FYS4411 - Computational Physics II

Project 2

Dorthea Gjestvang
Even Marius Nordhagen

May 3, 2018

- Github repository containing programs and results:
  `https://github.com/evenmn/FYS4411/tree/master/Project%202`

Abstract

Abstract

Contents

1 Introduction

2 Theory

2.1 Presentation of potential

In this project, we simulate a system of P electrons trapped in a harmonic oscillator potential, with a Hamiltonian given by

$$\hat{H} = \sum_{i=1}^{P}(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2) + \sum_{i<j}\frac{1}{r_{ij}} \tag{1}$$

where ω is the harmonic oscillator potential and $r_i = \sqrt{x_i^2 + y_i^2}$ is the position of electron i. The term $\frac{1}{r_{ij}}$ is the interacting term, where $r_{ij} = |r_i - r_j|$ is the distance between a given pair of interacting electrons. Natural units have been used, such that $\hbar = c = m_e = e = 1$.

Since electrons are fermions, we need an antisymmetric wavefunction under exchange of two coordinates, and we need to take the Pauli principle into account. A Slater determinant is therefore needed for multiple fermions to ensure that the total wavefunction is antisymmetric. In this project we will study particles in the ground state only, and according to the Pauli principle we can in this case study a maximum of two particles with spin $s = \pm 1/2$. The slater determinant for two particles read

$$\Psi_T = \begin{vmatrix} \Phi_1(\boldsymbol{r}_1) & \Phi_2(\boldsymbol{r}_1) \\ \Phi_1(\boldsymbol{r}_2) & \Phi_2(\boldsymbol{r}_2) \end{vmatrix} = \Phi_1(\boldsymbol{r}_1)\Phi_2(\boldsymbol{r}_2) - \Phi_2(\boldsymbol{r}_1)\Phi_1(\boldsymbol{r}_2) \tag{2}$$

where $\Phi_i(\boldsymbol{r})$ is the single particle wave function (SPF) of state i. This contains a spatial part and a spin part, and we assume that it can be splitted up such that the spin part takes the antisymmetry property and does not affect the energy. Therefore we only need a symmetric spatial part to calculate the energies.

2.2 Solving this with machine learning

When solving a system of particles as the one described in the previous system with Variational Monte Carlo, as described in [3], we would need an anzats for the wave function, where we use our physical intuition to create the form of a wave function with different variational parameters, and then let it be up to the computer to find the optimal parameters through a minimization

method. However, this method is only as good as the physical intuition; if the form of the wave function is unrealistic, the results will be the same

This challenge can be mended by using machine learning. There are several different types of machine learning systems, and the one we will present and utilize in this project has the ability to learn and sample from a probability distribution. This is perfect for quantum mechanical problems, as we know from quantum mechanics the wave function Ψ is nothing more than a probability denisty, giving that Ψ^2 is a probability distribution that says something about where a given particle most probabliy can be found. As we are solely interested in the energy of the two-fermion system, and not the exact wave function, the fact that the machine learning program does not explicitly give the wave function is therefore of no consequence. We still have to give some guidelines for the form of the probability distribution, but the machine learning program can sample from a larger variety of probability distributions compared to the form used in VMC.

2.2.1 Machine learning

With the goal of solving the quantum mechanical system presented in section 2.1 in mind, we should start by explaining what machine learning is. Machine learning is the idea that a computer can be trained to learn to yield certaint outputs, without directly being told exactly what to give. Examples on this is pattern recognizion, where the computer first is shown for example pictures of wolves and huskies. After training the computer on pictures where the computer sees huskies and wolves and is told the correct answer, it should after a sufficiently long training period, be able to recognize huskies and wolves by itself.

The example described above is what we call supervised learning, where the correct output answer is known during the training program. A machine learning program could also be unsupervised, where the correct answer is unknown, or based on reinforcement learning, where the the program learns by conducting trial-and-error experiments.

2.2.2 Neural network

This sound amazing, and maybe even impossible. Therefore the question now is: how to program computers to learn, just like humans? The answer is, fittingly, that we should make the program run like the the human brain by implementing what is called a neural network. Inspired by neurons in the

human brain, a neural network is a programmed network of variables, called nodes, that comminucate in a given manner. Each node preforms a simple process: based on the input it receives, and how that input is weighted, it decides wheter or not to fire. The mathematical model of an example of a neural node was presented by McCulloch and Pitts in 1943 [2], shown in figure 1, where the input is mared x_i, the weights deciding how much the input should count is W_i, and the output from the node based on x_i and W_i is called h.

