Bose Einstein Condensate, Variational Monte Carlo

Linus Ekstrøm

March 2021

Abstract

The main goal of this project was to use Variational Monte Carlo to simulate a Bose-Einstein Condensate in an alkali gas. We managed to reproduce the expected energy for the harmonic oscillator without interaction. We studied various approaches: *brute force*, *importance sampling*, *gradient methods* and we produced good results for these methods. We saw that adding importance sampling greatly increased the accepted metropolis step ratio. In addition the gradient method outperformed the brute force method for our one parameter grid search. The complete repository with all my work is located at `https://github.com/Linueks/Fys4411-Variational-Monte-Carlo`

Contents

1 Introduction

In this project we will look into the behaviour of a Bose-Einstein condensate. First we present the theoretical background for our work, then our method of implementation, following this we present our results and discuss them.

2 Theory

In this section we present the necessary theoretical background to follow along with our numerical experiments.

2.1 Bose-Einstein Condensates

A *Bose-Einstein condensate* (BEC) is a state of matter, typically formed when gases of bosons are cooled down close to absolute zero. In this condition a large amount of the bosons are able to occupy the lowest energy quantum state. This allows quantum phenomena commonly only visible in the microscopic real to be studied for macroscopic systems. This shared lowest energy state was first predicted by Albert Einstein [1] the 10th of July 1924 following a paper written by Satyendra Nath Bose. Interesting factors to study for the condensates is the fraction of condensed atoms, the nature of the condensate, the number of excitations above the condensate, the atomic density in the trap as a function of the temperature as well as the critical temperature required to achieve condensation.

2.1.1 Bose-Einstein Condensates in Alkali Gases

Trapped alkali systems are dilute, meaning the effective atom size is sufficiently small compared with the trap size as well as the inter-atom spacing. The condition for diluteness is given by $na_{Rb,Na,Li} \approx 10^{-6}$ where $n = N/V$ is the number density and $a_{Rb,Na,Li}$ is the s-wave scattering length of ^{87}Rb, ^{23}Na and ^{7}Li respectively. For the rest of this project we will focus primarily on Rubidium atoms unless specified otherwise. The characteristic dimensions for a typical Rubidium trap is $a_{ho} = \left(\frac{\hbar}{m\omega_\perp}\right)^{\frac{1}{2}} = 1.5 \cdot 10^4$Å. The scattering length lies in the range $85a_0 < a_{Rb} < 140a_0$ where a_0 is the Bohr radius. For the rest of the calculations performed in this project we will stick to a value of $a_{Rb} = 100a_0$.

One common framework for study of BEC in gases in alkali gases is the Gross-Pitaevskii (GP) equation. Where the time-independent version of the equation is

$$\mu\Psi = \Big(\frac{-\hbar^2}{2m}\nabla^2 + V + q|\Psi|^2\Big)\Psi$$

which is valid when the number of particles is conserved. Here μ is the chemical potential. From this one can find the structure of the BEC when external potentials are present. The time-dependent GP equation is given by

$$i\hbar\frac{\partial\Psi}{\partial t} = \Big(\frac{-\hbar^2}{2m}\nabla^2 + V + g|\Psi|^2\Big)\Psi$$

A key point for this description is the aforementioned diluteness condition, when this condition is met the physics is dominated by two-body collisions. In recent experiments it is possible for the local gas parameter to exceed the dilute condition due to tuning the scattering length in the presence of a Fashbach resonance. In such a situation, improved many-body methods such as Monte Carlo calculations may prove necessary. The main goal of this computational project is to use Variational Monte Carlo (VMC) methods to evaluate the ground state energy of a trapped, hard sphere BEC gas for a variable number of particle with a specific trial wave function.

2.2 Potential and Hamiltonian for BEC

The trial function mentioned above is used to study the properties of the condensate as well as non-condensate properties. The trap we will be studying is either a spherical or an elliptical trap defined by

$$V_{ext}(\vec{r}) = \begin{cases} \frac{1}{2}m\omega_{ho}^2 r^2 & (S) \\ \frac{1}{2}m[\omega_{ho}^2(x^2+y^2) + \omega_r^2 z^2] & (E) \end{cases}$$

When dealing with inter-boson interaction we will use V_{int} to represent the potential on each atom caused by all the other atoms.

$$V_{int}(\vec{r}) = \begin{cases} \infty & |\vec{r}_i - \vec{r}_j| \leq a \\ 0 & |\vec{r}_i - \vec{r}_j| > a \end{cases}$$

where a is the so-called hard-core diameter of the bosons. Now we can define our Hamiltonian for the BEC

$$H = \sum_i^N \Big(\frac{-\hbar^2}{2m}\nabla_i^2 + V_{ext}(\vec{r}_i)\Big) + \sum_{i<j}^N V_{int}(\vec{r}_i, \vec{r}_j)$$

where we have defined

$$\sum_{i<j} V_{ij} \equiv \sum_{i=1}^N \sum_{j=i+1}^N V_{ij}$$

which is a sum running over all the internal interactions between the bosons. In the Hamiltonian expression ω_{ho}^2 defines the trap potential strength.

2.3 Local Energy

The local energy is defined as

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

where the Ψ_T is the so called trial wave function for the ground state of a Bose-Einstein condensate with N atoms

$$\Psi_T(\vec{r}) = \Big[\prod_i g(\alpha,\beta,\vec{r}_i)\Big]\Big[\prod_{j<k} f(a,|\vec{r}_j-\vec{r}_k)\Big] \quad (2)$$

Here α and β are variational parameters. Above we see the trial wave function is split into two parts: a single-particle wave function and the correlation wave function.

$$g(\alpha,\beta,\vec{r}_i) = e^{-\alpha(x_i^2+y_i^2+\beta z_i^2)}$$

2.3.1 Single Particle in Harmonic Oscillator Trap

For the harmonic oscillator potential we have $a = 0$. We initially use $\beta = 1$, thus our single-particle wave function becomes

$$\Psi_T(x) = e^{-\alpha x^2}$$

Plugging into the local energy equation 1 we have

$$\begin{aligned} E_L(x) &= \frac{1}{\Psi_T(x)} H\Psi_T(x) \\ &= \frac{1}{\Psi_T(x)}\Big(\frac{-\hbar^2}{2m}\frac{d^2\Psi_T(x)}{dx^2} + \frac{1}{2}m\omega_{ho}^2 x^2\Psi_T(x)\Big) \\ \frac{d^2\Psi_T(x)}{dx^2} &= -2\alpha e^{-\alpha x^2} + 4\alpha^2 x^2 e^{-\alpha x^2} \end{aligned}$$

We now use natural units : $\hbar = m = \omega_{ho} \equiv 1$

$$\begin{aligned} E_L(x) &= -\frac{1}{2}\Big(-2\alpha + 4\alpha^2 x^2\Big) + \frac{1}{2}x^2 \\ &= \alpha + \Big(\frac{1}{2} - 2\alpha^2\Big)x^2 \end{aligned}$$

2.3.2 N Non-Interacting Particles in Harmonic Oscillator Trap

When extending the above calculation into three dimensions it is beneficial to use spherical coordinates. The laplacian becomes

$$\nabla^2 = \frac{1}{r^2}\frac{\partial}{\partial r}\Big(r^2\frac{\partial}{\partial r^2}\Big)$$

Whilst the trial function becomes

$$\begin{aligned} \Psi_T(\vec{r}) &= \Big[\prod_i g(\alpha,\beta,\vec{r}_i)\Big] = \Big[\prod_i e^{-\alpha|\vec{r}|^2}\Big] \\ &= e^{-\alpha\sum_i r_i^2} \end{aligned}$$

We use this in the same way as for the one-dimensional case. Inserting into the expression for the local energy gives

$$\begin{aligned} E_L(r) &= \frac{1}{\Psi_T(r)} H\Psi_T(r) \\ \nabla^2\Psi_T(r) &= \frac{1}{r^2}\frac{\partial}{\partial r}\Big(r^2\frac{\partial\Psi_T(r)}{\partial r}\Big) \\ r_j^2\frac{\partial\Psi_T(r)}{\partial r_j} &= -2\alpha r_j^3 e^{-\alpha\sum_i r_i^2} \\ \frac{1}{r_j^2}\frac{\partial\Psi_T(r)}{\partial r} &= \Big(4\alpha r_j^2 - 6\alpha r_j^2\Big)e^{-\alpha\sum_i r_i^2} \\ E_L(r) &= \sum_j -\frac{1}{2}(4\alpha^2 r_j^2 - 6\alpha) + \frac{1}{2}r_j^2 \\ &= 3N\alpha + \Big(\frac{1}{2} - 2\alpha^2\Big)\sum_j r_j^2 \end{aligned}$$

For the general case of N particles in D dimensions we have

$$E_L(r;\alpha) = DN\alpha + \Big(\frac{1}{2} - 2\alpha^2\Big)\sum_j r_j^2$$

In addition the expression for the local energy of the BEC without interaction in an elliptical trap is

$$E_L(r;\alpha,\beta) = (2\alpha + \alpha\beta)N + \Big(\frac{1}{2} - 2\alpha^2\Big)\sum_j r_j^2$$

2.3.3 N Interacting Particles in Harmonic Oscillator Trap

For the full problem, we define

$$f(r_{ij}) = \exp(\sum_{i<j} u(\vec{r}_{ij})) \quad (3)$$

and use this to rewrite the trial function from equation (2) to

$$\Psi_T(\vec{r}) = \Big[\prod_i g(\alpha,\beta,\vec{r}_i)\Big]\exp\Big(\sum_{j<k} u(\vec{r}_{j,k})\Big)$$

We also define

$$\phi(\vec{r}_i) = e^{-\alpha(x_i^2+y_i^2+\beta z_i^2)}$$

Which gives

$$\Psi_T(\vec{r}) = \Big[\prod_i \phi(\vec{r}_i)\Big]\exp\Big(\sum_{j<k} u(\vec{r}_{j,k})\Big)$$

We now calculate the first derivative for particle k using the product rule. For the Pi notation we take the derivative as we would with any product of functions, e.g.

$$\frac{d\Big(a(x)b(y)c(z)\Big)}{dy} = \frac{db(y)}{dy}\Big(a(x)c(z)\Big)$$

The rest of the functions are left unchanged. In product notation we get

$$\nabla_k\Big[\prod_i \phi(\vec{r}_i)\Big] = \nabla_k\phi(\vec{r}_k)\Big[\prod_{i\neq k}\phi(\vec{r}_i)\Big]$$

And using this with the product rule gives

$$\begin{aligned}\nabla_k \Psi_T(\vec{r}) &= \nabla_k \left(\Big[\prod_i \phi(\vec{r_i}) \Big] \exp\Big(\sum_{j<m} u(\vec{r}_{j,m}) \Big) \right) \\ &= \nabla_k \phi(\vec{r_k}) \Big[\prod_{i\neq k} \phi(\vec{r_i}) \Big] \exp\Big(\sum_{j<m} u(\vec{r}_{j,m}) \Big) \\ &+ \Big[\prod_i \phi(\vec{r_i}) \Big] \exp\Big(\sum_{j<m} u(\vec{r}_{j,m}) \Big) \sum_{l\neq k} \nabla_k u(\vec{r}_{k,l})\end{aligned}$$

where we used the chain rule in the last step.

