Study of Circular Quantum Dots using Machine Learning to Reduce the need of Physical Intuition

by

Even Marius Nordhagen

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

June 26, 2019

LaTeX

Abstract

This thesis aims the study of circular quantum dots using machine learning to reduce the need of physical intuition

Acknowledgements

5 years at Blindern

Thanks to Morten Hjorth-Jensen for all the help, motivation and believing in me.

Thanks to Håkon Kristiansen and Alocias Mariadason for answering all my questions and for you patience.

Thanks to my family for all the support, and my friends who always are there when I need to disconnect from the studies.

Last, but not least, I would like to say thank you to the Computational Physics group... So many talented guys in the group.

I have always believed that everything has a reason, which I used to point out whenever people talked about things that could not easily be explained. First when I was introduced to quantum mechanics, I realized that it might not be that simple, which is one of the reasons why this beautiful theory caught my attention. Even though my philosophy has changed to a less deterministic direction, I still like to think that everything can be explained out of some first principles.

Contents

Chapter 1

Introduction

The properties and behavior of quantum many-body systems are determined by the laws of quantum physics which have been known since the 1930s. The time-dependent Schrödinger equation describes the bounding energy of atoms and molecules, as well as the interaction between particles in a gas. In addition, it has been used to determine the energy of artificial structures like quantum dots, nanowires and ultracold condensates. [Electronic Structure Quantum Monte Carlo Michal Bajdich, Lubos Mitas]

Even though we know the laws of quantum mechanics, many challenges are encountered when calculating real-world problems. First, interesting systems often involve large number of particles, which causes expensive calculations. Second, the correct wave functions are seldomly known for a complex system, which is vital for measuring the observable correctly. Paul Dirac recognized those problems already in 1929:

"*The general theory of quantum mechanics is now almost complete... ...The underlying physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are thus completely known, and the difficulty is only that the exact application of these laws leads to equations much too complicated to be soluble.*"
-Paul Dirac, *Quantum Mechanics of Many-Electron Systems*, 1929

The purpose of this work is to look at machine learning as an approach to solving those problems, with focus on the former. The idea is to let a so-called restricted Boltzmann machine (RBM) define a flexible trial wave function, and then use a sampling tool to fit the function.

Lately, some effort has been put into this field, known as quantum machine learning. G.Carleo and M.Troyer demonstrated the link between RBMs and quantum Monte-Carlo (QMC) and named the states *neural-network quantum states* (NQS). They used the technique to study the Ising model and the Heisenberg model. [21] V.Flugsrud went further and investigated circular quantum dots with the same method.[**flugsrud˙vilde˙moe˙s**
We will extend the work she did to larger quantum dots and double quantum dots.

Why Quantum Dots?

VMC typically yields excellent results.

Quantum dots are often called artificial atoms because of their common features to real atoms, and their popularity is increasing due to their applications in semiconductor technology. For instance, quantum dots are expected to be the next big thing in display technology due to their ability of emitting photons of specific wavelength in addition to using 30% less energy than today's LED displays.[**manders˙8.3:˙2015**] Samsung already claim that they use this technology in their displays.[**noauthor˙2019˙nodate**]

Another reason why we are interested in simulating quantum dots, is that there exist experiments which we can compare our results to. Due to very strong confinement in the z-direction, the experimental dots, made by patterning GaAs/AlGaAs heterostructures, become essentially two-dimensional. [**marzin˙photoluminescence˙1994**][**br**
For that reason, our main focus in this work is two-dimensional dots, but also dots of three dimensions will be investigated.

Why Quantum Monte-Carlo?

History of QMC before and after the invention of electronic computers. Enrico Fermi 1930s similarities between imaginary time Schrödinger equation and stochastic processes in statistical mechanics. Metropolis VMC early 1950s. Kalos Greens's function Monte Carlo late 1950s. Ceperly and Alder 1980 homogeneous electron gas.

QMC appears to be method which has the best cost-to-performance ratio.

1.0.1 Why Computer Experiments?

"At the same time, advent of computer technology has offered us a new window of opportunity for studies of quantum (and many other) problems. It spawned a "third way" of doing science which is based on simulations, in contrast to analytical approaches and experiments. In a broad sense, by simulations we mean computational models of reality based on fundamental physical laws. Such models have value when they enable to make predictions or to provide new information which is otherwise impossible or too costly to obtain otherwise. In this respect, QMC methods represent an illustration and an example of what is the potential of such methodologies." [Electronic Structure Quantum Monte Carlo Michal Bajdich, Lubos Mitas]

Multi scale calculations are... A field of interest is how a systems behave when the interaction gets weaker. One way to model this, is to have a harmonic oscillator with a decreasing system frequency.

- Low frequency (weakly interacting electrons) field of interest - Multi scale calculations - Cartesian - Introduce the wave function - Mention the uncertainty principle and also quantum entanglement to catch the readers interest

1.1 Many-Body Problem

Not possible to solve analytically

1.2 Machine Learning

Branch of artificial intelligence

1.3 Goals and Milestones

- Investigate a new method to solve the Many-Body problem

1.4 Code

There exist multiple commercial code for solving the quantum many-body problem, and they are often optimized. In general it is wise to use already existing code and not try reinvent the wheel, but in our case we will investigate a new approach, which forces us to write the code from scratch.

Our variational Monte-Carlo (VMC) solver is written in object-orientated C++ inspired by Morten Ledum's example implementation, but our code is based on Cartesian coordinates. The goal is not to compete with the commercial software when it comes to performance, but we will still make some significant effort to make it efficient.

Additionally, the author has developed Hartree-Fock code and coupled cluster doubles (CCD) code written in Python, in cooperation with Stian Bilek. Also a full configuration interaction (FCI) code was developed. Alocias Mariadasons Hartree-Fock code was used to generate Hartree-Fock coefficients used in VMC code, because it lives in Cartesian coordinates and so does the VMC code. All code is open-source, and is freely available on `https://github.com/evenmn` under MIT license.

1.5 The Role of Ethics in Science

In science the ethics should always be prioritized.

Whenever one uses others work, no matter how much, the authors should be credited.

All the research that one does should always be detailed in a such way that it the experiments and results can be reproduced.

Computer science is no exception, all the points above are highly relevant.

Writing good code is a time consuming activity, and therefore author should be credited whenever some of their work is used by others.

1.6 Structure of Thesis

Fundamental theory, including many-body quantum physics and machine learning, is given in chapter 2-3. Methods for solving many-body systems follow thereafter in chapter 5-7, discussing quantum Monte-Carlo, the Hartree-Fock method and so-called post-Hartree-Fock methods. In chapter 8 and 9 we prepare for the implementation by deriving wave function elements and introduce minimization algorithms, and the final chapters 10-13 the implementation is justified, the results are given and discussed, and a brief conclusion is given.

Part I

Fundamental theory

Chapter 2

Quantum Many-Body Physics

If you are not completely
confused by quantum mechanics,
you do not understand it.

John Wheeler

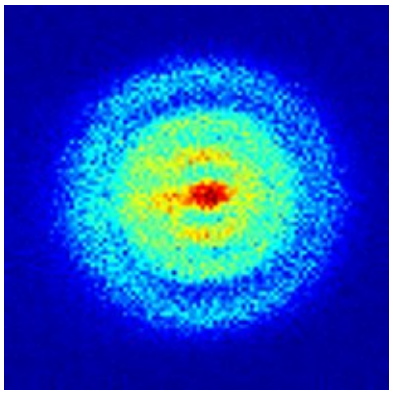


Figure 2.1: The first photograph of a Hydrogen atom was captured by an ultra sensitive camera in 2013. One can actually see the probability distribution $|\Psi(\boldsymbol{r})|^2$ with the naked eye. Published in Phys. rev. lett. 110, 213001 (2013), *Hydrogen atoms under magnification*. [16]

Around 1900, some physicists thought that there were nothing new to be discovered in physics and all that remained was more precise measurements, as Lord Kelvin famously pointed out. [13] He could not have been more wrong. In the following years, things were observed that could only be described by a quantized theory, led by Albert Einstein's explanation of the photoelectric effect in 1905.

Immense efforts were placed on completing the theory, and contributions from an array of scientists over a period of 20 years were necessary to get it finished. In 1929, Paul Dirac stated something similar to what Lord Kelvin said 30 years earlier, but apparently with greater accuracy.

2.1 Introductory Quantum Physics

In this section we will present the fundamentals of the quantum theory, that will make up the framework of this project. The theory is based on David Griffith's incredible textbook, *Introduction to Quantum Mechanics*, where the reader is relegated for further information.

Before we get started, we make a few assumptions in order to simplify our problem. The most important ones are specified below with an explanation why they are valid.

- **Point-like particles:** First, all particles involved will be assumed to be point-like, i.e, they lack spatial extension. For electrons this makes sense, since they, as far as we know, do not extent. The assumption also includes the nucleus in atomic systems, but it still makes sense since the distance from the nucleus to the electrons is known to be much larger than the nucleus extent.
- **Non-relativistic spacetime:** Second, we operate in the non-relativistic spacetime, which is an extremely good approximation as long as we do not approach the speed of light and we do not involve strong forces. Applying classical physics, we can find that the speed of the electron in a hydrogen atom is about 1% of the speed of light, and even though the electrons get higher speed in heavier atoms, we do not need to worry about it as we will stick to the lighter atoms. The forces acting are the weak Coulomb forces.
- For specific systems we might make new assumptions and approximations. For instance, for atomic systems we will assume that the nucleus is at rest. Those approximations will be discussed consecutively.

2.1.1 The Schrödinger Equation

In this work we will focus on solving the time-independent non-relativistic Schrödinger equation, which gives the energy eigenvalues of a system defined by a Hamiltonian $\hat{\mathcal{H}}$ and its eigenfunctions, $\Psi_n(\boldsymbol{r})$, which are the wave functions. $\boldsymbol{r}$ are the position coordinates of all the system's particles and n characterizes the state. The equation reads

$$\hat{\mathcal{H}}\Psi_n(\boldsymbol{r}) = \epsilon_n \Psi_n(\boldsymbol{r}) \tag{2.1}$$

where the Hamiltonian is the total energy operator. By analogy with the classical mechanics, this is given by

$$\hat{\mathcal{H}} = \hat{\mathcal{T}} + \hat{\mathcal{V}} \tag{2.2}$$

with $\hat{\mathcal{T}}$ and $\hat{\mathcal{V}}$ as the kinetic and potential energy operators respectively.

Again from classical mechanics, the kinetic energy for a moving particle of mass m yields $T = p^2/2m$ where p is the linear momentum, such that the kinetic energy operator can be represented as

$$\hat{\mathcal{T}} = \frac{\hat{\mathcal{P}}^2}{2m} \tag{2.3}$$

according to Ehrenfest's theorem. Further, the linear momentum operator is $\hat{\mathcal{P}} = -i\hbar\hat{\nabla}$ with $\hat{\nabla}$ as the differential operator and $\hbar$ as the reduced Planck's constant.

The potential energy can be split into an external part and an interaction part, where the latter is given by the Coulomb interaction. For two identical particles of charge q, the repulsive interaction gives the energy

$$V_I = k_e \frac{q^2}{r_{12}} \tag{2.4}$$

where r_{12} is the distance between the particles and k_e is Coulomb's constant. The total Hamiltonian of a system of N identical particles takes the form

$$\hat{\mathcal{H}} = -\sum_i^N \frac{\hbar^2}{2m}\nabla_i^2 + \sum_i^N u_i + \sum_i^N \sum_{j>i}^N k\frac{q^2}{r_{ij}} \tag{2.5}$$

which is the farthest we can go without specifying the external potential u_i. r_{ij} is the relative distance between particle i and j, defined by $r_{ij} \equiv |\boldsymbol{r}_i - \boldsymbol{r}_j|$.

Setting up equation (2.1) with respect to the energies, we obtain an integral,

$$\epsilon_n = \frac{\int d\boldsymbol{r}\Psi_n^*(\boldsymbol{r})\hat{\mathcal{H}}\Psi_n(\boldsymbol{r})}{\int d\boldsymbol{r}\Psi_n^*(\boldsymbol{r})\Psi_n(\boldsymbol{r})}, \tag{2.6}$$

which not necessarily is trivial to solve. If we take the wave function squared we get the probability distribution,

$$P(\boldsymbol{r}) = \Psi_n^*(\boldsymbol{r})\Psi_n(\boldsymbol{r}) = |\Psi_n(\boldsymbol{r})|^2 \tag{2.7}$$

so the nominator is basically the integral over all probabilities. If the wave function is normalized correctly, this should always give 1. Assuming that is the case, the expectation value can be expressed more elegantly by using Dirac notation,

$$E[\Psi] = \langle\Psi|\hat{\mathcal{H}}|\Psi\rangle\,, \tag{2.8}$$

where the first part, $\langle\Psi|$ is called a bra and the last part, $|\Psi\rangle$ is called a ket. At first this might look artificial and less informative, but it simplifies the notation significantly. More information about the notation is found in Appendix A.

[9]

2.1.2 The Variational Principle

In the equations above, the presented wave functions are assumed to be the exact eigenfunctions of the Hamiltonian. But often we do not know the exact wave functions, and we need to guess what the wave functions might be. In those cases we make use of the Rayleigh-Ritz variational principle or just the variational principle, which states that only the exact ground state wave function is able to give the ground state energy. All other wave functions that fulfill the required properties (see section 2.2) give higher energies, and mathematically we can express the statement

$$\epsilon_0 \leq \langle\Psi_T|\hat{\mathcal{H}}|\Psi_T\rangle\,. \tag{2.9}$$

Variational Monte-Carlo is a method based on (and named after) the variational principle, where we vary the trial wave function in order to obtain the lowest energy. It will be detailed in chaper (5).

2.1.3 Quantum Numbers

Up to this point, we have used n to indicate which state we are dealing with, but we have not explained what it actually means.

Unlike in classical physics, all the observable in quantum mechanics are discrete or *quantized*, hence the quantum name. The consequence is that n cannot take any number, but it needs to be a positive integer. In other words...

The quantum number $n \in [0, \infty\rangle$ is named the **principal** quantum number, and specifies the electronic shell of the state. As n increases, the energy will increase, which makes it the most important quantum number for energy measurements. The other quantum numbers are associated with angular momentum and spin, more about that in the next section.

2.1.3.1 Angular Momentum and Spin

If we again go back to the classical mechanics, the angular momentum $\boldsymbol{L}_r = \boldsymbol{R}\times\boldsymbol{p}$ around an axis at distance $|\boldsymbol{R}|$ from the mass center and the angular momentum $\boldsymbol{L}_c = I\boldsymbol{\omega}$ around its own mass center is a conserved quantity,

$$\boldsymbol{L}_{net} = \boldsymbol{L}_r + \boldsymbol{L}_c, \tag{2.10}$$

Since the net angular momentum $\boldsymbol{L}_{new}$ is just a sum over the angular momentum of all points in a continua around the rotational axis given by the definition of $\boldsymbol{L}_r$, both of them are actually the same thing.

In quantum mechanics we have again an analogy, where we define a **spin** s which describes a particles rotation around its own mass center and a **angular momentum** l which describes a particles rotation around an external rotational axis. Like in classical physics, the total spin S and the total angular momentum L is a conserved quantity,

$$J = L + S, \tag{2.11}$$

but the transition from L to S is rare compared to the transition from L_c to L_r. Spin-coupling. Azimuthal. Quantized.

We will summary all the quantum numbers through a table.

2.1.3.2 Hund's rules

Hund's rules define the filling order of electron structure shells with respect to the ground state energy.

2.1.4 Postulates of Quantum Mechanics

The quantum theory is built on six fundamental postulates, and without mentioning them our work would be incomplete. The postulates write:

1. *"The state of a quantum mechanical system is completely specified by the wave function $\Psi(\boldsymbol{r}, t)$."*
2. *"To every observable in classical mechanics, there corresponds a linear, Hermitian operator in quantum mechanics."*
3. *"In any measurement of the observable associated with an operator $\hat{\mathcal{A}}$, the only values that will ever be observed are the eigenvalues a which satisfy $\hat{\mathcal{A}}\Psi = a\Psi$."*
4. *"The expectation value of the observable corresponding to operator $\hat{\mathcal{A}}$ is given by*
$$\langle A \rangle = \frac{\int d\tau \Psi^* \hat{\mathcal{A}} \Psi}{\int d\tau \Psi^* \Psi}."$$
5. *"The wave function evolves in time according to the time-dependent Schrödinger equation,*
$$\hat{\mathcal{H}}\Psi(\boldsymbol{r}, t) = i\hbar \frac{\partial \Psi}{\partial t}."$$
6. *"The total wavefunction must be antisymmetric with respect to the interchange of all coordinates of one fermion with those of another. Electronic spin must be included in this set of coordinates."*

The postulates are taken from [**sherrill˙david˙postulates˙2003**].

The reader may recognize that some of the postulates were mentioned already in the first sections, which was necessary to give a proper introduction. Since we will be looking at stationary systems only, the time-independent Schrödinger equation and then postulate no.5 will not be used, but apart from that they all will play a significant role.

2.2 The Trial Wave Function

By the first postulate of quantum mechanics, the wave function contains all the information specifying the state of the system. This means that all observable in classical mechanics can also be measured from the wave function, which makes finding the wave function our main goal.

The trial wave function needs to meet some requirements in order to be used in the variational principle, and we thus need to make an educated guess on the wave function where the requirements are fulfilled. The requirements are the following:

1. **Normalizability:** The wave function needs to be normalizable in order to be physical. The total probability should always be 1, and a wave function that cannot be normalized will not have a finite total probability. The consequence is that the wave function needs to converge to zero when the positions get large.
2. **Cusp condition:** The cusp condition (also called the Kato theorem) states that the wave function should have a cusp where the potential explodes. An example on this is when charged particles come close to each other.

3. **Symmetry and anti-symmetry:** The wave function needs to be either symmetric or anti-symmetric under exchange of two coordinates, dependent on whether the particles are fermions or bosons. This is the statement of the sixth postulate, which will be further explained in the next section.

2.2.1 Anti-Symmetry and the Pauli Principle

Assume that we have a permutation operator $\hat{P}$ which exchanges two coordinates in the wave function,

$$\hat{P}(i \to j)\Psi_n(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_j, \ldots, \boldsymbol{x}_M) = p\Psi_n(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_j, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_M), \tag{2.12}$$

where p is just a factor which comes from the transformation. If we again apply the $\hat{P}$ operator, we should switch the same coordinates back, and we expect to end up with the initial wave function. For that reason, p must be either +1 or -1. [1]

The particles that have an antisymmetric (AS) wavefunction under exchange of two coordinates are called fermions, named after Enrico Fermi, and have half integer spin. On the other hand, the particles that have a symmetric (S) wavefunction under exchange of two coordinates are called bosons, named after Satyendra Nath Bose, and have integer spin.

It turns out that because of their anti-symmetric wave function, two identical fermions cannot be found at the same position at the same time, known as the Pauli principle. This causes some difficulties when dealing with multiple fermions, since they always spread to multiple states. The probability of finding two identical particles at the same position at the same time should therefore be zero, so technically we need to set the wave function to zero if it happens. To deal with this, we introduce a so-called Slater determinant, which automatically sets the wave function to zero if the Pauli principle is unsatisfied.

2.2.2 Slater determinant

For a system of more particles we can define a total wave function, which is a composition of all the single particle wave functions (SPF) and contains all the information about the system as the first postulate requires. For fermions, we need to combine the SPFs such that the Pauli principle is fulfilled at all times, which can be accomplished by a determinant.

Consider a system of two identical fermions with SPFs ϕ_1 and ϕ_2 at positions $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ respectively. The way we define the wavefunction of the system is then

$$\Psi_T(\boldsymbol{r}_1, \boldsymbol{r}_2) = \begin{vmatrix} \phi_1(\boldsymbol{r}_1) & \phi_2(\boldsymbol{r}_1) \\ \phi_1(\boldsymbol{r}_2) & \phi_2(\boldsymbol{r}_2) \end{vmatrix} = \phi_1(\boldsymbol{r}_1)\phi_2(\boldsymbol{r}_2) - \phi_2(\boldsymbol{r}_1)\phi_1(\boldsymbol{r}_2), \tag{2.13}$$

which is set to zero if the particles are at the same position. The determinant yields the same no matter the size of the system.

The Slater determinant is just a wave function ansatz to satisfy the Pauli principle, and we therefore need to denote it as the trial wave function. Additionally, the Slater determinant above contains the radial part only, because the single particle functions are the radial part by convention. For a general Slater determinant of N particles, the spin part needs to be included as well, giving

$$\Psi_T(\boldsymbol{r}) = \begin{vmatrix} \psi_1(\boldsymbol{r}_1) & \psi_2(\boldsymbol{r}_1) & \ldots & \psi_N(\boldsymbol{r}_1) \\ \psi_1(\boldsymbol{r}_2) & \psi_2(\boldsymbol{r}_2) & \ldots & \psi_N(\boldsymbol{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(\boldsymbol{r}_N) & \psi_2(\boldsymbol{r}_N) & \ldots & \psi_N(\boldsymbol{r}_N) \end{vmatrix} \tag{2.14}$$

where the ψ's are the true single particle wave functions, which are the tensor products

$$\psi = \phi \otimes \xi \tag{2.15}$$

with ξ as the spin part. In the next section, we will proceed further and show how the spin part can be factorized out of a general Slater determinant.

For bosonic systems, one can correspondingly construct a Slater permanent. The permanent of a matrix is similar to the determinant, but all negative signs are replaced by positive signs.

[1] Actually, in two-dimensional systems we have a third possibility which gives an *anyon*. The theory on this was developed by J.M. Leinaas and J. Myrheim during the 1970's. [5]

2.2.3 Splitting up the Slater Determinant

A determinant is relative computational expensive, and as the number of particles increases, it will certainly give us some problems. Fortunately, the Slater determinant can be split up when the particles have different spin, which is often hugely beneficial. Not only does the dimensions of the determinant reduce, but we also get rid of the spin part.

In this work, we will study fermions of spin $\sigma = \pm 1/2$ only, and we will therefore do the splitting for this specific case. However, this case is very important since both electrons and protons among others are spin-1/2 particles. The particles with $\sigma = +1/2 = \uparrow$ will be denoted as the spin-up particles, and the particles with $\sigma = -1/2 = \downarrow$ will be denoted as the spin-down particles. For simplicity, we will assume that the first coordinates $\boldsymbol{r}_1, \dots \boldsymbol{r}_{N_\uparrow}$ are the coordinates of the spin-up particles and the coordinates $\boldsymbol{r}_{N_\uparrow+1}, \dots \boldsymbol{r}_N$ are associated with the spin-down particles. $N_\uparrow$ is the number of spin-up particles and $N_\downarrow$ is the number of spin-down particles. The Slater determinant can then be written as

$$\Psi(\boldsymbol{r}) = \begin{vmatrix} \phi_1(\boldsymbol{r}_1)\xi_\uparrow(\uparrow) & \dots & \phi_{N_\uparrow}(\boldsymbol{r}_1)\xi_\uparrow(\uparrow) & \phi_{N_\uparrow+1}(\boldsymbol{r}_1)\xi_\downarrow(\uparrow) & \dots & \phi_N(\boldsymbol{r}_1)\xi_\downarrow(\uparrow) \\ \vdots & & \vdots & \vdots & & \vdots \\ \phi_1(\boldsymbol{r}_{N_\uparrow})\xi_\uparrow(\uparrow) & \dots & \phi_{N_\uparrow}(\boldsymbol{r}_{N_\uparrow})\xi_\uparrow(\uparrow) & \phi_{N_\uparrow+1}(\boldsymbol{r}_{N_\uparrow})\xi_\downarrow(\uparrow) & \dots & \phi_N(\boldsymbol{r}_{N_\uparrow})\xi_\downarrow(\uparrow) \\ \phi_1(\boldsymbol{r}_{N_\uparrow+1})\xi_\uparrow(\downarrow) & \dots & \phi_{N_\uparrow}(\boldsymbol{r}_{N_\uparrow+1})\xi_\uparrow(\downarrow) & \phi_{N_\uparrow+1}(\boldsymbol{r}_{N_\uparrow+1})\xi_\downarrow(\downarrow) & \dots & \phi_N(\boldsymbol{r}_{N_\uparrow+1})\xi_\downarrow(\downarrow) \\ \vdots & & \vdots & \vdots & & \vdots \\ \phi_1(\boldsymbol{r}_N)\xi_\uparrow(\downarrow) & \dots & \phi_{N_\uparrow}(\boldsymbol{r}_N)\xi_\uparrow(\downarrow) & \phi_{N_\uparrow+1}(\boldsymbol{r}_N)\xi_\downarrow(\downarrow) & \dots & \phi_N(\boldsymbol{r}_N)\xi_\downarrow(\downarrow) \end{vmatrix},$$

where spin-up wave functions require spin-up particles and vice versa. For that reason, half of the elements become zero and the determinant can be further expressed as

$$\Psi(\boldsymbol{r}) = \begin{vmatrix} \phi_1(\boldsymbol{r}_1)\xi_\uparrow(\uparrow) & \dots & \phi_{N_\uparrow}(\boldsymbol{r}_1)\xi_\uparrow(\uparrow) & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ \phi_1(\boldsymbol{r}_{N_\uparrow})\xi_\uparrow(\uparrow) & \dots & \phi_{N_\uparrow}(\boldsymbol{r}_{N_\uparrow})\xi_\uparrow(\uparrow) & 0 & \dots & 0 \\ 0 & \dots & 0 & \phi_{N_\uparrow+1}(\boldsymbol{r}_{N_\uparrow+1})\xi_\downarrow(\downarrow) & \dots & \phi_N(\boldsymbol{r}_{N_\uparrow+1})\xi_\downarrow(\downarrow) \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & \phi_{N_\uparrow+1}(\boldsymbol{r}_N)\xi_\downarrow(\downarrow) & \dots & \phi_N(\boldsymbol{r}_N)\xi_\downarrow(\downarrow) \end{vmatrix}.$$

This determinant can by definition be split up in a product of two determinants,

$$\Psi(\boldsymbol{r}) = |\hat{D}_\uparrow| \cdot |\hat{D}_\downarrow| \tag{2.16}$$

where $\hat{D}_\uparrow$ is the matrix containing all spin-up states and $\hat{D}_\downarrow$ is the matrix containing all spin-down states. Since all elements in the respective matrices contain the same spin function, it can be factorized out and omitted in the future study since the energy is independent of spin.

It is also worth to notice that the size of the spin-up determinant is determined by the number of spin-up particles, and it is similar for the spin-down determinant. This means that the we can change the total spin S by adjusting the relative sizes of the determinants.

This section was heavily inspired by D.Nissenbaum's dissertation, see appendix I in [14].

2.2.4 Basis set

To go further, we need to define a basis set, $\{\phi_1(\boldsymbol{r}), \phi_2(\boldsymbol{r}), \dots \phi_N(\boldsymbol{r})\}$ which should be chosen carefully based on the system. For a few systems, we know the exact basis of the non-interacting case, and it is thus a natural basis to use in the Slater determinant. For other systems, the choice of basis might depend on the situation, where we typically need to weigh computational time against accuracy. Concrete examples on both cases will be presented in chapter (3).

Often, one will see that the basis is optimized by the Hartree-Fock method. Using this basis in a single Slater determinant, we obtain the Hartree-Fock energy which sometimes is quite accurate. To get an even better energy estimate, we need to add more Slater determinants, which is the task of the post Hartree-Fock methods. More about this in chapter (5-7).

2.2.5 Jastrow factors

From electrostatics we know that identical, charged particles will repel each other. This means that the probability of finding two particles close to each other should be low, which needs to be baked into the wave function. One way to do this is to simply multiply the wave function with the distance between the particles; the smaller distance the lower probability. However, since we are going to work in the logarithmic space, dealing with exponential function will be much easier. This is the main idea behind the simple Jastrow factor.

2.2.5.1 Simple Jastrow

The simple Jastrow factor is just an exponential function with the sum over all particle distances. In addition, each distance r_{ij} is weighted by a parameter α_{ij}, and the factor becomes

$$J(\boldsymbol{r};\boldsymbol{\alpha}) = \exp\left(\sum_{i=1}^{N}\sum_{j>i}^{N}\alpha_{ij}r_{ij}\right). \tag{2.17}$$

All the α_{ij} are free variational parameters, which are expected to be symmetric since the distance matrix is symmetric.

One problem with this Jastrow factor, is that it does not create the cusp around each particle correctly. Basically, the Jastrow factor increases faster than it should when a particle is moved away from another. To solve this, we need to introduce a more complex Jastrow factor, the Padé-Jastrow.