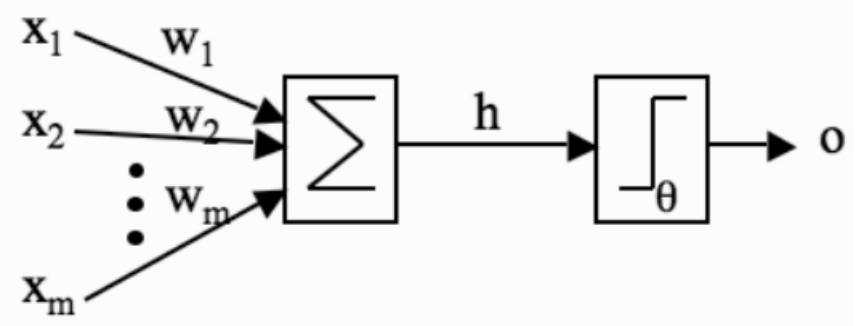

Figure 1: McCullochs and Pitts' model of neutrons visualized. Image reproduced from [2]

The neutrons are arranged in layers in the neural network, one visible layer that receives the input, and up to several hidden layers. The layers are arranged such that the output values from the visible nodes is the input values of the visible nodes. The nodes can also have bias values, that shift the output value h with a ceirtan number, and the nodes in different layers can be connected in different ways. An example of a neural network with two layers is shown in figure 2.

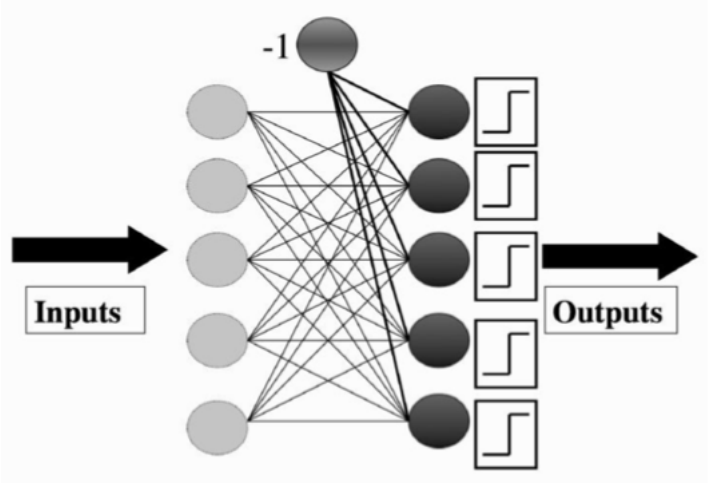

Figure 2: An example of a neural network with one visible and one hidden layer of nodes, and with bias -1 on the hidden nodes. Image reproduced from [2]

The idea behind machine learning is that the weights W_i, that decides how much a node puts emphazis on a given input, can be changed, and thus change the system's response to the same input. We will explain with the example from earlier, with huskies and the wolves: first, the program are shown pictures of wolves and huskies, and it is told the correct answer. The weights W_i are then updated such that when a husky is shown, the output "husky" is generated, and the same for wolves. After a sufficiently long training period, the program's weights are optimalized for recognizing wolves and huskies. When shown a picture, the program should then by itself be able to determine wheter it is a wolf or a husky that it sees.

2.2.3 Restricted Boltzmann Machines

There are plenty of ways to put together a neural network, as one can modify the number of nodes and layers, the bias, and also which nodes that are allowed to communicated. In this project, we will be using the so-called Restricted Boltzmann Machine (RBM). It is a two-layer network. The reason why it is named "restrictive" is that there are no connections between nodes in the same layer, but every node in the previous layer is connected to all the nodes in the next layer. The RBM can learn to draw samples from a probability distribution, which is just what we want in our project. In addition, we want to use a Gaussian-Binary RBM, where the hidden nodes have binary values, while the positions of the particles can take continous values, as they are, in fact, positions.

The joint probability distribution is known from statistical mechanics,

$$F(\boldsymbol{X}, \boldsymbol{H}) = \frac{1}{Z} e^{-\beta E(\boldsymbol{X}, \boldsymbol{H})} \tag{3}$$

where we set $\beta = 1/kT = 1$ and Z is the partition function, which can be ignored since it will vanish anyway.

The system energy of a Gaussian-Binary RBM is given by

$$E(\boldsymbol{X}, \boldsymbol{H}) = \sum_{i=1}^{M} \frac{(X_i - a_i)^2}{2\sigma_i^2} - \sum_{j=1}^{N} b_j H_j - \sum_{i,j=1}^{M,N} \frac{X_i W_{ij} H_j}{\sigma_i^2} \tag{4}$$

(Hinton 2010).