Next we want to find the second derivative as well, since our goal is to get the Hamiltonian $\frac{\nabla_k^2 \Psi_T(\vec{r})}{\Psi_T(\vec{r})}$. We apply the product rule again to get

$$\begin{aligned}\nabla_k^2 \Psi_T(\vec{r}) &= \nabla_k^2 \phi(\vec{r_k}) \Big[\prod_{i\neq k} \phi(\vec{r_i}) \Big] \exp\Big(\sum_{j<m} u(\vec{r}_{j,m}) \Big) \\ &+ 2\nabla_k \phi(\vec{r_k}) \Big[\prod_{i\neq k} \phi(\vec{r_i}) \Big] \\ &\quad \exp\Big(\sum_{j<m} u(\vec{r}_{j,m}) \Big) \sum_{l\neq k} \nabla_k u(\vec{r}_{k,l}) \\ &+ \Big[\prod_i \phi(\vec{r_i}) \Big] \exp\Big(\sum_{j<m} u(\vec{r}_{j,m}) \Big) \\ &\quad \sum_{i\neq k} \nabla_k u(\vec{r}_{k,i}) \sum_{l\neq k} \nabla_k u(\vec{r}_{k,l}) \\ &+ \Big[\prod_i \phi(\vec{r_i}) \Big] \exp\Big(\sum_{j<m} u(\vec{r}_{j,m}) \Big) \sum_{l\neq k} \nabla_k^2 u(\vec{r}_{k,l})\end{aligned}$$

Dividing by the trial wave function $\Psi_T(\vec{r})$ on both sides we get

$$\begin{aligned}\frac{\nabla_k^2 \Psi_T(\vec{r})}{\Psi_T(\vec{r})} &= \frac{\nabla_k^2 \phi(\vec{r_k})}{\phi(\vec{r_k})} \\ &+ \frac{2\nabla_k \phi(\vec{r_k})}{\phi(\vec{r_k})} \sum_{l\neq k} \nabla_k u(\vec{r}_{k,l}) \\ &+ \sum_{i\neq k} \nabla_k u(\vec{r}_{k,i}) \sum_{l\neq k} \nabla_k u(\vec{r}_{k,l}) \\ &+ \sum_{l\neq k} \nabla_k^2 u(\vec{r}_{k,l})\end{aligned} \tag{4}$$

$$\nabla_k u(\vec{r_{k,l}}) = \frac{\partial \vec{r}_{k,l}}{\partial \vec{r}_k} \frac{\partial}{\partial \vec{r}_k} \Big(u(r_{k,l}) \Big)$$

We do a change of variables, we have $\frac{\partial r_{k,l}}{\partial \vec{r}_k} = \frac{\vec{r}_k - \vec{r}_l}{r_{k,l}}$, so

$$\frac{\partial}{\partial \vec{r}_k} = \frac{\vec{r}_k - \vec{r}_l}{r_{k,l}} \frac{\partial}{\partial r_{k,l}}$$

Now we get

$$\begin{aligned}\nabla_k u(\vec{r_{k,l}}) &= \frac{\vec{r}_k - \vec{r}_l}{r_{k,l}} \frac{\partial}{\partial \vec{r}_{k,l}} \Big(u(r_{k,l}) \Big) \\ &= \frac{\vec{r}_k - \vec{r}_l}{r_{k,l}} u'(r_{k,l})\end{aligned}$$

And moving on to the second derivative we get

$$\begin{aligned}\nabla_k^2 u(\vec{r_{k,l}}) &= \frac{1}{r_{k,l}} u'(r_{k,l}) + u''(r_{k,l}) + \frac{1}{r_{k,l}} u'(r_{k,l}) \\ &= u''(r_{k,l}) + \frac{2}{r_{k,l}} u'(r_{k,l})\end{aligned}$$

Next we can insert into Eq. (4)

$$\begin{aligned}\frac{\nabla_k^2 \Psi_T(\vec{r})}{\Psi_T(\vec{r})} &= \frac{\nabla_k^2 \phi(\vec{r_k})}{\phi(\vec{r_k})} \\ &+ \frac{2\nabla_k \phi(\vec{r_k})}{\phi(\vec{r_k})} \Big(\sum_{l\neq k} \frac{\vec{r}_k - \vec{r}_l}{r_{k,l}} u'(r_{k,l}) \Big) \\ &+ \sum_{i\neq k} \frac{\vec{r}_k - \vec{r}_i}{r_{k,i}} u'(r_{k,i}) \sum_{l\neq k} \frac{\vec{r}_k - \vec{r}_l}{r_{k,l}} u'(r_{k,l}) \\ &+ \sum_{l\neq k} u''(r_{k,l}) + \frac{2}{r_{k,l}} u'(r_{k,l})\end{aligned}$$

or, rewriting the third term, we finally end up with

$$\begin{aligned}\frac{\nabla_k^2 \Psi_T(\vec{r})}{\Psi_T(\vec{r})} &= \frac{\nabla_k^2 \phi(\vec{r_k})}{\phi(\vec{r_k})} \\ &+ \frac{2\nabla_k \phi(\vec{r_k})}{\phi(\vec{r_k})} \Big(\sum_{l\neq k} \frac{\vec{r}_k - \vec{r}_l}{r_{k,l}} u'(r_{k,l}) \Big) \\ &+ \sum_{i\neq k} \sum_{l\neq k} \frac{(\vec{r}_k - \vec{r}_i)(\vec{r}_k - \vec{r}_l)}{r_{k,i} r_{k,l}} u'(r_{k,i}) u'(r_{k,l}) \\ &+ \sum_{l\neq k} u''(r_{k,l}) + \frac{2}{r_{k,l}} u'(r_{k,l})\end{aligned} \tag{5}$$

Which is what we were supposed to show, with l instead of j. Next in order to obtain an actual expression for our problem we need to find $\nabla^2\phi/\phi$, $\nabla\phi/\phi$, u' and u''.

$$\begin{aligned}\frac{\nabla^2 \phi}{\phi} &= \nabla^2 e^{-\alpha(x^2+y^2+\beta z^2)} \\ &= 4\alpha^2(x^2+y^2+\beta^2 z^2) - 2\alpha(2+\beta)\end{aligned}$$

Moving on we have

$$\frac{\nabla \phi}{\phi} = -2\alpha(x, y, \beta z)$$

where the parenthesis are representing the vector $(x, y, \beta z)$ for particle k. Next for $u'(r_{k,l})$ we have that $u = \ln(f(r_{k,l}))$ by (3) such that

$$\begin{aligned}u'(r_{k,l}) &= \frac{\partial}{\partial r_{k,l}} u(r_{k,l}) = \frac{1}{f(r_{k,l})} f'(r_{k,l}) \\ &= \frac{1}{\left(1 - \frac{a}{r_{k,l}}\right)} \frac{a}{r_{k,l}^2}\end{aligned} \tag{6}$$

which we can simplify to

$$u'(r_{k,l}) = \frac{a}{r_{k,l}(r_{k,l}-a)}$$

swapping the indices to k, i makes no difference in the derivation of (6). Now all that remains before we can input back into (5) is $u''(r_{k,l})$, we use the previous result such that

$$\begin{aligned} u''(r_{k,l}) &= \frac{\partial}{\partial r_{k,l}}\Big(\frac{a}{r_{k,l}\big(r_{k,l}-a\big)}\Big) \\ \textit{we use:}\quad w &= r_{k,l}(r_{k,l}-a), \quad \frac{dw}{dr_{k,l}} = 2r_{k,l}-a \\ &= -\frac{a}{w^2}\frac{dw}{dr_{k,l}} = -\frac{a(2r_{k,l}-a)}{\big(r_{k,l}(r_{k,l}-a)\big)^2} \\ &= \frac{a(a-2r_{k,l})}{(r_{k,l}^2 - r_{k,l}a)^2} \end{aligned}$$

Inserting everything we get the analytic expression for the second derivative (5) in three dimensions with interaction we get

$$\begin{aligned} \frac{1}{\Psi_T}\nabla^2\Psi_T &= 4\alpha^2(x^2+y^2+\beta^2z^2) - 2\alpha(2+\beta) \\ &+ 2(-2\alpha(x,y,\beta z))\cdot\Big(\sum_{l\neq k}\frac{\vec{r_k}-\vec{r_l}}{r_{k,l}}\frac{a}{r_{k,l}(r_{k,l}-a)}\Big) \\ &+ \sum_{i\neq k}\sum_{l\neq k}\Big(\frac{(\vec{r_k}-\vec{r_i})\cdot(\vec{r_k}-\vec{r_l})}{r_{k,i}r_{k,l}} \\ &\cdot\frac{a}{r_{k,i}(r_{k,i}-a)}\frac{a}{r_{k,l}(r_{k,l}-a)}\Big) \\ &+ \sum_{l\neq k}\Big(\frac{a(a-2r_{k,l})}{(r_{k,l}^2-r_{k,l}a)^2} + \frac{2}{r_{k,l}}\frac{a}{r_{k,l}(r_{k,l}-a)}\Big) \end{aligned}$$