2.2.5.2 Padé-Jastrow

The Padé-Jastrow factor is closely related to the simple Jastrow above, but a denominator is added to make the cusp correctly. It reads

$$J(\boldsymbol{r};\beta) = \exp\left(\sum_{i=1}^{N}\sum_{j>i}^{N}\frac{a_{ij}r_{ij}}{1+\beta r_{ij}}\right). \tag{2.18}$$

where β is a variational parameter. In addition, the fractions are multiplied with constants a_{ij} which depend on the particles i and j in the following way:

$$a_{ij} = \begin{cases} e^2/(D+1) & \text{if } i,j \text{ are particles of same spin} \\ e^2/(D-1) & \text{if } i,j \text{ are particles of opposite spin,} \end{cases} \tag{2.19}$$

for dimensions $D \in [2,3]$ where e is the elementary charge. We will later use natural and atomic units, and set $e = 1$, which for two dimensions gives $a_{ij} = 1/3$ (same spin) or $a_{ij} = 1$ (opposite spin) and for three dimensions $a_{ij} = 1/4$ (same spin) and $a_{ij} = 1/2$ (opposite spin).

This Jastrow factor is known to give accurate results for fermions and bosons because it gives the correct cusp condition, and it is the one we gonna use in the standard variational Monte-Carlo simulations.

2.3 Electron density

In quantum many-body computations, the electron density is frequently calculated, and there are several reasons for that. Firstly, the electron density can be found experimentally, such that the calculations can be benchmarked. Secondly, the electron density is very informative, since information about all particles can be gathered in one plot.

The P-body electron density can be found by integrating over all particles but P,

$$\rho_i(\boldsymbol{r}) = \int_{-\infty}^{\infty} d\boldsymbol{r}_P \dots d\boldsymbol{r}_N |\Psi(\boldsymbol{r}_1, \dots \boldsymbol{r}_N)|^2. \tag{2.20}$$

where $P < N$.

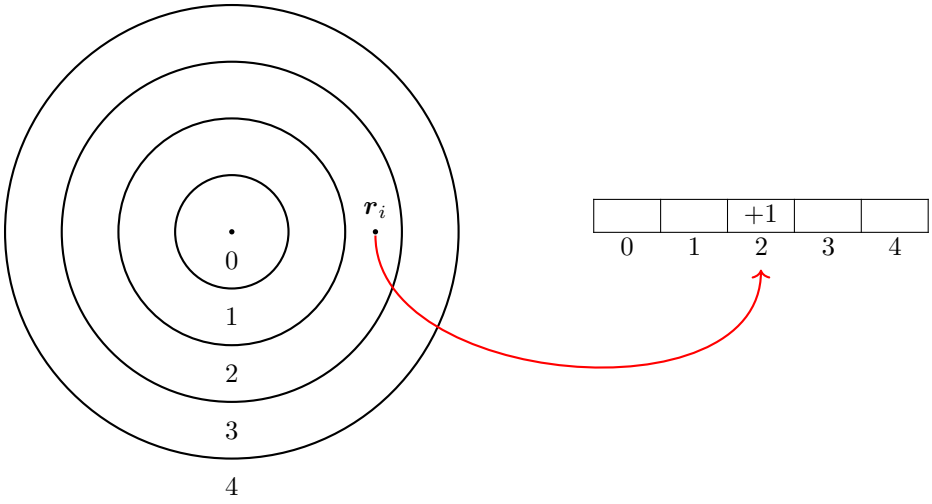


Figure 2.2: (color online) One-body

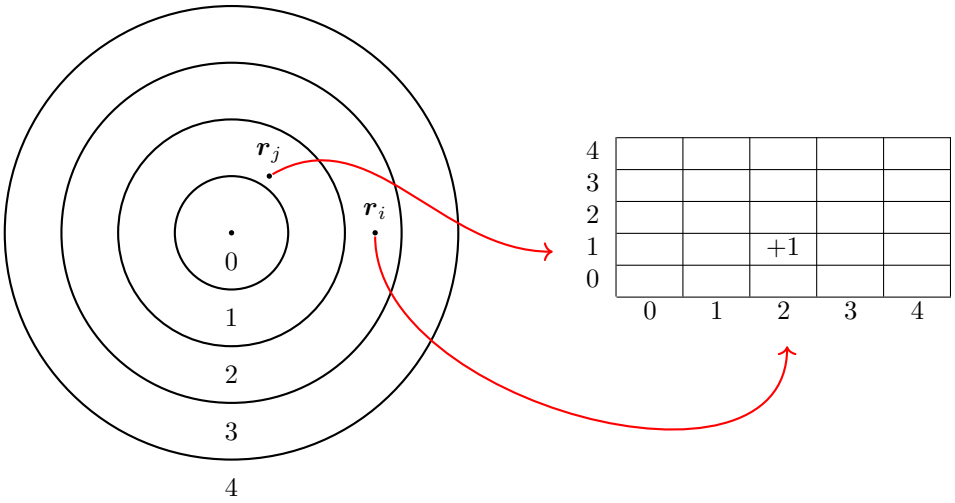


Figure 2.3: (color online) Two-body

2.3.1 One-body density

The one-body density is the most applied electron density, and is sometimes simply referred to as the electron density. For the two particle case, the one-body density gives the probability of finding one particle at a relative distance r to the other. For more particles, the one-body density gives the probability of finding the mass center of the remaining particles at a relative distance r from one of the particles.

Numerically, the one-body density is found by dividing the space into bins of equal sizes at different radii and counting the number of particles in each bin throughout the sampling. In practice, one often divide the space into bins where the radii are uniformly distributed, i.e, $r_i = i \cdot r_0$, see figure (**??**). In that case, one needs to divide each bin by its own volume afterwards in order to make all bins the same size. In two dimensions, the area of bin i is

$$A_i = (2i + 1)\pi d^2 \tag{2.21}$$

and in three dimensions the volume of bin i is

$$V_i = 4(i(i + 1) + 1/3)\pi d^3. \tag{2.22}$$

where d is the radial width of a bin.

2.3.2 Two-body density

For the one-body density, we integrate over all the particles but one, which corresponds to counting number of particles in each bin. For the two-body density, we integrate over all particles but a *particle pair*, which means that we numerically need to find the position of each particle pair.

2.3.3 Wigner Crystals

As the one-body density decreases, the particles move slower. At low one-body density, the potential energy dominates the kinetic energy and a phenomenon named Wigner crystallization might occur.

Wigner crystallization occurs when the

Chapter 3

Systems

> We must be clear that when it comes to atoms, language can be used only as in poetry.
>
> Niels Bohr, [4]

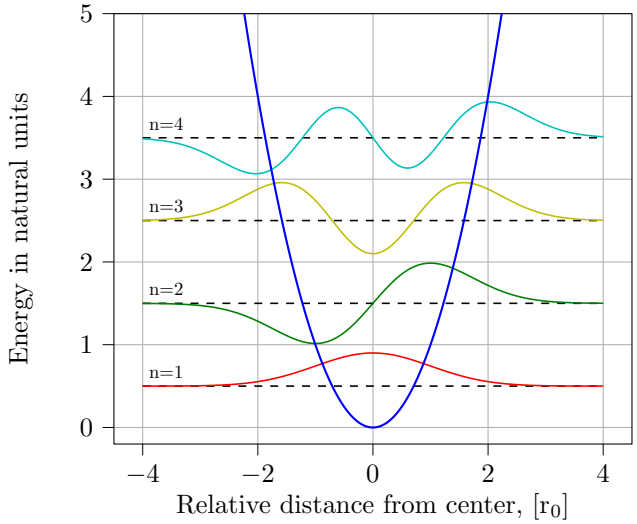


Figure 3.1: The quantum harmonic oscillator, with the Hermite functions represented up to 4th order. As in classical mechanics, the harmonic oscillator can describe various quantum systems, such as lattice vibration (phonons) and quantum fields.

When defining a system, we also need to specify the basis set to be used. The single particle functions are often known, and they are well-suited as a basis for the total

3.1 Quantum dots

Quantum dots are very small particles, and consist of fermions or bosons hold together by an external potential which is not created by a nucleus. Similar to atoms, these dots have discrete electronic states with a well-defined shell structure, and are therefore often called artificial atoms.

In this thesis we will study circular quantum dots with electrons affected by a harmonic oscillator potential. For an electron i, the potential reads

$$u_i = \frac{1}{2}m\omega^2 r_i^2, \tag{3.1}$$

where m is the mass of particle i, ω is the oscillator frequency and r_i is the relative distance from particle i to the center of the dot.

Using natural units as described in Appendix B, we can write the Hamiltonian as

$$\hat{\mathcal{H}} = \sum_{i=1}^{N}\Big(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2\Big) + \sum_{i<j}\frac{1}{r_{ij}} \tag{3.2}$$

where the energy is scaled with respect to Hartree units and lengths are scaled with respect to the Bohr radius.

The exact solutions of the non-interacting Hamiltonian are the Hermite functions,

$$\phi_n(x) = H_n(\sqrt{\omega}x)\exp\big(-\omega x^2/2\big) \tag{3.3}$$

which is a natural basis choice also for systems with interaction. $H_n(x)$ is the Hermite polynomial of n'th degree, and the first four Hermite functions are illustrated in figure (3.1). The energy of a particle with principal quantum number n in a D dimensional harmonic oscillator is given by

$$E_n = \omega\Big(n + \frac{D}{2}\Big) \qquad \forall n = 0, 1, 2... \tag{3.4}$$

We will study closed-shell systems only, since the Slater determinant in that case is unambiguous. The number of particles of closed-shell systems are called magic numbers, which in two dimensions are $N = 2, 6, 12, \ldots$. In general, the magic numbers are given by

$$N = s\binom{n+D}{D} \qquad \forall n = 0, 1, 2... \tag{3.5}$$

where s is the number of spin configurations (2), n is the principal quantum number and D is the number of dimensions. This is a direct consequence of the Pauli principle, where we in the ground state can have two particles with radial wave functions $\Phi_{n_x=0,n_y=0}$, in the next energy level we can have 4 particles with radial wave functions $\Phi_{n_x=1,n_y=0}$ and $\Phi_{n_x=0,n_y=1}$ with degeneracy 2 and so on.

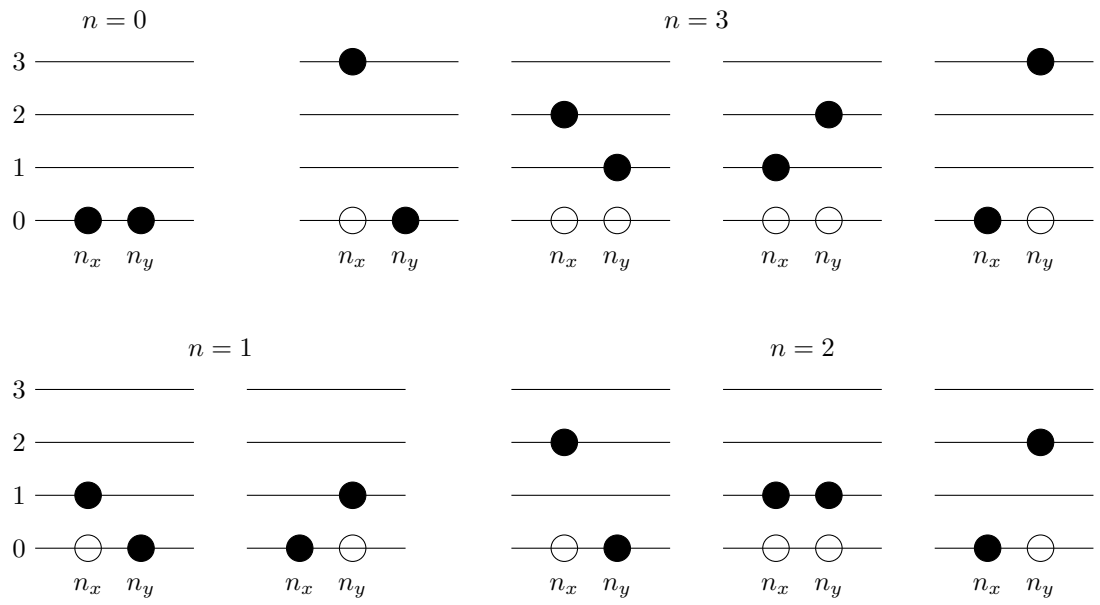


Figure 3.2: The possible states of a two-dimensional quantum dot for $n = n_x + n_y = 0, 1, 2, 3$. Recalling that the electrons can have spin ± 1, one can use this schematic to determine how many electrons there are in each closed shell and thus find the magic numbers.

3.2 Quantum double dots

Another historically important quantum system is the double dot, which similarly to the single dot can be solved analytically. For the same reason as the single dot often is called an artificial atom, the double dots are called artificial molecules.

Unlike the single dot, this system has not a potential that is definitely most used. In fact, there are multiple popular potentials, but when it comes to the symmetrical double-well, they are usually on the form

$$u_i = \frac{1}{2}\omega^2\left[|x_i|^a - \left(\frac{b}{2}\right)^a\right]^2 \tag{3.6}$$

in one dimension with b as the distance between the wells and a as an arbitrary integer. [**jelic˙double˙2012**] Setting $a = 1$ gives two parabolic wells with a sharp local maximum at $x = 0$, while $a = 2$ gives a smoother but steeper well. In figure (3.3) the potential is plotted for $a = 1, 2$ and 3.

For reference and benchmark reasons, we will focus on the case with $a = 1$ and $b = 2$, which can be written out as

$$u_i = \frac{1}{2}\omega^2\left[x_i^2 + \frac{1}{4}b^2 - b|x_i|\right], \tag{3.7}$$

still in one dimension. For more than one dimension, we assume that the double dot expands in the x-direction, which gives us the expression for all dimensions

$$u_i^{\mathrm{DW}} = \frac{1}{2}\omega^2\left[r_i^2 + \frac{1}{4}b^2 - b|x_i|\right] = u_i^{\mathrm{HO}} + \frac{1}{2}\omega^2\left[\frac{1}{4}b^2 - b|x_i|\right] \tag{3.8}$$

where HO means harmonic oscillator potential and DW means double-well potential. What we actually observe, is that the potential separates in a single-dot part and a double-dot part, which makes the double-dot Hamiltonian similar to the single-dot Hamiltonian,

$$\hat{\mathcal{H}} = \sum_{i=1}^{P}\left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\omega^2 r_i^2 + \frac{1}{2}\omega^2\left(\frac{1}{4}b^2 - b|x_i|\right)\right) + \sum_{i<j}\frac{1}{r_{ij}}. \tag{3.9}$$

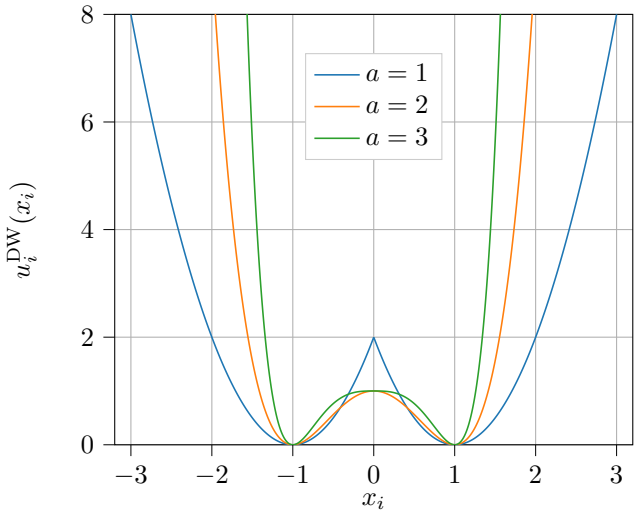


Figure 3.3: Double-well potentials plotted with $a = 1, 2$ and 3, and $b = 2$.

What is remaining is to find an appropriate basis set, and based on the observations above, a Hermite function expansion sounds reasonable. It will take the form

$$\left|\phi_n^{\mathrm{DW}}(x)\right\rangle = \sum_{\lambda=1}^{L} C_{n\lambda} \left|\phi_\lambda^{\mathrm{HO}}(x)\right\rangle, \tag{3.10}$$

where $L \in [1, \infty\rangle$ is the number of basis functions used and $C_{n\lambda}$ is the coefficient associated with the double-dot function n and the single-dot function λ, which is what we want to find. Inserting this into the double-dot Schrödinger equation and multiplying with $\left\langle\phi_\nu^{\mathrm{HO}}(x)\right|$ on the left-hand-side (LHS) gives

$$\sum_{\lambda=1}^{L} C_{n\lambda} \left\langle\phi_\nu^{\mathrm{HO}}(x)\right|\hat{\mathcal{H}}_{\mathrm{DW}}\left|\phi_\lambda^{\mathrm{HO}}(x)\right\rangle = \epsilon_n \sum_{\lambda=1}^{L} C_{n\lambda} \left\langle\phi_\nu^{\mathrm{HO}}(x)\middle|\phi_\lambda^{\mathrm{HO}}(x)\right\rangle \tag{3.11}$$

where the right-hand-side (RHS) sum collapses because the overlap is just the Kronecker delta, $\left\langle\phi_\nu^{\mathrm{HO}}(x)\middle|\phi_\lambda^{\mathrm{HO}}(x)\right\rangle = \delta_{\nu\lambda}$. By defining the matrix elements

$$\hat{h}_{\nu\lambda} \equiv \left\langle\phi_\nu^{\mathrm{HO}}(x)\right|\hat{\mathcal{H}}_{\mathrm{DW}}\left|\phi_\lambda^{\mathrm{HO}}(x)\right\rangle, \tag{3.12}$$

we can set up equation (3.11) as an eigenvalue problem on the form

$$\hat{h}\hat{C} = \epsilon\hat{C} \tag{3.13}$$

where our targets, $\hat{C}$, are just the eigenvectors of the $\hat{h}$-matrix. Now recall that the double-dot Hamiltonian is just an extension of the single-dot Hamiltonian, such that we can rewrite

$$\hat{h}_{\nu\lambda} = \left\langle\phi_\nu^{\mathrm{HO}}(x)\right|\hat{\mathcal{H}}_{\mathrm{HO}}\left|\phi_\lambda^{\mathrm{HO}}(x)\right\rangle + \left\langle\phi_\nu^{\mathrm{HO}}(x)\right|\hat{\mathcal{H}}_{+}\left|\phi_\lambda^{\mathrm{HO}}(x)\right\rangle \tag{3.14}$$

with $\hat{\mathcal{H}}_+ = (1/2)\omega\sum_{i=1}^{P}\Big((1/4)b^2 - b|x_i|\Big)$ as the extension. The former integrals are just the harmonic oscillator energies, presented in (3.4), while the latter integrals are trivial to calculate.

3.3 Atomic systems

We will also investigate real atoms, where we freeze out the nucleonic degrees of freedom known as the Born-Oppenheimer approximation. The electrons will in fact affect the nucleus, but due to the mass difference this effect will be negligible.

We again have Coulomb interaction between the electrons and the nucleus, and since we assume the latter to be at rest at the origin, the external potential affecting particle i is

$$u_i = -\frac{1}{2}k\frac{Ze^2}{r_i}, \tag{3.15}$$

where Z is the atomic number (number of protons inside the nucleus). The total Hamiltonian is given in (Hartree) atomic units,

$$\hat{\mathcal{H}} = \sum_{i=1}^{P}\Big(-\frac{1}{2}\nabla_i^2 - \frac{Z}{r_i} + \frac{l(l+1)}{2r_i^2}\Big) + \sum_{i<j}\frac{1}{r_{ij}}, \tag{3.16}$$

which also is discussed in Appendix B. For the non-interacting case, the energies are given by the Bohr formula

$$E_n = -\frac{Z^2}{2n^2}. \tag{3.17}$$

For atomic systems, it is convenient to use spherical coordinates, which allows us to split up the wave function in a radial part and an angular part,

$$\psi_{nlm}(r,\theta,\phi) = R_{nl}(r)Y_l^m(\theta,\phi) \tag{3.18}$$

The exact radial part for the non-interacting case is called the hydrogen-like orbitals, that is

$$R_{nl}(r) \propto r^l e^{-Zr/n}\Big[L_{n-l-1}^{2l+1}\Big(\frac{2r}{n}Z\Big)\Big] \tag{3.19}$$

where $L_q^p(x)$ are the *associated Laguerre polynomials* or *generalized Laguerre polynomials*. More about them and how to calculate them recursively can be found in Appendix C.

The angular part is given by the *spherical harmonics*

$$Y_l^m(\theta,\phi) \propto P_l^m(\cos\theta)e^{im\phi} \tag{3.20}$$

where $P_l^m(x)$ are the *associated Legendre polynomials*. The complex part in the spherical harmonics causes some difficulties, and we will therefore instead use the solid harmonics

$$S_l^m(r,\theta,\phi) \propto r^l P_l^{|m|}(\cos\theta)\begin{cases}\cos(m\phi) & \text{if} \quad m \geq 0\\ \sin(|m|\phi) & \text{if} \quad m < 0.\end{cases} \tag{3.21}$$

Again we will study closed shells only, but for atoms we will introduce subshells as well, which are dependent on the azimuthal quantum number l in addition to the principal quantum number n. In general, we have that $l \in [0, n-1]$ such that we only have one subshell for $n = 1$. Traditionally, the first few subshells are denoted with s, p, d and f, and the meaning can be found in table (3.1), together with number of electrons in each subshell.

Table 3.1: Table of the first subshells

Subshell label	l	Max electrons	Name
s	0	2	sharp
p	1	6	principal
d	2	10	diffuse
f	3	14	fundamental
g	4	18	alphabetic

For Helium, we have two electrons with $n = 1$, which means that both have $l = 0$ and both electrons are in the s-subshell. We can thus write the electron configuration as $1s^2$.

Similar as for the principal quantum number n, we can use the tumble rule the lower l the lower energy, such that for Beryllium all four electrons are still in the s-subshell. Beryllium therefore has electron

configuration $1s^2 2s^2$ or [He] $2s^2$. Since both subshells are fully occupied, Beryllium can be included in our closed-shell calculations.

If we continue with the same rules, we see that the next closed-shell atom has a fully occupied p-subshell as well, which is Neon with 10 electrons. This is a noble gas, and we can write the electron configuration as [Be] $2p^6$. All noble gases have endings Xs^2Xp^6, which is the reason why they always have 8 valence electrons.

We can now compare this to the periodical system, and observe that the two first rows agrees with the theory presented above: The first row has two elements and the second has eight. However, the third one also has eight elements, which does not fit our theory. The reason is that the angular momentum contribution is not taken into account, i.e., we need to include the Hamiltonian term

$$V_L = \frac{l(l+1)}{2r^2} \tag{3.22}$$

as well. If we do so, we see that the thumb rule defined above not always holds. Sometimes a low l in a higher n causes lower energy than a high l in a lower n.

"Colloquially, we call such solutions and derived properties as electronic structure."

Chapter 4

Machine Learning

In the early 1990's we were working with machine learning all the time, but back then we called it pattern recognition and regression.

Prof. Anne Solberg, UiO

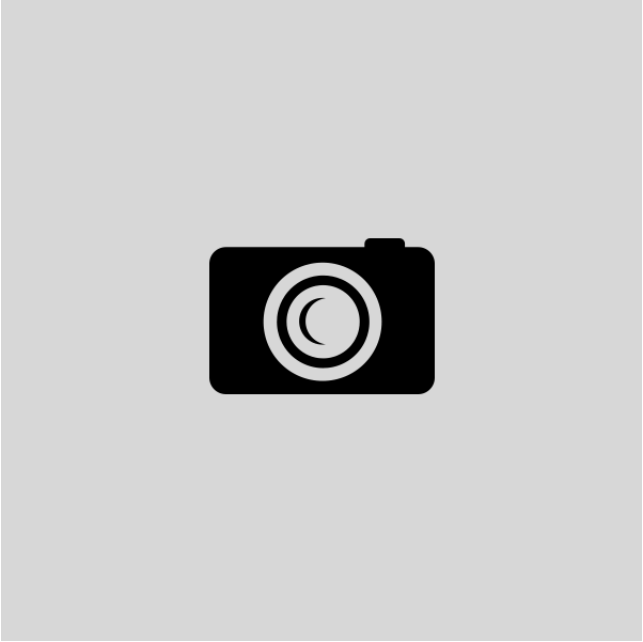


Figure 4.1: Caption

The use of the term *machine learning* has exploded over the past years, and sometimes it sounds like it is a totally new field. However, the truth is that many of the methods are relatively old, where for instance *linear regression* was known early in the 19th century. [1, 2] Those methods have just recently been taken under the machine learning umbrella, which is one of the reasons why the term is used more frequently than before. As the professor A.Solberg at the University of Oslo pointed out during one of her lectures, machine learning covers all methods where the goal is to minimize a loss function based on a set of parameters. This has made machine learning an extremely rich field. Unlike traditional algorithms, machine learning algorithms are not told what to do directly, but they use optimization tools to reproduce targets in the best way. As a consequence, we often do not know exactly what the algorithm does and why it behaves as is does, and

because of this behavior, the processing is often called artificial intelligence. In our search for a technique to solve quantum mechanical problems where less physical intuition is needed, machine learning appears as a natural tool.

The increasing popularity should also without any doubt be attributed to the dramatically improvement of a majority of the machine learning algorithms. Perhaps the most important contribution to this came in 2012, when the deep convolution neural network (CNN) **AlexNet** managed to classify 1.2 millions images into 1000 classes with a remarkable top-5 test error rate of 15.3%. [**krizhevsky˙imagenet˙2012**] Today, the CNNs have been further improved, and they are even able to beat humans in recognizing images! (The history began from AlexNet) Also voice recognition algorithms have lately been revolutionized, thanks to recurrent neural networks (RNNs), and especially long short-term memory (LSTM) networks. Their ability to recognize sequential (time-dependent) data made the technology good enough for an entry to millions of peoples everyday-life through services such as **Google Translate** (Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation), **Siri by Apple** (https://bgr.com/2016/06/13/ios-10-siri-third-party-apps/) and **Amazon Alexa** (https://www.allthingsdistributed.com/2016/11/amazon-ai-and-alexa-for-all-aws-apps.html). It is also interesting to see how machine learning has made computers eminent tacticians using recurrent neural networks. The **Google DeepMind** developed program **AlphaGo** demonstrated this by beating the 9-dan professional L. Sedol in the board game Go (https://www.alphagomovie.com/), before a improved version, **AlphaZero**, beat the at that time highest rated chess computer, **StockFish** (https://www.chess.com/news/view/google-s-alphazero-destroys-stockfish-in-100-game-match). Both these scenarios were unbelievable just a decade ago.

Even though all these branches are both exciting and promising, they will not be discussed further in this work, since they will simply not work for our purposes. The reason is that they initially require a data set with known outputs in order to be trained, they obey so-called *supervised* learning. Instead, we rely on *unsupervised* learning, which has the task of finding patterns in the data and is therefore not in need for known outputs. However, we will discuss some simpler supervised learning algorithms as an introduction and a motivation for the unsupervised learning section.

4.1 Supervised Learning

In supervised learning methods, we know the corresponding targets to the input data sets, which we use to train the model. This could for instance be linear regression, logistic regression or feed-forward neural networks.

Linear regression is perhaps the most intuitive example on this, where we want to find the line that fits some data points in the best possible way. In two dimensions, the x-coordinates are the inputs to the model and the y-coordinates are the targets. For instance, if we want to fit a second order polynomial,

$$f(x) = ax^2 + bx + c, \tag{4.1}$$

to a set of n points $\{(x_1, y_1), (x_2, y_2), \dots (x_n, y_n)\}$, one can set up a set of equations

$$\begin{array}{ccccccc} \hat{y}_1 & = & ax_1^2 & + & bx_1 & + & c \\ \hat{y}_2 & = & ax_2^2 & + & bx_2 & + & c \\ \vdots & & \vdots & & \vdots & & \vdots \\ \hat{y}_n & = & ax_n^2 & + & bx_n & + & c \end{array} \tag{4.2}$$

where $\hat{y}_i$ is the y-value of the line at $x = x_i$. Our goal is to minimize the distance between $\hat{y}_i$ and y_i by adjusting the parameters a, b and c, which is usually done using the mean square error (MSE) as a measurement of the error. The loss function (also called the cost function) then reads

$$\mathcal{L}(a, b, c) = \sum_{i=1}^{n} \Big(y_i - (ax_i^2 + bx_i + c) \Big)^2 \tag{4.3}$$

which can either be minimized by a matrix-vector product or an iterative minimization algorithm. Both these methods will be covered in the more general linear regression section below.

4.1.1 Linear Regression

In linear regression, the dependent variable y_i is a linear combination of the parameters, and for a dependent variable this can be written as

$$\hat{y}_i = \sum_j X_{ij}\beta_j \tag{4.4}$$

where β_j's are the unknown parameters to be found. In principle, X_{ij} can be an arbitrary function of the arguments x_i, but in the polynomial case it can be simplified by setting $X_{ij} = x_i^j$. We will proceed dealing with X_{ij} for general purposes.

