\text{new}}$ is accepted we

need to use an algorithm for updating the inverse matrix. We start by updating all but the i-th column of $\hat{D}^{-1}$. For each column $j \neq i$, we calculate

$$S_j = (\hat{D}(\mathbf{r}^{\text{new}}) \times \hat{D}^{-1}(\mathbf{r}^{\text{old}}))_{ij} = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{new}}) d_{lj}^{-1}(\mathbf{r}^{\text{old}}), \tag{A.2.10}$$

then we calculate the new elements of the j-th column of $\hat{D}^{-1}$ as follows

$$d_{kj}^{-1}(\mathbf{r}^{\text{new}}) = d_{kj}^{-1}(\mathbf{r}^{\text{old}}) - \frac{S_j}{R} d_{ki}^{-1}(\mathbf{r}^{\text{old}}) \quad \begin{matrix} \forall \;\; k \in \{1, \ldots, N\} \\ j \neq i \end{matrix} . \tag{A.2.11}$$

The last step is to update the i-th column of $\hat{D}^{-1}$ using the following equation

$$d_{ki}^{-1}(\mathbf{r}^{\text{new}}) = \frac{1}{R} d_{ki}^{-1}(\mathbf{r}^{\text{old}}) \quad \forall \;\; k \in \{1, \ldots, N\}. \tag{A.2.12}$$

Only the i-th row of the Slater matrix changes when differentiating the Slater determinant with respect to the coordinates of a single particle $\mathbf{r}_i$ as well, which means we can calculate the gradient and the Laplacian as follows

$$\frac{\vec{\nabla}_i |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \vec{\nabla}_i d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \vec{\nabla}_i \phi_j(\mathbf{r}_i) d_{ji}^{-1}(\mathbf{r}) \tag{A.2.13}$$

and

$$\frac{\nabla_i^2 |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i^2 d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{r}_i)\, d_{ji}^{-1}(\mathbf{r}). \tag{A.2.14}$$

Therefore in order to calculate the derivatives of the Slater determinant, we only need the derivatives of the single particle wave functions and the elements of the inverse Slater matrix.

The expectation value of the kinetic energy for electron i expressed in atomic units is

$$\langle \hat{K}_i \rangle = -\frac{1}{2} \frac{\langle \Psi | \nabla_i^2 | \Psi \rangle}{\langle \Psi | \Psi \rangle}, \tag{A.2.15}$$

and we have that

$$K_i = -\frac{1}{2} \frac{\nabla_i^2 \Psi}{\Psi}. \tag{A.2.16}$$

To find the kinetic energy we need the Laplacian of the wave function. We define the Slater determinant part of the wave function as Ψ_D and define the correlation part

(Jastrow factor) as Ψ_C. The Laplacian is then

$$\begin{aligned}\frac{\nabla^2\Psi}{\Psi} &= \frac{\nabla^2(\Psi_D\,\Psi_C)}{\Psi_D\,\Psi_C} = \frac{\nabla\cdot[\nabla(\Psi_D\,\Psi_C)]}{\Psi_D\,\Psi_C} = \frac{\nabla\cdot[\Psi_C\nabla\Psi_D+\Psi_D\nabla\Psi_C]}{\Psi_D\,\Psi_C}\\ &= \frac{\nabla\Psi_C\cdot\nabla\Psi_D+\Psi_C\nabla^2\Psi_D+\nabla\Psi_D\cdot\nabla\Psi_C+\Psi_D\nabla^2\Psi_C}{\Psi_D\,\Psi_C}\\ &= \frac{\nabla^2\Psi_D}{\Psi_D}+\frac{\nabla^2\Psi_C}{\Psi_C}+2\frac{\nabla\Psi_D}{\Psi_D}\cdot\frac{\nabla\Psi_C}{\Psi_C}.\end{aligned} \tag{A.2.17}$$

