Thesis

Christian Fleischer

1   Introduction

Part I

Theory

2   Variational Monte Carlo

The Variational Monte Carlo method is one of the computational methods 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

$$E[H] = \langle H \rangle = \frac{\int d\mathbf{R} \Psi_T^*(\mathbf{R}, \boldsymbol{\alpha}) H(\mathbf{R}) \Psi_T(\mathbf{R}, \boldsymbol{\alpha})}{\int d\mathbf{R} \Psi_T^*(\mathbf{R}, \boldsymbol{\alpha}) \Psi_T(\mathbf{R}, \boldsymbol{\alpha})} \tag{2.1}$$

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

$$E_0 \leq \langle H \rangle. \tag{2.2}$$

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)

$$P(\mathbf{R}) = \frac{|\Psi_T(\mathbf{R})|^2}{\int |\Psi_T(\mathbf{R})|^2 d\mathbf{R}} \tag{2.3}$$

Together with the local energy given in Eq.(7.13) we have that the approximation to the expectation value of the Hamiltonian is

$$\begin{aligned} E[H(\alpha)] &= \int P(\mathbf{R}) E_L(\mathbf{R}) d\mathbf{R} \\ &\approx \frac{1}{N} \sum_{i=1}^{N} P(\mathbf{R}_i, \boldsymbol{\alpha}) E_L(\mathbf{R}_i, \boldsymbol{\alpha}), \end{aligned} \tag{2.4}$$

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

$$\frac{A_{i \to j}}{A_{j \to i}} = \frac{p_i T_{i \to j}}{p_j T_{j \to i}}, \tag{2.5}$$

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

$$\frac{p_j}{p_i} = \frac{T_{i \to j} A_{i \to j}}{T_{j \to i} A_{j \to i}} \tag{2.6}$$

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

$$w = \frac{p_j}{p_i} = \frac{A_{i \to j}}{A_{j \to i}} = \frac{|\Psi_T(\mathbf{R}_j)|^2}{|\Psi_T(\mathbf{R}_i)|^2} \tag{2.7}$$

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.

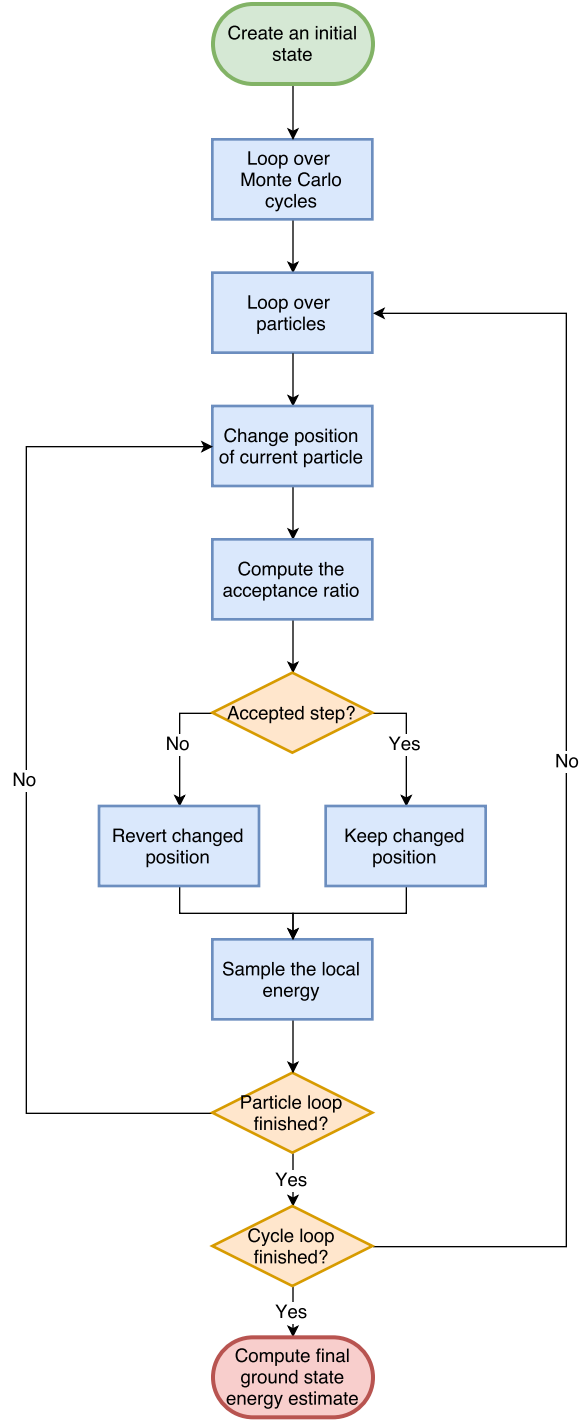- Finish and compute the final estimate for the ground state energy.

Figure 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).

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

$$\frac{\partial x(t)}{\partial t} = DF(x(t)) + \eta, \tag{2.8}$$

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

$$y = x + DF(x)\Delta t + \xi\sqrt{\Delta t}, \tag{2.9}$$

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

$$\frac{\partial P}{\partial t} = \sum_i D\frac{\partial}{\partial \mathbf{x}_i}\left(\frac{\partial}{\partial \mathbf{x}_i} - \mathbf{F}_i\right)P(\mathbf{x}, t), \tag{2.10}$$

where $\mathbf{F_i}$ is the ith 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.

$$\frac{\partial^2 P}{\partial \mathbf{x}_i^2} = P\frac{\partial}{\partial \mathbf{x}_i}\mathbf{F}_i + \mathbf{F}_i\frac{\partial}{\partial \mathbf{x}_i}P. \tag{2.11}$$

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

$$\frac{\partial^2 P}{\partial \mathbf{x}_i^2} = P\frac{\partial g}{\partial P}\left(\frac{\partial P}{\partial \mathbf{x}_i}\right)^2 + Pg\frac{\partial^2 P}{\partial \mathbf{x}_i^2} + g\left(\frac{\partial P}{\partial \mathbf{x}_i}\right)^2. \tag{2.12}$$

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

$$\mathbf{F} = 2\frac{1}{\Psi_T}\nabla\Psi_T, \tag{2.13}$$

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

$$G(y, x, \Delta t) = \frac{1}{(4\pi D\Delta t)^{3N/2}}\exp(-(y - x - D\Delta tF(x))^2/4D\Delta t), \tag{2.14}$$

which means that the ratio used in the Metropolis algorithm

$$w = \frac{|\Psi_T(\mathbf{R}_j)|^2}{|\Psi_T(\mathbf{R}_i)|^2} = q(y, x) = \frac{|\Psi_T(y)|^2}{|\Psi_T(x)|^2}, \tag{2.15}$$

is now replaced by

$$q(y, x) = \frac{G(x, y, \Delta t)|\Psi_T(y)|^2}{G(y, x, \Delta t)|\Psi_T(x)|^2}. \tag{2.16}$$

Using this ratio and Eq.(2.9), 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])

$$R = \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} \frac{\Psi_C^{\text{new}}}{\Psi_C^{\text{old}}}, \tag{2.17}$$

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

$$R_{SD} = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}), \tag{2.18}$$

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

$$d_{kj}^{-1}(\mathbf{r}^{\text{new}}) = \begin{cases} d_{kj}^{-1}(\mathbf{r}^{\text{old}}) - \frac{d_{ki}^{-1}(\mathbf{r}^{\text{old}})}{R} \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{new}}) d_{lj}^{-1}(\mathbf{r}^{\text{old}}) & \text{if } j \neq i \\[4mm] \frac{d_{ki}^{-1}(\mathbf{r}^{\text{old}})}{R} \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{old}}) d_{lj}^{-1}(\mathbf{r}^{\text{old}}) & \text{if } j = i \end{cases}$$

For the correlation part of R we simply have

$$R_C = \frac{\Psi_C^{\text{new}}}{\Psi_C^{\text{old}}} = \prod_{i<j}^{N} \exp\left(f_{ij}^{\text{new}} - f_{ij}^{\text{old}}\right) \tag{2.19}$$

$$= \exp\left(\sum_{i<j}^{N} f_{ij}^{\text{new}} - f_{ij}^{\text{old}}\right), \tag{2.20}$$

where

$$f_{ij} = \frac{ar_{ij}}{(1 + \beta r_{ij})}. \tag{2.21}$$

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

$$R_C = \exp\left(\sum_{i=1}^{k-1} f_{ik}^{\text{new}} - f_{ik}^{\text{old}} + \sum_{j=k+1}^{N} f_{kj}^{\text{new}} - f_{kj}^{\text{old}}\right)$$

$$= \exp\left(\sum_{i=1,i\neq k}^{N} f_{ik}^{\text{new}} - f_{ik}^{\text{old}}\right). \tag{2.22}$$

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

$$\mathbf{F} = 2\frac{1}{\Psi_T}\nabla\Psi_T. \tag{2.23}$$

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

$$\frac{\vec{\nabla}_i|\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \vec{\nabla}_i\phi_j(\mathbf{r}_i)d_{ji}^{-1}(\mathbf{r}), \tag{2.24}$$