The three most commonly used linear regression methods are *ordinary least square* (OLS) regression, Ridge regression and Lasso regression, where the former has the loss function

$$\mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{n}\Big(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\Big)^2, \qquad \text{OLS} \tag{4.5}$$

which is minimized when

$$\boldsymbol{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\boldsymbol{y}. \tag{4.6}$$

Quite often when we deal with large data sets, the design matrix contains singular values which give us problems with calculating $(\hat{X}^T\hat{X})^{-1}$. A way to avoid this is to introduce a penalty λ to ensure that all the diagonal values are non-zero, which can be accomplished by adding a small value to all diagonal elements. This is the idea behind Ridge regression, which has the loss function

$$\mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{n}\Big(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\Big)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 \qquad \text{Ridge} \tag{4.7}$$

and is minimized when

$$\boldsymbol{\beta} = (\hat{X}^T\hat{X} + \lambda I)^{-1}\hat{X}^T\boldsymbol{y}. \tag{4.8}$$

Finally, we have Lasso regression with a loss function given by

$$\mathcal{L}(\boldsymbol{\beta}) = \sum_{i=1}^{n}\Big(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\Big)^2 + \lambda\sum_{j=1}^{p}\beta_j \qquad \text{Lasso} \tag{4.9}$$

and without any expression for the optimized parameters. In order to optimize this loss function, we therefore need to use an iterative optimization algorithm. Such methods will later be used in the variational Monte-Carlo sampling, and some methods are therefore detailed in chapter (9).

To illustrate how the iterative optimization works, we will go back to equation (4.4) and use OLS for simplicity reasons. Imagine that we have four points $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$ that we want to fit to a curve of three parameters $\beta_1, \beta_2, \beta_3$. The simplest way to solve this interatively is to use **gradient descent** optimization, which goes as

$$\beta_k^+ = \beta_k - \eta\frac{\partial\mathcal{L}(\boldsymbol{\beta})}{\partial\beta_k} \tag{4.10}$$

with η specifying how much the parameters should be changed for each iteration, known as the learning rate. The output $\hat{y}_i$ can be expressed by

$$\hat{y}_i = X_{i1}\beta_1 + X_{i2}\beta_2 + X_{i3}\beta_3 \tag{4.11}$$

and the network can be illustrated with β_j as the input nodes, $\hat{y}_i$ as the output nodes and X_{ij} as lines in between all nodes, as in figure ...

ADD FIGURE

To go further, we will restrict ourselves to a second order polynomial, setting $X_{i1} = 1$, $X_{i1} = x_i$ and $X_{i1} = x_i^2$. The parameter update then yields

$$\beta_k^+ = \beta_k + 2\eta\sum_{i=1}^{3}(y_i - \beta_1 - \beta_2 x_i - \beta_3 x_i^2)x_i^{k-1}, \tag{4.12}$$

which can be run iteratively until the parameters have converged.

We will again stress that we go through this iterative method for illustration purposes only, it would be more efficient to apply equation (4.6). However, the thinking above is similar to the approach we do when training a neural network. An important difference is that we here search for an estimate of the left-hand-side (LHS) *nodes* b_j, while in a traditional feed-forward neural network (FNN) we want to find the *weights*, the parameters that connect the nodes together (here X_{ij}). The principle with multiplying the LHS nodes by the weights to obtain the right-hand-side (RHS) nodes is though the same, and we can use the same minimization tools. An example that is even more related to neural networks is *logistic regression*, which will be discussed in the next section.

4.1.2 Logistic Regression

In the previous section we presented how to fit a polynomial to a set of points using linear regression. In this chapter we will look at classification problems using logistic regression. Traditionally, the perceptron model was used for *hard classification*, which sets the outputs directly to binary values. However, often we are interested in the probability of a given category, which means that we need a continuous *activation function*. Logistic regression can, like linear regression, be considered as a function where weights are adjusted with the intention to minimize the error. An illustration of a simple perceptron is found in figure (4.2).

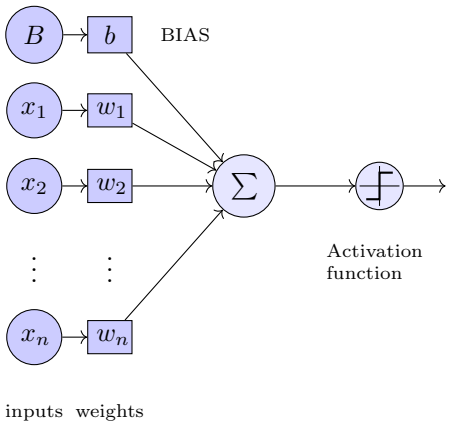


Figure 4.2: Logistic regression model with n inputs.

Consider first a classification problem with two possible outcomes, for example the ultimatum: will I pass or fail the exam? In that case an output node is sufficient, where 0 is pass, 1 is fail and an intermediate value gives the probability that I will fail. In order to map the output to a number between 0 and 1, we use an activation function.

But, how does the model know if I will pass or fail? In order to get any useful information from the perceptron, we need to train it first. In our case, one could for instance use data from other students as input, and train the model as long as we know if they passed or failed.

Now, what if we want to predict the grade instead of just pass/fail? In that case, we need a class for each grade one can get. A popular choice is to use *one hot encoding*, which fires

The very first step is to calculate the initial outputs (forward phase), where the weights usually are set to small random numbers. Then the error is calculated, and the weights are updated to minimize the error (backward phase). So far so good.

4.1.2.1 Forward phase

Let us look at it from a mathematical perspective, and calculate the net output. The net output seen from an output node is simply the sum of all the "arrows" that point towards the node, see figure (4.2), where each "arrow" is given by the left-hand node multiplied with its respective weight. For example, the contribution from input node 2 to the output node follows from $X_2 \cdot w_2$, and the total net output to the output O is

therefore

$$net = \sum_{i=1}^{I} x_i \cdot w_i + b \cdot 1. \tag{4.13}$$

Just some notation remarks: x_i is the value of input node i and w_i is the weight which connects input i to the output. b is the bias weight, which we will discuss later.

You might wonder why we talk about the net output all the time, do we have other outputs? If we look at the network mathematically, what we talk about as the net output should be our only output. Anyway, it turns out to be convenient mapping the net output to a final output using an activation function, which is explained further in section 4.1.2.5. The activation function, f, takes in the net output and gives the output,

$$out = f(net). \tag{4.14}$$

If not everything is clear right now, it is fine. We will discuss the most important concepts before we dive into the maths.

4.1.2.2 BIAS

As mentioned above, we use biases when calculating the outputs. The nodes, with value B, are called the bias nodes, and the weights, b, are called the bias weights. But why do we need those?

Suppose we have two inputs of value zero, and one output which should not be zero (this could for instance be a NOR gate). Without the bias, we will not be able to get any other output than zero, and in fact the network would struggle to find the right weights even if the output had been zero.

The bias value B does not really matter since the network will adjust the bias weights with respect to it, and is usually set to 1 and ignored in the calculations. [2]

4.1.2.3 Learning rate

In principle, the weights could be updated without adding any learning rate ($\eta = 1$), but this can in practice be problematic. It is easy to imagine that the outputs can be fluctuating around the targets without decreasing the error, which is not ideal, and a learning rate can be added to avoid this. The downside is that with a low learning rate the network needs more training to obtain the correct results (and it takes more time), so we need to find a balance.

4.1.2.4 Loss function

The loss function is what defines the error, and in logistic regression the cross-entropy function is a naturally choice. [3] It reads

$$c(\boldsymbol{W}) = -\sum_{i=1}^{n} \Big[y_i \log f(\boldsymbol{x}_i^T \boldsymbol{W}) + (1 - y_i) \log[1 - f(\boldsymbol{x}_i^T \boldsymbol{W})] \Big] \tag{4.15}$$

where $\boldsymbol{W}$ contains all weights, included the bias weight ($\boldsymbol{W} \equiv [b, \boldsymbol{W}]$), and similarly does $\boldsymbol{x}$ include the bias node, which is 1; $\boldsymbol{x} \equiv [1, \boldsymbol{x}]$. Further, the $f(x)$ is the activation function discussed in next section.

The cross-entropy function is derived from likelyhood function, which famously reads

$$p(y|x) = \hat{y}^y \cdot (1 - \hat{y})^{1-y}. \tag{4.16}$$

Working in the log space, we can define a log likelyhood function

$$\log \Big[p(y|x) \Big] = \log \Big[\hat{y}^y \cdot (1 - \hat{y})^{1-y} \Big] \tag{4.17}$$

$$= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \tag{4.18}$$

which gives the log of the probability of obtaining y given x. We want this quantity to increase then the loss function is decreased, so we define our loss function as the negative log likelyhood function. [7]

Additionally, including a regularization parameter λ inspired by Ridge regression is often convenient, such that the loss function is

$$c(\boldsymbol{W})^+ = c(\boldsymbol{W}) + \lambda||\boldsymbol{W}||_2^2. \tag{4.19}$$

We will later study how this regularization affects the classification accuracy.

4.1.2.5 Activation function

Above, we were talking about the activation function, which is used to activate the net output. In binary models, this is often just a step function firing when the net output is above a threshold. For continuous outputs, the logistic function given by

$$f(x) = \frac{1}{1+e^{-x}}. \tag{4.20}$$

is usually used in logistic regression to return a probability instead of a binary value. This function has a simple derivative, which is advantageous when calculating a large number of derivatives. As shown in section **??**, the derivative is simply

$$\frac{df(x)}{dx} = x(1-x). \tag{4.21}$$

$tanh(x)$ is another popular activation function in logistic regression, which more or less holds the same properties as the logistic function.

4.1.2.6 Backward phase

Now all the tools for finding the outputs are in place, and we can calculate the error. If the outputs are larger than the targets (which are the exact results), the weights need to be reduced, and if the error is large the weights need to be adjusted a lot. The weight adjustment can be done by any minimization method, and we will look at a couple of gradient methods. To illustrate the point, we will stick to the **gradient descent** (GD) method in the calculations, even though other methods will be used later. The principle of GD is easy: each weight is "moved" in the direction of steepest slope,

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot \frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}}, \tag{4.22}$$

where η is the learning rate and $c(\boldsymbol{w})$ is the loss function. We use the chain rule to simplify the derivative

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial c(\boldsymbol{w})}{\partial out} \cdot \frac{\partial out}{\partial net} \cdot \frac{\partial net}{\partial \boldsymbol{w}} \tag{4.23}$$

where the first is the derivative of the loss function with respect to the output. For the cross-entropy function, this is

$$\frac{\partial c(\boldsymbol{w})}{\partial out} = -\frac{y}{out} + \frac{1-y}{1-out}. \tag{4.24}$$

Further, the second derivative is the derivative of the activation function with respect to the output, which is given in (4.21)

$$\frac{\partial out}{\partial net} = out(1-out). \tag{4.25}$$

The latter derivative is the derivative of the net output with respect to the weights, which is simply

$$\frac{\partial net}{\partial \boldsymbol{w}} = \boldsymbol{x}. \tag{4.26}$$

If we now recall that $out = f(\boldsymbol{x}^T\boldsymbol{w})$, we can write

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]\boldsymbol{x} \tag{4.27}$$

and obtain a weight update algorithm

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]^T\boldsymbol{x}. \tag{4.28}$$

where the bias weight is included implicitly in $\boldsymbol{w}$ and the same applies for $\boldsymbol{x}$.

4.1.3 Neural network

If you have understood logistic regression, understanding a neural network should not be a difficult task. According to **the universal approximation theorem**, a neural network with only one hidden layer can approximate any continuous function. [8] However, often multiple layers are used since this tends to give fewer nodes in total.

In figure (4.3), a two-layer neural network (one hidden layer) is illustrated. It has some similarities with the logistic regression model in figure (4.2), but a hidden layer and multiple outputs are added. In addition, the output is no longer probabilities and can take any number, which means that we do not need to use the logistic function on the outputs anymore.

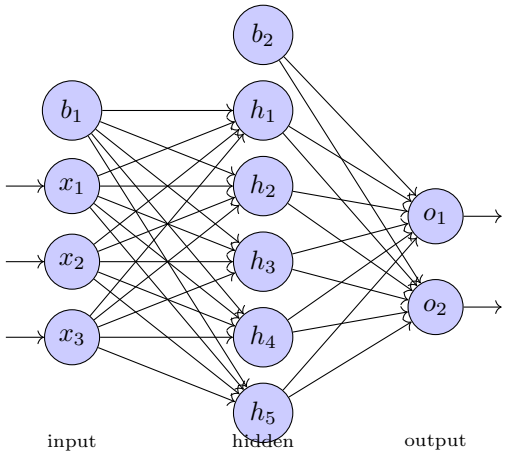


Figure 4.3: Neural network with 3 input nodes, 5 hidden nodes and 2 output nodes, in addition to the bias nodes.

Without a hidden layer, we have seen that the update of weights is quite straight forward. For a neural network consisting of multiple layers, the question is: how do we update the weights when we do not know the values of the hidden nodes? And how do we know which layer causing the error? This will be explained in section 4.1.3.3, where one of the most popular techniques for that is discussed. Before that we will generalize the forward phase presented in logistic regression.

4.1.3.1 Forward phase

In section 4.1.2.1, we saw how the output is found for a single perceptron. Since we only had one output node, the weights could be stored in an array. Generally, it is more practical to store the weights in matrices, since they will have indices related to both the node on left-hand side and the node on the right-hand side. For instance, the weight between input node x_3 and hidden node h_5 in figure (4.3) is usually labeled as w_{35}. Since we have two layers, we also need to denote which weight set it belongs to, which we will do by a superscript ($w_{35} \Rightarrow w_{35}^{(1)}$). In the same way, $\boldsymbol{W}^1$ is the matrix containing all $w_{ij}^{(1)}$, $\boldsymbol{x}$ is the vector containing all x_i's and so on. We then find the net outputs at the hidden layer to be

$$net_{h,j} = \sum_{i=1}^{I} x_i \cdot w_{ij}^{(1)} = \boldsymbol{x}^T \boldsymbol{W}_j^{(1)} \tag{4.29}$$

where the $\boldsymbol{x}$ and $\boldsymbol{W}^{(1)}$ again are understood to take the biases. This will be the case henceforth. The real output to the hidden nodes will be

$$h_j = f(net_{h,j}). \tag{4.30}$$

Further, we need to find the net output to the output nodes, which is obviously just

$$net_{o,j} = \sum_{i=1}^{H} h_i \cdot w_{ij}^{(2)} = \boldsymbol{h}^T \boldsymbol{W}_j^{(2)} \tag{4.31}$$

We can easily generalize this. Looking at the net output to a hidden layer l, we get

$$\boldsymbol{net}_{h_l} = \boldsymbol{h^{(l-1)}}^T \boldsymbol{W}^{(l)}. \tag{4.32}$$

4.1.3.2 Activation function

Before 2012, the logistic, the tanh and the pur linear functions where the standard activation functions, but then Alex Krizhevsky published an article where he introduced a new activation function called *Rectified Linear Units (ReLU)* which outperformed the classical activation functions. [4] The network he used is now known as AlexNet, and helped to revolutionize the field of computer vision. [5] After that, the ReLU activation function has been modified several times (avoiding zero derivative among others), and example of innovations are *leaky ReLU* and *Exponential Linear Unit (ELU)*. All those networks are linear for positive numbers, and small for negative numbers. Often, especially in the output layer, a straight linear function is used as well.

In figure (**??**), *standard RELU, leaky RELU* and *ELU* are plotted along with the logistic function.

4.1.3.3 Backward Propagation

Backward propagation is probably the most used technique for updating the weights, and is actually again based on equation (4.22). What differs, is the differentiation of the net input with respect to the weight, which gets more complex as we add more layers. For one hidden layer, we have two sets of weights, where the last layer is updated in a similar way as for a network without hidden layer, but the inputs are replaced with the values of the hidden nodes:

$$w_{ij}^{(2)+} = w_{ij}^{(2)} - \eta \cdot [f(h_i^T w_{ij}) - y_j]^T h_i. \tag{4.33}$$

We recognize the first part as δ_{ok}, such that

$$w_{ij}^{(1)+} = w_{ij}^{(1)} - \eta \cdot \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(2)} \cdot f'(out_{hj}) \cdot x_i \tag{4.34}$$

where we recall δ_{ok} as

$$\delta_{ok} = -(t_{ok} - out_{ok}) \cdot f'(out_{ok}).$$

For more layers, the procedure is the same, but we keep on inserting the obtained outputs from various layers.

4.1.3.4 Summary

Since it will be quite a lot calculations, I will just express the results here, and move the calculations to Appendix B. The forward phase in a three-layer perceptron is

$$\begin{aligned}
net_{hi} &= \sum_j w_{ji}^{(1)} \cdot x_j \\
out_{hi} &= \mathrm{f}(net_{hi}) \\
\\
net_{ki} &= \sum_j w_{ji}^{(2)} \cdot out_{hj} \\
out_{ki} &= \mathrm{f}(net_{ki}) \\
\\
net_{oi} &= \sum_j w_{ji}^{(3)} \cdot out_{kj} \\
out_{oi} &= \mathrm{f}(net_{oi})
\end{aligned} \tag{4.35}$$

which can easily be turned into vector form. The backward propagation follows from the two-layer example, and we get

$$w_{ij}^{(3)} = w_{ij}^{(3)} - \eta \cdot \delta_{oj} \cdot out_{ki}$$

$$w_{ij}^{(2)} = w_{ij}^{(2)} - \eta \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(3)} \cdot f'(out_{kj}) \cdot out_{hi}$$

$$w_{ij}^{(1)} = w_{ij}^{(1)} - \eta \sum_{k=1}^{O} \sum_{l=1}^{K} \delta_{ok} \cdot w_{lk}^{(3)} \cdot f'(out_{kl}) \cdot w_{jl}^{(2)} f'(out_{hj}) \cdot x_i$$

where we again use the short hand

$$\delta_{oj} = (t_j - out_{oj}) \cdot f'(out_{oj}).$$

If we compare with the weight update formulas for the two-layer case, we recognize some obvious connections, and it is easy to imagine how we can construct a general weight update algorithm, no matter how many layers we have.

Now over to the problem we want to solve using neural networks.

4.2 Unsupervised Learning

In unsupervised learning, a neural network is given the inputs only, and does not know what the output should look like. The task is then to find structures in the data, comparing data sets to each other and categorize the data sets with respect to their similarities and differences.

We have previously seen how the parameters can be adjusted using the backward propagation algorithm, but it does not work when we do not have prior known targets. Instead, we need to rely on a set of probabilities, and we therefore need to look carefully at the statistical foundation before we move on to the unsupervised algorithms.

4.2.1 Statistical foundation

In this section, we will explain the general relation between the joint probability distribution of two variables x and y, the marginal distributions and the conditional distributions. The expressions can either be set up with respect to the continuous space or the discrete space, and we will do the latter since we in practice will deal with discrete data sets.

The joint probability of measure both x and y is given by

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x) \tag{4.36}$$

where $p(x|y)$ and $p(y|x)$ are the conditional distributions of x and y respectively. $p(x)$ and $p(y)$ are called the marginal probabilities for x and y, and by reordering equation (4.36), we obtain Bayes' theorem

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \tag{4.37}$$

where $p(x)$ is the *prior* probability, $p(y|x)$ is the *likelyhood* function and $p(x|y)$ is the *posterior* probability. $p(y)$ can sometimes be found by a sum over the joint probability,

$$p(y) = \sum_i p(x_i, y) = \sum_i p(y|x_i)p(x_i), \tag{4.38}$$

but often this summation is intractable. Different techniques require different approaches to this problem, and for our case we will use Markov chain Monte-Carlo methods to bypass it. More about that in section (5).

For supervised learning, the The parameters will always be updated such that the probability is maximized. For instance,

Next challenge is that we do not have the posterior

Kullback-Leibler divergence gives a measure of how much information is loss when one goes from one probability distribution to another.

4.2.1.1 Marginal Distributions

4.2.1.2 Conditional Distributions

4.2.2 Boltzmann Machines

Boltzmann Machines are based on the more primitive Hopfield network, where a system of nodes is set up which defines the system energy. Inspired by statistical mechanics, the probability of finding the system in a state of energy E is given by the Boltzmann distribution,

$$P(\boldsymbol{s}) = \frac{1}{Z}\exp(-E(\boldsymbol{s})/k_B T), \tag{4.39}$$

hence the name Boltzmann machines. $\boldsymbol{s}$ includes all the nodes, k_B is known as Boltzmann's constant and T is the system temperature, but henceforth they both will be omitted by scaling $E'(\boldsymbol{s}) = E(\boldsymbol{s})/k_B T$. Z is known as the partition function, which is the sum over all possible probabilities.

In the most general form, all nodes are connected to all other nodes, that is an unrestricted Boltzmann machine, see figure (4.4) for an illustration.

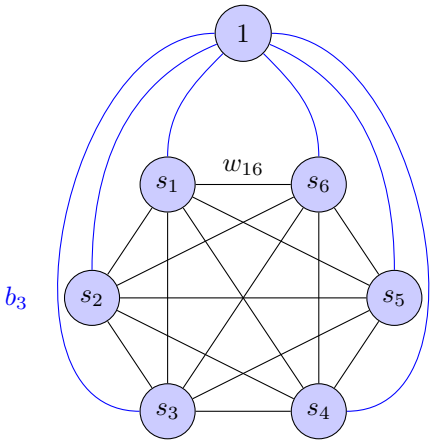


Figure 4.4: Unrestricted Boltzmann machine. Black lines are connections between all the nodes, where for instance the line between s_1 and s_6 is related to the weight w_{16}. The blue lines are related to the bias weights, and, for instance, the line going from the bias node to s_3 is related to b_3.

In the same manner as for a feed-forward neural network, we can directly multiply each node s_i with all its respective inner weights w_{ij} and then with the other nodes s_j. To obtain the total system energy, we also need to include the bias weights, i.e, multiply s_i with b_i. This gives the energy

$$E(\boldsymbol{s}) = -\sum_{i=1}^{N} s_i b_i - \sum_{i=1}^{N}\sum_{j=i}^{N} s_i w_{ij} s_j \tag{4.40}$$

for a system of N nodes, which is the so-called binary-binary network and the most basic architecture. During training, the weights are adjusted in order to maximize the probability...

4.2.3 Restricted Boltzmann Machines

When there is an unrestricted guy, a restricted guy must exist as well. What the term restricted means in this case, is that we ignore all the connections between nodes in the same layer, and keep only the inter-layer

ones. In the same manner as in equation (4.40), we can look at the linear case, where each node is multiplied with the corresponding weight, but now we need to distinguish between a visible node x_i and a hidden node h_j. For the same reason, all the bias weights need to divided into a group connected to the visible nodes, a_i and a group connected to the hidden nodes, b_j. The system energy then reads

$$E(\boldsymbol{x}, \boldsymbol{h}) = -\sum_{i=1}^{F} x_i a_i - \sum_{j=1}^{H} h_j b_j - \sum_{i=1}^{F}\sum_{j=i}^{H} x_i w_{ij} h_j \tag{4.41}$$

which is called binary-binary units or Bernoulli-Bernoulli units. F is the number of visible nodes and H is number of hidden nodes. In figure (4.5), a restricted Boltzmann machine with three visible nodes and three hidden nodes is illustrated.

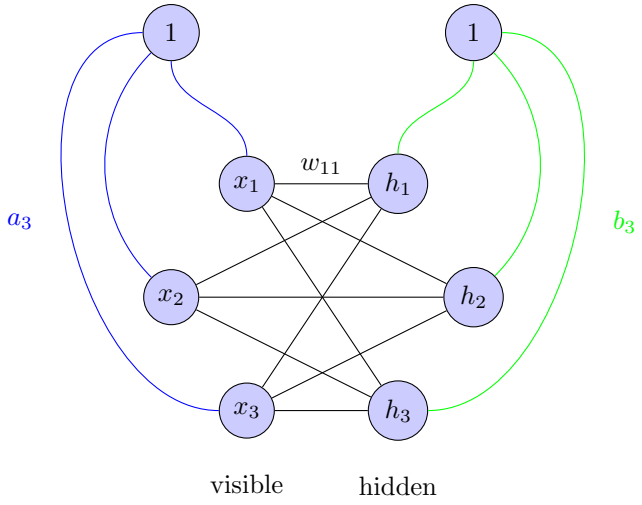


Figure 4.5: Restricted Boltzmann machine. Black lines are the inter-layer connections, where for instance the line between x_1 and h_1 is related to the weight w_{11}. The blue lines are related to the input bias weights, and, for instance, the line going from the bias node to x_3 is called a_3. Similarly, the green lines are connections between the hidden nodes and the bias, and, for instance, the line going from the bias node to h_3 is called b_3.

Until now we have discussed the linear models only, but as for feed-forward neural networks, we need non-linear models to solve non-linear problems. A natural next step is the Gaussian-binary units, which has a Gaussian mapping between the visible node bias and the visible nodes. The simplest such structure gives the following system energy:

$$E(\boldsymbol{x}, \boldsymbol{h}) = \sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{j=1}^{H} h_j b_j - \sum_{i=1}^{F}\sum_{j=i}^{H} \frac{x_i w_{ij} h_j}{\sigma_i^2} \tag{4.42}$$

where σ_i is the width of the Gaussian distribution, which can be set to an arbitrary number. Inserting the energy expression into equation (4.39), we obtain the general expression

$$P(\boldsymbol{x}, \boldsymbol{h}) = \exp\Big(-\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \exp\Big(h_j b_j + \sum_{i=1}^{F} \frac{h_j w_{ij} x_i}{\sigma^2}\Big). \tag{4.43}$$

which is the Gaussian-binary joint probability distribution. Generative sampling algorithms, as Gibbs' sampling, use this distribution directly, while other sampling tools, as Metropolis sampling, need the marginal distribution. Since the hidden nodes are binary, we just need to sum the joint probability distribution over $h = 0$ and $h = 1$ to find the marginal distributions. We obtain the expression

$$P(\boldsymbol{x}) = \exp\Big(-\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \Big(1 + \exp\Big(b_j + \sum_{i=1}^{F} \frac{w_{ij} x_i}{\sigma^2}\Big)\Big). \tag{4.44}$$

Similarly, the marginal distribution of the hidden nodes
More about the different sampling tools can be found in chapter 3.

4.2.4 Partly Restricted Boltzmann Machines

One can also imagine a partly restricted architecture, where we have connections inwards the visible nodes, but not the hidden nodes. This is what we have decided to call a partly restricted Boltzmann machine. A such neural network with three visible nodes and three hidden nodes is illustrated in figure (4.6).

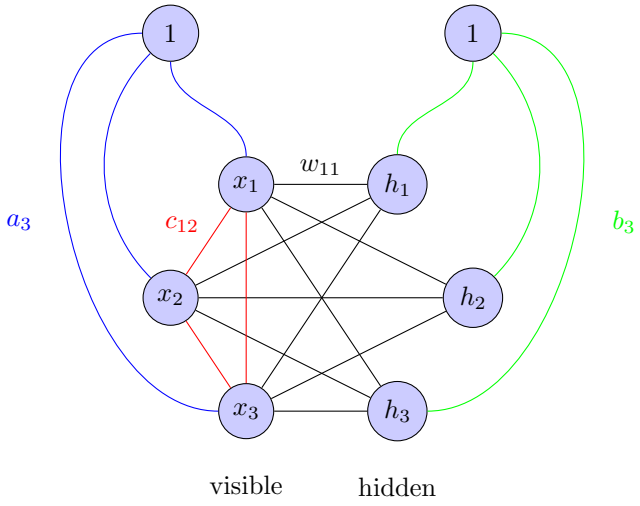


Figure 4.6: Partly restricted Boltzmann machine. Black lines are inter-layer connections, where for instance the line between x_1 and h_1 is related to the weight w_{11}. The blue lines are related to the input bias weights, and, for instance, the line going from the bias node to x_3 is related to a_3. Similarly, the green lines are related to the hidden nodes bias weights, and, for instance, the line going from the bias node to h_3 is related to b_3. Finally, the red lines are the intra-layer connections related to the intra-layer weights. The weight between node x_1 and x_2 is called c_{12}.

Compared to a standard restricted Boltzmann machine, we get an extra term in the energy expression where the visible nodes are connected. It is easy to find that the expression should be

$$E(\boldsymbol{x},\boldsymbol{h}) = \sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma_i^2} - \sum_{i=1}^{F}\sum_{j>i}^{F} x_i c_{ij} x_j - \sum_{j=1}^{H} h_j b_j - \sum_{i=1}^{F}\sum_{j=i}^{H}\frac{x_i w_{ij} h_j}{\sigma_i^2} \tag{4.45}$$

with c_{ij} as the weights between the visible nodes. For the later calculations, we are interested in the marginal distribution only, which reads

$$p(\boldsymbol{x}) = \exp\Big(-\sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma^2} + \sum_{i=1}^{F}\sum_{j>i}^{F} x_i c_{ij} x_j\Big)\prod_{j=1}^{H}\Big(1+\exp\Big(b_j+\sum_{i=1}^{F}\frac{w_{ij}x_i}{\sigma^2}\Big)\Big). \tag{4.46}$$

4.2.5 Deep Boltzmann Machines

We can also construct deep Boltzmann machines, where we just stack single-layer Boltzmann machines. There are many ways to construct those networks, where the number of layers, unit types, number of nodes and the degree of restriction can be chosen as the constructor wants. The number of combinations is endless, but in order to make use of the dept, all the layer should have different configurations. Otherwise, the deep network can be reduced to a shallower network. In figure (4.7) a restricted Boltzmann machine of two hidden layers is illustrated. We have chosen three hidden nodes in each layer, and three visible nodes. It should be trivial to imagine how the network can be expanded to more layers.