From eq. (A.2.17) we see that we need the gradient and Laplacian of both Ψ_D and Ψ_C. For Ψ_D the necessary expression are given by eq. (A.2.13) and eq. (A.2.14). We have that

$$\Psi_C = \prod_{i<j}\exp f(r_{ij}) = \exp\left\{\sum_{i<j}\frac{ar_{ij}}{1+\beta r_{ij}}\right\}, \tag{A.2.18}$$

and by differentiating with the chain rule similarly to what we did in eq. (A.1.5), we find that the gradient is

$$\frac{\nabla_k\Psi_C}{\Psi_C} = \sum_{j\neq k}\frac{\mathbf{r}_{kj}}{r_{kj}}\frac{\partial f(r_{kj})}{\partial r_{kj}} = \sum_{j\neq k}\frac{\mathbf{r}_{kj}}{r_{kj}}f'(r_{kj}) = \sum_{j\neq k}\frac{\mathbf{r}_{kj}}{r_{kj}}\frac{a}{(1+\beta r_{kj})^2}, \tag{A.2.19}$$

where

$$f'(r_{kj}) = \frac{\partial}{\partial r_{kj}}f(r_{kj}). \tag{A.2.20}$$

To find the Laplacian we need to calculate

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \frac{1}{\Psi_C}\nabla_k\sum_{j\neq k}\frac{\mathbf{r}_{kj}}{r_{kj}}f'(r_{kj})\Psi_C. \tag{A.2.21}$$

We follow the calculations from Ref. [4] and Ref. [5]. Using the product rule we get

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \frac{1}{\Psi_C}\sum_{j\neq k}\left(\frac{\mathbf{r}_{kj}}{r_{kj}}f'(r_{kj})\nabla_k\Psi_C+\frac{\mathbf{r}_{kj}}{r_{kj}}\Psi_C\nabla_k f'(r_{kj})+f'(r_{kj})\Psi_C\nabla_k\frac{\mathbf{r}_{kj}}{r_{kj}}\right) \tag{A.2.22}$$

$$= \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}f'(r_{ki})f'(r_{kj})+\sum_{j\neq k}\left(f''(r_{kj})+\frac{d-1}{r_{kj}}f'(r_{kj})\right), \tag{A.2.23}$$

where d is the number of dimensions, $d=2$ in our case. We end up with the following expression

$$\frac{\nabla_k^2\Psi_C}{\Psi_C} = \sum_{ij\neq k}\frac{(\mathbf{r}_k-\mathbf{r}_i)(\mathbf{r}_k-\mathbf{r}_j)}{r_{ki}r_{kj}}\frac{a}{(1+\beta r_{ki})^2}\frac{a}{(1+\beta r_{kj})^2}+\sum_{j\neq k}\left(\frac{a}{r_{kj}(1+\beta r_{kj})^2}-\frac{2a\beta}{(1+\beta r_{kj})^3}\right). \tag{A.2.24}$$

Appendix B

Program Structure

The simulations in this thesis were done by two programs implemented in C++ using object-orientation. The VMC simulation program consists of several classes responsible for different parts of the simulations. The following classes are responsible for the VMC simulations in this thesis. Any other classes from the vmc-solver folder in the Github repository [17] are outside the scope of this thesis.

- **Hamiltonian:** A super-class for different Hamiltonians. The subclasses calculate the local energy for their specific Hamiltonian. They also calculate the single particle wave functions if these are not approximated by expanding in a basis of single harmonic oscillator functions (not implemented for the square well). Since calculating the kinetic energy with numerical differentiation is done the same for all Hamiltonians, this super-class is responsible for that. The subclasses are:
  - **HarmonicOscillatorElectrons:** Calculates the local energy for quantum dots in a single harmonic oscillator potential well.
  - **DoubleHarmonicOscillator:** Calculates the local energy for quantum dots in a double harmonic oscillator potential well.
  - **SquareWell:** Calculates the local energy for quantum dots in a finite square well potential.
- **HermitePolynomials:** Stores the expressions for Hermite polynomials and their derivatives and double derivatives in individual subclasses. All subclasses are included in the superclass file. The first 50 Hermite polynomials are implemented, which means a total of 150 subclasses (50 each for polynomial, derivative and double derivate).
- **WaveFunction:** A super-class for different wave functions. The subclasses evaluate their specific wave function and also calculate the gradient, the Laplacian and the derivative w.r.t. the variational parameters α and β. The subclasses are:
  - **ManyElectrons:** WaveFunction subclass for many-body quantum dots This includes all Slater determinant functionality. This class uses single/double harmonic oscillator single particle wave functions directly, which it gets from the corresponding Hamiltonian subclasses.

  - **ManyElectrons_Coefficients:** Similar to ManyElectrons, but with the one key difference that it approximates the single particle wave functions by expansion in a single harmonic oscillator basis.

- **InitialState:** A super-class for different initial states. The subclasses set up the initial state. The subclasses are:

  - **RandomUniform:** Sets up an initial state with uniformly distributed particle positions. The particles are distributed around the well center (or centers in the double well case).