By setting $\Psi = F(\boldsymbol{X}, \boldsymbol{H})$, we know have a general form of the probability distribution. In equation 5 the form of the wave function is shown, when we have used that we have M visible nodes and N hidden nodes, and that the hidden nodes should take binary values.

$$\Psi = \frac{1}{Z} e^{\sum_i^M \frac{(X_i - a_i)^2}{2\sigma^2}} \prod_j^N (1 + e^{b_j + \sum_i M \frac{X_i W_{ij}}{\sigma^2}}) \tag{5}$$

2.3 Energy calculation

We want to calculate the energy of the two-fermion system, given the wave function described in 5. As explained in [3], the energy of the system is the expectation value of the Hamiltonian, but this is hard to compute directly.

$$E_L(\vec{r}) = \frac{1}{\Psi_T(\vec{r})} \hat{H} \Psi_T(\vec{r}). \tag{6}$$

By defining the local energy given in equation 6, the energy can be expressed as equation 7, and this can be solved with a Monte Carlo loop.

$$E = \int |\Psi_T^2| E_L d\vec{r} \tag{7}$$

2.3.1 Exact energy for non-interacting case

The wave function of one electron in a two-dimensional harmonic oscillator is given by

$$\phi_{n_x, n_y} = A H_{nx} \sqrt{\omega} x H_{ny} \sqrt{\omega} y e^{-\frac{\omega(x^2 + y^2)}{2}} \tag{8}$$

where $H_{nx}\sqrt{\omega}x$ is the Hermite polynomials and A is the normalization constant, and n_x and n_y are the principal quantum numbers. If we denote ϕ_1 the spatial wave function for the first electron, and ϕ_2 the spatial wave function for the second electron, then the total spatial wave function for the two-particle system is $\phi_1\phi_2$. As explained in section 2.1, the spin wave functions are anti-symmetric, and therefore it is fine that the spatial wave function is symmetric.

For a one-electron system the energy can be shown to take the value

$$H\left|\phi_1\right\rangle = E_1\left|\phi_1\right\rangle = \omega(n_{x,1} + n_{y,1} + 1)\left|\phi_1\right\rangle \quad \rightarrow E_1 = \omega(n_x + n_y + 1) \tag{9}$$

where we have used natural units, and all the energies are given in atomic units a.u.

When calculating the energy for the two-electron system we then get the following

$$H\left|\phi_1\phi_2\right\rangle = \left|H\phi_1\right\rangle\left|\phi_2\right\rangle + \left|\phi_1\right\rangle\left|H\phi_2\right\rangle = E_1\left|\phi_1\phi_2\right\rangle + E_2\left|\phi_1\phi_2\right\rangle \tag{10}$$

For the ground state $n_x = n_y = 0$, $E_1 = E_2 = \omega$ and therefore the ground state energy is simply $E_1 + E_2 = 2\omega$.

For the non-interacting case with $\omega = 1$ it is known the ground state energy is 3 a.u. [4].

2.4 Onebody density

A quantity that often is handy to calculate is the onebody density. The onebody density is the density of particles in the potential, and may give a better understanding of the placement of the particles than their positions. The onebody density ρ_i with respect to a given particle i is given by equation 11, which can be solved by Monte Carlo integration, as described in [3].

$$\rho_i = \int_{-\infty}^{\infty} d\vec{r}_1 \ldots d\vec{r}_{i-1} d\vec{r}_{i+1} \ldots d\vec{r}_N |\Psi(\vec{r}_1, \ldots \vec{r}_N)|^2. \tag{11}$$

Apart from the fact that the one body density is a good visual representation of the positions of the particles, it is also a qunatity that is often calculated experimentally. Therefore, it is simpler to compare computational simulations to experimental results when having calculated the onebody density.

2.5 Error estimation

As a physicist one should always be sceptical to measurements presented without an estimation of the uncertainties. When someone says they have measured their height to be 1.90 cm, is the uncertainty $\pm 1cm$ or $\pm 10cm$? If they used $\pm 1cm$, we have a fairly good idea of the height of the person, but if it is $\pm 10cm$, we do not really know anything about the real height of the person.

The error in a measurement is the difference between the true value of the mesurand, x, and the mesured value x_i:

$$error = x - x_i \tag{12}$$

However, the true value x is rarely known. When x is not known, it is thus impossible to give the error in the measurement. We must therefore be content with providing estimates of the error. It is common to give this uncertainty as the standard deviation σ; when preforming a mesurement n times, such that we have n measured points $x_1, .., x_n$ where the average value is $\langle x \rangle$, then the standard deviation is defined as

$$\sigma_\theta = \sqrt{\frac{1}{n-1}\sum_{k=1}^{n}(x_i - \langle x \rangle)^2} \tag{13}$$

When giving the uncertainty, one writes $x_i \pm \sigma$, where σ is so that $\approx 68\%$ of the measured values lies withing the interval $\pm\sigma$.

There are different kinds of sources of error when conducting computer experiments. Systematic errors originate from faults in the theory used in the experiment, and this might give consistently wrong results. Systematic errors are hard to handle, as one might get seemingly good looking results. One way to avoid systematic results are to benchmark the code with either experimental results, or with other's computer simulation results.