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

$$\frac{\nabla_k\Psi_C}{\Psi_C} = \sum_{j\neq k} \frac{\mathbf{r}_{kj}}{r_{kj}}\frac{a}{(1+\beta r_{kj})^2}. \tag{2.25}$$

2.2 Splitting the Slater Determinant

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

$$\Phi(\mathbf{r}_1, \mathbf{r}_2,, \mathbf{r}_3, \mathbf{r}_4, \alpha, \beta, \gamma, \delta) = \frac{1}{\sqrt{4!}}\begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) & \psi_{100\uparrow}(\mathbf{r}_3) & \psi_{100\uparrow}(\mathbf{r}_4) \\ \psi_{100\downarrow}(\mathbf{r}_1) & \psi_{100\downarrow}(\mathbf{r}_2) & \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) & \psi_{200\uparrow}(\mathbf{r}_3) & \psi_{200\uparrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_1) & \psi_{200\downarrow}(\mathbf{r}_2) & \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}, \tag{2.26}$$

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

$$\Phi(\mathbf{r}_1, \mathbf{r}_2,, \mathbf{r}_3, \mathbf{r}_4, \alpha, \beta, \gamma, \delta) = \det\uparrow(1,2)\det\downarrow(3,4) - \det\uparrow(1,3)\det\downarrow(2,4)$$
$$- \det\uparrow(1,4)\det\downarrow(3,2) + \det\uparrow(2,3)\det\downarrow(1,4) \tag{2.27}$$
$$- \det\uparrow(2,4)\det\downarrow(1,3) + \det\uparrow(3,4)\det\downarrow(1,2),$$

where

$$\det\uparrow(1,2) = \frac{1}{\sqrt{2}}\begin{vmatrix} \psi_{100\uparrow}(\mathbf{r}_1) & \psi_{100\uparrow}(\mathbf{r}_2) \\ \psi_{200\uparrow}(\mathbf{r}_1) & \psi_{200\uparrow}(\mathbf{r}_2) \end{vmatrix}, \tag{2.28}$$

and

$$\det \downarrow (3,4) = \frac{1}{\sqrt{2}} \begin{vmatrix} \psi_{100\downarrow}(\mathbf{r}_3) & \psi_{100\downarrow}(\mathbf{r}_4) \\ \psi_{200\downarrow}(\mathbf{r}_3) & \psi_{200\downarrow}(\mathbf{r}_4) \end{vmatrix}. \tag{2.29}$$

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

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, , \mathbf{r}_3, \mathbf{r}_4, \alpha, \beta, \gamma, \delta) \propto \det \uparrow (1,2) \det \downarrow (3,4), \tag{2.30}$$

and in general we can use the approximation

$$\Phi(\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_N) \propto \det \uparrow \det \downarrow, \tag{2.31}$$

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 $\downarrow$

$$|\hat{D}| = |\hat{D}|_\uparrow \cdot |\hat{D}|_\downarrow \tag{2.32}$$

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

$$\frac{|\hat{D}|^{\text{new}}}{|\hat{D}|^{\text{old}}} = \frac{|\hat{D}|_\uparrow^{\text{new}}}{|\hat{D}|_\uparrow^{\text{old}}} \cdot \frac{|\hat{D}|_\downarrow^{\text{new}}}{|\hat{D}|_\downarrow^{\text{old}}}, \tag{2.33}$$

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 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

$$\sigma = \sqrt{\frac{1}{n}(\langle H^2 \rangle - \langle H \rangle^2)}. \tag{2.34}$$

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

$$\sigma = \sqrt{\frac{1 + 2\tau/\Delta t}{n}(\langle H^2 \rangle - \langle H \rangle^2)}, \tag{2.35}$$

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.(2.34) 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
```

```
35        maxBlockSize = 10000
36        numberOfSizes = (maxBlockSize−minBlockSize)/deltaBlockSize + 1
37        largestBlockSize = minBlockSize + (numberOfSizes−1)*deltaBlockSize  #9910
38
39        #blockSizes = np.zeros(numberOfSizes)
40        blockSizes = np.linspace(minBlockSize, largestBlockSize, numberOfSizes).
              astype(int)
41        blockAmounts = len(energies)/blockSizes#np.zeros(numberOfSizes)
42        means = np.zeros(numberOfSizes)
43        variances = np.zeros(numberOfSizes)
44
45        for i in range(numberOfSizes):
46            #blockSize = minBlockSize + i*deltaBlockSize
47            #blockAmount = len(energies)/blockSize
48            #mean, variance = blocking(energies, blockAmount, blockSize)
49            mean, variance = blocking(energies, blockAmounts[i], blockSizes[i])
50            means[i] = mean
51            variances[i] = variance
52
53        standardDeviation = np.sqrt(abs(variances)/(blockAmounts−1.))
54        plt.plot(blockSizes, standardDeviation)
55        plt.xlabel("Block Size")
56        plt.ylabel(r"Standard Deviation $\sigma$")
57        plt.title("N=%i" %N)
58        plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))
59        plt.show()
```

Blocking in Python

2.4   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 and 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. The Steepest Descent method then finds a local minimum for this function by iteratively moving in the direction given by the negative of the gradient of the function at the current point (Ref. [6]). At each step the move is proportional to a chosen step length γ, and for α, one step is then

$$\alpha_{i+1} = \alpha_i - \gamma \bar{E}_\alpha, \tag{2.36}$$

where

$$\bar{E}_\alpha = \frac{d\langle E_L[\alpha]\rangle}{d\alpha} \tag{2.37}$$

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

$$\bar{\psi}_\alpha = \frac{d\psi[\alpha]}{d\alpha}. \tag{2.38}$$

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

$$\bar{E}_\alpha = 2\left(\left\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]}E_L[\alpha]\right\rangle - \left\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]}\right\rangle\langle E_L[\alpha]\rangle\right), \tag{2.39}$$

10

so we need to compute

$$\left\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]} E_L[\alpha] \right\rangle, \tag{2.40}$$

and

$$\left\langle \frac{\bar{\psi}_\alpha}{\psi[\alpha]} \right\rangle, \tag{2.41}$$

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.(2.40), Eq.(2.41) and the expectation value for the ground state energy (as usual). Using the expectation values we calculate the gradient Eq.(2.39) and find new parameters using Eq.(2.36). 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.

3   Diffusion Monte Carlo

4   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. (7.12). 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.

4.1   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

$$-\frac{\hbar^2}{2m} \frac{\partial^2 \psi(x)}{\partial x^2} + V(x)\psi(x) = E\psi(x), \tag{4.1}$$

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

$$-\frac{1}{2} \frac{\partial^2 \psi(x)}{\partial x^2} + V(x)\psi(x) = E\psi(x). \tag{4.2}$$

To find the eigenvalues and eigenvectors we first discretize this equation. Using the central finite difference scheme for the second derivative we get

$$\frac{\partial^2 \psi(x)}{\partial x^2} = \frac{\psi(x+h) - 2\psi(x) + \psi(x-h)}{h^2}, \tag{4.3}$$

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

$$h = \frac{x_{\max} - x_{\min}}{N}. \tag{4.4}$$

We then have that an arbitrary value of x is

$$x_i = x_{\min} + ih \qquad i = 0, 1, 2, \ldots, N. \tag{4.5}$$

Inserting this into Eq.(4.2) we get the following Schrödinger equation for x_i

$$-\frac{\psi(x_i + h) - 2\psi(x_i) + \psi(x_i - h)}{2h^2} + V(x_i)\psi(x_i) = E\psi(x_i), \tag{4.6}$$

or, using a more compact notation

$$-\frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{2h^2} + V_i\psi_i = E\psi_i. \tag{4.7}$$

We define the diagonal matrix elements

$$d_i = \frac{2}{2h^2} + V_i = \frac{1}{h^2} + V_i, \tag{4.8}$$

and the off-diagonal matrix elements

$$e_i = -\frac{1}{2h^2}. \tag{4.9}$$

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

$$d_i\psi_i + e_{i-1}\psi_{i-1} + e_{i+1}\psi_{i+1} = E\psi_i, \tag{4.10}$$

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

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 & \ldots & 0 & 0 \\ e_1 & d_2 & e_2 & 0 & \ldots & 0 & 0 \\ 0 & e_2 & d_3 & e_3 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & d_{N-2} & e_{N-2} \\ 0 & \ldots & \ldots & \ldots & \ldots & e_{N-2} & d_{N-1} \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \ldots \\ \ldots \\ \ldots \\ \psi_{N-1} \end{pmatrix} = E \begin{pmatrix} \psi_1 \\ \psi_2 \\ \ldots \\ \ldots \\ \ldots \\ \psi_{N-1} \end{pmatrix} \tag{4.11}$$

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

4.2 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, i.e.

$$C_{n',n} = \langle \psi_{n'} | \phi_n \rangle = \sum_{i=0}^{N-1} \psi_{n'}(x_i) \phi_n(x_i) \qquad n', n = 0, 1, 2, \ldots, \tag{4.12}$$

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

$$\phi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi \hbar} \right)^{1/4} e^{-\frac{m\omega x^2}{2\hbar}} H_n \left(\sqrt{\frac{m\omega}{\hbar}} x \right), \tag{4.13}$$

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

$$\phi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{\omega}{\pi} \right)^{1/4} e^{-\frac{\omega x^2}{2}} H_n \left(\sqrt{\omega} x \right). \tag{4.14}$$

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

$$\frac{d^2 H(x)}{dx^2} - 2x \frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0, \tag{4.15}$$

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

$$H_{n+1}(x) = 2x H_n(x) - 2n H_{n-1}(x), \tag{4.16}$$

with

$$H_0(x) = 1, \tag{4.17}$$

$$H_1(x) = 2x. \tag{4.18}$$

4.3 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

$$\psi_{n'}(x) = \sum_{n_x=0}^{\Lambda} C_{n',n_x} \phi_{n_x}(x), \tag{4.19}$$

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. (4.14). 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. (4.19) is a product on the form

$$T = C_{n',n_x} \phi_{n_x}(x) C_{n',n_y} \phi_{n_y}(y) C_{n',n_z} \phi_{n_z}(z). \tag{4.20}$$