- **Particle:** Responsible for creating particles and adjusting their positions.

- **System:** Responsible for running the Monte Carlo simulation. It performs the Metropolis and Metropolis-Hastings algorithms.

- **Sampler:** Responsible for sampling interesting quantities and computing averages. It is also responsible for providing the data to the user, both by printing to terminal and saving to file.

- **SteepestDescent:** Responsible for optimizing variational parameters using the Steepest Descent method. Once the optimal parameter has been found, the System class is tasked with running a large Monte Carlo simulation.

- **Random:** Responsible for generating pseudo-random numbers according to different distributions.

There is also a main program which sets the necessary parameters and makes calls to the classes to start the simulation. The code is also fully parallelized with MPI. An advantage to using an implementation like this is that it makes it easy to add functionality, like other potentials and alternative methods for optimizing variational parameters (e.g. the Conjugate Gradient method).

The other program used is the diagonalization program, which is responsible for diagonalizing the single particle problem for a given potential, and then find the overlap with the single harmonic oscillator basis. The overlap coefficients are then saved to a file so they can be used by the VMC solver. This program consists of:

- **WaveFunction:** This superclass contains subclasses for different potentials. The subclasses also calculate the harmonic oscillator basis functions. The "WaveFunction" superclass also computes the Hermite polynomials used in the basis functions. If other types of basis functions are implemented, a new basis functions superclass should be made, and the harmonic oscillator basis functions should be calculated by a subclass of that basis functions superclass.

  - **DoubleWell:** Subclass for the double harmonic oscillator potential. This subclass is also used for the single harmonic oscillator potential by setting the L parameter for each dimension to 0.
  - **SquareWell:** Subclass for the square well potential. In this thesis only single square wells were simulated, no double (or multi) square wells.

- **System:** This class is responsible for doing the diagonalization, and finding the overlap coefficients. It also has functionality for approximating the single particle wave functions by expansion in a single harmonic oscillator basis. This functionality was only added for testing purposes, as the VMC solver needs to do this itself mid-simulation (for varying particle positions).

Here as well, there is a main program which sets the necessary parameters, and gets the "System" class started. It also saves necessary data to files (e.g. the overlap coefficients). The data analysis is done in Python, with the programs **blocking.py** and **density.py**. Testing of the diagonalization program was done by the Python program **plot_data.py**, and plots of the potentials were created using **potential.py**. Finally, the code for the Hermite polynomial subclasses in the VMC solver was generated automatically with SymPy in the program **hermitePolynomialGenerator.py**. For the C++ programs the library Armadillo [26] was used extensively for matrix operations etc. Open MPI [27] was used for parallel computing. For the Python programs the following packages were used: Matplotlib [28], NumPy [29], and SymPy [30].

Appendix C

Code Generation with SymPy

In order to optimize the VMc solver we wanted to have explicit expressions for a lot of Hermite polynomials and corresponding derivatives and double derivatives. Not only would it be an extensive amount of work to calculate the expressions by hand, but writing the code to implement the expressions would also be a lot of work. In addition the risk of committing error during the implementation would be great. Instead we can use SymPy [30] for this task, which is a Python library for symbolic mathematics. [30] This means that we can for example do symbolic differentiation with a Python program. This is extremely useful in our case since we need to symbolically differentiate a lot of expressions. We also use SymPy to symbolically calculating new Hermite polynomials using the recursive relation of Hermite polynomials. First we specify what variables SymPy should treat as symbols with the "symbols" function:

```
1 x, a, w, aw = symbols('x, m_alpha, m_omega, m_alphaomega')
```

Now we can simply set up the recursion relation of the Hermite polynomials, with these symbols in place of x, α and ω values:

```
1 H[0] = 1
2
3 for n in range(1, nMax):
4         H[n] = simplify(2*sqrt(aw)*x*H[n-1] - diff(H[n-1], x))
```