Another source of error is the statistical error, which is based on how many measurements you have done. In the example with the height measurement in the beginning of this section, if the experimentalist only measured the height one time, we can be fairly certain that this is not the true value of the person. If they, however, repeated the same measurement several times and then maybe used the average as the estimate of the mesurand, we can be more confident that they have a better estimate, compared to when they only used one point. A common way to estimate the statistical uncertainty is with equation 14.

$$\sigma^2 \approx \langle x^2 \rangle - \langle x \rangle^2 \tag{14}$$

However, as explained in [3], this does not account for the covarinace between measurements. This leads to equation 14 being an underestimation of σ^2, and is thus more a guideline for the size of the uncertainty, more than an actual estimate of it. Luckily, there are computational methods that can calculate σ^2 with the covariance contribution included. The method used in this project is the blocking method, presented in section 3.5.

3 Method

3.1 Variational Monte Carlo

The Variational Monte Carlo method, often referred to as VMC, is a method in computational physics. The goal of VMC is to approximate the ground state energy of a quantum mechanical system by using the variational principle. The variational principle says that for any given trial wave function Ψ_T, the energy E given this Ψ_T is always more than or equal to the ground state energy E_0 of the system:

$$E = \frac{\langle \Psi_T | \, H \, | \Psi_T \rangle}{\langle \Psi_T | \Psi_T \rangle} \geq E_0 \tag{15}$$

In VMC, the user inputs a trial wave function for the quantum mechanical system. Over a Monte Carlo loop, the particles in the system are moved randomly, and for each proposed move, the new position of the particle is accepted or rejected depending on if the new position is more probable according to Ψ_T, compared to the previous position. For the ground state wave function, the position which yields a minimum in the energy of the quantum mechanical system is the most probable position. The result is therefore that the particles in the system are moved towards lower energies. When the method has converged, the energy E is an upper estimate of the true ground state energy E_0. If Ψ_T in addition is the true ground state wave function, then $E = E_0$.

3.2 Metropolis algorithm

Above, we described that the VMC method had to reject or accept proposed moves of the particles in the system. The task of handeling this is often left up to the Metropolis algorithm. The Metropolis algorithm uses the theory

that the probability of a system to transition from a state i to state j is given by

$$W_{i\to j} = T_{i\to j} \cdot A_{i\to j} \tag{16}$$

where $T_{i\to j}$ is the transition probability and $A_{i\to j}$ is the acceptance probability. If we denote the probability for being in a given state for p, and then look at the transition from a state i to a state j and back again, then it can be shown that we obtain the following relation between the transition and acceptance probabilities [3]:

$$\frac{A_{j\to i}}{A_{i\to j}} = \frac{p_i T_{i\to j}}{p_j T_{j\to i}} = w \tag{17}$$

As the Metropolis algorithm's job is to evaluate probable moves between states, the expression in equation 17 is used as the acceptance criteria. Thus the ratio $\frac{A_{j\to i}}{A_{i\to j}}$ describes if the system is more likely to move one way or the other. There are several methods that can be used for accepting and rejecting the moves, and these use different expressions for T and p in equation 17. In this project, we will study the brute force and the Hastings sampling algorithms.

3.2.1 Brute force

The brute force got is name because when proposing and accepting moves, the algorithm does so without "thinking" which way the system will most probably move. A random particle with position $\vec{R}$ is chosen, and its new position $\vec{R}_{new}$ is proposed at random as follows:

$$\boldsymbol{R}_{new} = \boldsymbol{R} + r \cdot \text{step}. \tag{18}$$

where r is a random number and *step* is the steplength.

The brute force algorithm does not concider the transitions probabilities, and simply puts $T_{i\to j} = T_{j\to i}$. To check the probability of a system being found in a given state, it uses $p(\vec{R}) = |\Psi_T(\vec{R})|^2$ for both position, such that the probability w for the transition is given by

$$w = \frac{P(\vec{R}_{new})}{P(\vec{R})} = \frac{|\Psi_T(\vec{R}_{new})|^2}{|\Psi_T(\vec{R})|^2}. \tag{19}$$

A random number $r \in [0, 1]$ is then drawn, and compared to w. If w is larger than r, then the new position is accepted, and if not, the move is rejected and the particle is moved back to its original position:

$$\text{New position: } \begin{cases} \text{accept} & \text{if} \quad w > r \\ \text{reject} & \text{if} \quad w \leq r. \end{cases} \tag{20}$$

This way, the algorithm will over many iterations move the particles towards the most probable state.