2.4 Rewriting Hamiltonian for 3D Interacting Case

As we have seen the Hamiltonian can be expressed as

$$\begin{aligned} H &= \sum_i^N\Big(\frac{-\hbar^2}{2m}\nabla_i^2 + V_{ext}(\vec{r}_i)\Big) + \sum_{i<j}^N V_{int}(\vec{r}_i,\vec{r}_j) \\ &= \frac{1}{2}\sum_i^N\Big(\frac{-\hbar^2}{m}\nabla_i^2 + m\big(\omega_{ho}^2(x_i^2+y_i^2)+\omega_z^2z_i^2\big)\Big) \\ &\quad + \sum_{i<j}^N V_{int}(\vec{r}_i,\vec{r}_j) \end{aligned}$$

introducing length and energy units of $r' = r/a_{ho}$, $h\omega_{ho}$ we rewrite as

$$\begin{aligned} H &= \frac{1}{2}\sum_i^N\Big(\frac{-\hbar^2}{ma_{ho}^2}\nabla_i'^2 + ma_{ho}^2\big(\omega_{ho}^2(x_i'^2+y_i'^2) \\ &\quad + \omega_z^2z_i'^2\big)\Big) + \sum_{i<j}^N V_{int}(\vec{r}_i,\vec{r}_j) \\ &= \frac{1}{2}\sum_i^N\Big(-\hbar\omega_{ho}\nabla_i^2 + \hbar\omega_{ho}\big((x_i'^2+y_i'^2) \\ &\quad + \frac{\omega_z^2}{\omega_{ho}^2}z_i'^2\big)\Big) + \sum_{i<j}^N V_{int}(\vec{r}_i,\vec{r}_j) \end{aligned}$$

where we can define $\gamma = \frac{\omega_z}{\omega_{ho}}$ and we have used the relation between ∇ and ∇' imposed by the length and energy units. γ is thus the ratio defining the ellipticity of the harmonic oscillator trap.

2.5 Variational Monte Carlo Method

To evaluate the ground state energy of our model computationally, we are going to use the *variational Monte Carlo method*. Monte Carlo methods use random sampling to make numerical estimations. We apply the Variational method by using a wave function $\Psi_T(\vec{r},\alpha)$ which depends on a parameter (or parameters) α, and minimizing the local energy (1) to find the optimal values for α. The variational Monte Carlo method is known as a Quantum Monte Carlo method, whose main goal is to find a good approximation of the quantum many-body problem. The difficult part of the VMC method is searching for the variational parameters that minimize the expectation value of the Hamiltonian

$$E[H] = \langle H\rangle = \frac{\int d\vec{R}\Psi_T^*(\vec{R},\alpha,\beta)H(\vec{R})\Psi_T(\vec{R},\alpha,\beta)}{\int d\vec{R}\Psi_T(\vec{R},\alpha,\beta)^*\Psi_T(\vec{R},\alpha,\beta)}$$

The lecture notes on the VMC method [2] show that this can be rewritten into

$$\begin{aligned} E[H] &= \int P(\vec{R})E_L(\vec{R})d\vec{R} \\ &\approx \frac{1}{N}\sum_i P(\vec{R}_i,\alpha,\beta)E_L(\vec{R}_i,\alpha,\beta) \end{aligned}$$

2.6 The Metropolis Algorithm

The Metropolis algorithm is a a method to sample a normalized probability distribution by a stochastic process. One chooses the probability distribution one wishes to minimize, for our problem it will be the expectation value for the energy. Next, sample a new possible state stochasticly. Accept the new state with probability $A_{i\to j}$ and repeat from the new state, deny with conjugate probability (1 - $A_{i\to j}$) and repeat from the

previous state. The lecture notes [2] goes into details of how to derive the required properties for the probabilities such that starting from any distribution, the method converges to the correct distribution. We are here concerned with the result that for very large n we require that $\vec{P}_i^{(n\to\infty)} = p_i$ which is expressed as

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

Where $A_{j\to i}$ represents the probability to accept the new state i from state j, $A_{i\to j}$ is the probability to accept the new state j from state i, $T_{i\to j}$ is the probability to sample the state j from the state i, and finally $T_{j\to i}$ is the probability to sample state i from j. The Metropolis choice is to maximize

$$A_{j\to i} = \min\Big(1, \frac{p_i T_{i\to j}}{p_j T_{j\to i}}\Big)$$

Doing this strictly stochasticly is known as the brute force Metropolis algorithm. The problem with this is that the direction of the stochastic process is uniformly distributed.

2.7 Importance Sampling

One way to ensure the stochastic process converges more efficiently is to replace the brute force walk with a walk biased by the trial wave function. This approach is based on the Fokker-Planck equation

$$\frac{\partial P}{\partial t} = D\frac{\partial}{\partial x}\Big(\frac{\partial}{\partial x} - F\Big)P(x,t)$$

as well as the Langevin equation.

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

Where F is a drift term, D is the diffusion coefficient, and η is a random variable. The Fokker-Planck equation is a partial differential equation describing the time evolution of the probability distribution P for a diffusion process. Where the Fokker-Planck describes the distribution of velocities in a diffuse gas, the Langevin equation describes the motion of the particles. The goal of importance sampling is to bias the walk appropriate to the trial function, and this is what is achieved by generating the trajectory of the stochastic process with the Langevin equation. Even more detail is found in the lecture notes [2] and this book [3] (p.219). It can be shown the drift term is given by

$$\vec{F} = 2\frac{1}{\Psi_T}\nabla\Psi_T$$

this is known as the so-called *quantum force*. This is the term responsible for pushing the stochastic process towards regions where the trial function is large. This increases the efficiency of the simulation compared to the brute force method.

2.7.1 Drift Force

Next we derive the analytic expression for the drift force which is used in importance sampling. We first deal with the single particle case without interaction. In one dimension we have

$$\begin{aligned} F &= \frac{2\nabla\Psi_T}{\Psi_T} \\ &= 2\frac{\nabla e^{-\alpha x_i^2}}{e^{-\alpha x_i^2}} \\ &= -4\alpha x_i \end{aligned}$$

expanding this expression to N particles we have

$$F = -4\alpha\sum_i x_i$$

Expanding to three dimensions with N particles we have

$$F = -4\alpha\sum_i \vec{r}$$

where $\vec{r} = (x_i, y_i, \beta z_i)$

2.7.2 Green's Function

Now with the *quantum force* in hand the transition probability is expressed in terms of Green's function

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

Replacing our brute force Metropolis algorithm $q(y,x) = |\Psi_T(y)|^2/|\Psi_T(x)|^2$ with

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

2.8 Steepest Descent

In the following section we briefly go over the steepest descent optimization method. A much more detailed explanation is available in the lecture notes [8]. In order increase the efficiency of our search for the optimal α value the steepest descent method was used. Even though it is the simplest form of gradient descent it is sufficient for our task at hand. Due to the convex nature of our trial function any minima we find in our search space is guaranteed to be a global minima [9]. From the lecture notes we define the derivative of the energy with respect to the variational parameter α as

$$\begin{aligned} \bar{E_\alpha} &= \frac{d\langle E_L[\alpha]\rangle}{d\alpha} \\ &= 2\Big(\langle\frac{\bar{\psi_\alpha}}{\psi[\alpha]}E_L[\alpha]\rangle - \langle\frac{\bar{\psi_\alpha}}{\psi[\alpha]}\rangle\langle E_L[\alpha]\rangle\Big) \end{aligned}$$

where we also defined

$$\bar{\psi_\alpha} = d\psi[\alpha]/d\alpha$$

This quantity $\bar{E_\alpha}$ is the quantity we wish to minimize in the steepest descent method. We do this by calculating the three expectation values in the expression above, and 'walking' our parameters a set distance γ 'along' this derivative.[1]

$$\alpha_{i+1} = -\gamma\alpha_i$$

2.9 Resampling Techniques

Resampling methods are used to draw out additional information about the data which is to be analyzed. For example, one can train a neural network on different samples from the training data and average the results in order to obtain information which would not be available from just one fit.

2.9.1 Blocking Method

The blocking method was made popular by Flyvberg and Pedersen [4]. In the blocking method one groups the 'training' samples into so called blocks of increasing size. Subsequently, if one computes the standard deviation for each block it is expected to see the variance increase as a function of the block size. However, the variance should increase until it reaches a plateau. Once this has happened we have reached a point where the sample from one block is no longer correlated with the sample from an adjacent block. The block size for the point of convergence essentially functions as an estimate for the *autocorrelation time* τ of our computational experiment. The autocorrelation time and its derivation is described in more detail in the lecture notes on statistical analysis [6].

$$\tau = 1 + 2\sum_{d=1}^{n-1} \kappa_d$$

where κ_d is known as the autocorrelation function

$$\kappa_d = \frac{f_d}{\mathrm{var}(x)}$$

this is also more closely looked at in the lecture notes. For our application we use our calculated local energies as our time series data X. We now define the so called *blocking transformations*. The main idea is to take the mean of subsequent elements from X and form a new vector X_1 and iterating to generate X_n where n is a large number. Thus we define

$$(X_0)_k \equiv X_k$$
$$(X_{i+1})_k \equiv \frac{1}{2}\big((X_i)_{2k-1} + (X_i)_{2k}\big)$$

this is where the term *blocking* stems from. The covariance we calculate is referred to as the *first order autocovariance*

$$\gamma_i = \frac{1}{n}\Big(\sum_{k=1}^{n}((X_i)_k - \bar{X})\Big)\Big(\sum_{k=0}^{n-1}((X_i)_k - \bar{X})\Big)$$

here i represents the current iteration in the blocking method and k is the index in the time series X. For these blocking transformations it is easier to work with time series data of length $n = 2^d$ where d is an integer. We are interested in finding out when the first order auto-covariance becomes zero. An new analysis of the blocking method described above was done by Marius Jonsson in his master thesis [**?**] and the code he produced is available at his github repository which is linked to in the blocking method lecture notes. The python version of his code was used on measurements of the cumulative local energy in the simulation.

3 Method

In this section we describe the implementation in C++ of the concepts introduced in the above section. For this project a code-skeleton was given, and since I have never written code in C++ before I opted to adapt this to solve the project. A lot of time went to fiddling with various intricacies regarding C++

3.1 Code Structure

The code skeleton used to solve this project was forked from `https://github.com/mortele/variational-monte-carlo-fys4411`, along with all the cmake stuff. Following we will describe the structure of the code skeleton.