```
                    _____
                     nx=2
                     n=2

                    _____
                     nx=1
                     n=1

                    _____
                     nx=0
                     n=0
```
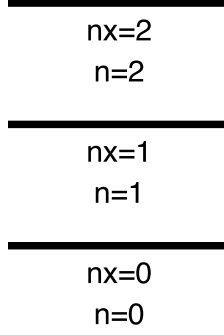
Figure 2: This figure shows the eigenstates for the three lowest energy levels in one dimension. Each line is one eigenstate, and a given eigenstate is on a higher energy level than those below it in the figure. The eigenstates are indexed by n, and nx is the quantum number corresponding to the eigenstate. In one dimension we only have one eigenstate per energy level and therefore we always have $nx = n$. However, as seen in Figure 3 this is not the case for higher dimensions.

```
   _____            _____            _____
  nx=2, ny=0          nx=1, ny=1          nx=0, ny=2
     n=3                 n=4                 n=5

         _____            _____
        nx=1, ny=0          nx=0, ny=1
           n=1                 n=2

              _____
             nx=0, ny=0
                n=0
```
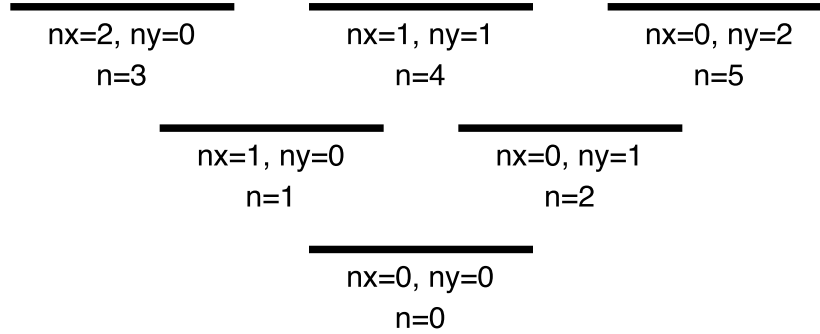
Figure 3: This figure shows the eigenstates for the three lowest energy levels in two dimensions. Each line is one eigenstate and a given eigenstate is on a higher energy level than those below it in the figure. The eigenstates are indexed by n. Each eigenstate has two quantum numbers nx and ny. Unlike in one dimension (Figure 2), when we have two dimensions we can have multiple eigenstates per energy level (e.g. eigenstates $n = 1$ and $n = 2$).

5   Many Particle Theory

6   Hartree-Fock

The Hartree-Fock method is well described in REF basicMB, which this section 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

$$\hat{h}^{HF}\psi_\alpha = \epsilon_\alpha \psi_\alpha, \tag{6.1}$$

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

$$\hat{h}^{HF} = \hat{t} + \hat{u}_{ext} + \hat{u}^{HF}. \tag{6.2}$$

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

$$\langle \hat{H} \rangle = E^{HF} = \langle \Phi_0 | \hat{H} | \Phi_0 \rangle, \tag{6.3}$$

14

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.

$$E^{HF} \geq E_0. \tag{6.4}$$

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 REF SEC Basis Functions).

6.1

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

$$\hat{H} = \hat{H}_0 + \hat{H}_1 = \sum_{i=1}^{N} \hat{h}_0(x_i) + \sum_{i<j}^{N} V(r_{ij}), \tag{6.5}$$

where

$$\hat{H}_0 = \sum_{i=1}^{N} \hat{h}_0(x_i) = \sum_{i=1}^{N} \left(\hat{t}(x_i) + \hat{u}_{\text{ext}}(x_i) \right) \tag{6.6}$$

is the non-interacting part, and

$$\hat{H}_1 = \sum_{i<j}^{N} V(r_{ij}) \tag{6.7}$$

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,

$$\left[\hat{H}, \hat{P} \right] = 0. \tag{6.8}$$

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

$$\hat{P}_{ij} \Psi_\lambda(x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_N) = \beta \Psi_\lambda(x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_N), \tag{6.9}$$

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,

$$\Phi(x_1, x_2, \ldots, x_N, \alpha, \beta, \ldots, \nu) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_\alpha(x_1) & \psi_\alpha(x_2) & \ldots & \ldots & \psi_\alpha(x_N) \\ \psi_\beta(x_1) & \psi_\beta(x_2) & \ldots & \ldots & \psi_\beta(x_N) \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \psi_\nu(x_1) & \psi_\nu(x_2) & \ldots & \ldots & \psi_\nu(x_N) \end{vmatrix}, \tag{6.10}$$

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.

$$\hat{h}_0(x_i) = \hat{t}(x_i) + \hat{u}_{\text{ext}}(x_i), \tag{6.11}$$

and

$$\hat{h}_0(x_i)\psi_\alpha(x_i) = \epsilon_\alpha \psi_\alpha(x_i). \tag{6.12}$$

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

$$\det(\mathbf{A}) = |\mathbf{A}| = \begin{vmatrix} a_{11} & a_{12} & \ldots & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & \ldots & \ldots & a_{nn} \end{vmatrix}, \tag{6.13}$$

which can also be written as

$$|\mathbf{A}| = \sum_{i=1}^{n!} (-1)^{p_i} \hat{P}_i a_{11}, a_{22}, \ldots, a_{nn}, \tag{6.14}$$

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.(6.10) is the trial function for the Hartree-Fock method, and we can now rewrite the determinant as

$$\Phi(x_1, x_2, \ldots, x_N, \alpha, \beta, \ldots, \nu) = \frac{1}{\sqrt{N!}} \sum_p (-1)^p \hat{P} \psi_\alpha(x_1)\psi_\beta(x_2)\ldots\psi_\nu(x_N) = \sqrt{N!}\hat{A}\Phi_H, \tag{6.15}$$

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

$$\hat{A} = \frac{1}{N!} \sum_p (-1)^p \hat{P}, \tag{6.16}$$

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

$$\Phi_H(x_1, x_2, \ldots, x_N, \alpha, \beta, \ldots, \nu) = \psi_\alpha(x_1)\psi_\beta(x_2)\ldots\psi_\nu(x_N). \tag{6.17}$$

6.2 Expectation Value of the Hamiltonian

From the variational principle we know that

$$E_0 \leq E[\Phi] = \int \Phi^* \hat{H} \Phi d\tau, \tag{6.18}$$

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.

$$\int \Phi^* \Phi d\tau = 1. \tag{6.19}$$

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 $\hat{A}$

$$\left[\hat{H}_0, \hat{A}\right] = \left[\hat{H}_1, \hat{A}\right] = 0. \tag{6.20}$$

In addition, $\hat{A}$ satisfies

$$\hat{A}^2 = \hat{A}, \tag{6.21}$$

since every permutation of the Slater determinant reproduces it.

6.2.1 One-body Hamiltonian

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

$$\int \Phi^* \hat{H}_0 \Phi d\tau = N! \int \Phi_H^* \hat{A} \hat{H}_0 \hat{A} \Phi_H d\tau, \tag{6.22}$$