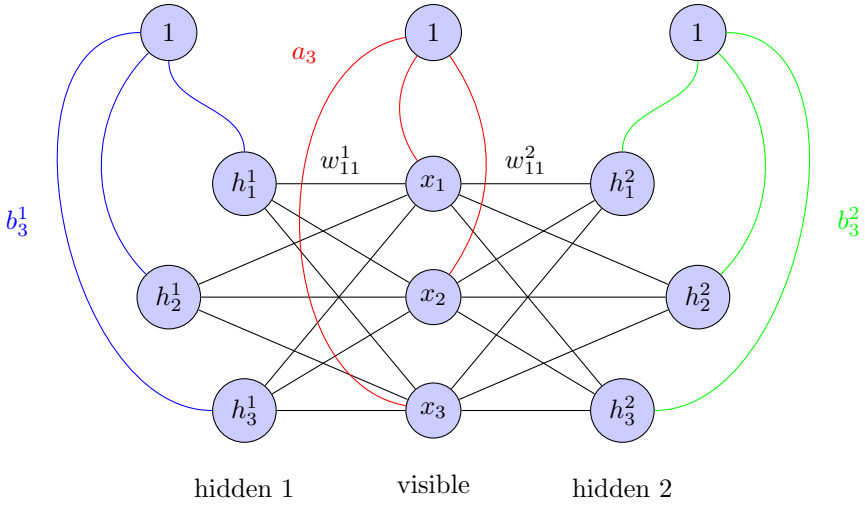


Figure 4.7: Deep restricted Boltzmann machine. Black lines the inter-layer connections, where for instance the line between x_1 and h_1 is related to the weight w_{11}. The blue lines are related to the input bias weights, and, for instance, the line going from the bias node to x_3 is related to a_3. Similarly, the green lines are related to the hidden nodes bias weights, and, for instance, the line going from the bias node to h_3 is related to b_3.

As the main focus so far has been restricted Boltzmann machines, also the deep networks will be assumed to be restricted, although both partly restricted and unrestricted can be constructed. The system energy of a deep restricted Boltzmann machine of L layers can be expressed as

$$E(\boldsymbol{x}, \boldsymbol{h}) = \sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{l=1}^{L}\sum_{j=1}^{H} h_j^l b_j^l - \sum_{l=1}^{L}\sum_{i=1}^{F}\sum_{j=i}^{H} \frac{x_i w_{ij}^l h_j^l}{\sigma_i^2} \tag{4.47}$$

which gives the marginal probability distribution

$$p(\boldsymbol{x}) = \exp\Big(-\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{l=1}^{L}\prod_{j=1}^{H} \Big(1 + \exp\Big(b_j^l + \sum_{i=1}^{F} \frac{w_{ij}^l x_i}{\sigma^2}\Big)\Big). \tag{4.48}$$

Part II

Advanced Theory

Chapter 5

Quantum Monte-Carlo Methods

Great quote.

Author

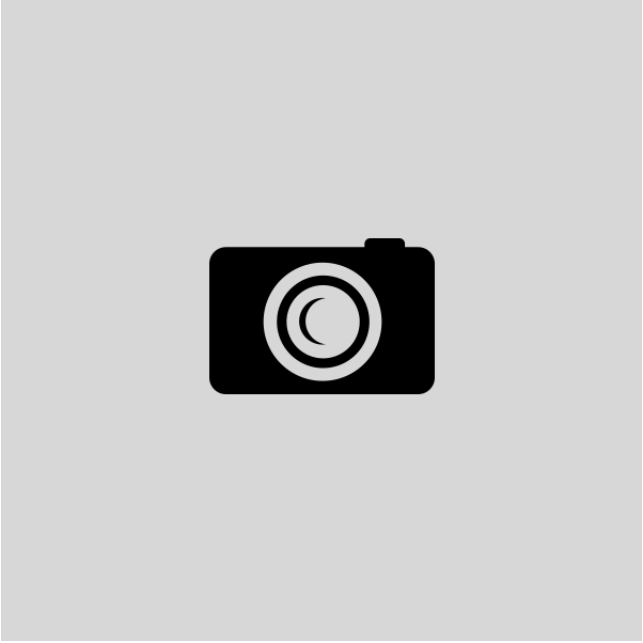


Figure 5.1: Caption

Some great methods for solving many-body quantum mechanical problems have been developed over the past decade.

Configuration interaction, Hartree-Fock, Coupled-Cluster, quantum Monte-Carlo methods. In this chapter we will discuss variational Monte-Carlo with and without a machine learning trial wave function.

For our work, we will focus on variational Monte-Carlo We will focus on the Hartree-Fock and variational Monte Carlo methods and give a detailed explanation of those methods. Additionally, the well-known methods configuration interaction and coupled cluster will be described briefly for some kind of completeness.

Monte Carlo methods in quantum mechanics are a bunch of methods that are built on diffusion processes, and includes Variational Monte Carlo (VMC), Diffusion Monte Carlo (DMC) and others. The common denominator is that we move particles in order to find the optimal configuration, usually where the energy is minimized.

5.1 Variational Monte-Carlo

The variational Monte-Carlo (hereafter VMC) method is today widely used when it comes to the study of ground state properties of quantum mechanical systems. It is a Markov chain Monte-Carlo method which makes use of Metropolis sampling, and has been used in studies of fermionic systems since the 1970's. [18] If we go back to the variational principle in equation (2.9), we see that by choosing a wave function which satisfies the criteria, we will get an energy larger or equal to the ground state energy.

There are two main problems we need to solve

1. We seldomly know the correct wave function
2. The integral we need to find the energy is hard or impossible to solve

Let us first determine the last problem, which often is considered as the root of all evil. Solving this integral analytically is impossible, but we can approximate it with a sum,

$$\begin{aligned} E &\leq \frac{\int d\boldsymbol{r}\Psi_T(\boldsymbol{r})^*\hat{\mathcal{H}}\Psi_T(\boldsymbol{r})}{\int d\boldsymbol{r}\Psi_T(\boldsymbol{r})^*\Psi_T(\boldsymbol{r})} \\ &= \int P(\boldsymbol{r})E_L(\boldsymbol{r})d\boldsymbol{r} \\ &\approx \frac{1}{M}\sum_{i=1}^{M} E_L(\boldsymbol{r}_i) \end{aligned} \tag{5.1}$$

which is a common trick in statistical physics. The local energy is defined as

$$E_L(\boldsymbol{r}) \equiv \frac{1}{\Psi_T(\boldsymbol{r})}\hat{\mathcal{H}}\Psi_T(\boldsymbol{r}) \tag{5.2}$$

and the $\boldsymbol{r}_i$ is withdrawn from the probability distribution $P(\boldsymbol{r})$, which is given by

$$P(\boldsymbol{r}) = \frac{|\Psi_T(\boldsymbol{r})|^2}{\int d\boldsymbol{r}|\Psi_T(\boldsymbol{r})|^2}. \tag{5.3}$$

When increasing the number of energies drawn from the distribution, M, henceforth denoted as Monte-Carlo cycles, the standard error decreases and we get a more accurate energy. The error goes as $\mathcal{O}(1/\sqrt{M})$, and in the limit when M goes to infinity, the error goes to zero,

$$\langle E\rangle = \lim_{M\to\infty}\frac{1}{M}\sum_{i=1}^{M} E_L(\boldsymbol{r}_i). \tag{5.4}$$

For more statistical details, see [18].

So far, so good, but how about the first problem stated above? How do we find the correct wave function? In VMC, we define a wave function with variational parameters, which are adjusted in order to minimize the energy for every iteration. Of course, we need a decent initial guess, which is usually based on our physical intuition. We will later examine how much physical intuition we need to get an acceptable result.

For every iteration, we run M Monte-Carlo cycles where we withdraw a new position $\boldsymbol{r}_i$. Whether or not the proposed move should be accepted is determined by the Metropolis algorithm.

5.2 The Metropolis Algorithm

Metropolis sampling is a method of accepting or rejecting moves in Markov chains, and is today often the preferred sampling algorithm in quantum Monte-Carlo. The genius of this algorithm, is that the acceptance of a move is not based on the probabilities themselves, but the ratio between the new and the old probabilities. In that way, we avoid calculating the sum over all probabilities, which often is expensive or even impossible to calculate.

If we denote $\boldsymbol{r}$ as the current state, and $\boldsymbol{r}'$ as the proposed state, we have a transition rule $P(\boldsymbol{r}'|\boldsymbol{r})$ for going from $\boldsymbol{r}$ to $\boldsymbol{r}'$ and a transition rule $P(\boldsymbol{r}|\boldsymbol{r}')$ for going the other way around. If we then assume that the rules satisfy *ergodicity* and *detailed balance*, we have the following relationship:

$$P(\boldsymbol{r}'|\boldsymbol{r})P(\boldsymbol{r}) = P(\boldsymbol{r}|\boldsymbol{r}')P(\boldsymbol{r}'). \tag{5.5}$$

The next step is to rewrite the transition rules in terms of a proposal distribution $T(\boldsymbol{r}'|\boldsymbol{r})$ and an acceptance probability $A(\boldsymbol{r}', \boldsymbol{r})$,

$$P(\boldsymbol{r}'|\boldsymbol{r}) = T(\boldsymbol{r}'|\boldsymbol{r})A(\boldsymbol{r}', \boldsymbol{r}). \tag{5.6}$$

In order to satisfy the detailed balance, we need to choose $A(\boldsymbol{r} \to \boldsymbol{r}')$ such that

$$A(\boldsymbol{r}', \boldsymbol{r}) = \min\left[1, \frac{T(\boldsymbol{r}|\boldsymbol{r}')P(\boldsymbol{r}')}{T(\boldsymbol{r}'|\boldsymbol{r})P(\boldsymbol{r})}\right], \tag{5.7}$$

since A cannot be larger than 1. If the acceptance is higher than a random number between 0 and 1, the move is accepted.

5.2.1 Brute-Force Sampling

In its simplest form, the move is proposed randomly both in magnitude and direction. Mathematically, we can write this as

$$\boldsymbol{r}' = \boldsymbol{r} + sd\boldsymbol{r} \tag{5.8}$$

where s is a random number which determines the distance to move and $d\boldsymbol{r}$ is a random direction (typically which particle to move). We obtain the naive acceptance probability when requiring $T(\boldsymbol{r}'|\boldsymbol{r}) = T(\boldsymbol{r}|\boldsymbol{r}')$, such that the it simplifies to

$$A(\boldsymbol{r}', \boldsymbol{r}) = \min\left[1, \frac{P(\boldsymbol{r}')}{P(\boldsymbol{r})}\right]. \tag{5.9}$$

However, with this approach a lot of moves will be rejected, which results in a significant waste of computing power. A better method is **importance sampling**.

5.2.2 Importance Sampling

Importance sampling is a more intelligent sampling method than the brute-force sampling, since the new position is based on an educated guess. To understand how it works, we need to take a quick look at diffusion processes. We start from the Fokker-Planck equation,

$$\frac{\partial P(\boldsymbol{r}, t)}{\partial t} = D\nabla\left(\nabla - \boldsymbol{F}\right)P(\boldsymbol{r}, t) \tag{5.10}$$

which describes how a probability distribution $P(\boldsymbol{r}, t)$ evolves in appearance of a drift force $\boldsymbol{F}$. In the case $\boldsymbol{F} = 0$, the equation reduces to the diffusion equation with D as the diffusion constant. This simplifies to $D = 1/2$ in atomic units.

The Langevin equation states that a diffusion particle tends to move parallel to the drift force in the coordinate space, but because of a random variable $\boldsymbol{\eta}$ this is not always true. The equation reads

$$\frac{\partial \boldsymbol{r}(t)}{\partial t} = D\boldsymbol{F}(\boldsymbol{r}(t)) + \boldsymbol{\eta}. \tag{5.11}$$

Given a position $\boldsymbol{r}$, the new position $\boldsymbol{r}'$ can be be found by applying forward-Euler on equation (5.11),

(5.12)

where Δt is a fictive time step and $\boldsymbol{\xi}$ is a Gaussian random variable. The next thing we want to find, is an expression for the drift force $\boldsymbol{F}$ which makes the system converge to a stationary state.

A stationary state is found when the probability density $P(\boldsymbol{r})$ is constant in time, i.e, when the left-hand-side of Fokker-Planck is zero. In that case, we can write the equation as

$$\nabla^2 P(\boldsymbol{r}) = P(\boldsymbol{r})\nabla\boldsymbol{F}(\boldsymbol{r}) + \boldsymbol{F}(\boldsymbol{r})\nabla P(\boldsymbol{r}). \tag{5.13}$$

In the next, we assume that the drift force has the form $\boldsymbol{F}(\boldsymbol{r}) = g(\boldsymbol{r})\nabla P(\boldsymbol{r})$, since the force should point to a higher probability. We can then go further and write

$$\nabla^2 P(\boldsymbol{r})\big(1 - P(\boldsymbol{r})g(\boldsymbol{r})\big) = \nabla\big(g(\boldsymbol{r})P(\boldsymbol{r})\big)\nabla P(\boldsymbol{r}) \tag{5.14}$$

which is satisfied when $g(\boldsymbol{r}) = 1/P(\boldsymbol{r})$. We then get the drift force

$$\boldsymbol{F}(\boldsymbol{r}) = \frac{\nabla P(\boldsymbol{r})}{P(\boldsymbol{r})} = 2\frac{\nabla \Psi_T(\boldsymbol{r})}{\Psi_T(\boldsymbol{r})}, \tag{5.15}$$

which is also known as the *quantum force*.

The remaining part is how to decide if a proposed move should be accepted or not. For this, we need to find the sampling distributions $T(\boldsymbol{r}'|\boldsymbol{r})$ from equation (5.7), which are just the solutions of the Fokker-Planck equation. The solutions read

$$G(\boldsymbol{r}, \boldsymbol{r}', \Delta t) \propto \exp\Big(- \big(\boldsymbol{r} - \boldsymbol{r}' - D\Delta t \boldsymbol{F}(\boldsymbol{r})\big)^2 / 4D\Delta t\Big) \tag{5.16}$$

which is called Green's functions. They correspond to the normal distribution $\mathcal{N}(\boldsymbol{r}|\boldsymbol{r}' + D\Delta t\boldsymbol{F}(\boldsymbol{r}), 2D\Delta t)$. The acceptance probability for importance sampling can finally be written as

$$A(\boldsymbol{r}'|\boldsymbol{r}) = \min\ \left[1, \frac{G(\boldsymbol{r}, \boldsymbol{r}', \Delta t)P(\boldsymbol{r}')}{G(\boldsymbol{r}', \boldsymbol{r}, \Delta t)P(\boldsymbol{r})}\right], \tag{5.17}$$

where the marginal probabilities are still given by equation (5.3). As a summary, we set up the actual algorithm, known as Metropolis-Hasting's algorithm, see algorithm (1).

Algorithm 1: The Metropolis-Hastings algorithm

Data: Initialize particle positions $\boldsymbol{r}$ and parameters θ randomly.
Result: The optimized trial wave function.
$\boldsymbol{r}' = \mathcal{N}(0, 1)$;
$\theta = \mathcal{N}(0, 1)$;
$P(\boldsymbol{r}'; \theta) = |\Psi_T(\boldsymbol{r}'; \theta)|^2$;
G;
while *not converged* **do**
  $\boldsymbol{r} = \boldsymbol{r}'$;
  $P(\boldsymbol{r}; \theta) = P(\boldsymbol{r}'; \theta)$;
  $G(\boldsymbol{r}, \boldsymbol{r}', \Delta t; \theta) = G(\boldsymbol{r}', \boldsymbol{r}, \Delta t; \theta)$;
  ;
  $\boldsymbol{r}' = \boldsymbol{r} + D\boldsymbol{F}(\boldsymbol{r}; \boldsymbol{\theta})\Delta t + \boldsymbol{\xi}\sqrt{\Delta t}$;
  $p = P(\boldsymbol{r}'; \theta)/P(\boldsymbol{r}; \theta)$;
  $g = G(\boldsymbol{r}', \boldsymbol{r}, \Delta t; \theta)/G(\boldsymbol{r}, \boldsymbol{r}', \Delta t; \theta)$;
  $w = gp$;
  $r = \mathcal{U}(0, 1)$;
  if $w < r$ **then**
    $\boldsymbol{r}' = \boldsymbol{r}$;
  else
    keep going;
  end
end

5.2.3 Gibbs Sampling

In the machine learning community, Gibbs sampling is widely used when it comes to training Boltzmann machines. It is an instance of the Metropolis-Hastings algorithm, but since the units are updated to maximize the probabilities, all the moves are accepted. The algorithm will be discussed for the restricted Boltzmann case only.

Given an initial set of coordinates $\boldsymbol{r}$ and $\boldsymbol{h}$, one can use the conditional probability $P(\boldsymbol{r}|\boldsymbol{h})$ to find a new set of

5.3 Diffusion Monte-Carlo

Diffusion Monte-Carlo is based on the time-imaginary Schrödinger equation.

Chapter 6

The Hartree-Fock Method

Hartree-Fock is like a good kitchen tool; it can be used to prepare several different dishes.

Morten Hjorth-Jensen

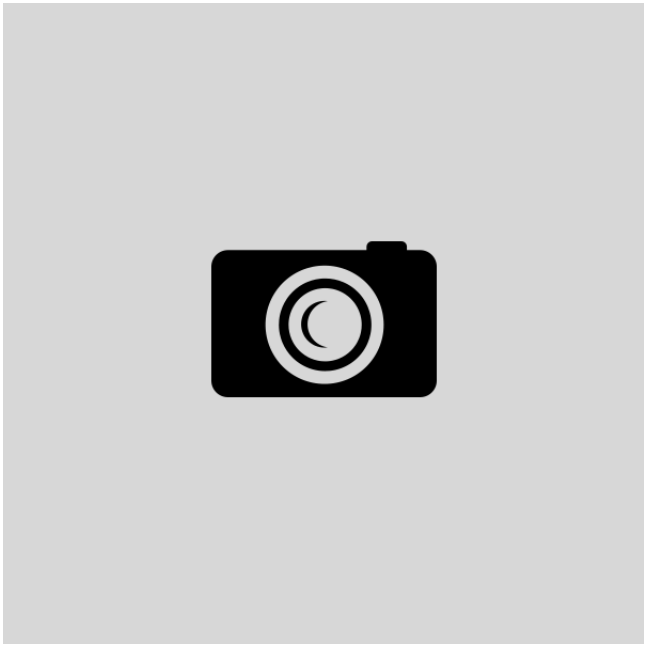


Figure 6.1: Caption

Hartree-Fock is an algorithm for finding an approximative expression for the ground state energy of a Hamiltonian. Ever since the childhood of many-body quantum mechanics, it has been an important quantum many-body method due to its accuracy contra expensiveness.

The method sets up a mean-field potential, often called the Hartree-Fock potential, which replaces the two-body repulsive interaction found in electronic structure calculations. Thereafter, a new basis is expanded in a given basis, and the optimal coefficients with respect to the mean-field constraint are generated. Traditionally, the method has been called the self-consistent field method, but today Hartree-Fock is mostly used.

Since we are replacing the interaction term, we need to define a new operator to replace the Hamiltonian, known as the Hartree-Fock operator,

$$\hat{h}^{\mathrm{HF}} = \hat{t} + \hat{u}_{\mathrm{ext}} + \hat{u}^{\mathrm{HF}}. \tag{6.1}$$

The $\hat{u}^{\mathrm{HF}}$ is a single-particle potential, and will later be determined by the algorithm. The operator can be used directly in the Schrödinger equation

$$\hat{h}^{\mathrm{HF}} |p\rangle = \epsilon_\alpha |p\rangle \tag{6.2}$$

where the obtained energy is called the Hartree-Fock energy. The Hartree-Fock basis is found from the following expansion,

$$|p\rangle = \sum_\lambda C_{p\lambda} |\lambda\rangle , \tag{6.3}$$

where the initial basis is denoted with Greek letters and the new basis is denoted with Roman letters. $C_{p\lambda}$ are the coefficients that form an orthogonal basis.

6.1 Restricted Hartree-Fock

When we calculated the reference energy above, our basis contained only one Slater determinant, more specifically the ground state. In Hartree-Fock, we still have a single Slater determinant basis, but we now construct new SPFs with the constraint of minimizing the energy.

In general, one can change from one single-particle basis to another by a unitary transform,

$$|p\rangle = \sum_\alpha c_{p\alpha} |\alpha\rangle , \tag{6.4}$$

where we use greek letters for the old basis and roman letters for the new one. If we then insert into (**??**), we get a find energy formula with coefficients, $C_{p\lambda}$, that we can vary

$$E = \sum_p^N \sum_{\alpha\beta} C^*_{p\alpha} C_{p\beta} \langle\alpha|\hat{h}_0|\beta\rangle + \frac{1}{2} \sum_{pq}^N \sum_{\alpha\beta\gamma\delta} C^*_{p\alpha} C^*_{q\beta} C_{p\gamma} C_{q\delta} \langle\alpha\beta|\hat{v}|\gamma\delta\rangle_{\mathrm{AS}} . \tag{6.5}$$

Further, we assume that also our new basis is orthonormal, i.e,

$$\langle p|q\rangle = \sum_\alpha c^*_{p\alpha} c_{q\alpha} \langle\alpha|\alpha\rangle = \sum_\alpha c^*_{p\alpha} c_{q\alpha} = \delta_{pq} \tag{6.6}$$

$$\Rightarrow \sum_\alpha c^*_{p\alpha} c_{q\alpha} - \delta_{pq} = 0 \tag{6.7}$$

We now have a function, E, that we want to minimize with respect to a constraint given in equation (6.7). This is a typical situation where Lagrange Multipliers is convenient to use, which in this case can be written as

$$\mathcal{L}(\{C_{p\alpha}\}) = E(\{C_{p\alpha}\}) - \sum_a \varepsilon_a \Big(\sum_\alpha c^*_{p\alpha} c_{q\alpha} - \delta_{pq} \Big). \tag{6.8}$$

The variation in reference energy is then find to be

$$\delta E = \sum_{k\alpha} \frac{\partial E}{\partial C^*_{k\alpha}} \delta C^*_{k\alpha} + \sum_{k\alpha} \frac{\partial E}{\partial C_{k\alpha}} \delta C_{k\alpha} - \sum_{k\alpha} \varepsilon_k (C_{k\alpha} \delta C^*_{k\alpha} + C^*_{k\alpha} \delta C_{k\alpha}) \tag{6.9}$$

which is zero when E is minimized. Each coefficient $C_{k\alpha}$ and $C^*_{k\alpha}$ is independent, so they can be varied independently. Thus

$$\left(\frac{\partial E}{\partial C^*_{k\alpha}} - \varepsilon_k C_{k\alpha} \right) \delta C^*_{k\alpha} = 0, \tag{6.10}$$

which is satisfied if and only if

$$\frac{\partial E}{\partial C^*_{k\alpha}} - \varepsilon_k C_{k\alpha} = 0 \qquad \forall\ k, \alpha \tag{6.11}$$

The first term can be derived from (6.9), and reads

$$\frac{\partial E}{\partial C^*_{k\alpha}} = \sum_\beta C_{k\beta} \langle\alpha|\hat{h}_0|\beta\rangle + \sum_p^N \sum_{\beta\gamma\delta} C^*_{p\beta} C_{k\gamma} C_{p\delta} \langle\alpha\beta|\hat{v}|\gamma\delta\rangle_{\mathrm{AS}} . \tag{6.12}$$

This results in the equation

$$\sum_{\gamma} \hat{h}^{\mathrm{HF}}_{\alpha\gamma} C_{k\gamma} = \varepsilon_k C_{k\gamma} \tag{6.13}$$

where we have defined

$$\hat{h}^{\mathrm{HF}}_{\alpha\gamma} \equiv \langle \alpha | \hat{h}_0 | \gamma \rangle + \sum_{p}^{N} \sum_{\beta\delta} C^*_{p\beta} C_{p\delta} \langle \alpha\beta | \hat{v} | \gamma\delta \rangle_{\mathrm{AS}} \,. \tag{6.14}$$

We recognize that (6.13) can be written as a matrix-vector product

$$\hat{h}^{\mathrm{HF}} C_k = \varepsilon^{\mathrm{HF}}_k C_k \tag{6.15}$$

where C_k are columns in our coefficient matrix and $\varepsilon^{\mathrm{HF}}_k$ are just the eigenvalues of $\hat{h}^{\mathrm{HF}}$, they have no physical significance. We will use this equation to find the optimal SPFs (optimal C_k's) and then find the energy from equation (6.9).

$$\hat{h}^{\mathrm{HF}}(C^{i+1}_k) C^i_k = \varepsilon^{\mathrm{HF}}_k C^i_k \tag{6.16}$$

Usually one initialize this with $\hat{C} = \hat{\mathcal{I}}$, the identity matrix.

6.2 Unrestricted Hartree-Fock

Chapter 7

Post Hartree-Fock Methods

Great quote.

Author

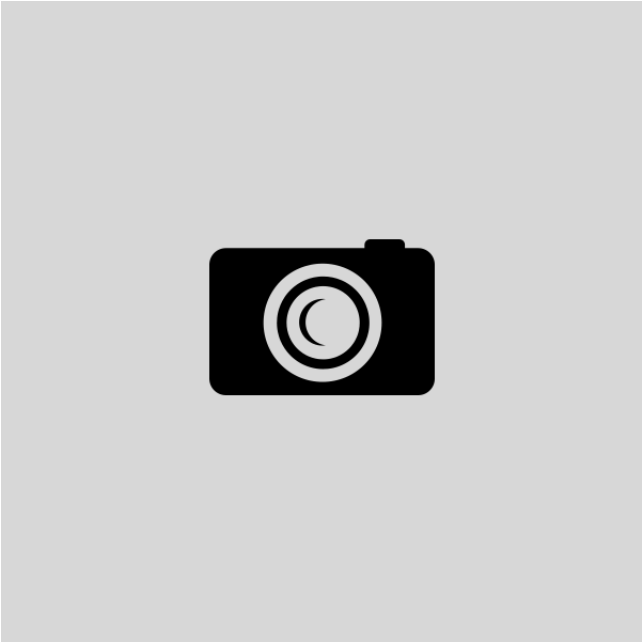


Figure 7.1: Caption

Post Hartree-Fock methods are usually *ab initio* methods where Hartree-Fock can be used as input. The term *ab initio* means from first principles, implying that only physical constants are put into the methods. The Monte-Carlo methods are not considered *ab initio* as nonphysical hyper parameters are required. The methods we will discuss here are the **configuration interaction** method and the **coupled cluster** method.

7.1 Configuration Interaction

The configuration interaction method is in many ways the the most intuitive method, and some will argue that it is the natural starting point.

Often, we know the true wavefunctions $|\Phi_i\rangle$ in the external potential, but are off when interaction is added

$$\hat{H}_0 |\Phi_i\rangle = \varepsilon_i |\Phi_i\rangle , \qquad (\hat{H}_0 + \hat{H}_I) |\Phi_i\rangle \neq \varepsilon_i |\Phi_i\rangle . \tag{7.1}$$

However, the Slater determinants form a ket basis, meaning we can write out eigenstates of $\hat{H}_I$ as a linear combination of the determinants

$$\begin{array}{ccccccccc}
|\Psi_0\rangle & = & C_0^{(0)} |\Phi_0\rangle & + & C_1^{(0)} |\Phi_1\rangle & + & \dots & + & C_{N-1}^{(0)} |\Phi_{N-1}\rangle \\
|\Psi_1\rangle & = & C_0^{(1)} |\Phi_0\rangle & + & C_1^{(1)} |\Phi_1\rangle & + & \dots & + & C_{N-1}^{(1)} |\Phi_{N-1}\rangle \\
|\Psi_2\rangle & = & C_0^{(2)} |\Phi_0\rangle & + & C_1^{(2)} |\Phi_1\rangle & + & \dots & + & C_{N-1}^{(2)} |\Phi_{N-1}\rangle \\
\vdots & & \vdots & & \vdots & & & & \vdots \\
|\Psi_{N-1}\rangle & = & C_0^{(N-1)} |\Phi_0\rangle & + & C_1^{(N-1)} |\Phi_1\rangle & + & \dots & + & C_N^{(N-1)} |\Phi_{N-1}\rangle
\end{array} \tag{7.2}$$

such that

$$\hat{H} |\Psi_p\rangle = \varepsilon_p |\Psi_p\rangle . \tag{7.3}$$

The Hamiltonian can be rewritten as a double sum over all states using the so-called *completeness relation*,

$$\hat{H} = \sum_{ij} |\Phi_i\rangle \, \langle\Phi_i|\hat{H}|\Phi_j\rangle \, \langle\Phi_j| \tag{7.4}$$

such that the Schrödinger equation can be rewritten as

$$\begin{pmatrix}
\langle\Phi_0|\hat{H}|\Phi_0\rangle & \langle\Phi_0|\hat{H}|\Phi_1\rangle & \dots & \langle\Phi_0|\hat{H}|\Phi_{N-1}\rangle \\
\langle\Phi_1|\hat{H}|\Phi_0\rangle & \langle\Phi_1|\hat{H}|\Phi_1\rangle & \dots & \langle\Phi_1|\hat{H}|\Phi_{N-1}\rangle \\
\vdots & \vdots & \ddots & \vdots \\
\langle\Phi_{N-1}|\hat{H}|\Phi_0\rangle & \langle\Phi_{N-1}|\hat{H}|\Phi_1\rangle & \dots & \langle\Phi_{N-1}|\hat{H}|\Phi_{N-1}\rangle
\end{pmatrix}
\begin{pmatrix} c_0^{(p)} \\ c_1^{(p)} \\ \vdots \\ c_{N-1}^{(p)} \end{pmatrix}
= \varepsilon_p
\begin{pmatrix} c_0^{(p)} \\ c_1^{(p)} \\ \vdots \\ c_{N-1}^{(p)} \end{pmatrix} \tag{7.5}$$

Until now, we have not made any assumptions, such that the equation above will give exact results when all single particle functions are included. The problem is that the matrix scales so badly, the number of Slater determinant that we need to include goes as

$$N_{\text{FCI}} = \binom{N_{\text{single orbitals}}}{N_{\text{electrons}}} \tag{7.6}$$

which is exploding. This is quite annoying since we in principle know how to solve the problems exact.