3.1.1 Main

As is customary in C++ we have our main.cpp which is where we set up all our hyper-parameters and global variables to be used throughout the code, and is also where we set up our system, and run the Monte Carlo simulation. There are a bunch of variables set up in this file, and I won't list them all here, but some important ones for the physics of the system is: $\alpha = 0.5$, $\gamma = \beta = 2.82843$ and $a = 0.0043$. After initializing the various variables we come to four boolean 'flags' to specify what type of simulation we wish to run. Command line interfacing / reading these from file would be something to improve with further work on the project. We have *calculateNumericDerivative*, *interactionOrNot*,

[1] I hope this explanation is sufficient.

[2] Note that the interacting part could not be completed in the allotted time for the project, something I am quite dissapointed in myself for.

activateImportanceSampling and *writeToFile*, these explain themselves.[2] After these flags, we have the brunt of the simulation set up.

3.1.2 System

Next we move onto the file system.cpp along with its header file system.h, here is where we start getting into the functions of the program. We have a bunch of set and get type functions, which just set the variables defined in main.cpp as variables accessible through the System class we won't go into details on these here as they just set a variable, with the exception of:

```
void System::setHamiltonian(
    Hamiltonian* hamiltonian)
void System::setWaveFunction(
    WaveFunction* waveFunction)
void System::setInitialState(
    InitialState* initialState)
```

These three functions set up other classes which are used for the simulation. The first one, setHamiltonian takes in a Hamiltonian class instance and sets it such that it can be used through the system class throughout the calculation. The other functions do respectively the same thing but with WaveFunction and InitialState instances. To the actual calculation functions defined in our system file, we have

```
double System::quantumForce(
    int particleIndex,
    int dimensionIndex)
bool System::metropolisStep()
void System::runMetropolisStep(
    int numberOfMetropolisSteps)
```

The first function is the quantum force described in 2.7.1 which returns a number which keeps the sampling more within the area of the state space relevant for the wave function anzats and is used to evaluate Green's function. The second function is the step described in 2.6 where we do a random sampling and check whether this is larger or smaller than ratio of the new wavefunction from the proposed step squared over the old wavefunction squared, and returns a yes or no depending on whether the Metropolis test passes. The third function runs over the number of Metropolis steps sat in main.cpp and runs the metropolisStep function for each cycle. In the metropolisStep function we first run into the WaveFunction Class which we will describe next.

3.1.3 WaveFunction Class

The WaveFunction class is the superclass which lets us define any trial wavefunction we wish, it has three virtual double functions which all subclasses must implement.

```
virtual double evaluate(
    std::vector<class Particle*> particles) = 0
virtual double computeDoubleDerivative(
    std::vector<class Particle*> particles) = 0
virtual double alphaDerivative(
    std::vector<class Particle*> particles) = 0
```

for this project the subclasses: *simplegaussian* and *interacting* were implemented, but in principle adding different anzatses is possible due to the modularity of the code. As previously stated, all subclasses must implement the functions descibed above. The evaluate function does what it's named and evaluates the wavefunction. The computeDoubleDerivative function is necessary to compute the local energy (1) as the laplacian appears in the Hamiltonian.

3.1.4 Hamiltonian Class

Much the same as with the WaveFunction class we also have a superclass called Hamiltonian which has two virtual double function

```
virtual double computeKineticEnergy(
    std::vector<Particle*> particles)
virtual double computePotentialEnergy(
    std::vector<Particle*> particles)
```

in addition to the two functions above the Hamiltonian class also has a function *computeLocalEnergy* which allows us to differ between numeric and analytic differentiation such that it is possible to implement wavefunction anzatses that do not necessarily have an analytic expression for the Laplacian. Two subclasses of the Hamiltonian class were implemented in this work, corresponding to the two subclasses of the WaveFunction class described above.

3.1.5 Particle Class

Now in both the sections above we have passed in a vector with the type class Particle. This particle class is defined in the file particle.cpp, and the class is very simple. It contains four functions where one is a get-function and one is a set-function. The two 'non-trivial' are the two following functions

```
void setPosition(
    const std::vector<double> &position)
void adjustPosition(
    double change, int dimension)
```

These simply do what the name describes, and are used throughout the code in system.cpp and are accessed other places in the code through the system class.

3.1.6 Sampler Class

Moving on in our description of the implementation of the variational Monte Carlo simulation. In the *runMetropolisSteps* function we let the system equilibrate according to our equilibration fraction which was set using the *setEquilibrationFraction* method. Once the system has equilibrated we start sampling the system using the sampler class. The sampler class consists of a couple of set/get-functions, the *sample* function and *printOutputToTerminal / printEnergiesToFile*. It is in the sample function the important part of the calculations utilizing the WaveFunction and Hamiltonian classes are performed. Here we calculate the local energy, alpha derivative for gradient descent and in principle many other observables of the system. In this work we focus on the local energy for our analysis.

3.2 Blocking Analysis

There is not too much to report on in the implementation of the blocking analysis, as we were instructed to use blocking code made by Marius Jonsson [**?**] to do our analysis. The reason being just implementing the same thing would not really offer too much learning outcome. The blocking method needs the number of samples collected to be a power of two. For our analysis 2^23 points were used, and to for an easy fix regarding the equilibration fraction we simply sat the fraction to zero for this analysis. This will have some result on our result, but since we were running for this large a number of cycles the effect should be negligible. So to save the data our sampler class simply has a savetofile flag which can be activated in the main.cpp file and then it just saves the energy data to a file in the data folder of the project. Running Marius' code on this data we obtain a value for the standard error of $9.47183 \cdot 10^{-5}$

4 Results

Below are the results generated by the code discussed in the sections above. All the results were generated on Windows 10 Home Edition using Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz, 3096 Mhz, 6 Core Processor. All the data used to generate the following plots can be found in the data folder of the project on Github. The file plotting.py is where the plots are generated. With future work on this project one area of improvement would be to make the plotting functionality more modular and easier to use for someone who isn't the author of the program. As of now, the filename which is looked up is hard-coded such that it must be changed in the file to look at another set of data.

4.1 Brute Force Metropolis

For our initial test of the code we used α as the only variational parameter and we performed the calculations for $N \in [1, 10, 100, 500]$ atoms in one-, two- and three-dimensions.

(a) $D = 1$

N	E	$\bar{t}$	σ_t
1	0.5	581ms	4.60ms
10	5	2495.2ms	535.46ms
100	50	18252.4ms	95.06ms
500	250	88560.8ms	600.69ms

(b) $D = 2$

N	E	$\bar{t}$	σ_t
1	1	673.8ms	4.53ms
10	10	3837ms	848.17ms
100	100	19350.8ms	73.55ms
500	500	89351ms	1062.30ms

(c) $D = 3$

N	E	$\bar{t}$	σ_t
1	1.5	585.6ms	5.47ms
10	15	2241.2ms	4.21ms
100	150	18296.2ms	74.89ms
500	750	89315.4ms	1145.79ms

Table 1: All of these runs were for $N_{mc} = 1000000$ and $\alpha = 0.5$. We see that the computation time increases with increasing the number of particles, I also expected the computation time to increase with the number of dimensions, but this is not what was seen. This makes no sense to me. Tests to check if the loops run for more than one dimension was added and the indices in all my loops run from 0 to 2 when i run the program in 3D, which they should, which should take three times the time as the 1D-case, however for some reason I could not find my program does not work like this.

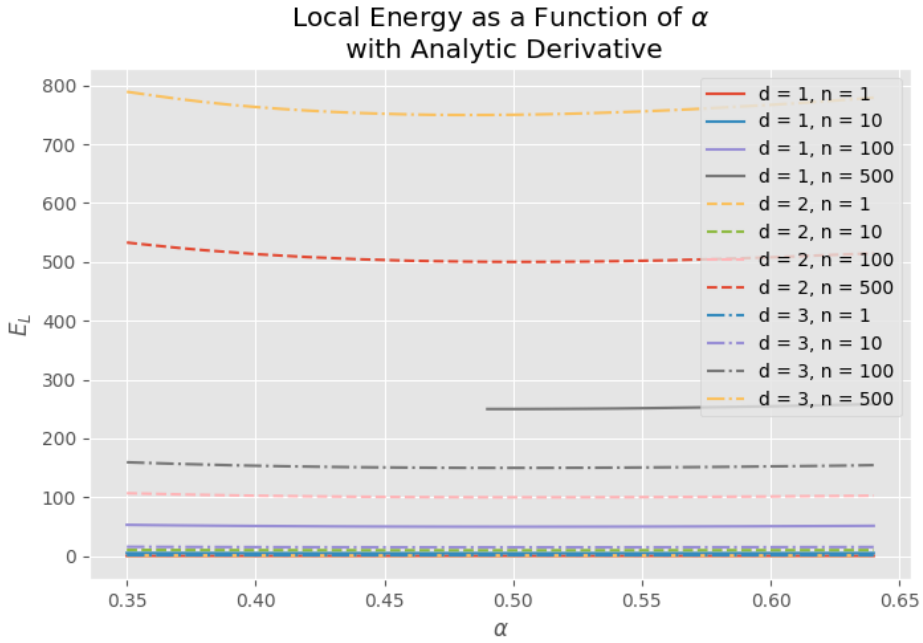


Figure 1: The local energy as a function of the variational parameter α, over the number of dimensions and number of particles in the system. For this simulation $N_{mc} = 500000$ as a compromise for the run-time on the total simulation. The program was ran in parallel on Windows using mpiexec.exe from the command line to obtain better statistics. For some reason some of the data for $D = 1, N = 500$ was lost in the simulation.