and using Eq.(6.20) and (6.21), we can reduce it to

$$\int \Phi^* \hat{H}_0 \Phi d\tau = N! \int \Phi_H^* \hat{H}_0 \hat{A} \Phi_H d\tau. \tag{6.23}$$

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

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{i=1}^{N} \sum_p (-1)^p \int \Phi_H^* \hat{h}_0 \hat{P} \Phi_H d\tau. \tag{6.24}$$

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

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{i=1}^{N} \int \Phi_H^* \hat{h}_0 \Phi_H d\tau. \tag{6.25}$$

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

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{\mu=1}^{N} \int \psi_\mu^*(\mathbf{r}) \hat{h}_0 \psi_\mu(\mathbf{r}) d\mathbf{r}. \tag{6.26}$$

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

$$\langle \mu | \hat{h}_0 | \mu \rangle = \int \psi_\mu^*(\mathbf{r}) \hat{h}_0 \psi_\mu(\mathbf{r}) d\mathbf{r}, \tag{6.27}$$

and rewrite Eq.(6.26) as

$$\int \Phi^* \hat{H}_0 \Phi d\tau = \sum_{\mu=1}^{N} \langle \mu | \hat{h}_0 | \mu \rangle \tag{6.28}$$

6.2.2 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

$$\int \Phi^* \hat{H}_1 \Phi d\tau = N! \int \Phi_H^* \hat{A} \hat{H}_1 \hat{A} \Phi_H d\tau, \tag{6.29}$$

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

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \sum_{i \leq j=1}^{N} \sum_p (-1)^p \int \Phi_H^* V(r_{ij}) \hat{P} \Phi_H d\tau. \tag{6.30}$$

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

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \sum_{i \leq j=1}^{N} \int \Phi_H^* V(r_{ij})(1 - P_{ij}) \Phi_H d\tau, \tag{6.31}$$

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

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \frac{1}{2} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \left[\int \psi_\mu^*(x_i) \psi_\nu^*(x_j) V(r_{ij}) \psi_\mu(x_i) \psi_\nu(x_j) dx_i dx_j \right.$$
$$\left. - \int \psi_\mu^*(x_i) \psi_\nu^*(x_j) V(r_{ij}) \psi_\nu(x_i) \psi_\mu(x_j) dx_i dx_j \right]. \tag{6.32}$$

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

$$\psi_\alpha(x) = \langle x | \alpha \rangle. \tag{6.33}$$

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

$$\langle \mu\nu | \hat{v} | \mu\nu \rangle = \int \psi_\mu^*(x_i) \psi_\nu^*(x_j) V(r_{ij}) \psi_\mu(x_i) \psi_\nu(x_j) dx_i dx_j, \tag{6.34}$$

and

$$\langle \mu\nu | \hat{v} | \nu\mu \rangle = \int \psi_\mu^*(x_i) \psi_\nu^*(x_j) V(r_{ij}) \psi_\nu(x_i) \psi_\mu(x_j) dx_i dx_j. \tag{6.35}$$

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

$$\langle \mu\nu | \hat{v} | \mu\nu \rangle_{\text{AS}} = \langle \mu\nu | \hat{v} | \mu\nu \rangle - \langle \mu\nu | \hat{v} | \nu\mu \rangle, \tag{6.36}$$

which for a general matrix element is

$$\langle \mu\nu | \hat{v} | \sigma\tau \rangle_{\text{AS}} = \langle \mu\nu | \hat{v} | \sigma\tau \rangle - \langle \mu\nu | \hat{v} | \tau\sigma \rangle. \tag{6.37}$$

This anti-symmetrization element has the symmetry property

$$\langle \mu\nu|\hat{v}|\sigma\tau\rangle_{\mathrm{AS}} = -\langle \mu\nu|\hat{v}|\tau\sigma\rangle_{\mathrm{AS}} = -\langle \nu\mu|\hat{v}|\sigma\tau\rangle_{\mathrm{AS}}, \tag{6.38}$$

and it is hermitian, which implies that

$$\langle \mu\nu|\hat{v}|\sigma\tau\rangle_{\mathrm{AS}} = \langle \sigma\tau|\hat{v}|\mu\nu\rangle_{\mathrm{AS}}. \tag{6.39}$$

We can now rewrite Eq.(6.32) as

$$\int \Phi^* \hat{H}_1 \Phi d\tau = \frac{1}{2} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \langle \mu\nu|\hat{v}|\mu\nu\rangle_{\mathrm{AS}}. \tag{6.40}$$

By combining Eq.(6.28) and (6.40) we end up with the energy functional

$$E\left[\Phi\right] = \sum_{\mu=1}^{N} \langle \mu|\hat{h}_0|\mu\rangle + \frac{1}{2} \sum_{\mu=1}^{N} \sum_{\nu=1}^{N} \langle \mu\nu|\hat{v}|\mu\nu\rangle_{\mathrm{AS}}. \tag{6.41}$$

6.3 Polar Coordinates

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

$$\langle \mu\nu|\hat{v}|\sigma\tau\rangle = \int \psi_\mu^*(x_i)\psi_\nu^*(x_j)V(r_{ij})\psi_\sigma(x_i)\psi_\tau(x_j)dx_i dx_j, \tag{6.42}$$

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

$$x = r\cos\theta \tag{6.43}$$

$$y = r\sin\theta \tag{6.44}$$

$$r = \sqrt{x^2 + y^2}, \tag{6.45}$$

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

$$\psi(r,\theta) = R(r)Y(\theta). \tag{6.46}$$

The normalized solution for the angular part in two dimensions is

$$Y(\theta) = \frac{1}{\sqrt{2\pi}} e^{im\theta}. \tag{6.47}$$

Since the total wave function must satisfy the physical condition

$$\psi(r,\theta) = \psi(r,\theta+2\pi), \tag{6.48}$$

the quantum number m is restricted to integral values

$$m = 0, \pm 1, \pm 2, \ldots \tag{6.49}$$

The solution for the radial part is

$$r_{nm}(r) = \sqrt{\frac{2n!}{(n+|m|)!}} \beta^{\frac{1}{2}(|m|+1)} r^{|m|} e^{-\frac{1}{2}\beta r^2} L_n^{|m|}(\beta r^2), \tag{6.50}$$

19

where n is the principal quantum number

$$n = 0, 1, 2, 3, \ldots \tag{6.51}$$

and m is the angular momentum quantum number given in Eq.(6.49). β is defined as

$$\beta = \frac{m_e \omega}{\hbar}, \tag{6.52}$$

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

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2} \right) L(x) = 0, \tag{6.53}$$

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

$$L_0(x) = 1, \tag{6.54}$$

$$L_1(x) = 1 - x, \tag{6.55}$$

$$L_2(x) = 2 - 4x + x^2, \tag{6.56}$$

$$L_3(x) = 6 - 18x + 9x^2 - x^3, \tag{6.57}$$

and

$$L_4(x) = 24 - 96x + 72x^2 - 16x^3 + x^4. \tag{6.58}$$