7.2 Coupled Cluster

The coupled cluster method is the *de facto* standard wave function-based method for electronic structure calculations. [10] The method approximates the wave function with an exponential expansion,

$$|\Psi_{\text{CC}}\rangle = e^{\hat{T}} |\Phi_0\rangle \tag{7.7}$$

where $\hat{T}$ is the cluster operator, entirely given by $\hat{T} = \hat{T}_1 + \hat{T}_2 + \hat{T}_3 + \dots$ with

$$\hat{T}_n = \left(\frac{1}{n!}\right)^2 \sum_{abc\dots} \sum_{ijk\dots} t_{ijk\dots}^{abc\dots} a_a^\dagger a_b^\dagger a_c^\dagger \cdots a_k a_j a_i. \tag{7.8}$$

We again want to solve the Schrödinger equation,

$$\hat{H} |\Psi\rangle = \hat{H} e^{\hat{T}} |\Phi_0\rangle = \epsilon e^{\hat{T}} |\Phi_0\rangle , \tag{7.9}$$

which can be simplified by multiplying with $e^{-\hat{T}}$ from the left. This introduces us to the **similarity transformed Hamiltonian**

$$\bar{H} = e^{-\hat{T}} \hat{H} e^{\hat{T}}. \tag{7.10}$$

If we on one hand now multiply with the reference bra on the left hand side, we easily observe that

$$\langle \Phi_0 | \bar{H} | \Phi_0 \rangle = \epsilon \tag{7.11}$$

which is the coupled cluster energy equation. On the other hand, we can multiply with an excited bra on left hand side, and find that

$$\left\langle \Phi_{ijk\ldots}^{abc\ldots} \right| \bar{H} \left| \Phi_0 \right\rangle = 0 \tag{7.12}$$

which are the coupled cluster amplitude equations. The similarity transformed Hamiltonian can be rewritten using the Baker-Campbell-Hausdorff expansion

$$\begin{aligned} \bar{H} = \hat{H} &+ [\hat{H}, \hat{T}] \\ &+ \frac{1}{2}[[\hat{H}, \hat{T}], \hat{T}] \\ &+ \frac{1}{6}[[[\hat{H}, \hat{T}], \hat{T}], \hat{T}] \\ &+ \frac{1}{24}[[[[\hat{H}, \hat{T}], \hat{T}], \hat{T}], \hat{T}] \\ &+ \cdots \end{aligned} \tag{7.13}$$

and we are in principle set to solve the amplitude equations with respect to the amplitudes $t_{ijk\ldots}^{abc\ldots}$ and then find the energy. The expansion is able to reproduce the true wave function exactly using a satisfying number of terms and an infinite basis. This is, of course, not possible, but even by limiting us to the first few coupled cluster operators, the results are often good compared to other methods. [12]

Part III

Implementation Preparatory

Chapter 8

Derivation of Wave Function Elements

In chapter (2) we presented the basic principles behind a many-body trial wave function, including the Slater determinant and the well-known Padé-Jastrow factor. Further in chapter (3), the common basis functions of the quantum dot and atomic systems were given, and in the previous chapter, (4), we explained how to create wave functions using Boltzmann machines. This means that all wave function elements used in this thesis already are presented, and in this chapter they are all collected, together with their derivatives and various optimizations. The calculations below are based on two main assumptions:

1. For each time step, we change one position coordinate only, i.e, move a particle along one of the principal axis.

2. A variational parameter α_i appears in only one of the wave function elements.

The first assumption is useful when updating position dependent arrays. Typically, we only need to update a coordinate of an array or a row of a matrix when this assumption is raised, which is hugely beneficial with respect to the computational time. The last assumption is makes all wave function elements independent, which obviously makes life easier. We will now show how the local energy can be split up and calculated independently for the different elements, and then we will take a look at how the parameters are updated.

8.1 Kinetic Energy Calculations

The local energy, defined in equation (5.2), is

$$E_L = \frac{1}{\Psi_T(\boldsymbol{r})}\hat{\mathcal{H}}\Psi_T(\boldsymbol{r}) \tag{8.1}$$

$$= \sum_{k=1}^{M}\Big[-\frac{1}{2}\Big(\frac{1}{\Psi_T(\boldsymbol{r})}\nabla_k^2\Psi_T(\boldsymbol{r})\Big)+\mathcal{V}\Big]. \tag{8.2}$$

The first term, which is the kinetic energy term, is the only wave function-dependent one. It will in this chapter be evaluated for various wave function elements. From the definition of differentiation of a logarithm, we have that

$$\frac{1}{\Psi_T(\boldsymbol{r})}\nabla_k\Psi_T(\boldsymbol{r}) = \nabla_k \ln\Psi_T(\boldsymbol{r}), \tag{8.3}$$

which provides the following useful relation

$$\frac{1}{\Psi_T(\boldsymbol{r})}\nabla_k^2\Psi_T(\boldsymbol{r}) = \nabla_k^2 \ln\Psi_T(\boldsymbol{r}) + (\nabla_k \ln\Psi_T(\boldsymbol{r}))^2. \tag{8.4}$$

Consider a trial wave function, Ψ_T, consisting of a product of p wave function elements, $\{\phi_1, \phi_2 \dots \phi_p\}$,

$$\Psi_T(\boldsymbol{r}) = \prod_{i=1}^{p}\phi_i(\boldsymbol{r}). \tag{8.5}$$

The kinetic energy related to this trial wave function is then computed by

$$\frac{1}{\Psi_T(\boldsymbol{r})}\nabla_k^2\Psi_T(\boldsymbol{r}) = \sum_{i=1}^{p}\nabla_k^2\ln\phi_i(\boldsymbol{r}) + \Big(\sum_{i=1}^{p}\nabla_k\ln\phi_i(\boldsymbol{r})\Big)^2, \tag{8.6}$$

which can be found when all local derivatives $\nabla_k^2\ln\phi_i(\boldsymbol{r})$ and $\nabla_k\ln\phi_i(\boldsymbol{r})$ are given. For each wave function element given below, those local derivatives will be evaluated. In addition, we need to know the derivative of local energy with respect to the variational parameters in order to update the parameters correctly.

8.2 Parameter Update

In gradient based optimization methods, as we use, one needs to know the gradient of the expectation value of local energy with respect to all variational parameters α_i,

$$\partial_{\alpha_i}\langle E_L\rangle \equiv \frac{\partial\langle E_L(\alpha_i)\rangle}{\partial\alpha_i}. \tag{8.7}$$

Since we are dealing with an expectation value, this gradient can be found from

$$\partial_{\alpha_i}\langle E_L\rangle = 2\Big(\langle E_L\partial_{\alpha_i}\ln\Psi_T\rangle - \langle E_L\rangle\langle\partial_{\alpha_i}\ln\Psi_T\rangle\Big) \tag{8.8}$$

which means that we need to calculate the expectation values $\langle E_L\partial_{\alpha_i}\ln\Psi_T\rangle$ and $\langle\partial_{\alpha_i}\ln\Psi_T\rangle$ in addition to the local energy. Those expectation values are found from the integrals

$$\langle\partial_{\alpha_i}\ln\Psi_T\rangle = \int_{-\infty}^{\infty} d\boldsymbol{r}P(\boldsymbol{r})\partial_{\alpha_i}\ln\Psi_T(\boldsymbol{r}) \tag{8.9}$$

and

$$\langle E_L\partial_{\alpha_i}\ln\Psi_T\rangle = \int_{-\infty}^{\infty} d\boldsymbol{r}P(\boldsymbol{r})E_L(\boldsymbol{r})\partial_{\alpha_i}\ln\Psi_T(\boldsymbol{r}), \tag{8.10}$$

which can also be found by Monte-Carlo integration.

8.3 Optimizations

How much a wave function element can be optimized heavily depends on the specific form of the element. For instance, sometimes the previous and present $\nabla_k\ln\phi_i$ are closely related, and only differ with a few calculations, while for some other elements they are not related at all. Those subjective optimizations will therefore be described when presenting each wave function element.

However, there are still optimizations that apply to all elements and give great speed-up. An example is when calculating the ratio between the previous and present wave functions for all wave function elements instead of the wave function itself. Firstly, this is usually cheaper to calculate than the wave function itself because we are working in the logarithm space. Secondly, the ratio is actually what we use in the sampling, so it is a natural thing to calculate. The total wave function ratio is just the product of all the wave function element ratios

$$\frac{\Psi_T^{\text{new}}}{\Psi_T^{\text{old}}} = \prod_{i=1}^{p}\frac{\phi_i^{\text{new}}}{\phi_i^{\text{old}}}$$

8.4 Derivatives

8.4.1 Simple Gaussian

A natural starting point is the Gaussian function, since it appears in standard variational Monte-Carlo computations of quantum dot systems. For P number of particles and F free dimensions, the function is given by

$$\Psi(\boldsymbol{x};\alpha) = \exp\Big(-\frac{1}{2}\omega\alpha\sum_{i=1}^{P}r_i^2\Big) = \exp\Big(-\frac{1}{2}\omega\alpha\sum_{i=1}^{F}x_i^2\Big)$$

similarly to the function presented in section (**??**). ω is the oscillator strength and α is a variational parameter, which for non-interacting atoms is 1. Due to the presence of r_i^2, the function can easily be treated both in Cartesian and spherical coordinates, but in this thesis we will focus on the former.

The gradient with respect to coordinate x_k is

$$\nabla_k \ln \Psi(\alpha) = -\omega\alpha x_k$$

and the corresponding Laplacian is

$$\nabla_k^2 \ln \Psi(\alpha) = -\omega\alpha.$$

Finally, for the parameter update we have that

$$\partial_\alpha \ln \Psi = -\frac{1}{2}\omega \sum_{i=1}^{M} x_i^2. \tag{8.11}$$

Since this wave function element is quite simple, there is no special optimization available that will cause a noticeable performance improvement. One can calculate $\omega\alpha$ once to save a few floating point operations, and of course calculate the probability ratio. Collecting all expressions, we end up with

$$\begin{aligned}
\frac{\Psi_{\text{new}}^2}{\Psi_{\text{old}}^2} &= \exp\left(\omega\alpha(x_{i,\text{old}}^2 - x_{i,\text{new}}^2)\right) \\
\nabla_k \ln \Psi &= -\omega\alpha x_k \\
\nabla_k^2 \ln \Psi &= -\omega\alpha \\
\partial_\alpha \ln \Psi &= -\frac{1}{2}\omega \sum_{i=1}^{M} x_i^2,
\end{aligned} \tag{8.12}$$

where i is the changed coordinate.

8.4.2 Padé-Jastrow Factor

The Padé-Jastrow factor is introduced in order to take care of the correlations. It is specified in equation (2.18),

$$J(\boldsymbol{r};\beta) = \exp\left(\sum_{i=1}^{P}\sum_{j>i}^{P} \frac{a_{ij}r_{ij}}{1+\beta r_{ij}}\right),$$

where P is the number of particles, r_{ij} is the relative distance between particle i and j and β is a variational parameter. One challenge is that we operate in Cartesian coordinates, while the expressed Jastrow factor obviously is easier to handle in spherical coordinates. Since we need to differentiate this with respect to all free dimensions, we need to be careful not confuse the particle indices and coordinate indices. Let us define i as the coordinate index and i' as the index on the corresponding particle. With that notation, the gradient and Laplacian read

$$\nabla_k \ln J = \sum_{j' \neq k'=1}^{P} \frac{\beta_{k'j'}}{(1+\gamma r_{k'j'})^2} \frac{x_k - x_j}{r_{k'j'}}$$

and

$$\nabla_k^2 \ln J = \sum_{j' \neq k'=1}^{P} \frac{\beta_{k'j'}}{(1+\gamma r_{k'j'})^2}\left[1 - \left(1 + 2\frac{\gamma r_{k'j'}}{1+\gamma r_{k'j'}}\right)\frac{(x_k - x_j)^2}{r_{k'j'}^2}\right]\frac{1}{r_{k'j'}}$$

respectively, where j is a coordinate index for the same direction as k ($k = ndi$ with d as the number of dimensions and n as an integer).

By defining

$$f_{ij} = \frac{1}{1+\gamma r_{ij}} \quad g_{ij} = \frac{x_i - x_j}{r_{i'j'}} \quad h_{ij} = \frac{r_{ij}}{1+\gamma r_{ij}}$$

the equations can be written as

$$\begin{aligned}
\frac{J^2_{\text{new}}}{J^2_{\text{old}}} &= \exp\Big(2\sum_{j'=1}^{N}\beta_{i'j'}(h^{\text{new}}_{i'j'} - h^{\text{old}}_{i'j'})\Big) \\
\nabla_k \ln J &= \sum_{j'\neq k'=1}^{N} \beta_{k'j'}\cdot f^2_{k'j'}\cdot g_{kj} \\
\nabla^2_k \ln J &= \sum_{j'\neq k'=1}^{N} \frac{\beta_{k'j'}}{r_{k'j'}} f^2_{k'j'}\Big[1-(1+2\gamma h_{k'j'})g^2_{kj}\Big] \\
\partial_\gamma \nabla_k \ln J &= \sum_{j'\neq k'=1}^{N} \beta_{k'j'}\cdot f^3_{k'j'}(x_k - x_j) \\
\partial_\gamma \nabla^2_k \ln J &= \sum_{j'\neq k'=1}^{N} \beta_{k'j'}\cdot f^3_{k'j'}\Big[1=4\gamma h_{k'j'}\cdot g^2_{kj}\Big],
\end{aligned} \tag{8.13}$$

with marked indices (i') as the particle related ones and the unmarked (i) as the coordinate related ones. i' is the moved particle.

8.4.3 Slater Determinant

The Slater determinant is added to introduce anti-symmetry into the wave function, and as discussed in section (2.2.2), it can be split up in a spin-up part and a spin-down part,

$$\Psi(\boldsymbol{x}) = |\hat{D}_\uparrow(\boldsymbol{x}_\uparrow)|\cdot|\hat{D}_\downarrow(\boldsymbol{x}_\downarrow)|.$$

$x_\uparrow$ are the coordinates of particles with spin up (defined as the first half of the coordinates) and $x_\downarrow$ are the coordinates of particles with spin down (defined as the last half of the coordinates).

We can now utilize the logarithmic scale,

$$\ln\Psi = \ln|\hat{D}_\uparrow(\boldsymbol{x}_\uparrow)| + \ln|\hat{D}_\downarrow(\boldsymbol{x}_\downarrow)|$$

such that we only need to care about one of the determinants when differentiating, dependent on whether the coordinate we differentiate with respect to is among the spin-up or the spin-down coordinates:

$$\nabla_k \ln\Psi = \begin{cases} \nabla_k \ln|\hat{D}_\uparrow(\boldsymbol{x}_\uparrow)| & \text{if} \quad k < F/2 \\ \nabla_k \ln|\hat{D}_\downarrow(\boldsymbol{x}_\downarrow)| & \text{if} \quad k \geq F/2. \end{cases}$$

Before we go further, we will introduce a more general notation which cover both cases:

$$\hat{D} \equiv \hat{D}_\sigma(\boldsymbol{x}_\sigma)$$

where σ is the spin. When summing, the sum is always over all relevant coordinates.

Furthermore, we have that

$$\nabla_k \ln|\hat{D}| = \frac{\nabla_k|\hat{D}|}{|\hat{D}|}$$

and

$$\nabla^2_k \ln|\hat{D}| = \frac{\nabla^2_k|\hat{D}|}{|\hat{D}|} - \left(\frac{\nabla_k|\hat{D}|}{|\hat{D}|}\right)^2$$

The first derivative of a determinant is given by Jacobi's formula, which reads

$$\frac{\nabla_i|\hat{A}|}{|\hat{A}|} = \text{tr}\Big(\hat{A}^{-1}\nabla_i\hat{A}\Big), \tag{8.14}$$

and the second derivative is then

$$\frac{\nabla_i^2|\hat{A}|}{|\hat{A}|} = \Big(\mathrm{tr}\left(\hat{A}^{-1}\nabla_i\hat{A}\right)\Big)^2 + \mathrm{tr}\left(\hat{A}^{-1}\nabla_i^2\hat{A}\right) - \mathrm{tr}\left(\hat{A}^{-1}\nabla_i\hat{A}\hat{A}^{-1}\nabla_i\hat{A}\right)$$

where $\mathrm{tr}\left(\hat{B}\right)$ is the trace of matrix $\hat{B}$, i.e, the sum of all diagonal elements. $\nabla_i\hat{A}$ means that we differentiate the matrix component-wise with respect to coordinate i. The traces can then be written as sums,

$$\mathrm{tr}\Big(\hat{A}^{-1}\nabla_i\hat{A}\Big) = \sum_j A_{ji}^{-1}\nabla_i A_{ij}.$$

and

$$\mathrm{tr}\Big(\hat{A}^{-1}\nabla_i^2\hat{A}\Big) = \sum_j a_{ji}^{-1}\nabla_i^2 a_{ij}.$$

Using all the general matrix operations presented above, we end up with

$$\nabla_k \ln|\hat{D}| = \sum_j d_{jk}^{-1}\nabla_k d_{kj}$$

and

$$\nabla_k^2 \ln|\hat{D}| = \sum_j d_{jk}^{-1}\nabla_k^2 d_{kj} - \Big(\sum_j d_{jk}^{-1}\nabla_k d_{kj}\Big)^2$$

8.4.3.1 Efficient calculation of Slater determinants

As you might already have noticed, we need to calculate the inverse of the matrices every time a particle is moved. This is a pretty heavy task for the computer, where the standard way, LU decomposition, requires $\sim \mathrm{N}^3$ floating point operations for an N×N matrix. [11].

The good thing is that, by exploiting that only one row in the Slater matrix is updated for each step, we can update the inverse iteratively.

Before we start finding an algorithm for this, we will introduce the reader to some common linear algebra concepts. First of all, the inverse of a matrix is given by the *comatrix* transposed over its determinant

$$\hat{A}^{-1} = \frac{\hat{C}^T}{|\hat{A}|} \tag{8.15}$$

where the comatrix is defined by the inner determinants of the matrix. [**weisstein'matrix'nodate**] As a consequence, the determinant can be written as

$$|\hat{A}| = \sum_{i,j} a_{ij}c_{ij}. \tag{8.16}$$

As always, we are interested in the ratio between the wave functions, and since only a row is updated every time we move a particle, the ratio between the determinants can be expressed as

$$R \equiv \frac{|\hat{D}^{\mathrm{new}}|}{|\hat{D}^{\mathrm{old}}|} = \frac{\sum_j d_{ij}^{\mathrm{new}}c_{ij}^{\mathrm{new}}}{\sum_j d_{ij}^{\mathrm{old}}c_{ij}^{\mathrm{old}}} \tag{8.17}$$

where the particle associated with the i'th row is moved. The i'th row of the comatrix is independent of the i'th row of the matrix itself, such that $c_{ij}^{\mathrm{new}} = c_{ij}^{\mathrm{old}}$.

In the end, we will take advantage of the fact that we only move one particle at a time. This means that one of the two determinants cancel when calculating the probability ratio used in Metropolis sampling. Since we do not have any variational parameters in the Slater determinant, we end up with three expressions for each determinant:

$$
\begin{aligned}
&\text{if} \quad k < F/2: \\
&\frac{|\Psi_{\text{new}}|^2}{|\Psi_{\text{old}}|^2} = |\hat{D}_\uparrow(\boldsymbol{x}_\uparrow^{\text{new}})|^2 / |\hat{D}_\uparrow(\boldsymbol{x}_\uparrow^{\text{old}})|^2 \\
&\nabla_k \ln|\hat{D}_\uparrow| = \sum_{j=1}^{M/2} \nabla_k d_{jk} d_{kj}^{-1} \\
&\nabla_k^2 \ln|\hat{D}_\uparrow| = \sum_{j=1}^{M/2} \nabla_k^2 d_{jk} d_{kj}^{-1} - \Big(\sum_{j=1}^{M/2} \nabla_k d_{ik} d_{ki}^{-1} \Big)^2
\end{aligned}
\tag{8.18}
$$

$$
\begin{aligned}
&\text{if} \quad k \geq F/2: \\
&\frac{|\Psi_{\text{new}}|^2}{|\Psi_{\text{old}}|^2} = |\hat{D}_\downarrow(\boldsymbol{x}_\downarrow^{\text{new}})|^2 / |\hat{D}_\downarrow(\boldsymbol{x}_\downarrow^{\text{old}})|^2 \\
&\nabla_k \ln|\hat{D}_\downarrow| = \sum_{j=M/2}^{M} \nabla_k d_{jk} d_{kj}^{-1} \\
&\nabla_k^2 \ln|\hat{D}_\downarrow| = \sum_{j=M/2}^{M} \nabla_k^2 d_{jk} d_{kj}^{-1} - \Big(\sum_{j=M/2}^{M} \nabla_k d_{ik} d_{ki}^{-1} \Big)^2
\end{aligned}
\tag{8.19}
$$

8.4.4 NQS-Gaussian

Now over to the real deal; the machine learning inspired wave function elements. The total NQS wave function, presented in equation (**??**), was decided split up in case we wanted to run them separately. The first part will henceforth be denoted as the NQS-Gaussian,

$$
\Psi(\boldsymbol{x}; \boldsymbol{a}) = \exp\left(- \sum_{i=1}^{M} \frac{(x_i - a_i)^2}{2\sigma^2} \right) \tag{8.20}
$$

while the last part will be denoted as the NQS-Jastrow and is presented in the next subsection.

The derivatives of the NQS-Gaussian are similar to those of the simple Gaussian, they are therefore just listed up in equation (8.21).

$$
\begin{aligned}
\frac{\Psi_{\text{new}}^2}{\Psi_{\text{old}}^2} &= \exp\Big((x_i^{\text{old}} + x_i^{\text{new}} - 2a_i)(x_i^{\text{old}} - x_i^{\text{new}}) \Big) \\
\nabla_k \ln\Psi &= -\frac{x_k - a_k}{\sigma^2} \\
\nabla_k^2 \ln\Psi &= -\frac{1}{\sigma^2} \\
\partial_{a_l} \nabla_k \ln\Psi &= \frac{1}{\sigma^2} \\
\partial_{a_l} \nabla_k^2 \ln\Psi &= 0
\end{aligned}
\tag{8.21}
$$

8.4.5 NQS-Jastrow Factor

$$
J(\boldsymbol{x}; \boldsymbol{b}, \boldsymbol{W}) = \prod_{j=1}^{N} \left[1 + \exp\Big(b_j + \sum_{i=1}^{M} \frac{W_{ij} x_i}{\sigma^2} \Big) \right]
$$

$$
\nabla_k \ln J = \sum_{j=1}^{N} \frac{W_{kj}}{\sigma^2} \frac{\exp\big(b_j + \sum_{i=1}^{M} \frac{W_{ij} x_i}{\sigma^2} \big)}{1 + \exp\big(b_j + \sum_{i=1}^{M} \frac{W_{ij} x_i}{\sigma^2} \big)}
$$

$$\nabla_k^2 \ln J = \sum_{j=1}^{N} \frac{W_{kj}^2}{\sigma^4} \frac{\exp\left(b_j + \sum_{i=1}^{M} \frac{W_{ij}x_i}{\sigma^2}\right)}{\left(1 + \exp\left(b_j + \sum_{i=1}^{M} \frac{W_{ij}x_i}{\sigma^2}\right)\right)^2}$$

$$\partial_{b_l} \nabla_k \ln J = \frac{W_{kl}}{\sigma^2} \frac{\exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)}{\left(1 + \exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)\right)^2}$$

$$\partial_{b_l} \nabla_k^2 \ln J = \frac{W_{kl}^2}{\sigma^4} \frac{\exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)\left(1 - \exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)\right)}{\left(1 + \exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)\right)^3}$$

$$\partial_{W_{ml}} \nabla_k \ln J = \frac{1}{\sigma^2} \frac{\exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)}{1 + \exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)} \delta_{mk} + \frac{W_{kl}x_m}{\sigma^4} \frac{\exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)}{\left(1 + \exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)\right)^2}$$

$$\partial_{W_{ml}} \nabla_k^2 \ln J = 2\frac{W_{kl}}{\sigma^4} \frac{\exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)}{\left(1 + \exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)\right)^2} \delta_{mk} + \frac{W_{kl}^2 x_m}{\sigma^4} \frac{\exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)}{\left(1 + \exp\left(b_l + \sum_{i=1}^{M} \frac{W_{il}x_i}{\sigma^2}\right)\right)^3}$$

where δ_{ij} is the Kronecker delta. Defining

$$p_j \equiv \frac{1}{1 + \exp\left(+ b_j + \sum_{i=1}^{M} \frac{W_{ij}x_i}{\sigma^2}\right)} \quad \text{and} \quad n_j \equiv \frac{1}{1 + \exp\left(- b_j - \sum_{i=1}^{M} \frac{W_{ij}x_i}{\sigma^2}\right)}$$

the expressions above can be simplified in the following fashion

$$\begin{aligned}
\frac{J_{\text{new}}^2}{J_{\text{old}}^2} &= \prod_{j=1}^{N} \frac{p_j^{\text{old}}}{p_j^{\text{new}}} \\
\nabla_k \ln J &= \sum_{j=1}^{N} \frac{W_{kj}}{\sigma^2} n_j \\
\nabla_k^2 \ln J &= \sum_{j=1}^{N} \frac{W_{kj}^2}{\sigma^4} p_j n_j \\
\partial_{b_l} \nabla_k \ln J &= \frac{W_{kl}}{\sigma^2} p_l n_l \\
\partial_{b_l} \nabla_k^2 \ln J &= \frac{W_{kl}^2}{\sigma^4} p_l n_l (p_l - n_l) \\
\partial_{W_{ml}} \nabla_k \ln J &= \frac{1}{\sigma^2} n_l \delta_{mk} + \frac{W_{kl}x_m}{\sigma^4} p_l n_l \\
\partial_{W_{ml}} \nabla_k^2 \ln J &= 2\frac{W_{kl}}{\sigma^4} p_l n_l \delta_{mk} + \frac{W_{kl}^2 x_m}{\sigma^6} p_l n_l (p_l - n_l)
\end{aligned} \tag{8.22}$$

8.4.6 Hydrogen-Like Orbitals

$$\Psi(\alpha, \boldsymbol{r}) = \exp\Big[-\frac{1}{2}\alpha \sum_{i=1}^{N} r_i \Big] \tag{8.23}$$

where the derivative with respect to coordinate r_k is

$$\nabla_k \ln \Psi(\alpha) = -\alpha \tag{8.24}$$

and the second derivative is

$$\nabla_k^2 \ln \Psi(\alpha) = 0 \tag{8.25}$$

The gradients for those derivatives are

$$\partial_\alpha \nabla_k \ln \Psi(\alpha) = -1 \tag{8.26}$$

and

$$\partial_\alpha \nabla_k^2 \ln \Psi(\alpha) = 0 \tag{8.27}$$

respectively.

Chapter 9

Optimization and resampling

Great quote.