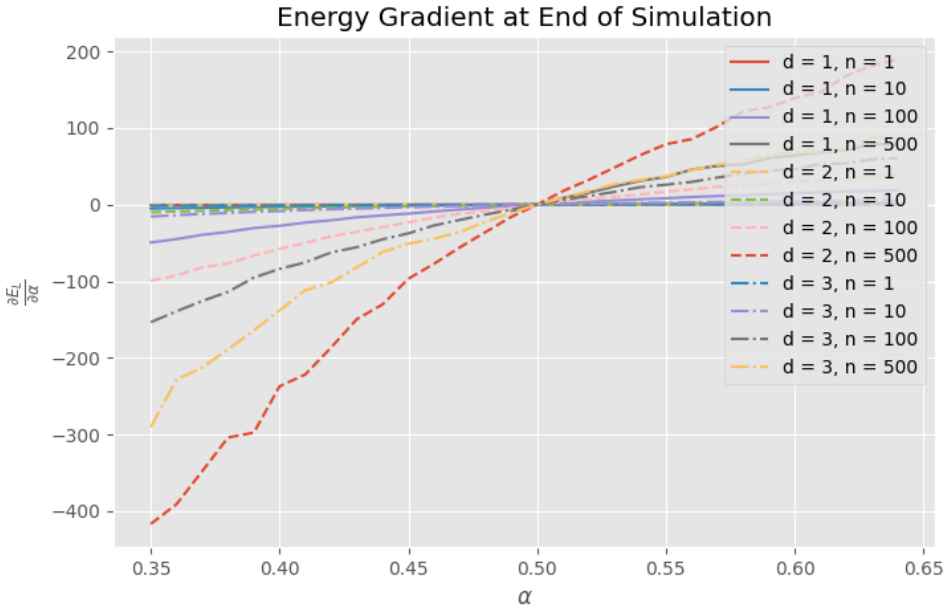
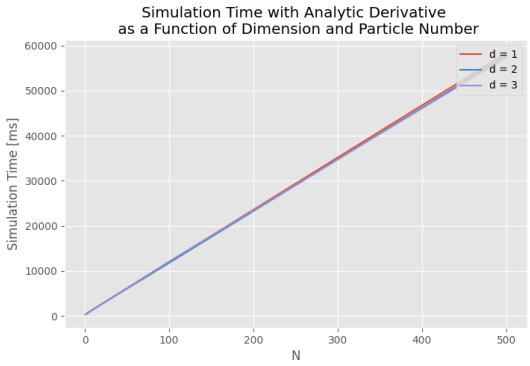


Figure 2: The Energy Gradient w.r.t α at the end of the simulation. We see all of them cross over at 'the correct' α-value $\alpha = 0.5$. This graph shows why it is extremely computationally beneficial to use the simple gradient descent method for this problem.

(a) The simulation time, using the analytic expression for the derivative as a function of the number of particles and dimension of the system.



(b) Zoomed picture

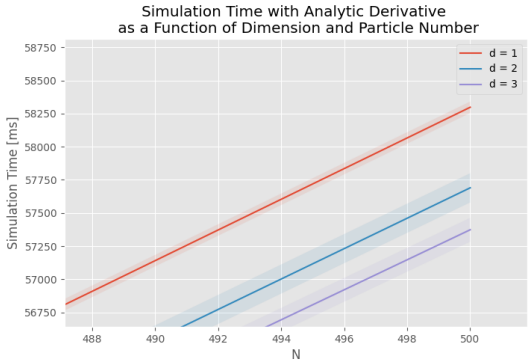


Figure 3: As mentioned the runs were ran in parallel the most naive way possible, but it does prove effective for our problem as running the code multiple times will provide better statistical analysis.

	D	1	2	3
N				
1		113.28ms	7.94ms	8.99ms
10		183.49ms	173.48ms	28.93ms
100		428.98ms	277.46ms	434.76ms
500		2645.56ms	1021.06ms	1249.60ms

Table 2: Table of the standard deviation of the simulation time measurements for $D = 1, 2, 3$ and $N = 1, 10, 100, 500$. All the data presented here was with $N_{mc} = 500000$. The values here are averaged over 5 parallel runs.

4.2 Importance Sampling

Below are the results of the simulations after adding importance sampling. The most important result from the importance sampling implementation is that an increase in the acceptance rate of approx 37% was observed from the brute force method.

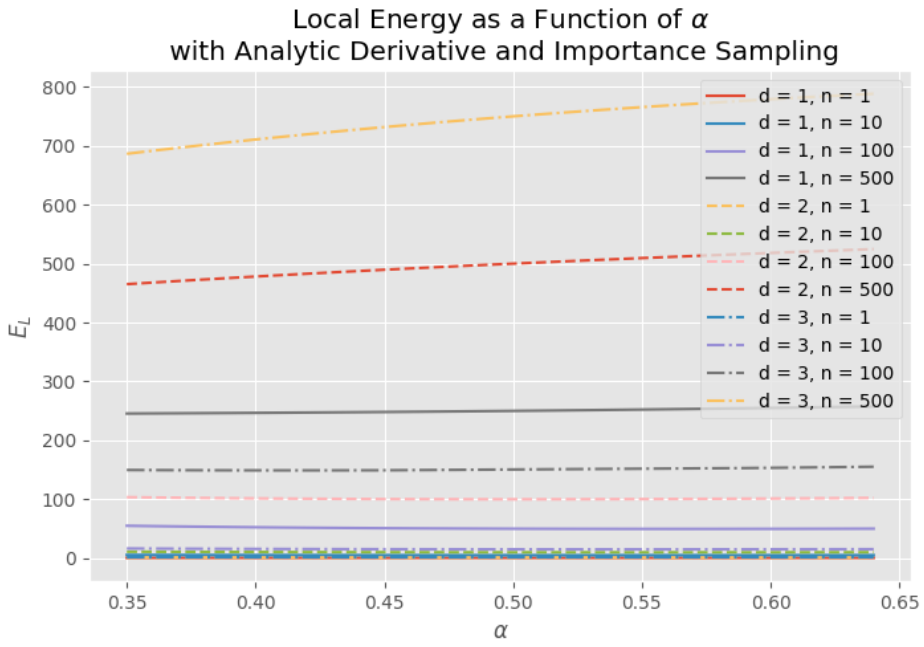


Figure 4: The local energy as a function of α with importance sampling. It is obvious there is some error here for higher N's. Have not been able to exactly pinpoint the source of this error. We see the local energy curves have been 'flipped' upside down for higher N's

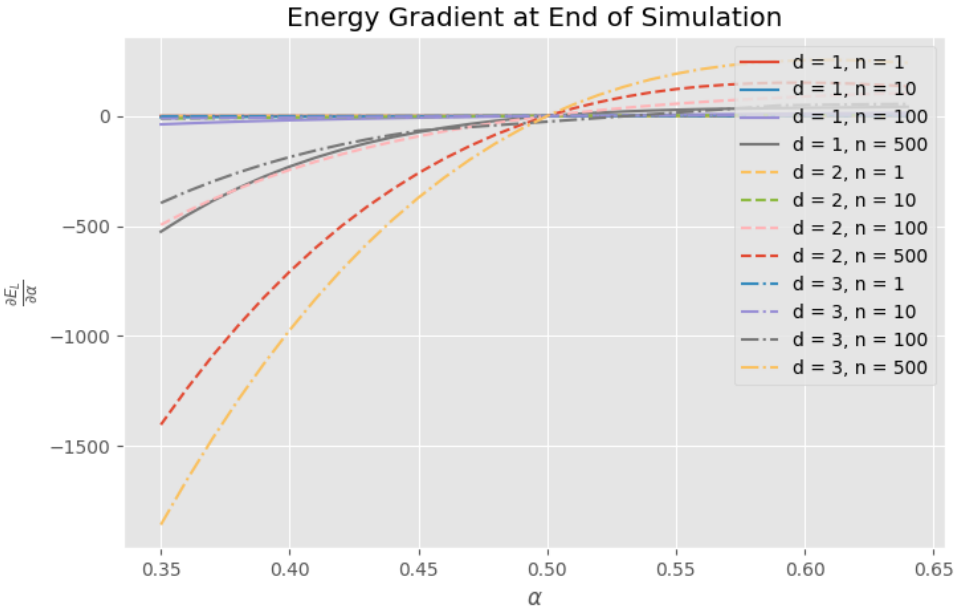


Figure 5: The energy gradient also shows the same weird behavior. Where the higher N lines have been skewed a bit from their supposed zero-point at $\alpha = 0.5$

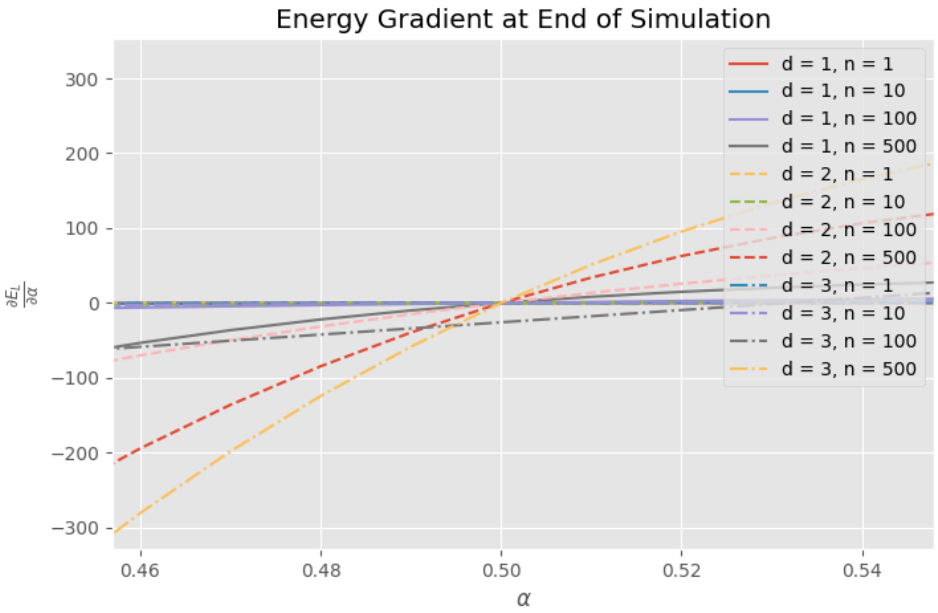
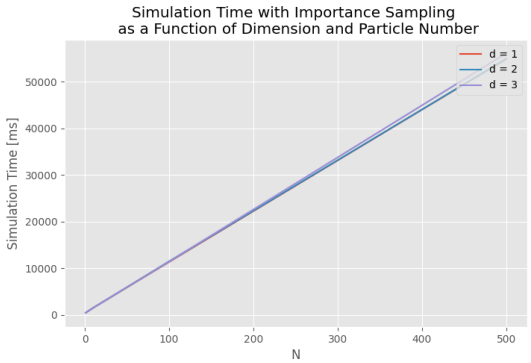


Figure 6: Zoomed version of the above plot

(a) The simulation time, using the analytic expression for the derivative as a function of the number of particles and dimension of the system.