3.2.2 Metropolis-Hastings

The Metropolis-Hastings algorithm, also known as importance sampling, have a more refined approach to choosing new positions and accepting them, compared to the brute force method. Instead of proposing the new position at random, the Metropolis-Hastings algorithm considers the drift force F, given by equation 21, which says something about which direction the particle is pushed in.

$$F(R) = \frac{2\nabla\psi_T}{\psi_T} \tag{21}$$

The equation that describes the motion of a particle pushed around by the drift force F, is called the Langevin equation. The Langevin equation can be derived from the Fokker-Planch equation, which describes the time-evolution of the probability density function. The Langevin equation can be solved using the Euler method. The result, presented in equation 22 , describes how the Metropolis-Hastings algorithm chooses a new position for the random particle:

$$R_{new} = R + DF(R)\Delta t + \xi\sqrt{\Delta t} \tag{22}$$

where $D = \frac{1}{2}$ is the diffusion constant and Δt is the timestep, that decides how long steps the method should take for each move. This is a much more educated guess compared to the brute force algorithm.

When regaring the transition probabilities, the Metropolis-Hastings algoritm does not go the easy way with $T_{i\to j} = T_{j\to i}$. Instead it uses that the transition probabilities are given as the Green's function, also found using the Fokker-Planch equation:

$$\begin{aligned} G(R_{new}, R, \Delta t) &= \frac{1}{(4\pi D\Delta t)^{3N/2}} \exp\big[-(R_{new} - R - D\Delta t F(R))^2/4D\Delta t\big] \\ &= T_{R\to R_{new}} \end{aligned} \tag{23}$$

such that the probability parameter w becomes

$$w = \frac{G(R, R_{new}, \Delta t)|\Psi_T(R_{new})|^2}{G(R_{new}, R, \Delta t)|\Psi_T(R)|^2}. \tag{24}$$

Thereafter a random number r is drawn and compared to w, just like in the brute force method. This more complicated way of proposing and accepting positions makes the Metropolis-Hastings slightly slower than the brute force method, but also quicker to converge towards the ground state.

One thing that is worth to note with the brute force and the Metropolis-Hastings algorithms when doing machine learnign projects, is that both of them in this project are implemented so that only the positions, ie the visible nodes, are updated. This makes it so that we lose the opportunity to modify half of the syste, which maybe yield a worse convergence than if we varied the whole system.

3.3 Gibbs sampling

As an alternative to the Metropolis algorithm, one can use the Gibbs sampling method. In both the brute force method and Metropolis-Hastings, one has to accound for that proposed particle moves can be rejected, and this "wastes" one Monte Carlo cycle with doing several flops, when no change is inflicted on the system.

The Gibbs sampling algorithm differs from the Metropolis in several ways. In section 2.2.3 we wrote that we chose the trial wave function $\Psi = F(\boldsymbol{X}, \boldsymbol{H})$, where F is defined in equation 3. In Gibbs sampling, we instead use the trial wave function $\Psi_T = \sqrt{F}$.

Furthermore, instead of updating only the visible nodes, as is done in Metropolis, the Gibbs sampling algorithm is a two-step sampling process where first a new position is chosen, and then one of the hidden nodes are updated. The positions are sampled from a Gaussian probability distribution, shown in equation 25, while the probability for sampling $h = 1$ is shown in equation 26.

$$P(\boldsymbol{X}|\boldsymbol{h}) = \mathcal{N}(\boldsymbol{X}; \boldsymbol{a} + \boldsymbol{W}\boldsymbol{h}\sigma^2) \tag{25}$$

$$P(H_j = 1|\boldsymbol{X}) = \frac{1}{1 + e^{-b_j - \frac{\boldsymbol{X}^T \boldsymbol{W}_{\boldsymbol{cols,j}}}{\sigma^2}}} \tag{26}$$

When we say that the visible and hidden nodes here are "sampled", we mean that they are drawn from the probability distribution, and then au-

tomatically accepted. This is different from Metropolis, and should mean that the Gibbs sampling method converges faster than both brute force and Metropolis-Hansings, as no Monte Carlo cycles are lost on the move not being accepted.

3.4 Gradient descent

We use stochastic gradient descent to update the weights in order to minimize the local energy. The process itself is quite similar to that one typically used in Variational Monte Carlo (VMC) where the weights are our variational parameters, but compared with a normal VMC we have a lot more parameters to vary. The updating algorithm for updating the variational parameter α_i goes as

$$\alpha_i^+ = \alpha_i - \eta d\alpha_i \tag{27}$$

where the gradient of the local energy with respect to parameter α_i is

$$d\alpha_i = \frac{\partial \langle E_L \rangle}{\partial \alpha_i} = 2\bigg(\Big\langle E_L \frac{1}{\Psi}\frac{\partial \Psi}{\partial \alpha_i} \Big\rangle - \Big\langle E_L \Big\rangle \Big\langle \frac{1}{\Psi}\frac{\partial \Psi}{\partial \alpha_i} \Big\rangle \bigg) \tag{28}$$

where η is the learning rate. We need to do this for all the parameters, and for our purposes it is convenient to vectorize the gradient such that $d\boldsymbol{a}$ and $d\boldsymbol{b}$ are vectors and $d\boldsymbol{W}$ is a matrix. The gradients to implement are

$$d\boldsymbol{a} = \frac{1}{\Psi}\frac{\partial \Psi}{\partial \boldsymbol{a}} = \frac{\boldsymbol{X} - \boldsymbol{a}}{\sigma^2} \tag{29}$$

$$d\boldsymbol{b} = \frac{1}{\Psi}\frac{\partial \Psi}{\partial \boldsymbol{b}} = \tag{30}$$

3.4.1 Adaptive stochastic gradient descent

A problem with the standard stochastic gradient descent method is that it will either converge fast, but the precision is poor, or the precision is good, but it converges slowly. To avoid this problem, we want a large learning rate when the energy is far from converging and a small learning rate when the energy converges.