Author

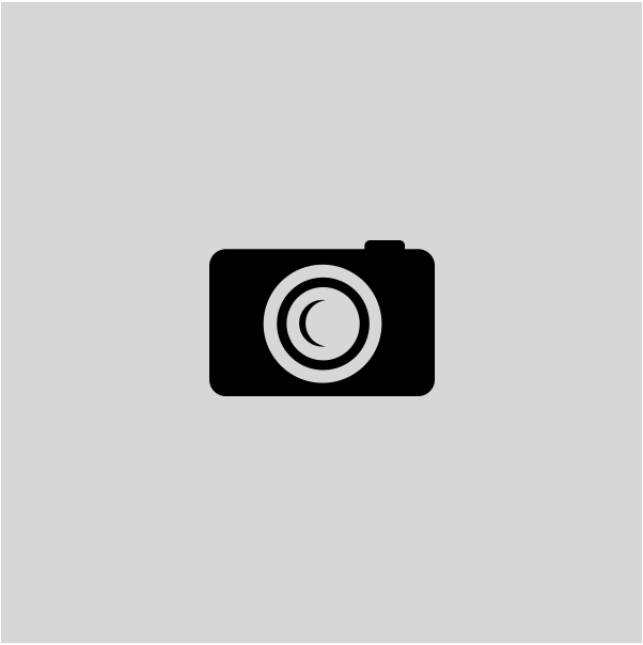


Figure 9.1: Caption

Optimization is a wide term..

9.1 Minimization Algorithms

Suppose we have a very simple model trying to fit a straight line to data points. In that case, we could manually vary the coefficients and find a line that fits the points quite good. However, when the model gets more complicated, this can be a time consuming activity. Would it not be good if the program could do this for us?

In fact, there exist plenty of methods capable of doing this. Some of them rely on the gradients only, and are therefore called gradient methods. Other need both the gradient and the Hessian matrix, and find the minimum based on both the slope and the curvature of the cost function. For our purpose, gradients methods have provided good results over decades, and there is no need for more complicated algorithms. The standard gradient descent method will be discussed firstly, before we move to its stochastic brother. Momentum will be added for both methods. Finally, we examine the ADAM optimizer, which is stochastic by nature and is equipped with momentum by default.

9.1.1 Gradient Descent

Perhaps the simplest and most intuitive method for finding the minimum is the gradient descent method (GD), which reads

$$\alpha_i^{\text{new}} = \alpha_i - \eta \cdot \frac{\partial Q(\alpha_i)}{\partial \alpha_i} \tag{9.1}$$

where α_i^{new} is the updated weight α and η is the learning rate. The idea is to find the steepest slope of the cost function $Q(\vec{\alpha})$ with respect to a certain α_i, and move in the direction which minimizes the cost function. For every step, the cost function is thus minimized, and when the gradient approaches zero the minimum is found. A possible stop criterion is

$$\frac{\partial Q(\alpha_i)}{\partial \alpha_i} < \varepsilon. \tag{9.2}$$

where ε is a tolerance.

In cases where the cost function is not strictly increasing or decreasing, we will have both local and global minima. Often, it is hard to say whether we are stuck in a local or global minimum, and this is where the stochasticity enters the game.

9.1.2 Stochastic Gradient Descent

For standard gradient descent, we calculate the gradient based on all sampling points, we say that we have one batch.

Instead of calculating the gradient based on all sampling points, we can divide the data into multiple batches and calculate the gradient based on one batch and hope that it is a good approximation of the true gradient. Updating weight α_i based on batch j thus yields

$$\alpha_i^{\text{new}} = \alpha_i - \eta \cdot \frac{\partial Q_j(\alpha_i)}{\partial \alpha_i} \tag{9.3}$$

and a run through all batches is called an *epoch*.

The reader might ask herself why this helps us, will we not just get a bad gradient approximation? The answer is that the stochasticity adds some coincidence to the system, which makes it less likely to be stuck in local minima. Additionally, it might speed-up the training session.

9.1.3 ADAM

ADAM is a first-order stochastic optimization method which is widely used in machine learning. It was discovered by D.P. Kingma and J. Ba, and published in a 2014 paper. The article has already more than 20000 citations! [19] So what makes this method so great?

The main reason why it is so popular might be that it is straight-forward to implement, in the same time as it is efficient and capable of handle a large number of parameters. Additionally, the method has built-in momentum which makes it less likely to be stuck in a local minimum, more about that later.

The algorithm goes as following

Algorithm 2: The ADAM algorithm

```
Data: this text
Result: how to write algorithm with LaTeX2e
initialization;
while not at end of this document do
    read current;
    if understand then
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end
end
```

9.1.4 Adding momentum

We have already mentioned momentum, but what is it and why do we use it?

If we go back to an introductory mechanics course, you might remember that momentum is a quantity that maintains the motion of a body. Imagine a ball that rolls down a steep hill, but then there is a local minimum that it needs to escape to keep rolling. Because of its momentum, it will probably be able to escape.

Exactly the same idea lies behind the momentum used in optimization algorithms; the momentum will try to maintain the motion towards the global minimum, which makes the system less likely to be stuck in a local minimum. Se figure .. for illustration.

Momentum can be added to most optimization algorithms, also gradient descent and stochastic gradient descent. The way we do it is to ...

$$\boldsymbol{v} = \gamma \boldsymbol{v} + \eta \nabla E \tag{9.4}$$

9.2 Resampling

9.2.1 Blocking

dkdkdk

9.3 Random number generators

In the Monte-Carlo sampling we are drawing millions of numbers, and in order to get accurate estimations, they should all be random, independent and fast to get.

In C++ there are plenty of random number generator available, but not all of them meet our requirements. For instance the standard

Mersenne Twister needs to be mentioned here

Part IV

Implementation and Results

Chapter 10

Scientific Programming

Great quote.

Author

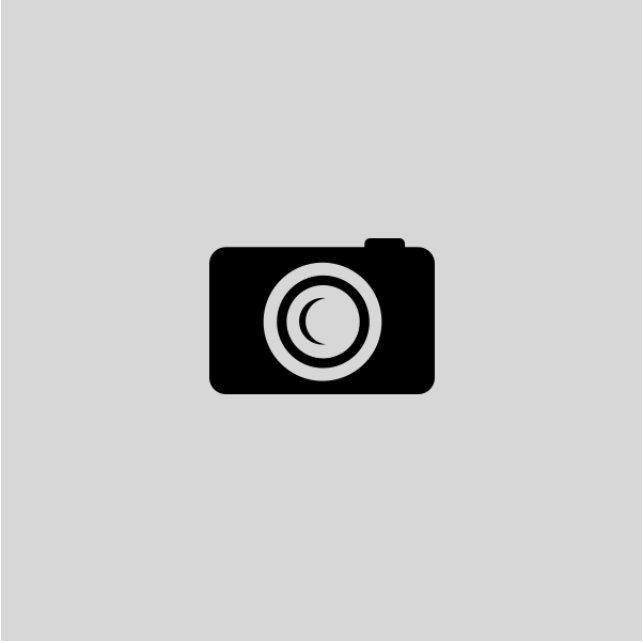


Figure 10.1: Caption

Since this thesis is much about writing code, it is natural including a few words about it.

The computer's language itself is binary, and is the lowest level. To translate commands to this language, we need a "translator", which is a language that fills the gap between the binary language and human commands. This language is categorized in levels based on how similar they are to the binary language. Low-level languages are similar to the binary language, which means fast but complicated. High-level languages are easy to work with, but are not as fast as low-level languages. Might mention grammar etc

One can either do *procedural programming* or *object oriented programming*. The former means that the code is written in the same order as the program flow goes, while one in the latter defines objects.

10.1 Object Orientated Programming

In the everyday life, we are surrounded by objects all the time which we can place in different categories. For instance, a *cat* is an object with a name, race, age and so on, and can be placed in the class *animals*. In object oriented programming, the class could be implemented as

```
class Animal:
  def __init__(self, animal, name, race, age):
    self.animal = animal
    self.name = name
    self.race = race
    self.age = age

  def __call__(self):
    return "%s is a %s that's %d years old and of the race %s"%(self.name, self.animal, self.age,
        ↪ self.race)
```

where...

The next step is to define an object, which in our case is the cat with name "Schrodinger":

```
Alma = Animal("cat", "Schroedeger", "Ragdoll", 4)
print(Alma())
```

This implies that "Schrodinger" is a cat of race "Ragdoll" and of age 4. When calling this class from "Schrodinger", the class returns

```
$python3 simple_class.py
>>> Schrodinger is a cat that's 4 years old and of the race Ragdoll
```

You might wonder how this is related to scientific programming. The answer is that it is often convenient to define various parts of the code as objects to increase the liability and maximize reuse of code. For example, we use various Hamiltonians, where each can be defined as a subclass of the Hamiltonian superclass.

The code above is written in Python, but the exact task could be performed in C++.

INCLUDE C++ IMPLEMENTATION

As one can see,

In C++ one needs to define constructors and destructors. All variables used inside the functions are defines in the constructor, while they are removed in the destructor to free up memory. In Python, the constructors are called `__init__` by default, and memory is handled automatically.

10.1.1 Inheritance

This is also called parent and child, respectively.

Parent and child Polymorphy: Child inherit from the parents. Virtual functions to achieve runtime polymorphism Should define virtual destructor as well

1. Single inheritance

2. Multiple inheritances

Python and C++ support multiple inheritances. Multilevel inheritance: Got child and grand child. Hierarchical inheritance: Parent got several children. https://www.geeksforgeeks.org/inheritance-in-python/

10.1.2 Pointers

Sometimes we do not want to send the object itself, but either its address, such that..

10.1.3 Virtual Functions

Often one wants to define a template of objects... where the super class defines which functions its objects should have. In C++, this can be achieved by virtual functions, functions with arguments specified but task undefined. Those functions are overwritten by the corresponding functions in the object (hence virtual),

10.1.4 Data types

To not lose precision, it is important to choose the correct data type. As a thumb rule, a large integer should be declared as a `long int` (or just a `long`), and an extremely large float should be declared as a `long double`. However, normally it is sufficient to declare integers as `int`s and floats as `double`s.

Some variables should always be non-negative, such as counters and variables that gives the number of something. In those situations, the prefix `unsigned` is useful because it does two things: 1. ensures that a negative number is never assigned to the variable, 2. increases the range in positive direction. In for-loops, the control variable will usually be declared as `unsigned int`.

When using a library some variable types are already set, for instance the length of a standard vector is always given as an `unsigned long`. In those cases, it is most appropriate to continue using that type.

When we tell the pros, we should also tell the cons. With `unsigned` types there is always a risk for underflow, which means that the number explode if it ...

As an additional note, we often want to ensure that a variable is not changed inside a function. To prevent this, the arguments can be passed with the feature `const`, which raises an error if the respective variable is tried changed.

10.1.4.1 Data types in Eigen

The open source template library for linear algebra, Eigen, will be used throughout the coding, and it comes with arrays of various properties. The most relevant ones are `VectorXi`, which has dynamic length and `int` data type and `VectorXd`, which has dynamic length and `double` data type. The Matrix class has equivalent objects.

In cases where we have *a priori* knowledge of the array size, we can replace the `X` with the actual size. A fixed 3×3 matrix of type `double` can for example be declared as `Matrix3d`. According to the Eigen documentation, using fixed size is *"...hugely beneficial to performance"*. `https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html`.

Chapter 11

Implementation

There are only two hard things in Computer Science: cache invalidation and naming things.

Phil Karlton, [25]

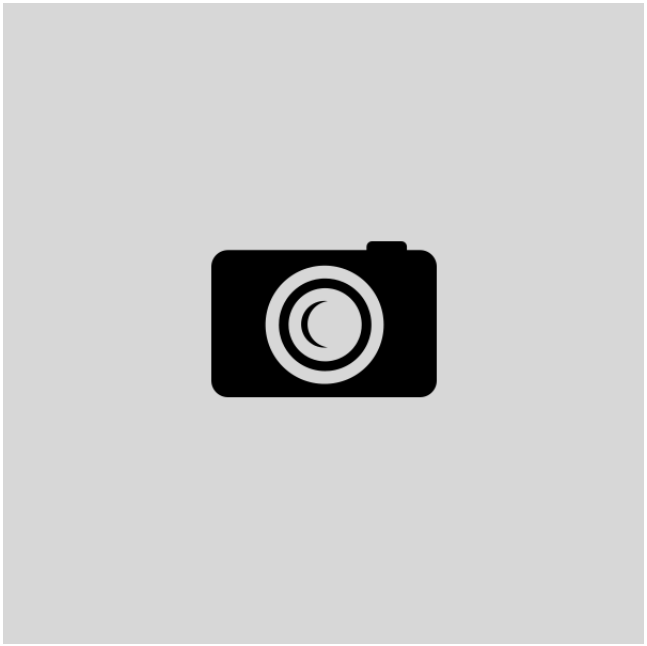


Figure 11.1: Caption

For many projects, planning is half the job, and so is true for a good VMC implementation. In fact, the program was restructured three times before we landed on this final version. It is based on Morten Ledum's VMC framework found at [20], which was meant as an example implementation in the course *FYS4411 - Computational Physics II: Quantum Mechanical Systems*.

For all matrix and array operations, the **Eigen** library was used. All source code can be found at [23].

The code was developed with three main goals. It should be

- readable,
- fast,
- flexible.

How we work to achieve the goals will be illustrated by code mainly picked from the `WaveFunction` class, which is the heart of the code.

11.1 Readability

To maximize the readability, we developed a highly object oriented code based on the theory in chapter (10). For instance, each wave function element was treated as an object, with the properties `updateArrays`, `setArrays`, `resetArrays`, `initializeArrays`, `updateParameters`, `evaluateRatio`, `computeGradient`, `computeLaplacian` and
`computeParameterGradient`. To ensure that all wave function elements have all the necessary properties, the super class `WaveFunctions` is equipped with the corresponding virtual functions

Listing 11.1: `wavefunction.h`

```
#pragma once
#include <Eigen/Dense>
#include <iostream>

class WaveFunction {
public:
    WaveFunction(class System *system);
    virtual void   updateArrays     (Eigen::VectorXd positions, int pRand)  = 0;
    virtual void   setArrays        () = 0;
    virtual void   resetArrays      () = 0;
    virtual void   initializeArrays(Eigen::VectorXd positions) = 0;
    virtual void   updateParameters(Eigen::MatrixXd parameters, int elementNumber) = 0;

    virtual double evaluateRatio    ()      = 0;
    virtual double computeGradient (int k) = 0;
    virtual double computeLaplacian()      = 0;

    virtual Eigen::VectorXd computeParameterGradient() = 0;

    virtual ~WaveFunction() = 0;

protected:
    int      m_numberOfParticles                = 0;
    int      m_numberOfDimensions               = 0;
    int      m_numberOfFreeDimensions           = 0;
    int      m_maxNumberOfParametersPerElement  = 0;
    class System* m_system = nullptr;
};
```

which serves a template for all the sub classes (wave function elements). As you might notice, we use the **lowerCamelCase** naming convention for function and variable names, which means that each word begins with a capital letter except the initial word. For classes, we use the **UpperCamelCase** to distinguish from function names. This is known to be easy to read, and apart from for example the popular **snake_case**, we do not need delimiters between the words, which saves some space. After the naming convention is decided, we are still responsible for giving reasonable names, which is not always an easy task, as Phil Karlton points out. When one sees the name, one should know exactly what the variable/function/class is or does. More about naming conventions can be read at [22].

11.2 Efficiency

The efficiency is mostly based on recursive computations, such that we do not need to calculate everything over again when it is enough to calculate the fraction between the old the new number. This is highly relevant when it comes to the Slater determinant, but we also do this for other wave function elements.

First of all, we need to initialize all position dependent arrays. Some of these are initialized once, and then updated for the rest of the run based on their earlier value. `initializeArrays` does all the magic here.

Every wave function element is equipped with a function, `updateArrays`, where all relevant arrays are updated immediately after a particle is moved. In this way we ensure that nothing is calculated twice inside any element. Those functions are by far the most expensive to calculate, but it is also easier to streamline a really expensive function than several quite expensive ones.

Inside `updateArrays`, we first need to update the old variables, typically named like `m_positionsOld` or so. This is generally done by calling the function `setArrays`, which ensures that all the old variables are

correct. We need to store the old variables in case a move is rejected and we need to go back to the old positions. Thereafter, we can update all the variables. The most basic example is the Gaussian function, where we basically only need to update the position and the probability ratio.

Listing 11.2: from `gaussian.cpp`

```
void Gaussian::updateArrays(Eigen::VectorXd positions, int changedCoord) {
    setArrays();
    m_positions             = positions;
    updateProbabilityRatio(changedCoord);
}

void Gaussian::setArrays() {
    m_positionsOld          = m_positions;
    m_probabilityRatioOld   = m_probabilityRatio;
}

void Gaussian::updateProbabilityRatio(int changedCoord) {
    m_probabilityRatio = exp(m_omega * m_alpha * (m_positionsOld(changedCoord) * \
    m_positionsOld(changedCoord)- m_positions(changedCoord)*m_positions(changedCoord)));
}
```

Similarly to the function `setArrays`, there is also a function `resetArrays`, which is called then a move is rejected. It works the exact opposite way, looking like

Listing 11.3: from `gaussian.cpp`

```
void Gaussian::resetArrays() {
    m_positions             = m_positionsOld;
    m_probabilityRatio      = m_probabilityRatioOld;
}
```

There are also a few arrays that are used inside multiple wave function elements, such as the distance matrix and the radial distance vector. They might also be used in the Hamiltonian. To ensure that they are not calculated more than necessary, we define them globally together with a respective Boolean which tells us whether or not the array is updated at the present cycle.

Listing 11.4: from `system.cpp`

```
NEED TO ADD EXACT IMPLEMENTATION
```

For profiling, we used **callgrind** with **kcachegrind** visualization, which are great tools when we want to find out which functions that steal CPU time.

11.3 Flexibility

Unlike many other VMC codes, our code was developed flexible with respect to the wave functions. This means that one can combine various wave function elements, where each element is implemented separately. For instance, the Gaussian function, the Slater determinant and the Padé-Jastrow factor were implemented separately, but they all can easily be combined. The way one does this in practice, is to append multiple wave function elements to the vector `WaveFunctionElements` in main. One can combine the Gaussian with the Padé-Jastrow factor and the Slater determinant in the following way

Listing 11.5: from `main.cpp`

```
System* quantumDot = new System();
std::vector<class WaveFunction*> WaveFunctionElements;
WaveFunctionElements.push_back(new Gaussian(quantumDot));
WaveFunctionElements.push_back(new PadeJastrow(quantumDot));
WaveFunctionElements.push_back(new SlaterDeterminant(quantumDot));
quantumDot->setWaveFunctionElements(WaveFunctionElements);
```

The big advantage of this implementation technique is that we do not need to hard code every possible combination of wave function elements, which reduces the number of code lines significantly. This also eases the operation of adding new elements, since we only need to calculate the derivatives of the particular element (do not need to worry about cross terms). Exactly how this is done can be read in chaper 8. The con is that the program will be slightly slower, since even canceling cross terms are calculated.

11.3.1 Energy calculation

The way we calculate the total kinetic energy then is based on the theory presented in chapter 8, where we explain that

$$T = -\frac{1}{2}\frac{1}{\Psi_T}\nabla_k^2\Psi_T = -\frac{1}{2}\left[\sum_{i=1}^{p}\nabla_k^2 \ln\phi_i + \left(\sum_{i=1}^{p}\nabla_k \ln\phi_i\right)^2\right]. \tag{11.1}$$

The corresponding implementation thus reads

Listing 11.6: from `system.cpp`

```
double System::getKineticEnergy() {
    double kineticEnergy = 0;
    for(auto& i : m_waveFunctionElements) {
        kineticEnergy += i->computeLaplacian();
    }
    for(int k = 0; k < m_numberOfFreeDimensions; k++) {
        double nablaLnPsi = 0;
        for(auto& i : m_waveFunctionElements) {
            nablaLnPsi += i->computeGradient(k);
        }
        kineticEnergy += nablaLnPsi * nablaLnPsi;
    }
    return - 0.5 * kineticEnergy;
}
```

11.3.2 Probability ratio calculation

In the same chapter we state the obvious fact that

$$\frac{\Psi_T^{\text{new}}}{\Psi_T^{\text{old}}} = \prod_{i=1}^{p}\frac{\phi_i^{\text{new}}}{\phi_i^{\text{old}}},$$

which can easily be implemented as

Listing 11.7: from `system.cpp`

```
double System::evaluateWaveFunctionRatio() {
    double ratio = 1;
    for(auto& i : m_waveFunctionElements) {
        ratio *= i->evaluateRatio();
    }
    return ratio;
}
```

11.3.3 Parameters

Another consequence of this flexible implementation is that we need to treat all parameters in the same way to make everything general. To do this, we create a global matrix of dimensions $n \times m$ where n is the number of wave function elements and m is the maximum number of parameters in a wave function element. Thus each element has its own row in the matrix, and one can easily track down a specific parameter.

For the parameter update, each element needs to provide an array of length m containing its respective parameter gradients. This array is calculated in the function `computeParameterGradient` for each element, and they are all collected in the function `getAllInstantGradients`:

Listing 11.8: from `system.cpp`

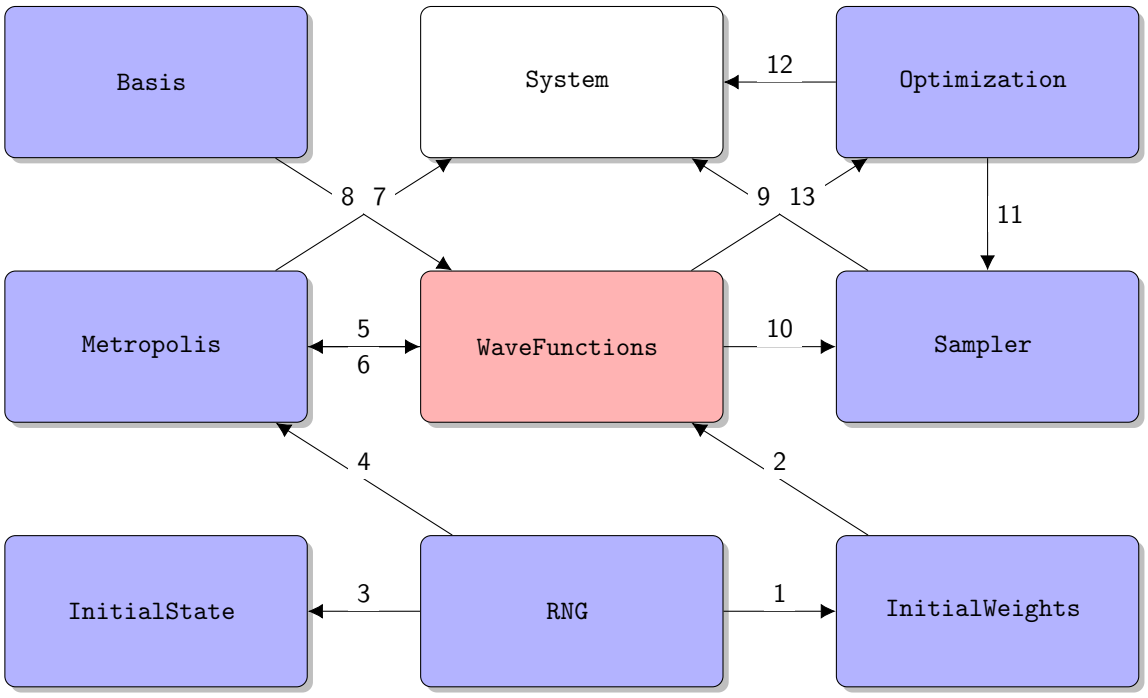
```
Eigen::MatrixXd System::getAllInstantGradients() {
    Eigen::MatrixXd gradients = Eigen::MatrixXd::Zero(m_numberOfWaveFunctionElements, \
    m_maxNumberOfParametersPerElement);
    for(int i = 0; i < m_numberOfWaveFunctionElements; i++) {
        gradients.row(i) = m_waveFunctionElements[i]->computeParameterGradient();
    }
    return gradients;
}
```

We need to stress that those are the instant gradients calculated every time a particle is moved. Exactly how the parameters are updated depends on an average of these, described in chapter 8.

Immediately after the parameters are updated, the new parameters need to be sent into the wave function elements. This is done through the functions `updateParameters`, which update all the parameters and weights in the elements. In addition, the function has the responsibility to order the elements such that none gets the same number.

11.4 Structure

How the classes are communicating is no easy task to explain, most classes are calling other classes, there is no tidy way to visualize the actual code flow. However, a simplified structure chart can still be informative, and in figure (**??**) the most important calls between the different classes are pointed out. We decided to leave out `main.cpp` since all it does is to set the different classes.



1 - Set random initial weights

2 - Set weights in wave functions

3 - Set random initial positions

4 - Propose random move

5 - Evaluate wave functions

6 - Update positions

7 - Accept or reject step

8 - Get basis used in Slater determinant

9 - Sample

10 - Compute local energy

11 - Calculate instant gradients

12 - Calculate energy derivatives

13 - Calculate average gradients

Figure 11.2: Structure chart of the implemented code, presenting super classes as tiles. The most important intra-class calls are represented with lines pointing from the sender class towards the receiver class.

This is the main aim of the flow, but the actual flow does also depend on system. For instance, when

using importance sampling, we will have an additional call between `WaveFunctions` and `Metropolis` due to calculations of the quantum force.

11.5 Foundation

The foundation of the code are all the super classes, nine in the number. They all have multiple sub classes, and the reader needs to specify which sub class to be used. The exception is the `WaveFunctions` class, as described above, where multiple sub classes can be used. Below, the role of all the super classes will be discussed briefly and the difference between various sub classes will be explained.

11.5.1 Super classes

11.5.1.1 The `Basis` class

In this class, one needs to choose which basis set that should be used in the Slater determinant. There are three required functions:

- `numberOfOrbitals()` gives the number of orbitals given the number of particles and dimensions. This is used in the Slater determinant.
- `evaluate(double x, int n)` gives the the value of element `n` for a given `x`.
- `evaluateDerivative(double x, int n)` gives the derivative of element `n` with respect to `x` for a given `x`.

Possible sub classes choices are `Hermite` and `HydrogenLike`, where the former is well-suited for quantum dots and the latter is used in atomic structure calculations.

11.5.1.2 The `Hamiltonians` class

In this class, one needs to specify the Hamiltonian of the system. The only required function is `computeLocalEnergy()`, which returns the local energy. One can choose between the Hamiltonians `AtomicNucleus` and `HarmonicOscillator`, where the first one sets up an external potential like the one we find in an atom, and takes the atomic number Z as an argument. The second one sets up a harmonic oscillator potential, and actually the only thing that distinguish the who classes is the external energy calculation.

11.5.1.3 The `InitialStates` class

In one way or another we need to initialize the particle positions, but how we want to do this depends on the situation. The implemented methods are randomly initialized positions drawn from a uniform or normal distribution, `RandomUniform` and `RandomNormal` respectively. They consist of the function `setupInitialState()`.

11.5.1.4 The `InitialWeights` class

In the same manner as the `InitialStates` class, we can initialize the weights in various ways. One way is to set all the weights to the same initial value, represented by the sub class `Constant`. It takes an argument `factor` which gives the initial value of all weights.

A second choice is random initial weights, where the class `Randomize` initializes the weights based on a uniform distribution. Also this class takes the `factor` argument, which defines the interval. By default, the interval is [-1,1], which corresponds to `factor=1`.

11.5.1.5 The `Metropolis` class

This class is the true sampling class, where the magic sampling is done. Three sampling methods are implemented:

- `BruteForce` is the standard Metropolis sampling, where a particle is moved in a totally random direction and the move is accepted if the new probability is high enough.

- `ImportanceSampling` is a more advanced version of the Metropolis algorithm, where the particle is moved in the same direction as the quantum force.
- `GibbsSampling` is not directly related to the Metropolis algorithm, it is a simple method which is widely used in Boltzmann machines.

The sub classes need to have the function `acceptMove()`, where the particle is moved and the the move is either accepted or rejected. To get the new positions, one need to call `updatePositions()`. which is member of the super class.

11.5.1.6 The `Optimization` class

The next class is the `Optimization` class, where the weight update is performed in the function `updateWeights()`. Also the instant gradients (the gradient for each step) is calculated here, in the function `getAllInstantGradients()`.

Two gradient based stochastic methods are implemented: `StochasticGradientDescent` and `ADAM`, with descriptive names. They both takes an argument `gamma` which is the prefactor in front of the momentum. The reader can consult chapter (9) for details on how the optimization methods work.

11.5.1.7 The `Plotter` class

Not sure if I will keep this as a class

11.5.1.8 The `RNG` class

The random number generator (RNG) was implemented as a class to ease the switch between different RNGs. Each subclass need to contain the following functions:

- `nextInt(int upperLimit)` returns the next number in the RNG sequence as an integer between 0 and `upperLimit`.
- `nextDouble()` returns the next number in the RNG sequence as a double between 0 and 1.
- `nextGaussian(double mean, double standardDeviation)` returns the next number in the RNG sequence, regenerated by a normal distribution with mean value `mean` and standard deviation `standardDeviation`.

The two available RNGs are the Mersenne Twister number generator, `MersenneTwister` and... . For the theory behind thoe methods, see section (9.3).