(b) Zoomed picture

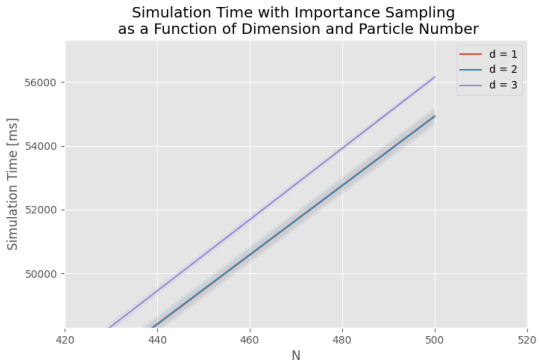


Figure 7: For this one the situation is flipped against our brute force runs. In other words the result observed with importance sampling is closer to what I expected.

D	1	2	3
N			
1	119.91ms	3.83ms	94.30ms
10	179.48ms	186.47ms	196.86ms
100	230.80ms	266.83ms	52.05ms
500	539.87ms	377.47ms	1398.97ms

Table 3: Table of the standard deviation of the simulation time measurements for $D = 1, 2, 3$ and $N = 1, 10, 100, 500$. All the data presented here was with $N_{mc} = 500000$. The values here are averaged over 5 parallel runs.

4.3 Numeric Derivative

Below are the results of the code when running with numeric derivative. This method is nice to test against, as it can be implemented the same regardless of the trial function in question.

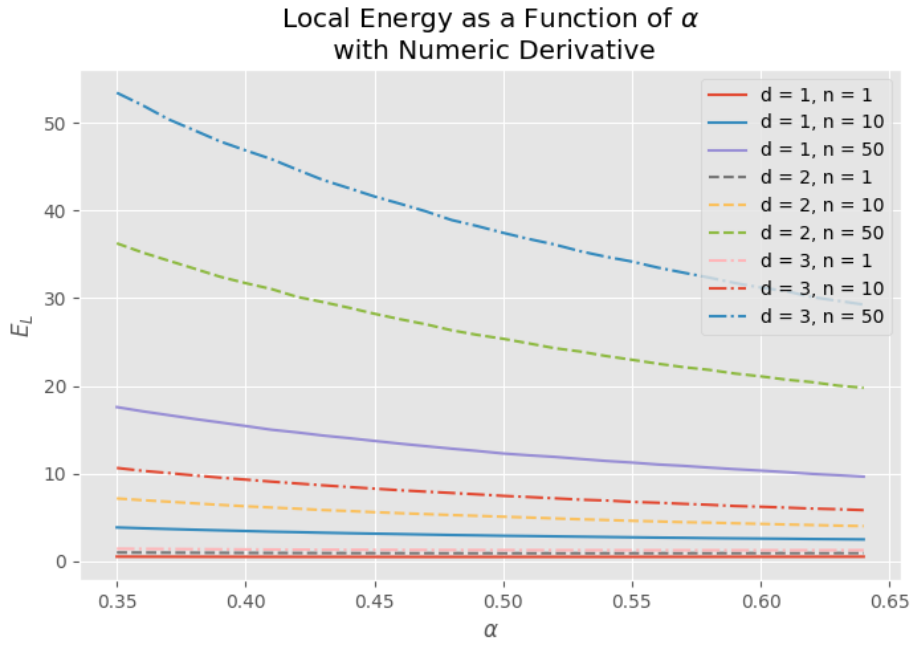


Figure 8: Full energy plot when using numeric derivative, with $N_{mc} = 500000$. We see some strange behaviour of the local energy for higher values of N.

It was necessary to limit the number of particles when using the numeric method to be able to obtain reasonable results in a reasonable amount of time. Running longer would hinder me in other subjects as there was no access to cluster-computing this year due to problems with the cooling system. For this reason spending a very long time simulating results for one course proved to be impossible.

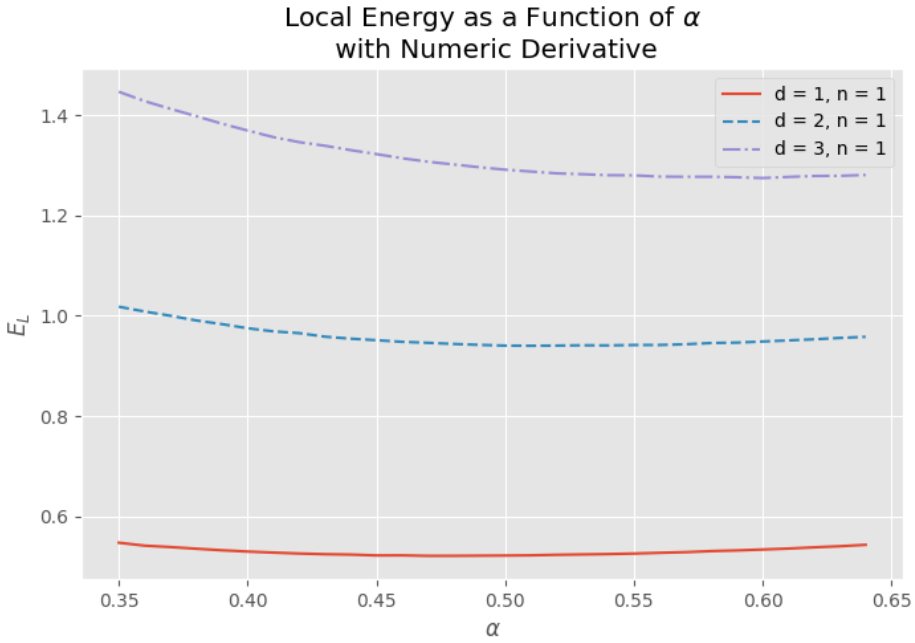


Figure 9: Limited view of the above data for clarity. We see that the $D = 1, 2$ cases reproduce the result from the analytical derivative while there turns out to be some problem for $D = 3$.

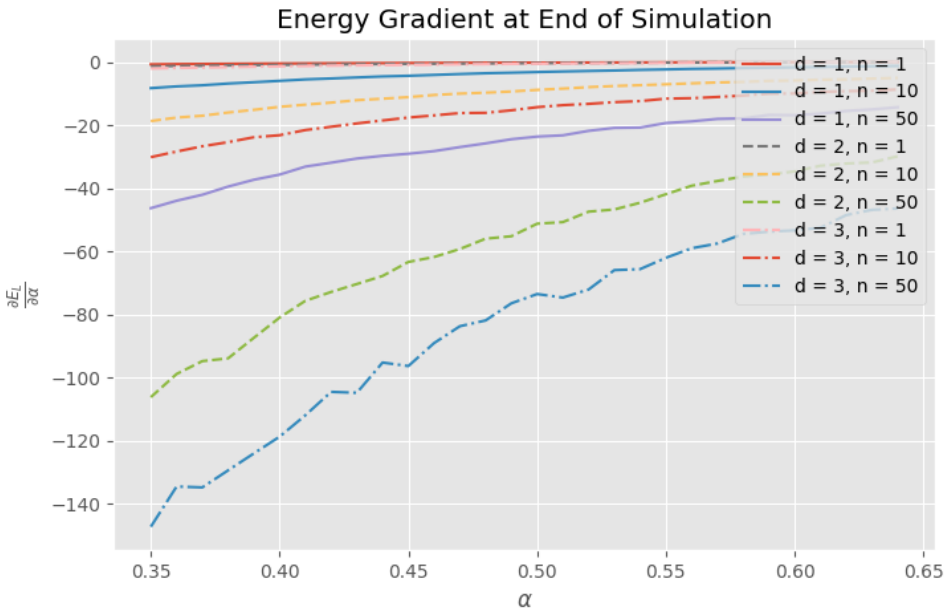


Figure 10: Energy gradient w.r.t α at the end of the simulation.

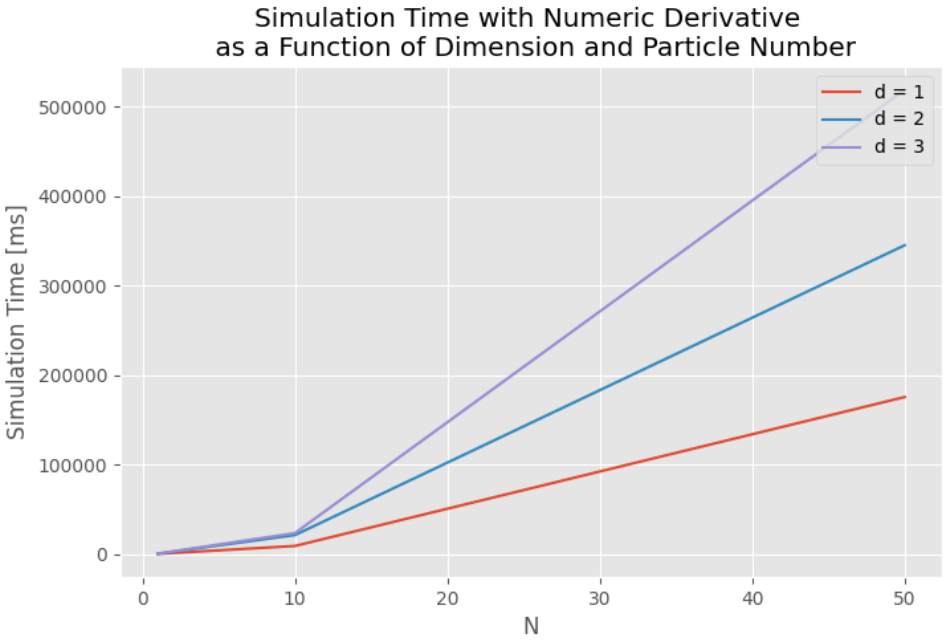


Figure 11: Caption

	D	1	2	3
N				
1		160.34ms	1112.39ms	8794.88ms
10		256.73ms	27019.77ms	15506.98ms
50		362.01ms	3475.47ms	20761.70ms

Table 4: Table of the standard deviation of the simulation time measurements for $D = 1, 2, 3$ and $N = 1, 10, 50$. All the data presented here was with $N_{mc} = 500000$. The values here are averaged over 5 parallel runs.