3.5 Blocking method

In section 2.5, we described the need of a proper estimation of the uncertainty in computational simulations, where the covariance was included in the calculation od σ. A quick and easy way to get a proper estimate of

the uncertainty, is by using the blocking method. If we have a data set $x_1, x_2, ..., x_n$ with n data points, where the mean value is $\langle x \rangle$. We can calculate the standard deviation using equation 13, however, as explained before, this does not include the covariance, as the data points $x_1, x_2, ..., x_n$ may be correlated. A way to get around this, it by transforming the data set such that

$$x'_i = \frac{1}{2}(x_{2i} + x_{2i+1}) \tag{31}$$

The number of points in the transformed data set is now n', and every x'_i consists of two of the original data points. The transformed x'_i's are referred to as blocks, and the standard deviation in each block σ_i can now be calculated. An estimate of the total standard deviation $\hat{\sigma}$ when we have n' transformed data points is

$$\hat{\sigma} = \frac{\sigma_1 + .. + \sigma_{n'}}{n'} \tag{32}$$

We now increase the block size, such that there are even more of the original data point in each block, and calculate the total standard deviation from equation 32 again. If we continue doing this and plot the resulting estimates of $\hat{\sigma}$, we will observe that it first increases, until it flats out. When we have reached this plateu, we know that the data blocks x'_i are no longer correlated. The covariance is then 0, and we have thus obtained a proper estimation of the uncertainty.

4 Code

In this section, we give an overview of the structure and implementation of our code, to make it easier for potential users to familiarize themselves with the code.

4.1 Structure

In order to keep the code structured, we have split the running of the program up in several files. The user only needs to interact with main.cpp, as all the parameters for running the machine lerning project are set here. Examples are that the user can choose between brute force, Metropolis-Hastings og Gibbs sampling, and set the number of visible and hidden nodes, along with the number of iterations. main.cpp calls on the function GradientDecent in gradient_decent.cpp, where the machine learning structure is implemented.

The machine learning structure consists of the VMC-loop, where the nodes are updated, and the gradient decent method, where the weights are updated. GradientDecent uses the class WaveFunction in wavefunction.cpp to calculate the value of the wave function, along with the energy of the system. Depending on whether the user chose brute force, Metropolis-Hastings or Gibbs sampling, GradientDecent calls functions in either wavefunction.cpp, hastings_tools.cpp or gibbs_tools.cpp to update the positions and calculate the transition probabilities. The test functions are found in test.cpp. The structure can be seen in figure 3.

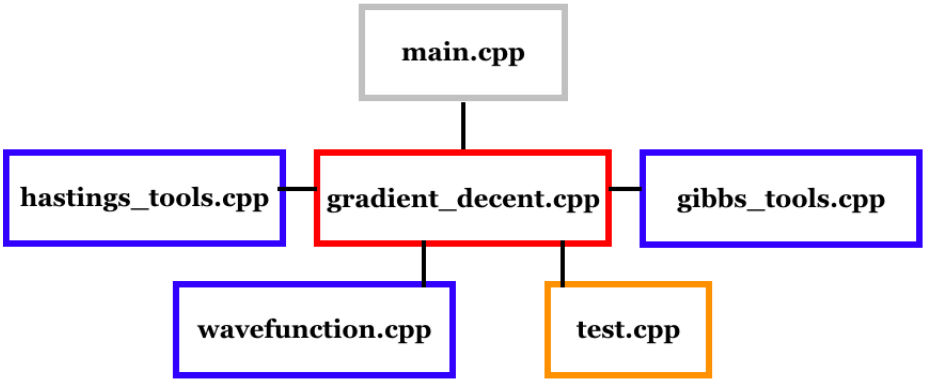

Figure 3: The structure of the machine learning program, showing the communication between the different files

4.2 Implementation

4.2.1 General implementation

```
Set up vectors X and h for visible and hidden nodes, fill with
    ↪ random numbers, X between -1 and 1, h with 0 and 1

Set opp vectors a, b, W for the weights, fill with random numbers
    ↪ between -1 and 1

Initialize class WaveFunction

for iter iterations
        for MC Monte Carlo cycles
                #do VMC

        #do GradientDecent

        end when stopping criterias == true, when method has
            ↪ converged
        print final energy
```