The "simplify" function ensures the result is on a simple form (or at least as simple as possible). The "diff" function does a symbolic differentiation on the first argument with respect to the second argument. Now we have all the Hermite polynomials we need on symbolic form. We continue by finding the derivatives with the following simple loop:

```
1 for n in range(0, nMax):
2         Hd[n] = simplify(diff(H[n], x))
```

and the double derivatives with:

```
1 for n in range(0, nMax):
2         Hdd[n] = simplify(diff(H[n], x, 2))
```

where the third argument of "diff" indicates that we want to differentiate twice. Now we have all the symbolic expressions we need, but with one more simple function we can get more out of SymPy. The function "codegen" can automatically create C functions for calculating our Hermite expressions, and print these expression to the terminal so we can copy them into our C++ code. With the simple code:

```
for n in range(0, nMax):
        print codegen(("HermitePolynomial_%i::eval" %n, H[n]), "c", "file",
            header=False)[0][1]
        print codegen(("dell_HermitePolynomial_%i::eval" %n, Hd[n]), "c", "file",
             header=False)[0][1]
        print codegen(("lapl_HermitePolynomial_%i::eval" %n, Hdd[n]), "c", "file"
            , header=False)[0][1]
```

we get terminal output on the following form:

```
#include "file.h"
#include <math.h>

double HermitePolynomial_0::eval() {

   double HermitePolynomial_0::eval_result;
   HermitePolynomial_0::eval_result = 1;
   return HermitePolynomial_0::eval_result;

}

#include "file.h"
#include <math.h>

double dell_HermitePolynomial_0::eval() {

   double dell_HermitePolynomial_0::eval_result;
   dell_HermitePolynomial_0::eval_result = 0;
   return dell_HermitePolynomial_0::eval_result;

}

#include "file.h"
#include <math.h>

double lapl_HermitePolynomial_0::eval() {

   double lapl_HermitePolynomial_0::eval_result;
   lapl_HermitePolynomial_0::eval_result = 0;
   return lapl_HermitePolynomial_0::eval_result;

}
```

This output still needs some modifications before it can be used in the VMC solver. SymPy has a lot more advanced functionality which could get us the exact output we want, but since the generated code we need is fairly general the modifications can be easily done with the find/replace option of a text editor. The first thing we do is replace all the "#include" lines with nothing, since we copy all the functions into the same file which we have set up beforehand. Next we replace the name of the variable in each function with something simple. To do this we replace the phrase "ermitePolynomial_0::eval_result" with nothing and do this again with the 0 changed to the numbers needed. In our case this involves 50 find/replace commands, but since the only thing we need to change in between is one number it's quickly done. Our variables then end up

as simply "H", "dell_H" and "lapl_H".

The modifications done so far are not strictly necessary, but was done to clean up the code somewhat. The next two modifications are needed for the VMC simulation to work properly. First we need to add an argument to the functions, since we want to send in the position x. This is done by replacing "eval()" with "eval(double x)". The final modification is needed due to an issue we run into with the int type in C++ when we have larger Hermite polynomials. For Hermite polynomial number 50, the expressions involve large integers, e.g. 126995894129839951916263049307749351424000000. Since SymPy simply prints the numbers like this, they are considered as int type when copied into the C++ program. However, the number is too large to be an int type in C++, so we need to change the type. This is done by simply adding a period to the end of the these numbers. These large numbers are always a constant in front of a pow function in our auto-generated code, so the code looks like "i*pow(", where "i" is the integer. We can then simply replace "0*pow(" with "0.*pow", and do this for $0, \ldots, 9$ so that the change is done to all relevant integers regardless of what the last digit is.

An alternative to using "codegen", is to use the SymPy function "printing.ccode" to print the Hermite expressions as C code instead of Python code (e.g. changing "1**2" to "pow(1, 2)"). We can then create our own custom string to print out with the variable names we want, no "include" lines, etc. This makes the code in "hermitePolynomialGenerator.py" somewhat messy, but in return the only modification we need to make after generating the code is to add the periiods discussed above. Instead of the "codegen" code, we would then use the following:

```
for n in range(0, nMax):
        print("""double HermitePolynomial_%i::eval(double x) {

    double H;
    H = %s;
    return H;

}




double dell_HermitePolynomial_%i::eval(double x) {

    double dell_H;
    dell_H = %s;
    return dell_H;

}




double lapl_HermitePolynomial_%i::eval(double x) {

    double lapl_H;
    lapl_H = %s;
    return lapl_H;

}""" %(n, printing.ccode(H[n]), n, printing.ccode(Hd[n]), n, printing.ccode(Hdd[n
    ])))
```

We also use this approach to auto-generate the class constructors and declarations, as well as the code to store the "HermitePolynomials" objects in arrays.

Bibliography

[1] M. Hjorth-Jensen, *Conjugate gradient methods and other optimization methods*, https://github.com/CompPhysics/ComputationalPhysics2/blob/master/doc/pub/cg/pdf/cg-print.pdf, Read: 17.03.2016.