The Laguerre polynomials fullfill the orthogonality relation

$$\int_0^\infty e^{-x} L_n(x)^2 dx = 1, \tag{6.59}$$

and the recursion relation

$$(n+1)L_{n+1}(x) = (2n + 1 - x)L_n(x) - nL_{n-1}(x). \tag{6.60}$$

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

$$\psi(r, \theta) = \sqrt{\frac{n!}{\pi(n + |m|)!}} \beta^{\frac{1}{2}(|m|+1)} r^{|m|} e^{-\frac{1}{2}\beta r^2} L_n^{|m|}(\beta r^2) e^{im\theta}, \tag{6.61}$$

with eigenvalue

$$E = \hbar\omega(2n + |m| + 1), \tag{6.62}$$

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

$$E = \hbar\omega(n_x + n_y + 1). \tag{6.63}$$

7   Systems

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

$$H = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + V_c(\mathbf{r}_i) \right) + \sum_{i<j} \frac{1}{r_{ij}}, \tag{7.1}$$

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

$$H_0 = \sum_{i=1}^{N} \left(-\frac{1}{2}\nabla_i^2 + V_c(\mathbf{r}_i) \right), \tag{7.2}$$

while the repulsive Coulomb interaction between two electrons is given by

$$H_1 = \sum_{i<j} \frac{1}{r_{ij}}, \tag{7.3}$$

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 $r_i = \sqrt{r_{i_x}^2 + r_{i_y}^2}$.

7.1   Potentials

7.1.1   Standard Harmonic Oscillator Well

For the standard harmonic oscillator well, the confinement potential is simply

$$\begin{aligned} V_c(\mathbf{r}) &= \frac{1}{2}m_e\omega^2\mathbf{r}^2 \\ &= \frac{1}{2}\omega^2\mathbf{r}^2, \end{aligned} \tag{7.4}$$

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 4.

7.1.2   Double Harmonic Oscillator Well

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

$$V_c(\mathbf{r}) = \frac{1}{2}m_e\omega^2\min\left[\sum_{j}^{M}(\mathbf{r} - \mathbf{L}_j)^2\right], \tag{7.5}$$

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 $\mathbf{L}_1 = (0, 0)$
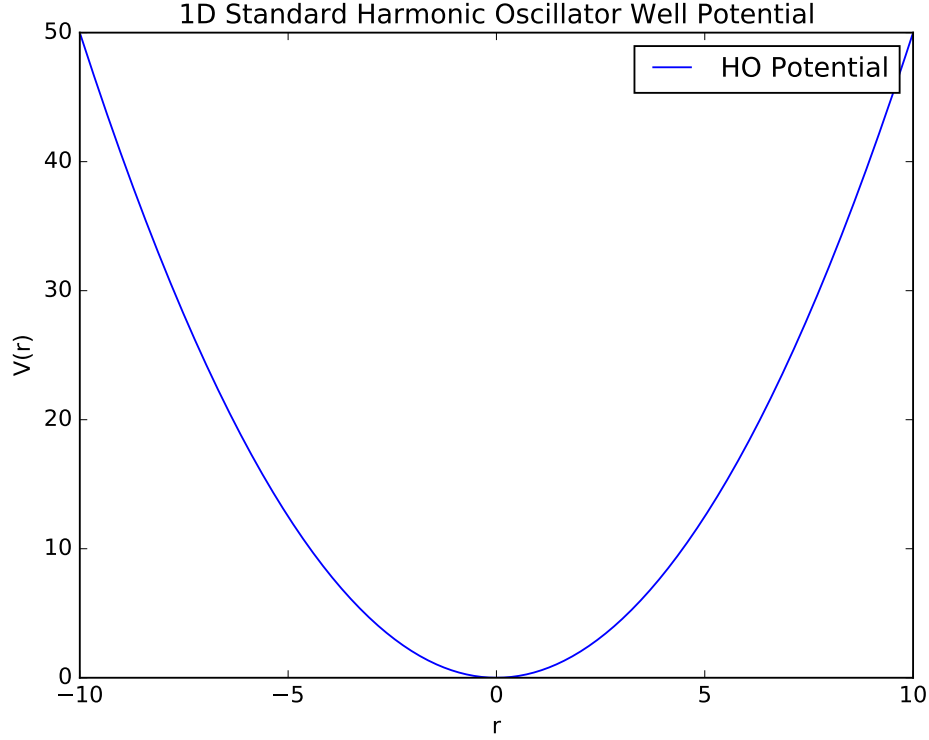
Figure 4: Standard Harmonic Oscillator Well Potential as described by Eq.(7.4), with $\omega = 1$.

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

$$
\begin{aligned}
V_c(x, y) &= \frac{1}{2} m_e \omega^2 [\mathbf{r}^2 - 2L_x|x| - 2L_y|y| + L_x^2 + L_y^2] \\
&= \frac{1}{2} m_e \omega^2 [\mathbf{r}^2 - 2L_x|x| + L_x^2].
\end{aligned}
\tag{7.6}
$$

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. As explained by Ref. [](Wikipedia: Logic gate), 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 shows an example of a double-well potential, where $L_x = 5$ and $\omega = 1$.

7.1.3 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
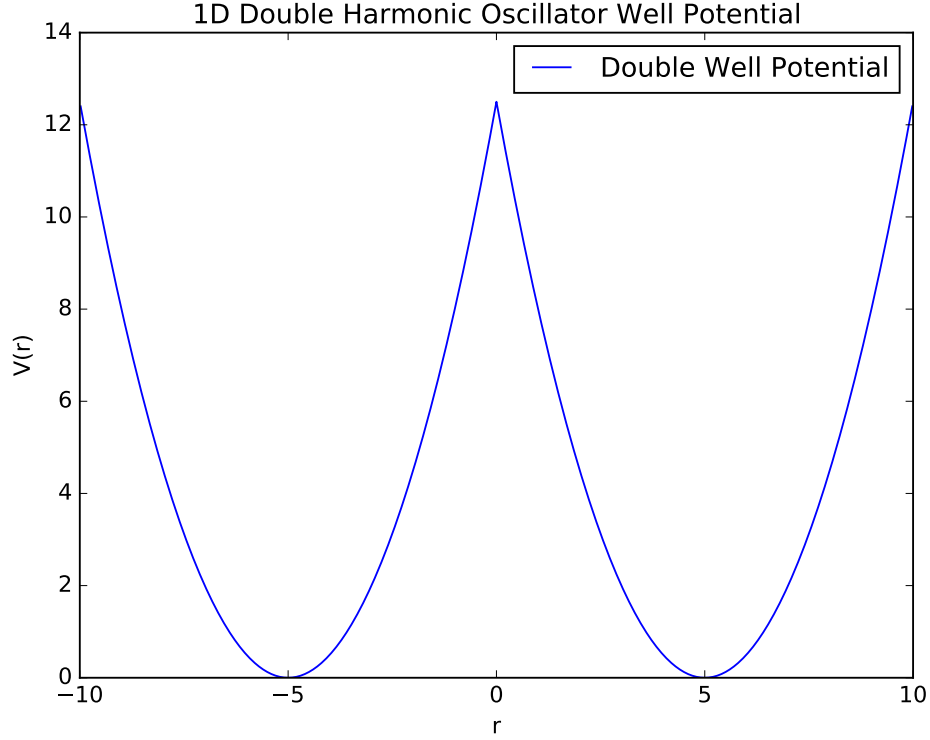
22

Figure 5: Double Harmonic Oscillator Well Potential as described by Eq.(7.6), with $L_x = 5$, $L_y = 0$, and $\omega = 1$.

well, the potential in one dimension can be written as

$$V_c(x) = \begin{cases} C_1, & \text{if } -L < x < L \\ C_2, & \text{otherwise} \end{cases}, \tag{7.7}$$

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

$$V_c(x) = \begin{cases} 0, & \text{if } -L < x < L \\ V_0, & \text{otherwise} \end{cases}. \tag{7.8}$$

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 6 shows a finite square well potential with $V_0 = 1$ and $L = 3$.

7.2 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

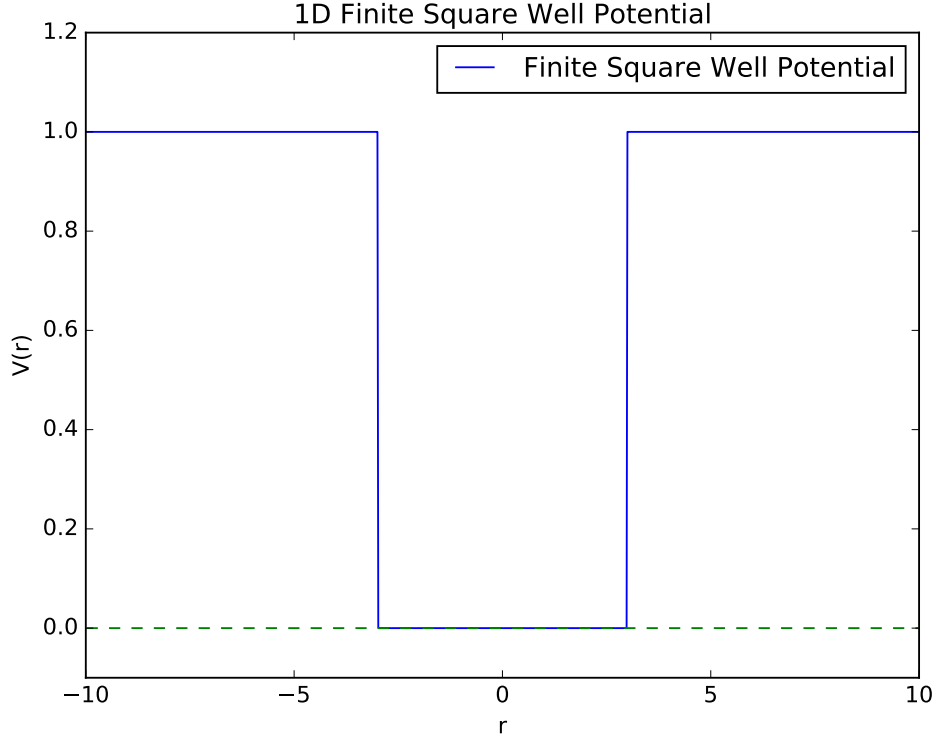$$\epsilon_n = \omega(n_x + n_y + 1). \tag{7.9}$$

Figure 6: Finite Square Well Potential as described by Eq.(7.8), with $L = 3$ and $V_0 = 1$.

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

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}) = C \exp\left(-\alpha\omega(r_1^2 + r_2^2)/2\right) \exp\left(\frac{ar_{12}}{(1 + \beta r_{12})}\right), \tag{7.10}$$

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.

7.3   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

$$\psi_T(\mathbf{r_1}, \mathbf{r_2}, \ldots, \mathbf{r_N}) = Det\left(\phi_1(\mathbf{r_1}), \phi_2(\mathbf{r_2}), \ldots, \phi_N(\mathbf{r_N})\right) \prod_{i<j}^{N} \exp\left(\frac{ar_{ij}}{(1+\beta r_{ij})}\right), \tag{7.11}$$

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

$$\phi_{n_x,n_y}(x, y) = AH_{n_x}(\sqrt{\omega}x)H_{n_y}(\sqrt{\omega}y)\exp\left(-\omega(x^2+y^2)/2\right). \tag{7.12}$$

The functions $H_{n_x}(\sqrt{\omega}x)$ are Hermite polynomials, 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.

7.4   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

$$E_L(\mathbf{R}) = \frac{1}{\Psi_T(\mathbf{R})}H\Psi_T(\mathbf{R}). \tag{7.13}$$

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

$$F = \frac{2\nabla\Psi_T}{\Psi_T}. \tag{7.14}$$

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.

7.5   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. We have that the energy for a given electron is given by Eq.(7.9). Table 1 contains the total energy of unperturbed closed shell systems up to $N = 20$ electrons.

N	E
2	2ω
6	10ω
12	28ω
20	60ω

Table 1: Benchmarks of the energy E for unperturbed closed shell systems in two dimensions with N electrons.

N	E
2	3.000
6	20.1597
12	65.700
20	155.868

Table 2: Benchmarks of the energy E for perturbed closed shell systems in two dimensions with N electrons and $\omega = 1$.

For the perturbed case we use the benchmarks given in table 2 provided by Ref. [3]. From Ref. [2] we also have an analytical result for the ground state energy, $E = 3$, for the two-body case with $\omega = 1$.

Part II

Implementation

8   Program Structure

8.1   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.

8.1.1 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.