4.2.2 VMC

```
for M Monte Carlo iterations

        if Metropolis:
                move random particle
                accept or reject move based on transition probability

        if Gibbs:
                sample new position and hidden node

        calculate energy of new position
        update sum of E_L

print estimated E
```

4.2.3 Gradient Decent

```
for iter iterations:

        for MC Monte Carlo cycles:
                #do VMC

        Check if stopping criteria == true:
                if true, break and print final values

        Calculate gradient for the weights a, b and W
        Update a, b and W
```

5 Results

In this section we will see how accurate results we achieve using the NQS wavefunction for the brute-force Metropolis-, the Metropolis-Hastings- and the Gibbs' sampling method. Firstly, we calculate the energy of one particle in one dimension to ensure that the method works and we obtain an energy close to the analytical energy. Secondly, we extend the system to two interacting electrons in a two-dimensional harmonic oscillator. Furthermore the onebody densities with and without interaction are presented and finally we study how the total energy is distributed between kinetic energy and potential energy for various harmonic oscillator frequencies.

5.1 Energy calculations, without interaction

We first study the energy of a non-interacting electron in one dimension, and in figure 4 we compare the energies produced by the three methods as a function of number of iterations.

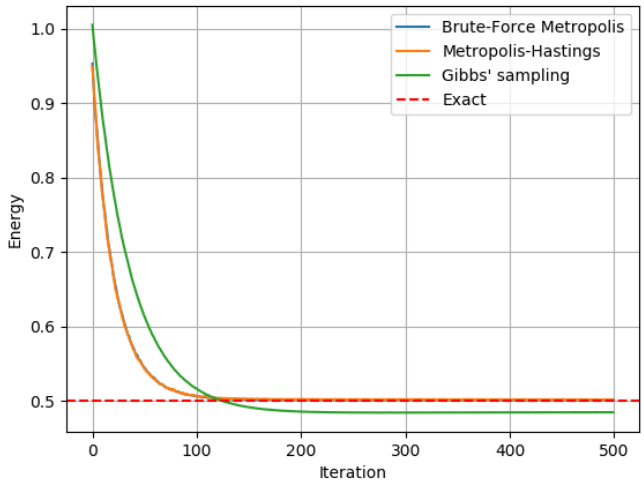

Figure 4: Energy as a function of number of iterations up to 500 iterations for one particle in one dimension. Brute-Force (BF) Metropolis (blue line), Metropolis-Hastings (MH) (orange line), Gibbs' (green line) and analytical value (dashed red line). All the samplings were conducted with MC=1000000 and $\eta = 0.01$. For BF we used steplength $dx = 1.0$, while for MH the timestep was $dt = 0.01$. See text for further description.

We observe that the energies produced by brute-force and Metropolis-Hastings are very similar, and in fact it is hard to distinguish them. They also give energies closer to the analytical energy compared to Gibbs' sampling and converge faster. In table 1 some chosen energies from figure 4 are listed, and we observe that brute-force has actually the smallest absolute error for all number of iterations.

Table 1:

Iterations	BF	MH	Gibbs	Exact
100	0.506332	0.507337	0.516750	0.5
200	0.501485	0.502318	0.485531	0.5
300	0.501363	0.501983	0.484165	0.5
400	0.501207	0.501983	0.484105	0.5
500	0.501323	0.501731	0.484648	0.5

5.1.1 Playing with learning rate and number of hidden nodes

To find the lowest energy, we experiment with different learning rates and a various number of hidden nodes. We use brute-force Metropolis and in figure 5 one can find energy plotted as functions of number of iterations. Initially the energy is overall higher for more hidden nodes, but we get a closer energy to the exact energy for lower number of hidden nodes.