[2] M. Taut, *Two electrons in an external oscillator potential: Particular analytic solutions of a Coulomb correlation problem*, Phys. Rev. A **48**, 3561 - 3566 (1993).

[3] M. .L. Pedersen, G. Hagen, M. Hjorth-Jensen, S. Kvaal, and F. Pederiva, *Ab initio computation of the energies of circular quantum dots*, Phys. Rev. B **84**, 115302 (2011).

[4] M. Hjorth-Jensen, *Computational Physics 2: Variational Monte Carlo methods*, https://github.com/CompPhysics/ComputationalPhysics2/blob/master/doc/pub/vmc/pdf/vmc-print.pdf, Read: 16.06.2016.

[5] M. Hjorth-Jensen, *Computational Physics Lecture Notes Fall 2015*, https://github.com/CompPhysics/ComputationalPhysics2/blob/master/doc/Literature/lectures2015.pdf, Chapter 16.10, Read: 16.06.2016.

[6] David j. Griffiths, *Introduction to Quantum Mechanics*, Pearson, 2nd edition edition, 2005.

[7] Aslak Tveito, Hans Petter Langtangen, Bjørn Frederik Nielsen and Xing Cai, *Elements of Scientific Computing*, Springer, 1st edition, 2010.

[8] B.H. Bransden and C.J. Joachain, *Physics of Atoms and molecules*, Longman, 1983. Chapter 2.

[9] http://hyperphysics.phy-astr.gsu.edu/hbase/quantum/Scheq.html, Read 23.05.17

[10] M. Hjorth-Jensen, https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Projects/2015/Project2/project2_2015.pdf

[11] M. Hjorth-Jensen, https://github.com/CompPhysics/ThesisProjects/blob/master/doc/pub/quantumdots/pdf/quantumdots-minted.pdf

[12] M. Hjorth-Jensen, *Hartree-Fock methods and introduction to Many-body Theory*, https://github.com/CompPhysics/ComputationalPhysics2/blob/gh-pages/doc/pub/basicMB/pdf/basicMB-print.pdf

[13] Sebastian Ruder, *An overview of gradient descent optimization algorithms*, 2016, arXiv:1609.04747 [cs.LG]

[14] https://en.wikipedia.org/wiki/Quantum_dot, Read: 16.06.2016.

[15] Bjarne Stroustrup, *Bjarne Stroustrup's C++ Glossary*, http://www.stroustrup.com/glossary.html#Gpolymorphism, Read: 09.05.2017.

[16] *Using the GNU Compiler Collection (GCC)*, section 3.10: *Options That Control Optimization*, http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, Read 10.05.2017

[17] Christian Fleishcer, Github Repository 2017, https://github.com/christianfleischer/Quantum-Dot-Project.

[18] https://en.wikipedia.org/wiki/Overhead_(computing), Read 14.05.2017

[19] https://en.wikipedia.org/wiki/Race_condition, Read 23.05.2017

[20] https://en.wikipedia.org/wiki/Logic_gate, Read 23.05.2017

[21] Yang Min Wang. Coupled-Cluster Studies of Double Quantum Dots. Master's thesis, University of Oslo, 2011.

[22] Jørgen Høgberget. Quantum Monte-Carlo Studies of Generalized Many-body Systems. Master's thesis, University of Oslo, 2013.

[23] Sarah Reimann. Quantum-mechanical systems in traps and Similarity Renormalization Group theory. Master's thesis, University of Oslo, 2013.

[24] Christoffer Hirth. Studies of quantum dots: Ab initio coupledcluster analysis using OpenCL and GPU programming. Master's thesis, University of Oslo, 2012.

[25] Veronica K. B. Olsen. Full Configuration Interaction Simulation of Quantum Dots. Master's thesis, University of Oslo, 2012.

[26] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based C++ library for linear algebra. Journal of Open Source Software, Vol. 1, pp. 26, 2016. http://dx.doi.org/10.21105/joss.00026

[27] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004.

[28] John D. Hunter. Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55, Publisher link: `http://aip.scitation.org/doi/abs/10.1109/MCSE.2007.55`

[29] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37, Publisher link: `http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37`

[30] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy: symbolic computing in Python. PeerJ Computer Science 3:e103 `https://doi.org/10.7717/peerj-cs.103`

[31] Daniel V. Schroeder, *Multiple Wells* `http://physics.weber.edu/schroeder/quantum/MultipleWells.pdf`