11.5.1.9 The `WaveFunctions` class

Last, but not least, the `WaveFunctions` class contains all the wave function related computations. We have already mentioned it, but all the details are still to be stressed.

The required functions in the wave function elements are

- `updateArrays(Eigen::VectorXd positions, int pRand)` which update position dependent arrays recursively with respect to the new positions, `positions` and the changed position index `pRand`.
- `resetArrays()` set the arrays back to the old values when a move is rejected.
- `initializeArrays(Eigen::VectorXd positions)` initialize all arrays at the beginning. This is the only moment when the arrays cannot be updated recursively.
- `updateParameters(Eigen::MatrixXd parameters, int elementNumber)` updates the weights. All weights of the system are stored in the parent matrix `parameters`, while each wave function element has child weight matrices and arrays which are mapped from the parent. They are all updated in this function. `elementNumber` is the number of the element, and is unique for all the wave function elements.
- `evaluateRatio()` returns the ratio between the new and the old probability, $|\Psi_T(\boldsymbol{r}_{\text{new}})|^2/|\Psi_T(\boldsymbol{r}_{\text{old}})|^2$

- `computeFirstDerivative(int k)` returns the first derivative of the wave function element with respect to the position index `k`.
- `computeSecondDerivative()` returns the second derivative of the wave function element with respect to all position indices.
- `computeFirstEnergyDerivative(int k)` returns the derivative of the position `k` first derivative of the wave function element with respect to all the weights, $\partial/\partial\alpha_i \nabla_k ln(\psi)$. The outcome is an array.
- `computeSecondEnergyDerivative()` returns the derivative of the position second derivative of the wave function element with respect to all the weights, $\sum_k \partial/\partial\alpha_i \nabla_k^2 ln(\psi)$. The outcome is an array.

The wave function elements implemented are

- `Gaussian` is the simple Gaussian function.
- `PadeJastrow` is the Padé-Jastrow factor.
- `SlaterDeterminant` is the Slater determinant.
- `MLGaussian` is the Gaussian part derived from the Boltzmann machines.
- `NQSJastrow` is the product part derived from the Boltzmann machines.

New wave function elements can easily be implemented, all one needs to do is to calculate all the derivatives and specify how to update the position dependent arrays recursively.

11.5.2 How to set sub classes?

We have now described all the available super classes and sub classes, but how do we set them? As hinted in the beginning of the chapter, the entire system should be specified in `main.cpp`. For example, a harmonic oscillator Hamiltonian can be set by

```
system->setHamiltonian(new HarmonicOscillator(system));
```

which is calling the function `setHamiltonian` in the class `System`. This function sets the official Hamiltonian object to `HarmonicOscillator`, such that every time we call the super class `Hamiltonian`, we are forwarded to `HarmonicOscillator`. The `System` class is basically filled with functions that set objects and scalars. To make those objects and scalars available in other classes, the `System` header is equipped with get-functions. For instance, there exist a function

```
class Hamiltonian* getHamiltonian() { return m_hamiltonian; }
```

which returns the correct Hamiltonian sub class. In the other classes where the `System` objects appears as `m_system`, the local energy can be found by

```
double localEnergy = m_system->getHamiltonian->computeLocalEnergy();
```

Similar functions exist for other essential objects, arrays and scalar.

11.6 Graphical User Interface (GUI)

Chapter 12

Results

Great quote.

Author

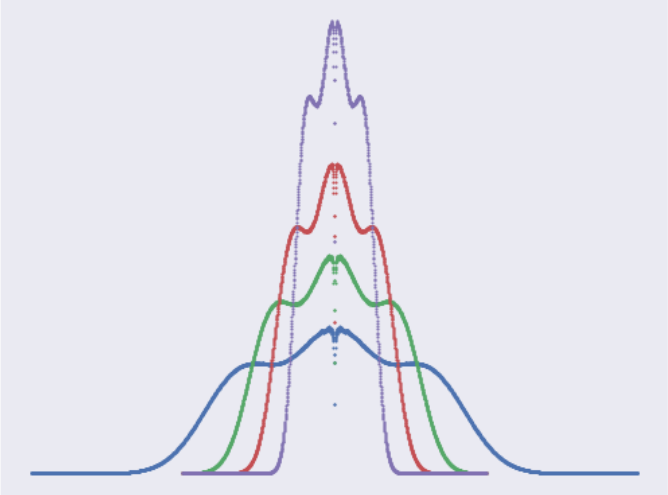


Figure 12.1: One-body density plots for a two-dimensional single quantum dot containing 12 electrons, popularly called an artificial Magnesium atom. The four graphs correspond to four different oscillator frequencies, where the weakest oscillator gives the broadest density distribution. It's quite artistic, isn't it?

After all, this thesis is related to a master in physics, and therefore the results and the physical insight is the interesting part. Before we move on to the physical results, we will take a quick look at some more technical results, more precisely the computational cost of various wave function structures and the energy convergence using various optimization tools.

For validation purposes, we will present a few selected results on the case without repulsive interaction and compare to analytical results. Thereafter, we study the case with repulsive interaction in a much larger scale, where we compare various wave function structures for different number of particles and oscillator strengths in two and three dimensions.

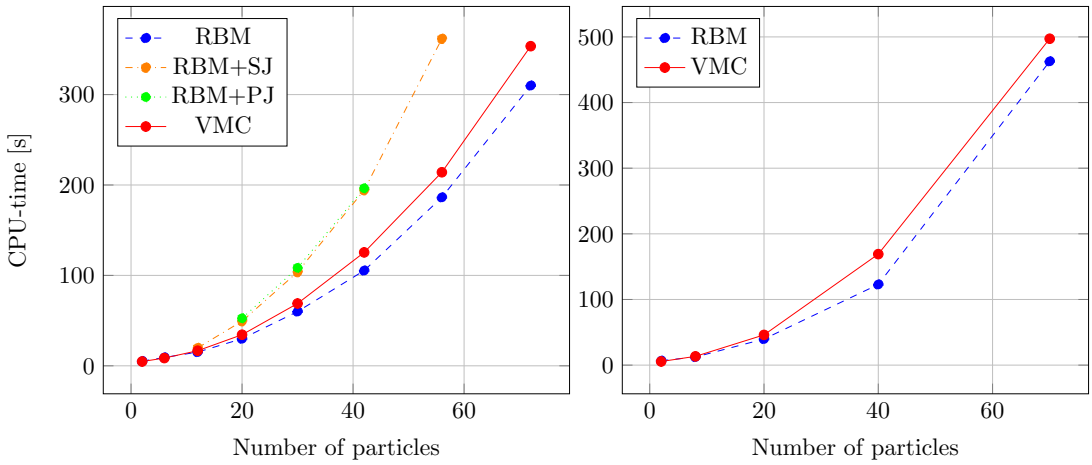


Figure 12.2: CPU-time per iteration as a function of number of particles for two and three dimensions. The solid line is standard variational Monte-Carlo (VMC), while the dashed lines are restricted Boltzmann machines without Jastrow factor (RBM), with simple Jastrow factor (RBM+SJ) and with Padé-Jastrow factor (RBM+PJ)

12.1 Computational cost

One of the major problems of simulating quantum many-body systems is the computational cost, which explodes as the system size increases. In figure (**??**) the CPU-time is plotted as a function of number of particles. We observe that the restricted Boltzmann machine (RBM) generally is cheaper to calculate compared to standard variational Monte-Carlo (VMC), which is a bit surprising. In the VMC trial wave function, we have only two variational parameters, while we in the RBM have $D \cdot N \cdot (1 + H) + H$ with N as number of particles, D as the number of dimensions and H as the number of hidden nodes. Throughout this thesis, we always set $H = N$, which gives 10,584 parameters for 72 particles in two dimensions and 14,980 parameters for 70 particles in three dimensions.

In other words, this evolve to be an optimization problem. The reason why the RBM still appears to have a cheaper cost, is probably that we do not need to calculate the distance matrix over and over again. For RBM+SJ and RBM+PJ, the cost is significantly higher due to the distance matrix.

As the applied theory used in quantum many-body simulations agrees perfectly with laboratory experiments, they can be considered as actual experiments. In that manner, one can use computer experiments to verify other experiments and even predict new things. Similarly to experiments in a laboratory, computer experiments are also dependent on external factors, especially when it comes to the CPU-time, and therefore it is important to do such measurements multiple times to find an accurate average time. The CPU-times above are the average from at least four independent runs for each number of particles. All the runs were performed on the Abel computational cluster, which is equipped with Supermicro X9DRT compute nodes with dual Intel E5-2670 CPUs running at 2.6 GHz. Different hardware might give different CPU-times, but the CPU-time ratios (the exponential factor) should be the same.

To estimate how fast the cost increases as we add more particles, we do linear regression with a function on the form $f(x) = \alpha x^{\beta}$ where x is the number of particles while α and β are the unknown parameters to be found. From the limited number of points, we have found the parameters and presented them in table (12.1).

Table 12.1: Optimal constants α and β for restricted Boltzmann machine (RBM), restricted Boltzmann machine with a simple Jastrow factor (RBM+SJ), restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ) and standard variational Monte-Carlo sampling (VMC).

	2D		3D	
	α	β	α	β
RBM	0.0840	1.92	0.0302	2.268
RBM+SJ	-	-	-	-
RBM+PJ	-	-	-	-
VMC	0.111	1.88	0.148	1.91

Although the RBM was found to be cheaper than VMC, we can see that the estimated exponential factor α is actually slightly larger. The prefactor β is significantly lower though.

12.2 Energy convergence

We want our calculations to converge fast and to be stable, and that is what the optimization tools are responsible for. In figure (**??**) we compare standard gradient descent to stochastic gradient descent and ADAM for two interacting electrons in a two- and three dimensional well.

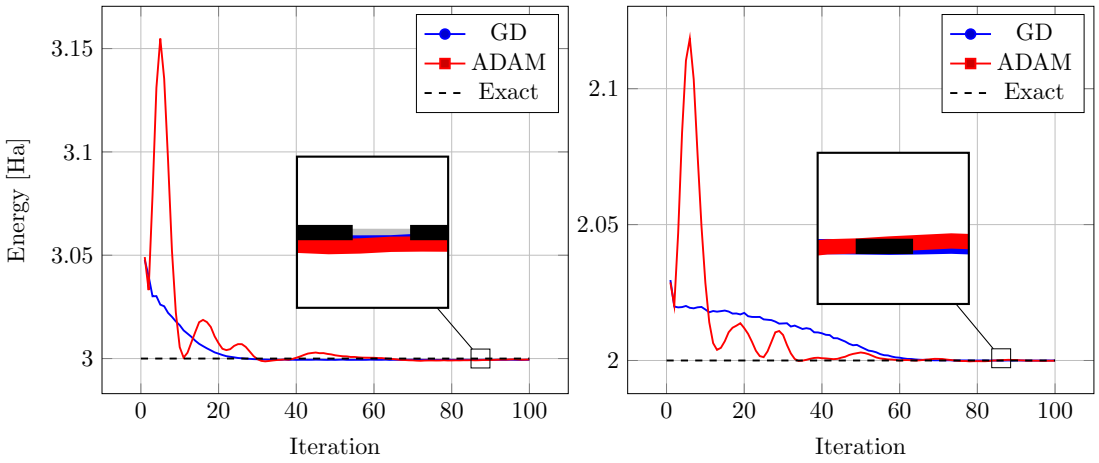


Figure 12.3: Energy convergence for two electrons in a two dimensional (a) and three dimensional (b) quantum dot. We use standard variational Monte-Carlo wave function, the learning rate was set to $\eta = 0.5$ and the number of Metropolis steps used for each iteration was $M = 2^{24}$.

We observe that the gradient descent methods in the cases above have a smoother convergence compared ADAM. However, this could be the case for the standard variational Monte-Carlo wave function only, so we better also look how well they do for other structures.

Even though the gradient descent methods seem to be the best choice based on those graphs, we experience that the energy occasionally explode when they are applied on more heavy systems, and therefore we will rely in the ADAM optimizer henceforth.

12.3 No Repulsive Interaction

We start with the non-interacting case in order to validate the implemented code. In this case we both know the exact energies and the exact one-body densities for an array of systems, and can therefore testify the flexibility of the code. We will focus on quantum dot systems.

12.3.1 Ground-state energy

The single quantum dot has analytical ground-state energies given by equation (3.4) and number of closed-shell particles given by equation (3.5). The first 5 closed-shell energies for $\omega = 0.5$ and $\omega = 1.0$ are presented in table (12.2).

Table 12.2: Energy of N non-interacting electrons trapped in a harmonic oscillator of frequency $\omega = 0.5$ and $\omega = 1.0$. RBM is a single Slater determinant with a plain Boltzmann machine baked in, while VMC is a standard variational Monte-Carlo Slater determinant.

		$\omega = 0.5$			$\omega = 1.0$		
	N	RBM	VMC	Exact	RBM	VMC	Exact
2D	2	1.00045(5)	1.0	1	2.00098(5)	2.0	2
	6	5.009(2)	5.0	5	10.010(1)	10.0	10
	12	-	14.0	14	-	28.0	28
	20	-	30.0	30	-	60.0	60
	30	-	55.0	55	-	110.0	110
3D	2	1.5050(3)	1.5	1.5	3.0063(2)	3.0	3
	8	-	9.0	9	-	18.0	18
	20	-	30.0	30	-	60.0	60
	40	-	75.0	75	-	150.0	150
	70	-	157.5	157.5	-	315.0	315

We observe that the restricted Boltzmann machine wave function is able to reproduce the exact energy for most of the cases, but when the number of particles get large, the statistical error gets significant.

12.3.2 One-body density

We will also focus on the one-body densities throughout the results, and comparing the obtained densities to the analytical ones is a good indicator on whether the implementation is correct or not. In figure (12.4) the one-body densities are plotted for quantum dots of two non-interacting electrons. The analytical one-body densities are found from the definition of one-body density in equation (2.20).

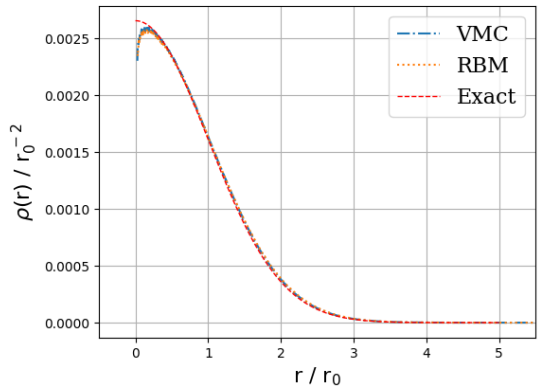
We observe that both the standard variational Monte-Carlo wave function (VMC) and the restricted Boltzmann machine (RBM) reproduce the analytical one-body density. The distribution gets narrower as the frequency is increased, and we also observe that the distributions are identical for two- and three dimensions with the same frequency.

12.4 With repulsive interaction

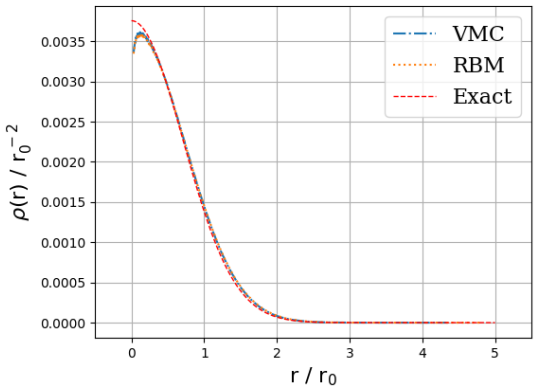
We now move on to the interesting case with repulsive interaction, where we no longer have analytical results, apart from a few semi-analytical energies and wave functions for the two- and three-dimensional single quantum dots.

To achieve good results, we apply an adaptive step number, which means that the number of steps per iteration is increased for the last iterations. Firstly, this makes the final energy more accurate due to better statistics. Secondly, we get less noisy electron density plots by using this technique. All results below are produced using $2^{20} = 1,048,576$ number of steps per iteration for the initial iterations. Then the number of steps is increased to $2^{24} = 16,777,216$ when we have 11 iterations left, and for the very last iteration we use $2^{28} = 268,435,456$ steps.

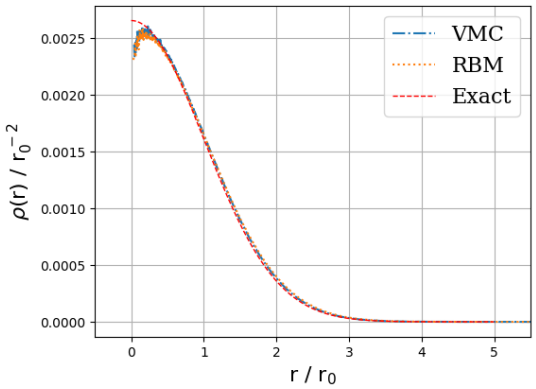
First we will look at the ground-state energy of closed shell quantum dots, and thereafter we move on to open shells. We will also have a look at double quantum dots, which we investigate with closed shells only.



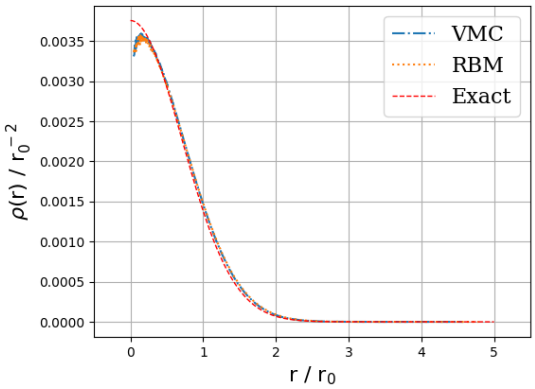
(a) 2D, $\omega = 0.5$



(b) 2D, $\omega = 1.0$



(c) 3D, $\omega = 0.5$



(d) 3D, $\omega = 1.0$

Figure 12.4: One-body densities of two non-interacting electrons in two- and three dimensions for $\omega = 0.5$ and $\omega = 1.0$.

12.4.1 Quantum dots

12.4.1.1 Ground-state energy, $S = 0$

By utilizing the symmetry of quantum dots of two electrons, M.Taut was able to obtain semi-analytical energies for some specific frequencies ω. More precisely, he found the energy to be $E = 3$ for the frequency $\omega = 1$ and $E = 2/3$ for the frequency $\omega = 1/6$ for the two-dimensional case, and $E = 2$ for the frequency $\omega = 1/2$ and $E = 1/2$ for the frequency $\omega = 1/10$ for the three-dimensional case. [6][7]

For other references, we need to rely on what researchers have found before us. Since diffusion Monte-Carlo (DMC) is known to give very accurate results, we will mainly compare our results to J. Høgberget's DMC computations, which exist for closed shell quantum dots of a maximum of 56 electrons in two dimensions and a maximum of 20 particles in three dimensions. [15]. Comparing the energy to the Hartree-Fock limit is also interesting, mainly because of the Boltzmann machines. We use A.Mariadason's computations for this for quantum dots of a maximum of 20 electrons in two dimensions, and a maximum of 8 particles in three dimensions. [**mariadason˙quantum˙2018**]

Ground state energy computations of two- and three dimensional quantum dots are found in tables (12.4) and (12.5) respectively. They are performed by a restricted Boltzmann machine (RBM), restricted Boltzmann machine with a simple Jastrow factor (RBM+SJ), restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ), partly restricted Boltzmann machine (PRBM) and standard variational Monte-Carlo (VMC). In

addition, the Hartree-Fock limit (HF) and diffusion Monte-Carlo (DMC) are present for reference purposes.

We observe that the method where less physical intuition is used, RBM, is the one that gives the highest energies. However, compared to HF, the energy is mostly lower. When we add more intuition in form of a simple Jastrow factor, the energy drops significantly. RBM+PJ and VMC are on the same level, with VMC pinched in front.

12.4.1.2 Ground-state energy, $S \neq 0$

When the number of particles in the quantum dot is among the magic numbers presented in equation (3.5), the ground state is guaranteed to be found at $L = S = 0$. However, if we want to look at dots of other sizes, this is often not true, and we will therefore look at some cases with $S \neq 0$. We will still stick to $L = 0$ and closed shell systems.

Imagine a dot of four particles with $S = 1$ (three have spin up and the last has spin down). The ground state is obviously found when a spin up and a spin down particle is in the lowest energy state, and the remaining particles are in the first excited energy state. Since the lowest states and the first excited states with spin up are filled up, this is a closed shell system.

Again, our results will be benchmarked to DMC, with references Ref.[**pederiva˙diffusion˙2000**] and Ref.[**ghosal˙incipient˙2007**]. The former reference also presents Hartree-Fock results which we also will use.

The results are listed in table (12.3), where we use a restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ) and standard variational Monte-Carlo (VMC).

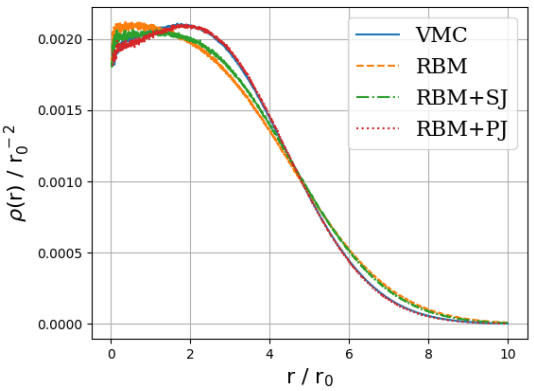
We find both the RBM+PJ and VMC to reproduce the reference energy in all the cases.

Table 12.3: The ground state energy of two-dimensional circular quantum dots of frequency ω for a given spin configuration (L,S). The results were obtained by a restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ) and standard variational Monte-Carlo (VMC). For reference, the Hartree-Fock limit results from Ref.[**pederiva˙diffusion˙2000**] (HF) and diffusion Monte-Carlo results from Refs.[**pederiva˙diffusion˙2000**],[**ghosal˙incipient˙2007**] (DMC) are listed. All energies are given in units of $\hbar$, and the numbers in parenthesis are the statistical uncertainties in the last digit.

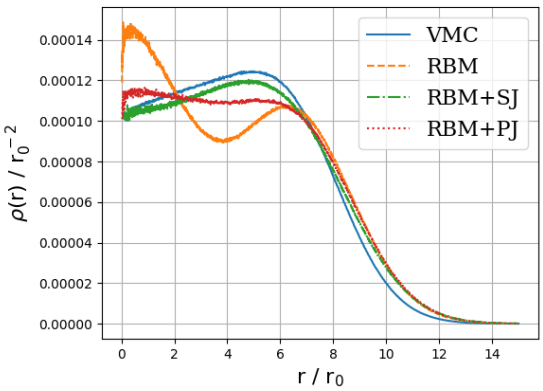
N	ω	L	S	RBM+PJ	VMC	HF (Ref.[**pederiva˙diffusion˙2000**])	DMC (Ref.[**pederiva˙diffusi** (Ref.[**ghosal˙incipien**
4	0.28	0	1	3.7475(2)	3.7711(5)	3.9033	3.7135(3)[a]
8	0.28	0	1	12.828(3)	12.849(1)	13.1887	12.6903(7)[a]
	0.25	0	1	11.807(1)	11.852(1)	-	11.6697(1)[b]
9	0.28	0	3/2	15.886(4)	15.812(4)	16.1544	15.4784(7)[a]
	3.0	0	3/2	97.164(4)	96.936(7)	-	97.0095(3)[b]
11	0.28	0	1/2	22.285(2)	22.252(1)	22.8733	22.0750(4)[a]

12.4.1.3 One-body density

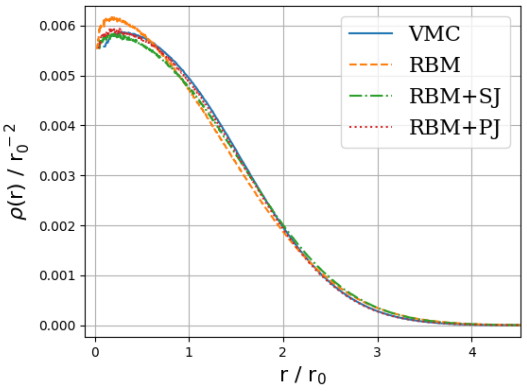
Another quantity of particular interest is the one-body density. We have produced one-body density plots using a restricted Boltzmann machine (RBM), a restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ) and standard variational Monte-Carlo (VMC) for various frequencies up to 72 electrons in two dimensions and 70 electrons in three dimensions. The plots can be found in figures (12.5-12.8) (two dimensions) and (12.9-12.10) (three dimensions).



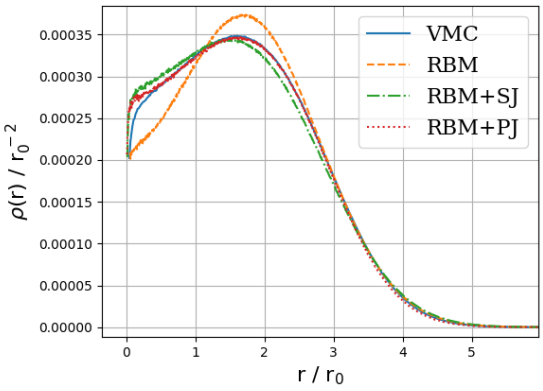
(a) 2P, $\omega = 0.1$



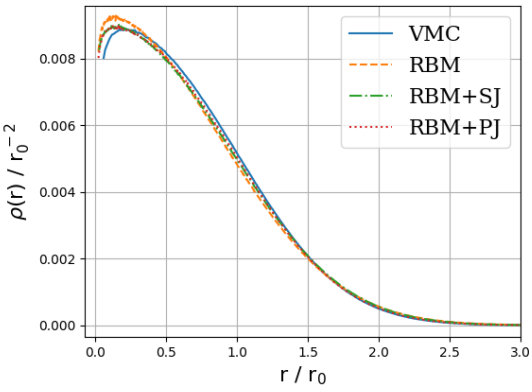
(b) 6P, $\omega = 0.1$



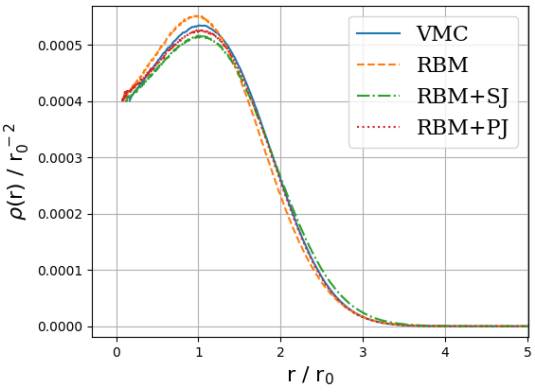
(c) 2P, $\omega = 0.5$



(d) 6P, $\omega = 0.5$



(e) 2P, $\omega = 1.0$



(f) 6P, $\omega = 1.0$

Figure 12.5: One-body densities of two and six interacting electrons in two dimensions for various oscillator frequencies produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM) and restricted Boltzmann machine with Padé-Jastrow factor (RBMPJ). Stochastic gradient descent was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268.435.456$.

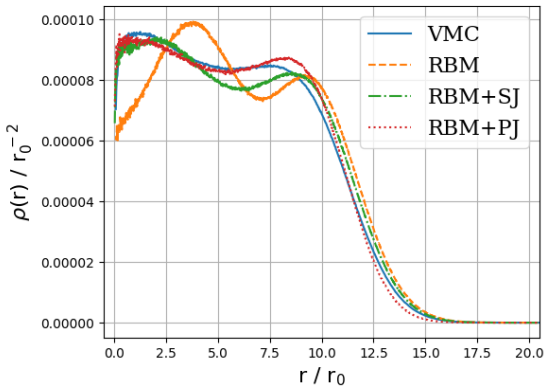
Table 12.4: The ground state energy of two-dimensional circular quantum dots of frequency ω obtained by various methods. The column on the left-hand-side represents restricted Boltzmann machine (RBM), followed by restricted Boltzmann machine with simple Jastrow factor (RBM+SJ), restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ), partly restricted Boltzmann machine (PRBM), the Hartree-Fock limit (HF), standard variational Monte-Carlo with Hartree-Fock basis (VMC+HF), standard variational Monte-Carlo with Hermite basis (VMC) and diffusion Monte-Carlo (DMC). Hartree-Fock results are taken from Ref.[**mariadason˙quantum˙2018**], DMC results are taken from [15] and semi-analytical results are taken from [7]. N is the number of electrons in the dot, and $L = S = 0$. The energy is given in units of $\hbar$, and the numbers in parenthesis are the statistical uncertainties in the last digit.