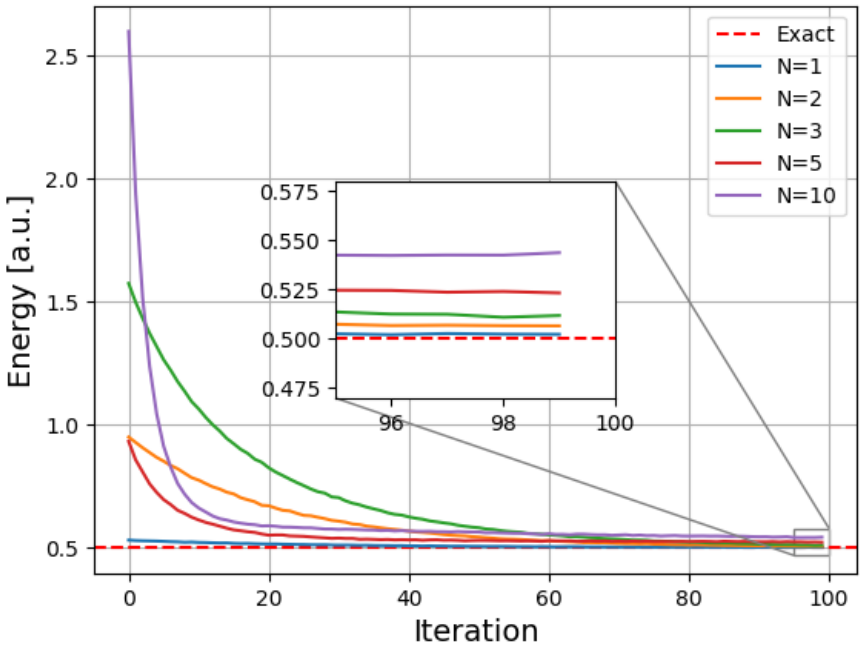

Figure 5: Energy as functions of iterations for N=1,2,3,5 and 10. Brute-Force Metropolis is used, MC=1000000, $\eta = 0.01$ and $dx = 1.0$.

Since the optimal number of hidden nodes is one in this case, we will use N=1 while optimizing the learning rate.

5.2 Energy calculations, with interaction

5.3 Onebody density

In figure **??** the onebody densities for two particles in two dimensions with and without interaction is plotted. From the energy calculations, we found Metropolis-Hastings method to be the better one for two particles, and henceforth we will therefore stick to it. They look similar, but

5.4 Average energies and distance

6 Discussion

7 Conclusion

8 Appendix A - Local energy calculations

The energy can be calculated by introducing a local energy, where the acerage local energy goes to the true energy with a sufficient amount of samples. This local energy can be splitted up in a kinetic part, a part from the harmonic oscillator potential and a interacting part,

$$E_L = \sum_{k=1}^{M}(E_{\text{KIN},k} + E_{\text{EXT},k}) + E_{\text{POT}}. \tag{33}$$

As discussed in [3], the kinetic part can be expressed as

$$E_{\text{KIN},k} = \frac{1}{\Psi_T}\nabla_k^2\Psi_T \tag{34}$$

$$= (\nabla_k \ln \Psi_T)^2 + \nabla_k^2 \ln \Psi_T \tag{35}$$

where we have used that

$$\frac{1}{\Psi_T}\nabla_k\Psi_T = \nabla_k \ln \Psi_T. \tag{36}$$

$$\frac{\partial}{\partial X_k} \ln \Psi_T = -\frac{X_k - a_k}{\sigma^2} + \sum_{j=1}^{N} \frac{W_{kj}}{\sigma^2}\text{Logistic}(v(j)) \tag{37}$$

$$\frac{\partial^2}{\partial X_k^2} \ln \Psi_T = -\frac{1}{\sigma^2} + \sum_{j=1}^{N} \frac{W_{kj}^2}{\sigma^4}\text{Logistic}^2(v(j))e^{v(j)} \tag{38}$$

where

$$\text{Logistic}(x) = \frac{1}{1+e^x} \tag{39}$$

and

$$v(j) = b_j + \sum_{i=1}^{M} \frac{X_i W_{ij}}{\sigma^2}. \tag{40}$$

Thus the local energy can be expressed as

$$E_L = \sum_{i=1}^{N} \frac{\boldsymbol{W}_{*i}^T \boldsymbol{W}_{*i}}{\sigma^4} \text{Logistic}^2\big(v(i)\big) \tag{41}$$

$$+ \sum_{i,j=1}^{N,N} \frac{\boldsymbol{W}_{*i}^T \boldsymbol{W}_{*j}}{\sigma^4} \text{Logistic}\big(v(i)\big)\text{Logistic}\big(v(j)\big) \tag{42}$$

$$- 2\sum_{i=1}^{N} \frac{\boldsymbol{W}_{*i}^T (\boldsymbol{X} - \boldsymbol{a})}{\sigma^4} \text{Logistic}(v(i)) \tag{43}$$

$$+ \frac{(\boldsymbol{X} - \boldsymbol{a})^T \cdot (\boldsymbol{X} - \boldsymbol{a})}{\sigma^4} - \frac{M}{\sigma^2} + \boldsymbol{X}^T \boldsymbol{X} + E_{\text{INT}} \tag{44}$$

9 References

[1] Morten Hjorth-Jensen. Computational Physics 2: Variational Monte Carlo methods, Lecture Notes Spring 2018. Department of Physics, University of Oslo, (2018).

[2] S. Marsland, *Machine Learning: An algorithmic Perspective, Second edition* (2015)

[3] D. Gjestvang, E. M. Nordhagen *Computational Physics II: Project 1* (2018)

[4] V. Flugsrud and M. Hjort-Jensen Project 2, The restricted Boltzmannn machine applied to the quantum many body problem. Deadline June 1 Department of Physics, University of Oslo, (2018).

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