4.4 Gradient Descent

In the following section are the results when optimizing for the variational parameter α using a simple steepest descent method. The learning rate has to be tuned according to the dimension and number of particles. To produce this graph learning rates $\gamma = 0.05$ was used for $(D = 1, N = 10)$, $(D = 2, N = 100)$, $(D = 3, N = 10)$ and $(D = 3, N = 100)$; $\gamma = 0.1$ was used for $D = 2, N = 500$; $\gamma = 0.15$ was used for $(D = 3, N = 1)$ $(D = 3, N = 500)$; $\gamma = 0.3$ was used for $(D = 1, N = 100)$, $(D = 1, N = 500)$, $(D = 2, N = 10)$ and finally $\gamma = 0.35$ was used for $D = 1, N = 1$ and $D = 2, N = 1$

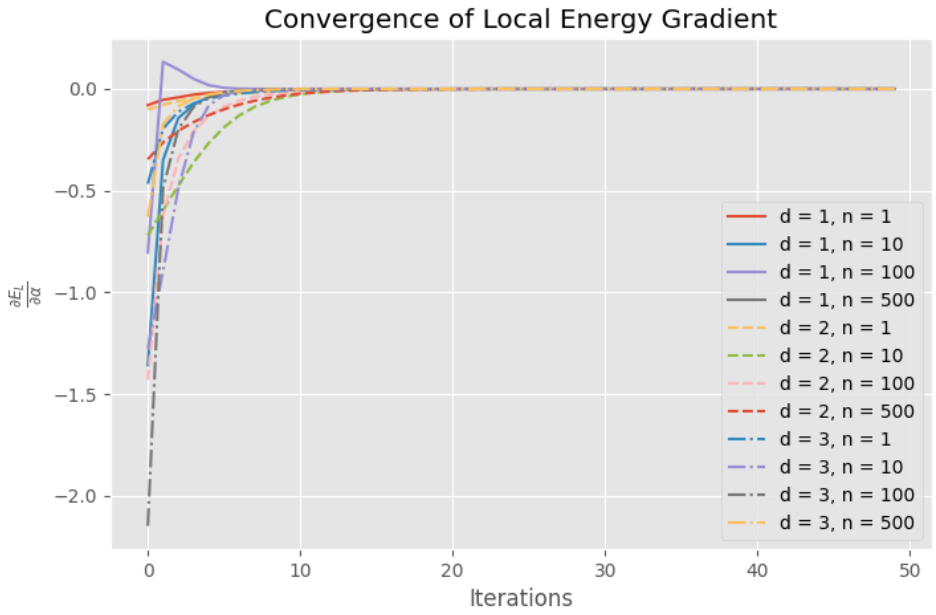


Figure 12: Convergence of the Gradient of the local energy w.r.t α as a function of the steepest descent iteration.

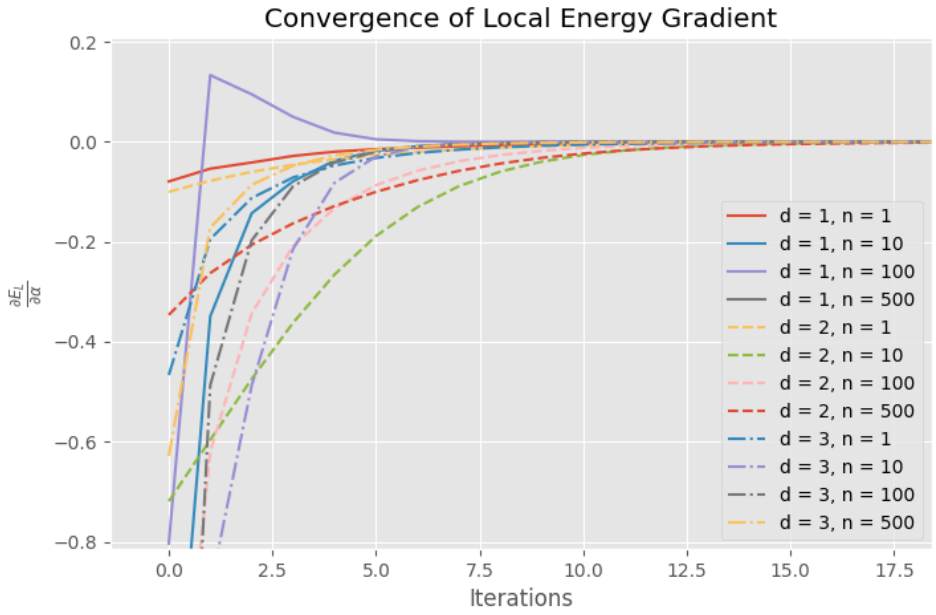


Figure 13: Zoomed in version of the figure above.

5 Discussion

In this section we look through the results presented in the previous section and discuss in a slightly more informal manner the interpretation of the results in addition to ways to improve the quality of the work given future work on the project.

5.1 Discussion of Results

For our experiments with just the brute force metropolis algorithm we see that we indeed get reproduce the analytical answers using our implementation of the variational monte carlo. Further we obtained the optimal value for $\alpha_{\text{opt}} = 0.5$. However, as the name suggests the number of cpu-cycles used is way more than opti-

mal. This was remedied by utilizing the power of gradient methods. An additional point is that with only one variational parameter it is feasible to run a grid search as was done in this report. When increasing the number of variational parameters the search space quickly grows out of control. Even for this simple formulation it was necessary to limit which parameters were searched over. Additional ones that would be interesting to study could be the Metropolis step size, importance sampling δt, the numerical derivative step size, the equilibration fraction. If one adapted to an even more complicated setup, i.e. fermionic simulations or quantum flavor or chromo-dynamics then you would have all of the aforementioned parameters in addition to the myriad of other parameters I assume one would have to add for such simulations[3] Now something to mention is that for our problem the steepest descent method suffices as our parameter space is convex and therefore guaranteed to have only a global minima, which one would converge to. For more complicated simulations this 'cost-landscape' is not guaranteed to be globally convex, much in the same way as for many machine learning problems. Thus more complicated optimization methods would have to be implemented, for instance: stochastic gradient descent (SGD), or maybe borrowing more from the machine learning field using a more advanced version of the standard SGD optimization algorithm such as the famous ADAM-optimizer could prove fruitful [10].

Now back to the actually implemented solutions, the steepest descent was in all cases able to find the minima for the given trial function in approximately 10 iterations of the algorithm. Note that each of the runs through the simulation only used $N_{mc} = 10000$ vs. the $N_{mc} = 500000$ of the brute force method. Although admittedly little optimization was used to figure out whether the brute force method would provide good results at lower step numbers than this. However, it was very evident that the brute force method was not at all able to obtain the local energy minima when running with the same number of steps as used in the steepest descent method. The gradient method was ran for 50 iterations every time it was ran, and there should have been added some check to see if the gradient had reached a sufficiently low absolute value to save on cycles.[4] Even with running approximately 40 unnecessary iterations of the steepest descent method, it still vastly outperformed the brute force search for every dimension and for every particle number.

5.2 Adding Interaction

In order to add interactions into the calculation a new Hamiltonian sub-class was implemented in addition to a WaveFunction sub-class. However, due to time constraints and a weird bug which I found it very hard to debug I was not able to correctly implement the interaction functionality into my final project code successfully. This is the most glaring part of the project which I feel bad for not being able to complete. I do have implementations of the sub-classes mentioned above, but there is some weird bug I did not have the time management skills to work out.

5.3 Bugs

The plots in figures 4, 6, 8, show bugs which I only saw when I started running the final big simulations. This is a learning opportunity for me as I was only testing with the smallest / easiest cases throughout my development, and extrapolated from correct results there that it would also be correct for more particles. As such, doing more complete tests during the development phase of my coding will something I will focus more on in future projects of this size, and most definitely in my master's thesis work next year.

Just to mention why I think these three figures represent bugs in my code: for 4 we see that the local energy function diverts from the expected form for higher particle cases, the bug in 6 does look like a different bug than the previous one, my reasoning behind this statement is that the energy gradient for $D = 3, N = 500$ for example does behave as expected, even though the local energy does not. As stated these bugs were found during the absolute last phase of my work on this project and other obligations / new projects have made it so that spending more time debugging on this project is unfeasible.

There is also the problem of the simulation times for the analytical derivative simulation does not scale as expected. Either I have done an un-intentional optimization or there is something wrong in my code, as there seemed to be minimal difference between the simulation times for $1, 2, 3$ dimensions 3, 7. The situation looks a bit better with importance sampling where at the very least the three dimensional case takes longer. For the numerical derivative calculation we do see the expected behavior, but this would be due to the nature of taking this derivative.

5.4 Code Improvements and Further Work

Disregarding the above comments on a few bugs, there are a number of features which would make my implementation a lot more elegant / programmatically pleasing. One such improvement would be being able to read all hyper-parameters from an external file, this would allow for running the code with new values without having to recompile the whole project. This change was

[3] Flavor and chromodynamics simulations are something I don't know that much about, but undoubtedly brute force methods would not be sufficient when performing such simulations.

[4] Adding this would be very easy, but since it worked so well with such a low step number it was deemed a bit unnecessary to do.

not as pressingly needed as it might have been with future work, as it only takes a few seconds to recompile. However, with a potentially much larger code-base the compilation time could become a real problem. An attempt to implement reading variables from file was implemented. Unfortunately, this being my first time ever coding in c++, spending a lot of time trying to implement a quality of life feature such as this was not at the top of the list of my priorities. It would be nice though. Another feature would be having some check to see if the local energy / local energy gradient had reached a sufficiently small delta between steps and stopping the simulation short if this was the case. Again, the value of this change was not realized before the end part of the project where it was apparent that running hours of simulations on my only available computer would be a hindrance to my progress in other courses. Thus it would have been a smart thing to spend a bit of time on. With further development on this code this would definitely be a priority.

As for further work on the topic, there is the possibility of adding flavor/chromo-dynamical elements to the models, fermionic Pauli exclusion and spin statistics, using neural networks to learn the trial functions, or even more cool using quantum machine learning and quantum boltzmann machines to learn or even quantum-mechanically simulate the systems.