```cpp
void RandomUniform::setupInitialState() {

    long idum = -1-m_my_rank;
    Random::setSeed(idum);

    // Create random positions for all particles:
    for (int i=0; i < m_numberOfParticles; i++) {
        std::vector<double> position = std::vector<double>();

        for (int j=0; j < m_numberOfDimensions; j++) {
            double pos = pow(-1, i%2)*Random::nextDouble();
            position.push_back(pos);
        }
        // Create particles:
        m_particles.push_back(new Particle());
        m_particles.at(i)->setNumberOfDimensions(m_numberOfDimensions);
        m_particles.at(i)->setPosition(position);
    }
}
```

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.

8.1.2 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.

```cpp
bool System::metropolisStep(int currentParticle) {
    /* Perform the actual Metropolis step: Take the current particle and
     * change it's position by a random amount, and check if the step is
     * accepted by the Metropolis test (compare the wave function evaluated
     * at this new position with the one at the old position).
     */

    // Change position of current particle by a random amount creating a trial
         state
    setCurrentParticle(currentParticle);
    std::vector<double> positionChange(m_numberOfDimensions);

    for (int i=0; i<m_numberOfDimensions; i++){
        positionChange[i] = (Random::nextDouble()*2-1)*m_stepLength;
    }

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

    // Check if trial state is accepted
    if (Random::nextDouble() <= qratio){
        m_waveFunction->updateSlaterDet(currentParticle);
        return true;
    }

    for (int i=0; i<m_numberOfDimensions; i++){
        // If trial state is not accepted, revert to old position for chosen
             particle (revert to old state)
        m_particles[currentParticle]->adjustPosition(-positionChange[i], i);
        m_waveFunction->updateDistances(currentParticle);
        m_waveFunction->updateSPWFMat(currentParticle);
        m_waveFunction->updateJastrow(currentParticle);
    }

    return false;
}
```

8.1.3 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.

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

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

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

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

Listing 2: Potential energy calculation for the regular harmonic oscillator well sub class of the Hamiltonian class.

```cpp
double potentialEnergy = 0;
    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 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 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 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 4, then the code in Listing 2 will be used, while if we chose line 2, then the code in Listing 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 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.

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: 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.

8.1.4 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.

```cpp
 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 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 2 and Listing 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 iteratively. There are advantages to each of these implementations. The iterative 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 iterative 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 iterative method is used, however it would be best to switch to the last method described in order to substantially reduce computation time.

```
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 8: Iterative 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.

8.1.5 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.

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

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

        for (int i=0; i < m_halfNumberOfParticles; i++) {
            for (int j=0; j < m_halfNumberOfParticles; j++) {
                spinUpSlater(i,j) = m_SPWFMat(i,j);
                spinDownSlater(i,j) = m_SPWFMat(i+m_halfNumberOfParticles,j);
            }
        }
    }

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

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

    return waveFunction;
}
```

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.35). 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. (2.18) we know that the expression for the Slater part of the ratio is

$$R_{SD} = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}}) d_{ji}^{-1}(\mathbf{r}^{\text{old}}), \tag{8.1}$$

where ϕ are the single particle wave functions, and d_{ji}^{-1} is element ji of the inverse Slater matrix. From Eq. (2.22) we know the expression for the Jastrow part

$$R_C = \exp\left(\sum_{i=1, i \neq k}^{N} f_{ik}^{\text{new}} - f_{ik}^{\text{old}} \right), \tag{8.2}$$

where N is the number of particles and

$$f_{ij} = \frac{a r_{ij}}{(1 + \beta r_{ij})}. \tag{8.3}$$

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.

```cpp
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 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. (4.19), and use the resulting single particle wave functions to fill the Slater matrices. We restate the equation here for convenience

$$\psi_{n'}(x) = \sum_{n_x=0}^{\Lambda} C_{n',n_x} \phi_{n_x}(x). \tag{8.4}$$

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 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.

```cpp
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 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 eigenstates 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.

```
1  int i = currentParticle;
2  if (i < m_halfNumberOfParticles) {
3      mat spinUpSlaterInverseOld = m_spinUpSlaterInverse;
4      for (int j=0; j < i; j++) {
5          double sum = 0;
6
7          for (int l=0; l <m_halfNumberOfParticles; l++) {
8              sum += m_SPWFMat(i,l)
9                      *spinUpSlaterInverseOld(l,j);
10         }
11         for (int k=0; k < m_halfNumberOfParticles; k++) {
12             m_spinUpSlaterInverse(k,j) = spinUpSlaterInverseOld(k,j)
13                                     -(sum/m_ratioSlaterDet)*
                                            spinUpSlaterInverseOld(k,i);
14         }
15     }
16     for (int j=i+1; j < m_halfNumberOfParticles; j++) {
17         double sum = 0;
18
19         for (int l=0; l <m_halfNumberOfParticles; l++) {
20             sum += m_SPWFMat(i,l)
21                     *spinUpSlaterInverseOld(l,j);
22         }
23         for (int k=0; k < m_halfNumberOfParticles; k++) {
24             m_spinUpSlaterInverse(k,j) = spinUpSlaterInverseOld(k,j)
25                                     -(sum/m_ratioSlaterDet)*
                                            spinUpSlaterInverseOld(k,i);
26         }
27     }
28     for (int k=0; k < m_halfNumberOfParticles; k++) {
29         m_spinUpSlaterInverse(k,i) = spinUpSlaterInverseOld(k,i)/
                m_ratioSlaterDet;
30     }
31 }
```

Listing 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 $j > i$.

8.1.6 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 2.4. 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. (2.36). 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.

```cpp
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++;
```

```
43          std::string upLine = "\e[A";
44
45          if (my_rank == 0) {
46              cout << "Iterations: " << iteration << endl;
47              for (int i=0; i < numberOfParameters; i++) {
48                  cout << "Parameter " << i+1 << ": " << parameters[i] << endl;
49                  upLine += "\e[A";
50              }
51              cout << upLine;                    //"\033[F";
52          }
53
54
55  }while(diff > tol && iteration < maxIterations);
56  // Loop ends when requested tolerance for optimal parameters has been reached
         or after max iterations.
```

8.1.7   Testing the Code

Non-interacting Case

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. From Table 3 and 4 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	Eigenstates
1	1	1ω	1	1
2	2	2ω	1	1
6	10	10ω	2	3
12	28	28ω	3	6

Table 3: 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 eigenstates column shows the number of eigenstates 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 eigenstates fill up. For each spin up and spin down particle pair we need one eigenstate, otherwise the Slater determinant becomes singular. So for 1 or 2 particles we only need 1 eigenstate, while for 6 particles we need 3 eigenstates. In general the number of eigenstates has to be greater or equal to half of the total amount of particles.

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

Table 4: 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 (3), we only need the minimum amount of basis functions to get benchmark consistent results. The eigenstates column shows the number of eigenstates 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 eigenstates fill up. For each spin up and spin down particle pair we need one eigenstate, otherwise the Slater determinant becomes singular. So for 1 or 2 particles we only need 1 eigenstate, while for 8 particles we need 4 eigenstates. In general the number of eigenstates has to be greater or equal to half of the total amount of particles.

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" over the potential barrier. The benchmarks in Table 5 and 7 assume that the wells can be treated as separate with no "spilling" between them and with the same amount of particles in each well.

From Table 5 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.

In the three dimensional case we see from Table 7 that the same problem occurs, but here it

N	E (VMC)	E (Benchmark)	Energy Levels	Eigenstates
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 5: 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 eigenstates column shows the number of eigenstates 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 eigenstates fill up.

N	E (VMC)	E (Benchmark)	Energy Levels	Eigenstates
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: More results for the same system as in Table 5. Here we check if the issue for 12 particles in Table 5 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.

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.

–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 8. 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 $(0, 0, 1)$.

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

N	E (VMC)	E (Benchmark)	Energy Levels	Eigenstates
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 7: 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 eigenstates column shows the number of eigenstates 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 eigenstates fill up.

#	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 8: 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.

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 9 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 10. 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 11). 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

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 9: 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.

computation times.

N (steps)	E (VMC)	E (Benchmark)	Energy Levels	Eigenstates
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 10: Here we see how the results for two particles in a finite square well potential depends on the number of basis functions (eigenstates) 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.

In Table 12 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, but unlike the in the double well case, we here get good results regardless of the number of particles, provided we use enough basis functions. 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

N (steps)	E (VMC)	E (Benchmark)	Energy Levels	Eigenstates	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 11: 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.

exact solution than the benchmarks themselves. The important thing to note is that the results are reasonably consistent with the benchmarks for all cases, which is a good indicator that the program is working correctly.

N (particles)	E (VMC)	E (Benchmark)	Energy Levels	Eigenstates
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 12: 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 eigenstates column shows the number of eigenstates 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 eigenstates 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 (eigenstates) 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.

From Table 13 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 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.

N (particles)	steps	E (VMC)	E (Benchmark)	Energy Levels	Eigenstates
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 13: 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 eigenstates column shows the number of eigenstates 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 eigenstates 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 (eigenstates) 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.

8.2 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 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.

8.2.1 Diagonalizing

The first step is to set up the tridiagonal matrix. See section 4.1 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 }
```

8.2.2   Finding the Overlap Coefficients

We know from section 4.2 that the overlap coefficients are given by Eq. (4.12), which we restate here for convenience.

$$C_{n',n} = \langle \psi_{n'} | \phi_n \rangle = \sum_{i=0}^{N-1} \psi_{n'}(x_i)\phi_n(x_i) \qquad n', n = 0, 1, 2, \ldots, \qquad (8.5)$$

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. (8.5), 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.

8.2.3 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. (4.19) from section 4.3, which is

$$\psi_{n'}(x) = \sum_{n_x=0}^{\Lambda} C_{n',n_x} \phi_{n_x}(x). \tag{8.6}$$

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.

8.2.4 Testing the Code

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:

$$||\psi_{n'}^{\text{diag}} - \psi_{n'}^{\text{exp}}||_2 = \sqrt{\sum_i^N |\psi_{n'}^{\text{diag}}(x_i) - \psi_{n'}^{\text{exp}}(x_i)|^2} < \epsilon. \qquad (8.7)$$

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. (8.6). The L_2 being 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'.

Another test we can do is to check that the norm of ψ^{diag} 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

$$||\psi|| = \sqrt{\langle \psi | \psi \rangle}, \qquad (8.8)$$

and we want the following to be true

$$||\psi_x^{\text{exp}}|| \approx ||\psi_y^{\text{exp}}|| \approx ||\psi_z^{\text{exp}}|| \approx 1, \qquad (8.9)$$

in the three dimensional case.

8.3 Hartree-Fock

8.3.1 Testing the Code

9 Optimizing Performance

We use parallelization of code and specific compiler options to optimize the performance of our program. To parallelize our program we use MPI, and we also optimize with the $O3$ compiler flag of the GNU C++ compiler. The $O3$ flag tells the compiler to make optimizations to the code where it is possible, in order to reduce computation time. By parallelizing our program, we make it possible to run the program on multiple processors at once. We can run the program on several processors on a single computer (typically 4 or 8 processors) to get some speedup. However, we can also run it on a super computer cluster in order to utilize even more processors (we'll use up to 64 processors in this project) and decrease computation time substantially. If the program is parallelized well we should see a speed-up of about a factor 2 when doubling the amount of processors. In general, Variational Monte Carlo simulations are simple to parallelize, so we should expect to achieve that amount of speed-up when running the code in parallel. 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.

Part III
Results

10

10.1 Wave Function Expanded in Harmonic Oscillator Basis

N	E (Shared start)	E (Split start)
2	2.00002	2.00002
4	4.00005	4.00005
6	8.00009	8.0001

Table 14: Unperturbed energies for the double well when approximating the single particle wave functions with harmonic oscillator basis functions. Results in the second column are for when all the particles starts in the same well and are then distributed between the wells by the program, while the results in the third column are for when we force half of the particles in each well at the start. Two dimensional system, $L_x = 4$, $\omega = 1$.

N	E (Shared start)	E (Split start)
2	3.00854	2.34952
4	6.53324	6.53747
6	14.1804	14.2108

Table 15: Perturbed energies for the double well when approximating the single particle wave functions with harmonic oscillator basis functions. Results in the second column are for when all the particles starts in the same well and are then distributed between the wells by the program, while the results in the third column are for when we force half of the particles in each well at the start. Two dimensional system, $L_x = 4$, $\omega = 1$.

10.2 Wave Function as Super Postion of two Harmonic Oscillator Functions

N	E (Shared start)	E (Split start)
2	2	2
4	4	4
6	8	8

Table 16: Unperturbed energies for the double well when approximating the single particle wave functions with a super position of two harmonic oscillator functions. Results in the second column are for when all the particles starts in the same well and are then distributed between the wells by the program, while the results in the third column are for when we force half of the particles in each well at the start. Two dimensional system, $L_x = 4$, $\omega = 1$.

N	E (Shared start)	E (Split start)
2	3.00852	2.3495
4	6.53322	6.53742
6	14.4748	13.872

Table 17: Perturbed energies for the double well when approximating the single particle wave functions with a super position of two harmonic oscillator functions. Results in the second column are for when all the particles starts in the same well and are then distributed between the wells by the program, while the results in the third column are for when we force half of the particles in each well at the start. Two dimensional system, $L_x = 4$, $\omega = 1$.

11 Conclusion

Hello world!

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

12 Appendix

Appendices

URL to github folder with source files etc.
`https://github.com/christianfleischer/Quantum-Dot-Project`

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

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

We define

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

$$v = \frac{ar_{12}}{(1 + \beta r_{12})} \tag{A.3}$$

Then we have

$$\nabla_1 \exp(u) = -\alpha\omega\mathbf{r_1}\exp(u). \tag{A.4}$$

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

$$\begin{aligned}
\frac{\partial}{\partial x_1}\exp(v(r_{12})) &= \frac{\partial r_{12}}{\partial x_1}\frac{\partial}{\partial r_{12}}\exp(v(r_{12})) \\
&= \frac{(x_1 - x_2)}{r_{12}}\frac{\partial}{\partial r_{12}}\exp(v(r_{12})) \\
&= \frac{(x_1 - x_2)}{r_{12}}\frac{a}{(1 + \beta r_{12})^2}\exp(v(r_{12})),
\end{aligned} \tag{A.5}$$

which gives

$$\nabla_1 \exp(v) = \frac{(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} \exp(v). \tag{A.6}$$

For the second particle we have

$$\nabla_2 \exp(u) = -\alpha \omega \mathbf{r}_2 \exp(u) \tag{A.7}$$
$$\nabla_2 \exp(v) = -\nabla_1 \exp(v). \tag{A.8}$$

The gradients for the full wave function are then

$$\nabla_1 \psi_T = \left(-\alpha \omega \mathbf{r}_1 + \frac{(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} \right) C \exp(u) \exp(v) \tag{A.9}$$

$$\nabla_2 \psi_T = \left(-\alpha \omega \mathbf{r}_2 - \frac{(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} \right) C \exp(u) \exp(v). \tag{A.10}$$

To find the Laplacian we need

$$\nabla_1(-\alpha \omega \mathbf{r}_1) = -\alpha \omega d = -2\alpha \omega, \tag{A.11}$$

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

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

We also have

$$\nabla_2(-\alpha \omega \mathbf{r}_2) = \nabla_1(-\alpha \omega \mathbf{r}_1), \tag{A.13}$$

and

$$\nabla_2 \left(-\frac{(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} \right) = \nabla_1 \left(\frac{(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} \right) \tag{A.14}$$

We end up with the Laplacians

$$\frac{\nabla_1^2 \psi_T}{\psi_T} = \left(-2\alpha \omega + \frac{1}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} - \frac{2a\beta}{(1 + \beta r_{12})^3} + \left(-\alpha \omega \mathbf{r}_1 + \frac{(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} \right)^2 \right) \tag{A.15}$$

$$\frac{\nabla_2^2 \psi_T}{\psi_T} = \left(-2\alpha \omega + \frac{1}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} - \frac{2a\beta}{(1 + \beta r_{12})^3} + \left(-\alpha \omega \mathbf{r}_2 - \frac{(\mathbf{r}_1 - \mathbf{r}_2)}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} \right)^2 \right). \tag{A.16}$$

The sum of the Laplacians in the Hamiltonian is then

$$
\begin{aligned}
\frac{\nabla_1^2 \psi_T}{\psi_T} + \frac{\nabla_2^2 \psi_T}{\psi_T} = &-4\alpha \omega + \frac{1}{r_{12}} \frac{a}{(1 + \beta r_{12})^2} - \frac{2a\beta}{(1 + \beta r_1 2)^3} \\
&+ \alpha^2 \omega^2 (r_1^2 + r_2^2) + \frac{2a^2}{(1 + \beta r_{12})^4} - \alpha \omega r_{12} \frac{2a}{(1 + \beta r_{12})^2}.
\end{aligned} \tag{A.17}
$$

The complete expression for the local energy is then

$$
\begin{aligned}
E_L =& \frac{1}{\psi_T} H \psi_T \\
=& 2\alpha\omega + \frac{1}{r_{12}} \frac{a}{(1+\beta r_{12})^2} - \frac{2a\beta}{(1+\beta r_1 2)^3} - \frac{1}{2}\alpha^2\omega^2(r_1^2 + r_2^2) \\
& - \frac{a^2}{(1+\beta r_{12})^4} + \alpha\omega r_{12}\frac{a}{(1+\beta r_{12})^2} + \frac{1}{2}\omega^2(r_1^2 + r_2^2) + \frac{1}{r_{12}} \\
=& 2\alpha\omega + \frac{1}{2}\omega^2(r_1^2 + r_2^2)(1-\alpha^2) - \frac{2a\beta}{(1+\beta r_1 2)^3} \\
& - \frac{a^2}{(1+\beta r_{12})^4} + \left(\alpha\omega r_{12} + \frac{1}{r_{12}}\right)\frac{a}{(1+\beta r_{12})^2} + \frac{1}{r_{12}}.
\end{aligned}
\tag{A.18}
$$

A.2 Many-body quantum dots

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

$$
\psi_T(\mathbf{r_1}, \mathbf{r_2}, \ldots, \mathbf{r_N}) = Det\left(\phi_1(\mathbf{r_1}), \phi_2(\mathbf{r_2}), \ldots, \phi_N(\mathbf{r_N})\right) \prod_{i<j} \exp\left(\frac{a r_{ij}}{(1+\beta r_{ij})}\right),
\tag{A.19}
$$