5.5 Learning Outcome

In this project I have learned so much it is hard to start listing of things. Firstly, getting an insight into C++, even though it was very frustrating at times, has proved very instructive in understanding how more serious numerical projects are ran. Having pretty much only coded in Python for the whole of my undergrad, it does feel like introducing C++ a bit earlier in the curriculum for physics students interested in numerical simulations would be something that the department of physics should look a bit into. However, some of the greatest stresses of this project is learning to browse around all the millions of articles on how to do simple things in C++. One anecdote I had was trying to take the difference of two arrays, like taking the difference of vectors. Maybe there are libraries for doing things like this simpler, but it literally took me over an hour to figure out how to define lists, how to run through their elements, this is different in different versions of C++. Now obviously this is something that would not be a problem for someone who is familiar with the programming language, and definitely would not have caused me any issue in Python, but there were what feels like countless such problems throughout the work on the project. So it felt like a lot of the time spent on the project was not related to the subject matter of the course, quantum mechanical computational physics, but more fiddling with aspects of C++. Do note that, the teachers have been excellent at providing help[5]. So thanks a lot for being great teachers Øyvind and Morten!

Continuing on our list, I have learned about the variational Monte Carlo method, and how we use this to make approximations to ground state wave functions in quantum mechanics. This seems like it is an exciting area of study that I have only really began to scratch the surface of really understanding. In addition I have learned various methods to optimize variational calculations, from using importance sampling, which is related to another interesting field of study: transport theory. Something another course I am currently taking: FYS4460 also somewhat scratches the surface of. Importance sampling allows us to spend more cycles in parts of Hilbert space where our trial function is larger. Further into our optimization methods we have learned about both gradient methods and stochastic gradient methods. I did already know a lot of this information from FYS-STK4155 and some from FYS3150 in addition to my own interest into the subject, primarily due to my interest in machine learning. Finally, even though I only used the most basic form of parallel computation in this project, running simultaneous instances of the whole program, as opposed to actually making the loops etc. parallel it is still good practice and refreshment of knowledge from previous subjects.

5.6 Comments on Difficulties and Potential Improvements to Project

Honestly the biggest difficulty for me in this project came from just doing basic things in C++. This was mentioned a bit at the start of the previous subsection, but simple stuff like imports, code structures. However most of these problems were solved by coming to the zoom labs, but one problem this semester which is completely out of our control is having to do problem solving online. I feel like the time between being able to ask questions is significantly higher when doing online labs.

Another comment for how the project is structured is: I felt like a lot of the time during my work on this project I sort of "forgot" the underlying physics as it sort of got lost in the mathematical language. So it might be helpful to mention more intuitions the instructor has for these very complex topics. Try to help us remember we're actually simulating a gas which is being kept together by a magnetic field / laser. Maybe find some good YouTube videos which visually explain the topic? Anyway this is just a thought.

Another difficulty specifically for me this semester was the way I was admitted into the courses, initially thinking I was going to a completely different country and then starting out a month behind. However, this is

[5] A lot of the time on questions I myself would deem as stupid.

of course nothing to do with the course at all and just a bi-product of the pandemic situation we're still in the middle of.

One thing I thought was really nice was having a code skeleton to work of.

6 Conclusion

This work has been on using the quantum variational principle, and Monte Carlo simulations to simulate many-body bosonic gases in a Bose-Einstein condensate. The fact that I am able to write this sentence and understand every word is a testament to the learning outcome of this project. It has been a great learning experience for my physics, mathematical and programmatical understanding. One regret I have is that I was not able to completely solve the project tasks to the standard I wished I could, this is due to various factors, but I will still look fondly back at my work on this project. Thanks again for being great teachers Morten and Øyvind!

A Interaction Implementation Problems

The following is just a copy paste of my piazza post detailing a bit about my problem with the interaction implementation. Or at least what I saw as the problem with my implementation. As such the formatting leaves a bit to be desired but I thought it helpful to include in my report.

Hi running into a problem with the implementation of the interaction. The problem arises with one particle in one dimension for my interacting code, which should just give the exact same answer as the uninteracting code. My two codes produce the exact same answer for the first 14 mc cycles, but after this they diverge. I don't know how to even begin finding the cause of this, have put prints throughout the whole calculation. I ran and tested to see where they diverge from each other and it is once we come to mc cycle number 14 (after equilibration)

PRINTOUT FROM INTERACTION:

metrostep: 4, interaction, i:0, d:0, term:0.264917
metrostep: 5, interaction, i:0, d:0, term:0.368217
metrostep: 6, interaction, i:0, d:0, term:0.80102
metrostep: 7, interaction, i:0, d:0, term:0.675074
metrostep: 8, interaction, i:0, d:0, term:0.650788
metrostep: 9, interaction, i:0, d:0, term:0.397991
metrostep: 10, interaction, i:0, d:0, term:0.41959
metrostep: 11, interaction, i:0, d:0, term:0.115665
metrostep: 12, interaction, i:0, d:0, term:0.160604
metrostep: 13, interaction, i:0, d:0, term:0.091369
metrostep: 14, interaction, i:0, d:0, term:0.252861
metrostep: 15, interaction, i:0, d:0, term:0.28035
metrostep: 16, interaction, i:0, d:0, term:0.0601848
metrostep: 17, interaction, i:0, d:0, term:0.260429
metrostep: 18, interaction, i:0, d:0, term:0.260429
metrostep: 19, interaction, i:0, d:0, term:0.0542116
metrostep: 20, interaction, i:0, d:0, term:0.190942
metrostep: 21, interaction, i:0, d:0, term:0.190942
metrostep: 22, interaction, i:0, d:0, term:0.308438
metrostep: 23, interaction, i:0, d:0, term:0.840831
metrostep: 24, interaction, i:0, d:0, term:0.571121
metrostep: 25, interaction, i:0, d:0, term:0.571121
metrostep: 26, interaction, i:0, d:0, term:0.892799
metrostep: 27, interaction, i:0, d:0, term:0.892799
metrostep: 28, interaction, i:0, d:0, term:1.07374
metrostep: 29, interaction, i:0, d:0, term:1.07374
metrostep: 30, interaction, i:0, d:0, term:0.668366
metrostep: 31, interaction, i:0, d:0, term:0.873743

PRINTOUT FOR NO INTERACTION:

metrostep: 4, no interaction, i:0, d:0, term:0.264917
metrostep: 5, no interaction, i:0, d:0, term:0.368217
metrostep: 6, no interaction, i:0, d:0, term:0.80102
metrostep: 7, no interaction, i:0, d:0, term:0.675074
metrostep: 8, no interaction, i:0, d:0, term:0.650788
metrostep: 9, no interaction, i:0, d:0, term:0.397991
metrostep: 10, no interaction, i:0, d:0, term:0.41959
metrostep: 12, no interaction, i:0, d:0, term:0.160604
metrostep: 13, no interaction, i:0, d:0, term:0.091369
metrostep: 14, no interaction, i:0, d:0, term:0.252861
metrostep: 15, no interaction, i:0, d:0, term:0.28035
metrostep: 16, no interaction, i:0, d:0, term:0.0601848
metrostep: 17, no interaction, i:0, d:0, term:0.260429
metrostep: 18, no interaction, i:0, d:0, term:0.0874611
metrostep: 19, no interaction, i:0, d:0, term:0.000839796
metrostep: 20, no interaction, i:0, d:0, term:0.0591986

metrostep: 21, no interaction, i:0, d:0, term:0.0170749
metrostep: 22, no interaction, i:0, d:0, term:0.0699012
metrostep: 23, no interaction, i:0, d:0, term:0.410284
metrostep: 24, no interaction, i:0, d:0, term:0.24316
metrostep: 25, no interaction, i:0, d:0, term:0.235402
metrostep: 26, no interaction, i:0, d:0, term:0.473158
metrostep: 27, no interaction, i:0, d:0, term:0.305034
metrostep: 28, no interaction, i:0, d:0, term:0.439913
metrostep: 29, no interaction, i:0, d:0, term:0.439913
metrostep: 30, no interaction, i:0, d:0, term:0.214584
metrostep: 31, no interaction, i:0, d:0, term:0.357786

The loops diverge between metrostep 17 and 18. Now the way i see it they should be producing the exact same thing, all im doing is adding up the positions in one dimension for one particle. This has been trivial for the whole project but now suddenly something's going wrong and I think ive been staring at it for too long.

Now it is important to mention that of course these values should be different for the interacting and non-interacting case. However this above test was ran using the parameters a and γ which make the interaction code and the non-interacting code technically equal, or at least they should be. The reason I mention this weird discrepancy is that following this the code for the interacting case blows up.

References

[1] Einstein, A (10 July 1924). "Quantentheorie des einatomigen idealen Gases", Königliche Preußische Akademie der Wissenschaften. Sitzungsberichte: 261–267.

[2] Hjorth-Jensen, M, Department of Physics, University of Oslo, Jan 14, 2020, http://compphysics.github.io/ComputationalPhysics2/doc/pub/vmc/html/vmc.html

[3] N.G. Van Kampen, Stochastic Processes in Physics and Chemistry, 3rd edition, 21st March 2007

[4] H. Flyvberg, H.G. Pedersen, The Journal of Chemical Physics, 31 August 1998 https://doi.org/10.1063/1.457480

[5] M. Jonsson, Physical Review E, 15 October 2018, Department of Physics, University of Oslo, N-0316 Oslo, Norway https://journals.aps.org/pre/abstract/10.1103/PhysRevE.98.043304

[6] Hjorth-Jensen, M, Department of Physics, University of Oslo, March 5th, 2020, http://compphysics.github.io/ComputationalPhysics2/doc/pub/statanalysis/html/statanalysis.html

[7] Hjorth-Jensen, M, Department of Physics, University of Oslo, Last Visit: 17th March 2021 https://github.com/CompPhysics/ComputationalPhysics2/blob/gh-pages/doc/pub/vmc/programs/c%2B%2B/vmcsolver.cpp#L52

[8] Hjorth-Jensen, M, Department of Physics, University of Oslo, Last Visit: 14 April 2021 http://compphysics.github.io/ComputationalPhysics2/doc/pub/week5/html/week5.html

[9] Hjorth-Jensen, M, Department of Physics, University of Oslo, Last Visit: 14 April 2021 http://compphysics.github.io/ComputationalPhysics2/doc/pub/week6/html/week6.html

[10] Wikipedia Article for Stochastic Gradient Descent https://en.wikipedia.org/wiki/Stochastic_gradient_descent, last visited 04.05.2021