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

$$
\phi_{n_x,n_y}(x,y) = A H_{n_x}(\sqrt{\omega}x) H_{n_y}(\sqrt{\omega}y) \exp\left(-\omega(x^2+y^2)/2\right.
\tag{A.20}
$$

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

$$
d_{ij} = \phi_j(x_i),
\tag{A.21}
$$

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

$$
d_{ij}^{-1} = \frac{C_{ji}}{|\hat{D}|},
\tag{A.22}
$$

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

$$
\sum_{k=1}^{N} d_{ik}d_{kj}^{-1} = \delta_{ij}.
\tag{A.23}
$$

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

$$R \equiv \frac{|\hat{D}(\mathbf{r}^{\text{new}})|}{|\hat{D}(\mathbf{r}^{\text{old}})|} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})C_{ij}(\mathbf{r}^{\text{new}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})C_{ij}(\mathbf{r}^{\text{old}})}. \tag{A.24}$$

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

$$C_{ij}(\mathbf{r}^{\text{new}}) = C_{ij}(\mathbf{r}^{\text{old}}) \quad \forall\, j \in \{1, \ldots, N\}. \tag{A.25}$$

We use eq. (A.22) and eq. (A.25) with eq. (A.24) in order to obtain

$$R = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})C_{ij}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})C_{ij}(\mathbf{r}^{\text{old}})} = \frac{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})d_{ji}^{-1}(\mathbf{r}^{\text{old}})}{\sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{old}})d_{ji}^{-1}(\mathbf{r}^{\text{old}})}, \tag{A.26}$$

where, by eq. (A.23), the denominator of the rightmost expression is unity, and we end up with

$$R = \sum_{j=1}^{N} d_{ij}(\mathbf{r}^{\text{new}})d_{ji}^{-1}(\mathbf{r}^{\text{old}}) = \sum_{j=1}^{N} \phi_j(\mathbf{r}_i^{\text{new}})d_{ji}^{-1}(\mathbf{r}^{\text{old}}). \tag{A.27}$$

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

$$S_j = (\hat{D}(\mathbf{r}^{\text{new}}) \times \hat{D}^{-1}(\mathbf{r}^{\text{old}}))_{ij} = \sum_{l=1}^{N} d_{il}(\mathbf{r}^{\text{new}})d_{lj}^{-1}(\mathbf{r}^{\text{old}}), \tag{A.28}$$

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

$$d_{kj}^{-1}(\mathbf{r}^{\text{new}}) = d_{kj}^{-1}(\mathbf{r}^{\text{old}}) - \frac{S_j}{R}d_{ki}^{-1}(\mathbf{r}^{\text{old}}) \quad \begin{array}{c} \forall\; k \in \{1, \ldots, N\} \\ j \neq i \end{array}. \tag{A.29}$$

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

$$d_{ki}^{-1}(\mathbf{r}^{\text{new}}) = \frac{1}{R}d_{ki}^{-1}(\mathbf{r}^{\text{old}}) \quad \forall\; k \in \{1, \ldots, N\}. \tag{A.30}$$

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

$$\frac{\vec{\nabla}_i |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \vec{\nabla}_i d_{ij}(\mathbf{r})d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \vec{\nabla}_i \phi_j(\mathbf{r}_i)d_{ji}^{-1}(\mathbf{r}) \tag{A.31}$$

and

$$\frac{\nabla_i^2 |\hat{D}(\mathbf{r})|}{|\hat{D}(\mathbf{r})|} = \sum_{j=1}^{N} \nabla_i^2 d_{ij}(\mathbf{r}) d_{ji}^{-1}(\mathbf{r}) = \sum_{j=1}^{N} \nabla_i^2 \phi_j(\mathbf{r}_i) \, d_{ji}^{-1}(\mathbf{r}). \tag{A.32}$$

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

$$\langle \hat{K}_i \rangle = -\frac{1}{2} \frac{\langle \Psi | \nabla_i^2 | \Psi \rangle}{\langle \Psi | \Psi \rangle}, \tag{A.33}$$

and we have that

$$K_i = -\frac{1}{2} \frac{\nabla_i^2 \Psi}{\Psi}. \tag{A.34}$$

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

$$\begin{aligned}
\frac{\nabla^2 \Psi}{\Psi} &= \frac{\nabla^2 (\Psi_D \, \Psi_C)}{\Psi_D \, \Psi_C} = \frac{\nabla \cdot [\nabla (\Psi_D \, \Psi_C)]}{\Psi_D \, \Psi_C} = \frac{\nabla \cdot [\Psi_C \nabla \Psi_D + \Psi_D \nabla \Psi_C]}{\Psi_D \, \Psi_C} \\
&= \frac{\nabla \Psi_C \cdot \nabla \Psi_D + \Psi_C \nabla^2 \Psi_D + \nabla \Psi_D \cdot \nabla \Psi_C + \Psi_D \nabla^2 \Psi_C}{\Psi_D \, \Psi_C} \\
&= \frac{\nabla^2 \Psi_D}{\Psi_D} + \frac{\nabla^2 \Psi_C}{\Psi_C} + 2 \frac{\nabla \Psi_D}{\Psi_D} \cdot \frac{\nabla \Psi_C}{\Psi_C}.
\end{aligned} \tag{A.35}$$

From eq. (A.35) 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.31) and eq. (A.32). We have that

$$\Psi_C = \prod_{i<j} \exp f(r_{ij}) = \exp \left\{ \sum_{i<j} \frac{a r_{ij}}{1 + \beta r_{ij}} \right\}, \tag{A.36}$$

and by differentiating with the chain rule similarly to what we did in eq. (A.5), we find that the gradient is

$$\frac{\nabla_k \Psi_C}{\Psi_C} = \sum_{j \neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{\partial f(r_{kj})}{\partial r_{kj}} = \sum_{j \neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} f'(r_{kj}) = \sum_{j \neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} \frac{a}{(1 + \beta r_{kj})^2}, \tag{A.37}$$

where

$$f'(r_{kj}) = \frac{\partial}{\partial r_{kj}} f(r_{kj}). \tag{A.38}$$

To find the Laplacian we need to calculate

$$\frac{\nabla_k^2 \Psi_C}{\Psi_C} = \frac{1}{\Psi_C} \nabla_k \sum_{j \neq k} \frac{\mathbf{r}_{kj}}{r_{kj}} f'(r_{kj}) \Psi_C. \tag{A.39}$$

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

$$\frac{\nabla_k^2 \Psi_C}{\Psi_C} = \frac{1}{\Psi_C} \sum_{j \neq k} \left(\frac{\mathbf{r}_{kj}}{r_{kj}} f'(r_{kj}) \nabla_k \Psi_C + \frac{\mathbf{r}_{kj}}{r_{kj}} \Psi_C \nabla_k f'(r_{kj}) + f'(r_{kj}) \Psi_C \nabla_k \frac{\mathbf{r}_{kj}}{r_{kj}} \right) \quad \text{(A.40)}$$

$$= \sum_{ij \neq k} \frac{(\mathbf{r}_k - \mathbf{r}_i)(\mathbf{r}_k - \mathbf{r}_j)}{r_{ki} r_{kj}} f'(r_{ki}) f'(r_{kj}) + \sum_{j \neq k} \left(f''(r_{kj}) + \frac{d-1}{r_{kj}} f'(r_{kj}) \right), \quad \text{(A.41)}$$

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

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

$$\text{(A.42)}$$

B   Program Structure

The methods were implemented with a C++ program using object-orientation. The program consists of several classes responsible for different parts of the simulations. The classes are:

- **Hamiltonian:** A super-class for different Hamiltonians. The subclasses calculate the local energy for their specific Hamiltonian. 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:

  - **HarmonicOscillator:** Calculates the local energy in the non-interacting case. [Project1]
  - **HarmonicOscillatorRepulsive:** Calculates the local energy in the interacting case. [Project1]
  - **HarmonicOscillatorElectrons:** Calculates the local energy for quantum dots. [Project2]

- **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 parameter(s) α (and β) using the analytical expressions. The subclasses are:

  - **SimpleGaussian:** WaveFunction subclass for the non-interacting case. [Project1]
  - **RepulsiveGaussian:** WaveFunction subclass for the interacting case. [Project1]
  - **TwoElectrons:** WaveFunction subclass for the two-body quantum dot case. [Project2]
  - **ManyElectrons:** WaveFunction subclass for the many-body quantum dot case. This includes all Slater determinant functionality. [Project 2]

- **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.

- **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 more wave functions, Hamiltonians and initial states, or alternative methods for optimizing variational parameters (e.g. the Conjugate Gradient method). The data analysis is done in Python, with the programs **blocking.py**, **density.py** and **performance.py**.

13 Bibliography

References

[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, Phys. Rev. A **48**, 3561 - 3566 (1993).

[3] M. .L. Pedersen, G. Hagen, M. Hjorth-Jensen, S. Kvaal, and F. Pederiva, 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] `https://en.wikipedia.org/wiki/Gradient_descent`, Read: 17.03.2016.

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

[8] `https://en.wikipedia.org/wiki/Virial_theorem`, Read: 16.06.2016.