N	ω	RBM	RBM+SJ	RBM+PJ	PRBM	HF (Ref.[**mariadason˙quantum˙2018**])	VMC+HF (Ref.[**mariadason˙quantum˙2018**])
2	0.1	0.4728(1)	0.44859(6)	0.44130(5)	0.4959(2)	0.525635	-
	1/6	0.7036(1)	0.67684(7)	0.66715(6)	0.7326(2)	0.768675	-
	0.28	1.0705(2)	1.03485(7)	1.02185(1)	1.0987(2)	1.14171	-
	0.5	1.7231(2)	1.67787(8)	1.65957(1)	1.7182(3)	1.79974	-
	1.0	3.0822(2)	3.0255(1)	3.00002(2)	3.0566(3)	3.16190	-
6	0.1	3.697(1)	3.6584(4)	3.57832(2)	-	3.85238	-
	0.28	7.924(1)	7.7503(4)	7.6245(2)	-	8.01957	-
	0.5	12.242(1)	11.9659(5)	11.8113(2)	11.995(3)	12.2713	-
	1.0	20.731(1)	20.4061(7)	20.1853(2)	20.483(1)	20.7192	-
12	0.1	12.705(2)	12.566(2)	12.3416(4)	-	12.9247	-
	0.28	26.389(2)	26.083(1)	25.7331(5)	-	26.5500	-
	0.5	40.440(3)	39.694(1)	39.2743(6)	-	40.2161	-
	1.0	67.632(3)	66.378(2)	65.9550(7)	-	66.9113	-

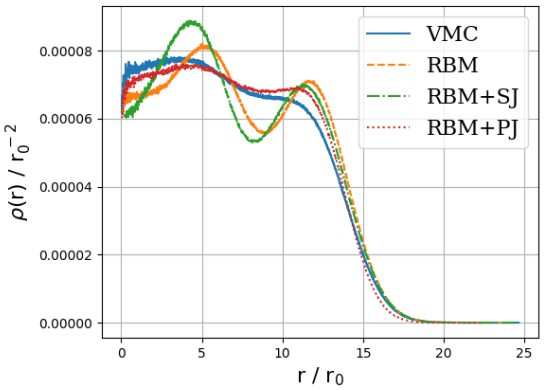
20	0.1	30.824(2)	30.567(3)	30.1553(9)	-	31.1902	-	30.0730(6)	29.9775(1)
	0.28	63.746(4)	62.811(3)	62.148(1)	-	63.5390	-	62.0598(7)	61.9263(1)
	0.5	97.166(5)	94.920(4)	94.104(1)	-	95.7328	-	94.0391(9)	93.8755(1)
	1.0	159.640(5)	157.209(4)	156.104(1)	-	158.004	-	156.106(2)	155.8823(1)
30	0.1	61.829(5)	61.351(4)	60.774(2)	-	-	-	60.585(1)	60.4203(2)
	0.28	126.958(6)	126.067(5)	124.437(2)	-	-	-	124.195(2)	123.9685(2)
	0.5	191.495(7)	188.995(5)	187.493(2)	-	-	-	187.325(3)	187.0426(2)
	1.0	315.364(8)	311.468(7)	308.989(2)	-	-	-	308.957(2)	308.5627(2)
42	0.1	109.892(6)	110.030(7)	-	-	-	-	107.928(2)	107.6389(2)
	0.28	224.462(8)	224.587(8)	-	-	-	-	220.224(2)	219.8426(2)
	0.5	337.523(8)	333.582(9)	331.410(3)	-	-	-	331.276(3)	330.6306(2)
	1.0	553.40(1)	549.76(1)	543.746(3)	-	-	-	543.738(7)	542.9428(8)
56	0.1	-	180.52(1)	-	-	-	-	176.774(3)	175.9553(7)
	0.28	364.85(1)	366.91(1)	-	-	-	-	359.63(1)	358.145(2)
	0.5	547.46(1)	545.74(1)	-	-	-	-	538.686(9)	537.353(2)
	1.0	894.12(2)	890.70(2)	-	-	-	-	880.352(5)	879.3986(6)
72	0.1	-	-	-	-	-	-	276.83(1)	-
	0.28	-	-	-	-	-	-	551.00(2)	-
	0.5	843.05(2)	-	-	-	-	-	822.82(1)	-
	1.0	-	-	-	-	-	-	1340.520(7)	-

Table 12.5: The ground state energy of three-dimensional circular quantum dots of frequency ω obtained by various methods. The column on the left-hand-side represents restricted Boltzmann machine (RBM), followed by restricted Boltzmann machine with simple Jastrow factor (RBM+SJ), restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ), partly restricted Boltzmann machine (PRBM), the Hartree-Fock limit (HF), standard variational Monte-Carlo with Hartree-Fock basis (VMC+HF), standard variational Monte-Carlo with Hermite basis (VMC) and diffusion Monte-Carlo (DMC). Hartree-Fock results are taken from Ref.[**mariadason˙quantum˙2018**], DMC results are taken from [15] and semi-analytical results are taken from [6]. N is the number of electrons in the dot, and $L = S = 0$. The energy is given in units of $\hbar$, and the numbers in parenthesis are the statistical uncertainties in the last digit.

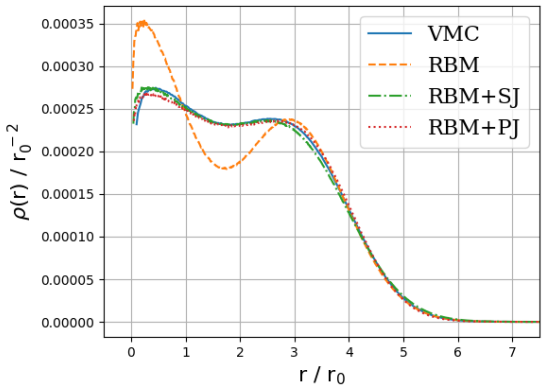
N	ω	RBM	RBM+SJ	RBM+PJ	PRBM	HF (Ref.[**mariadason˙quantum˙2018**])	VMC+HF (Ref.[**mariadason˙quantum˙2018**])	
2	0.1	0.5178(1)	0.50225(3)	0.50073(7)	-	0.529065	-	
	0.28	1.2259(1)	1.20471(4)	1.20201(5)	-	1.23722	-	
	0.5	2.0269(1)	2.00374(4)	2.00009(4)	-	2.03851	-	
	1.0	3.7571(1)	3.73559(4)	3.73032(4)	-	3.77157	-	
8	0.1	6.549(7)	5.7995(5)	5.8448(7)	-	5.86255	-	
	0.28	13.098(2)	12.2512(4)	12.2084(2)	-	12.3987	-	
	0.5	19.487(2)	19.0266(4)	18.9831(2)	-	19.1916	-	
	1.0	33.302(1)	32.739(4)	32.6883(2)	-	32.9246	-	
20	0.1	27.813(2)	-	-	-	-	-	
	0.28	57.700(4)	-	-	-	-	-	
	0.5	87.840(4)	-	-	-	-	-	
	1.0	146.292(4)	-	-	-	-	-	
40	0.1	-	-	-	-	-	88.182(1)	-
	0.28	182.714(6)	-	-	-	-	179.567(1)	-
	0.5	275.262(7)	-	-	-	-	269.746(1)	-
	1.0	452.732(8)	-	-	-	-	442.602(2)	-
70	0.1	-	-	-	-	-	227.082(6)	-
	0.28	-	-	-	-	-	456.941(2)	-
	0.5	693.52(2)	-	-	-	-	682.277(5)	-
	1.0	1129.40(2)	-	-	-	-	1108.950(4)	-



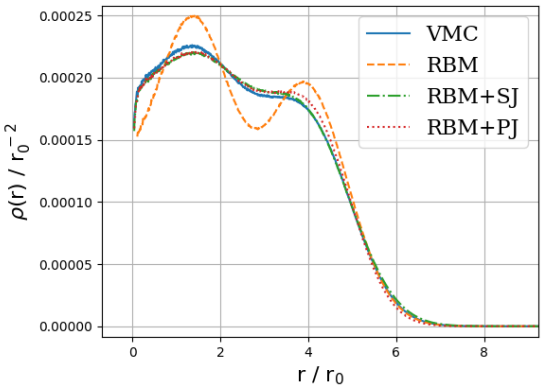
(a) 12P, $\omega = 0.1$



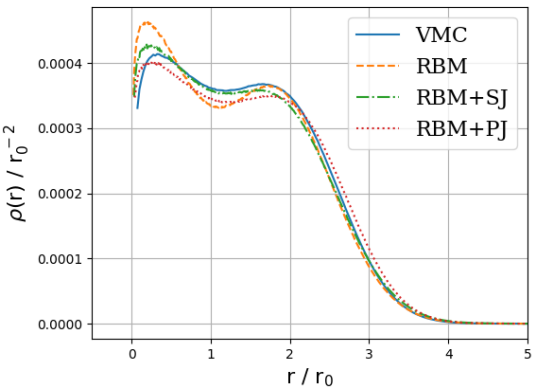
(b) 20P, $\omega = 0.1$



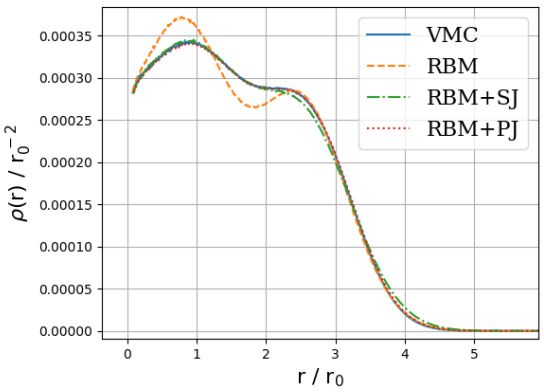
(c) 12P, $\omega = 0.5$



(d) 20P, $\omega = 0.5$

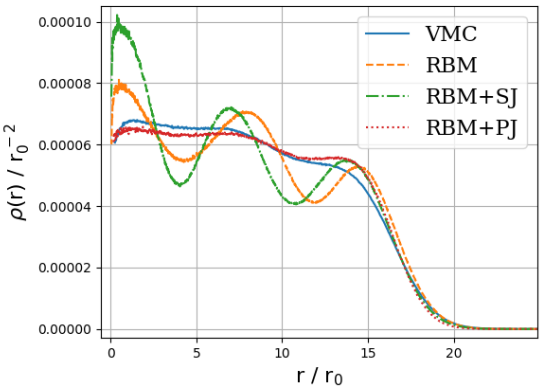


(e) 12P, $\omega = 1.0$

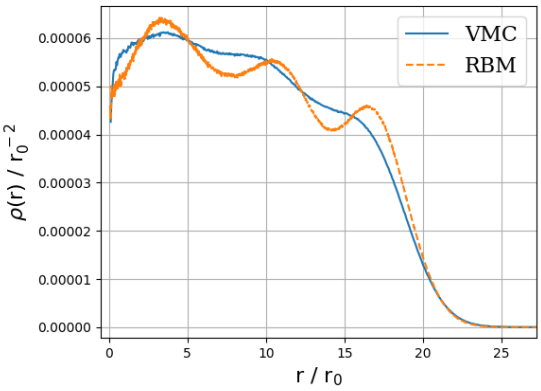


(f) 20P, $\omega = 1.0$

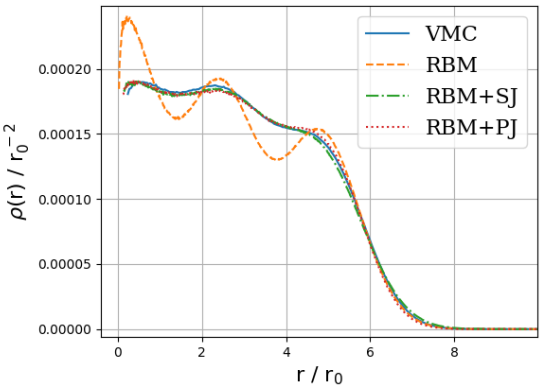
Figure 12.6: One-body densities of 12 and 20 interacting electrons in two dimensions for various oscillator frequencies produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM) and restricted Boltzmann machine with Padé-Jastrow factor (RBMPJ). Stochastic gradient descent was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268.435.456$.



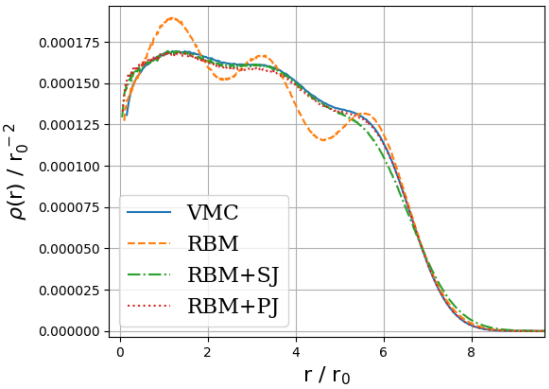
(a) 30P, $\omega = 0.1$



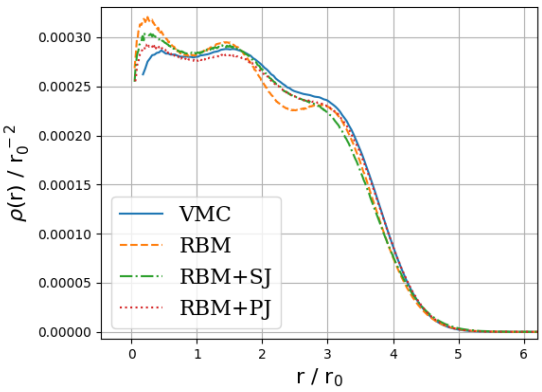
(b) 42P, $\omega = 0.1$



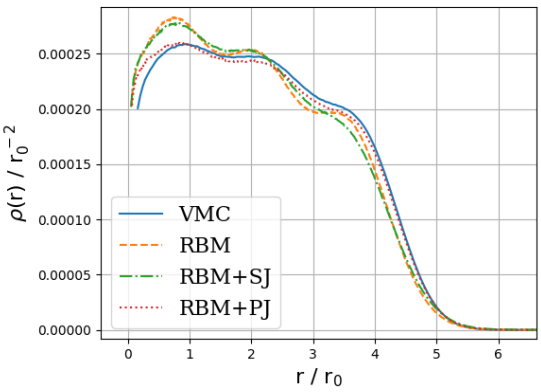
(c) 30P, $\omega = 0.5$



(d) 42P, $\omega = 0.5$

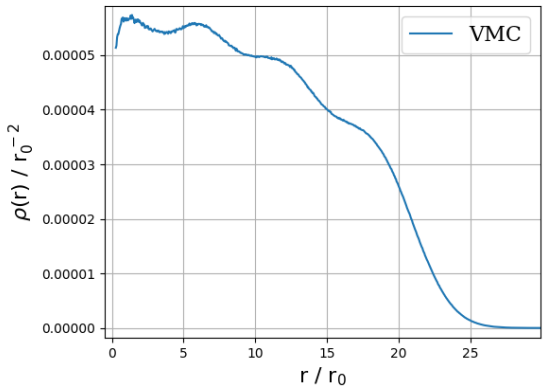


(e) 30P, $\omega = 1.0$

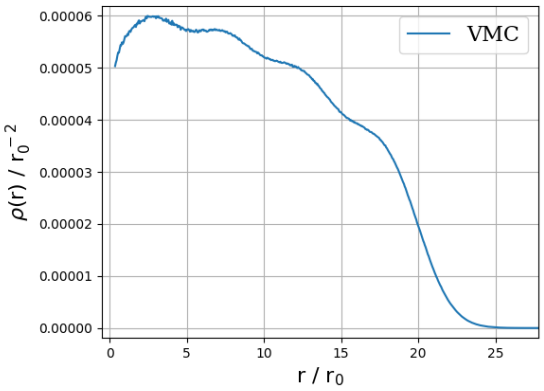


(f) 42P, $\omega = 1.0$

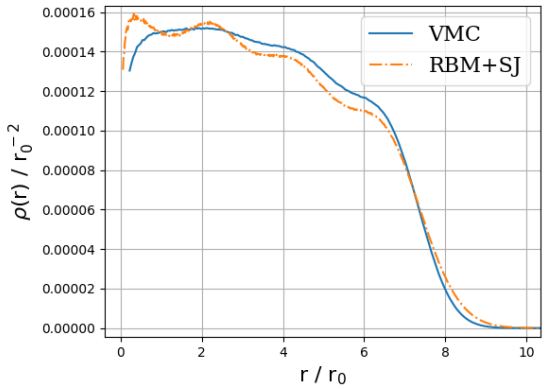
Figure 12.7: One-body densities of 30 and 42 interacting electrons in two dimensions for various oscillator frequencies produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM) and restricted Boltzmann machine with Padé-Jastrow factor (RBMPJ). Stochastic gradient descent was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268.435.456$.



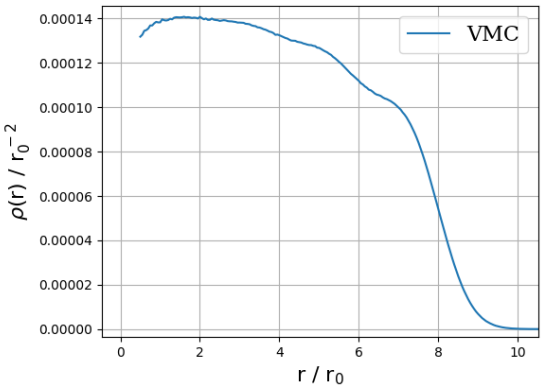
(a) 56P, $\omega = 0.1$



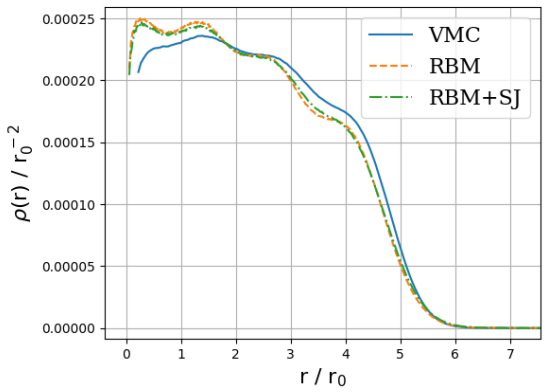
(b) 72P, $\omega = 0.1$



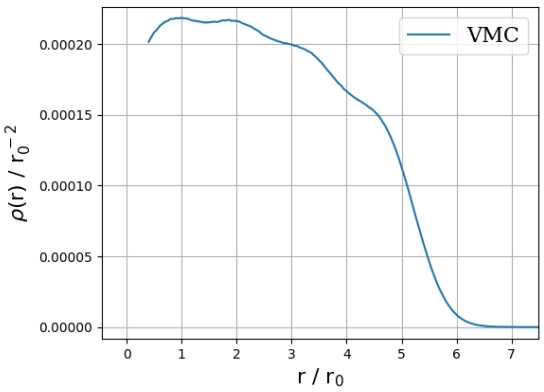
(c) 56P, $\omega = 0.5$



(d) 72P, $\omega = 0.5$

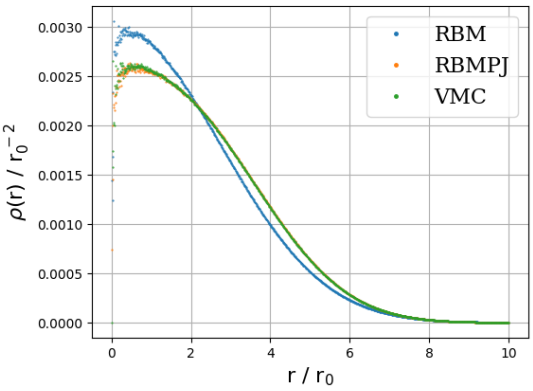


(e) 56P, $\omega = 1.0$

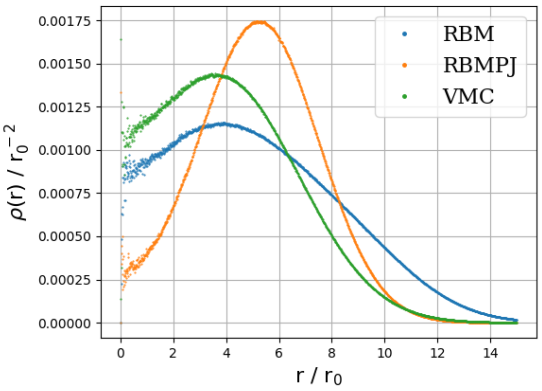


(f) 72P, $\omega = 1.0$

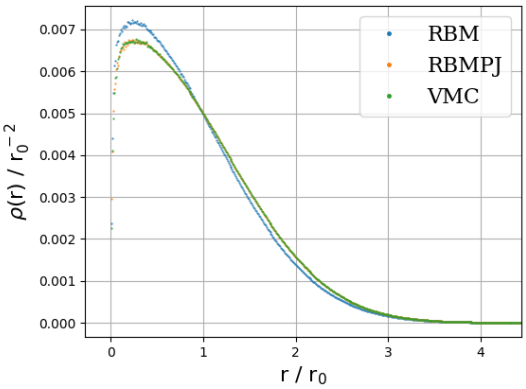
Figure 12.8: One-body densities of 56 and 72 interacting electrons in two dimensions for various oscillator frequencies produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM) and restricted Boltzmann machine with Padé-Jastrow factor (RBMPJ). Stochastic gradient descent was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268.435.456$.



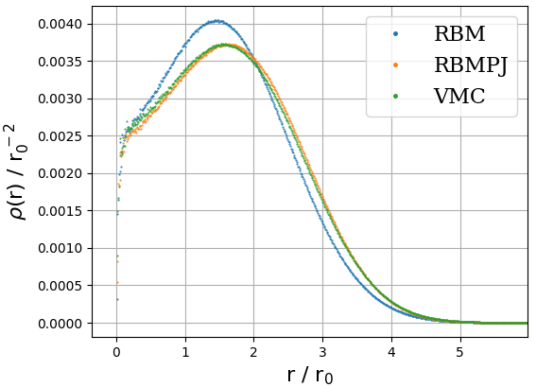
(a) 2P, $\omega = 0.1$



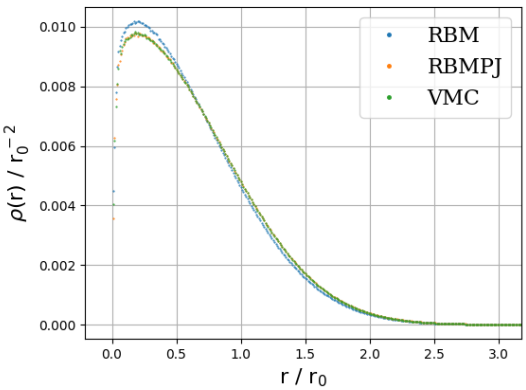
(b) 8P, $\omega = 0.1$



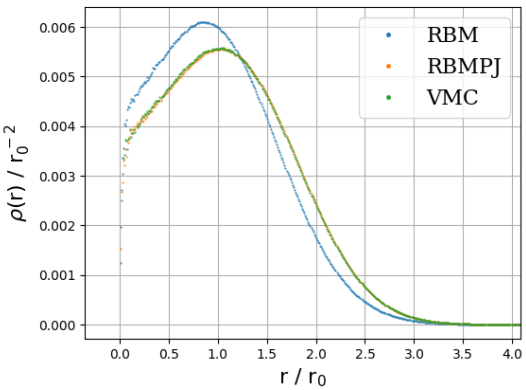
(c) 2P, $\omega = 0.5$



(d) 8P, $\omega = 0.5$

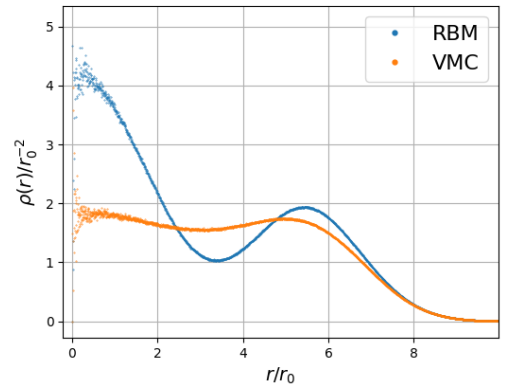


(e) 2P, $\omega = 1.0$

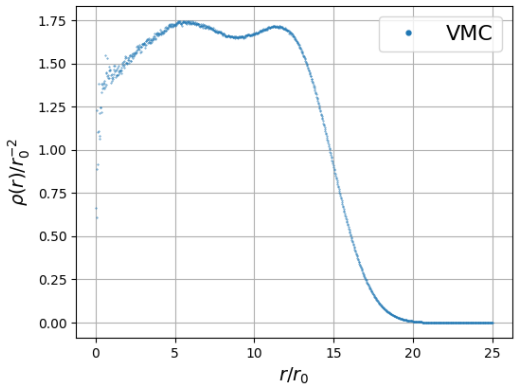


(f) 8P, $\omega = 1.0$

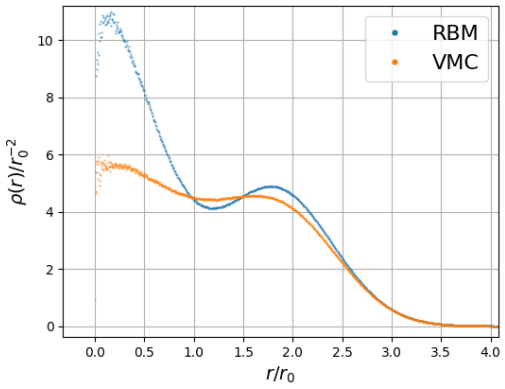
Figure 12.9: One-body densities of two and eight interacting electrons in three dimensions for various oscillator frequencies produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM) and restricted Boltzmann machine with Padé-Jastrow factor (RBMPJ). Stochastic gradient descent was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268.435.456$.



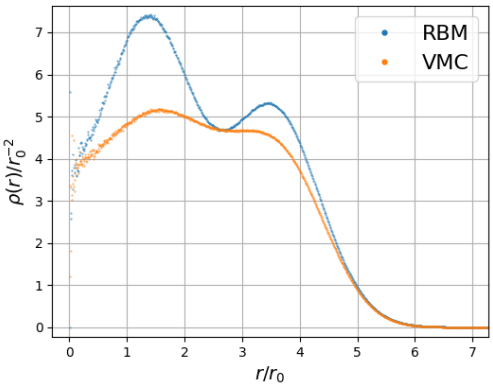
(a) 2P, $\omega = 0.1$



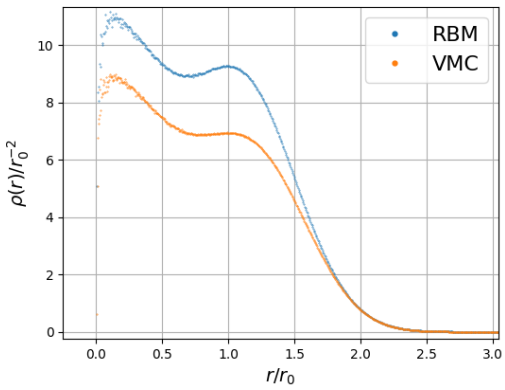
(b) 8P, $\omega = 0.1$



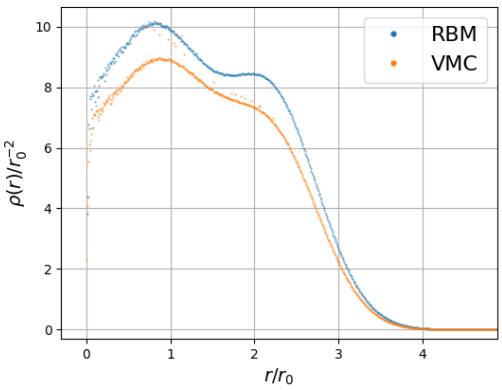
(c) 2P, $\omega = 0.5$



(d) 8P, $\omega = 0.5$



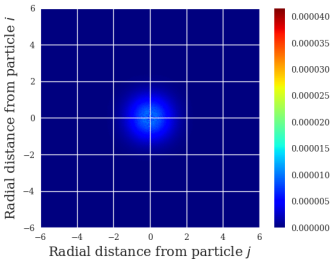
(e) 2P, $\omega = 1.0$



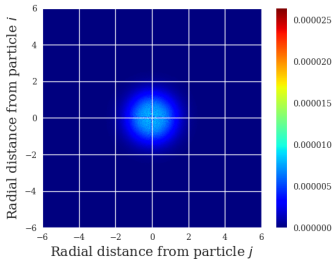
(f) 8P, $\omega = 1.0$

Figure 12.10: One-body densities of 20 and 40 interacting electrons in three dimensions for various oscillator frequencies produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM) and restricted Boltzmann machine with Padé-Jastrow factor (RBMPJ). Stochastic gradient descent was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268.435.456$.

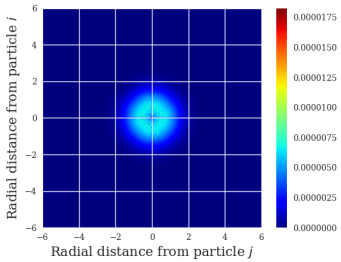
12.4.1.4 Two-body density



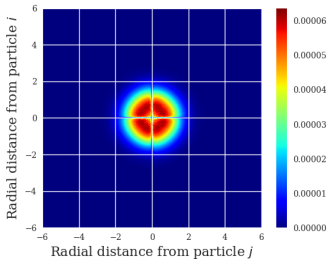
(a) RBM, 2P



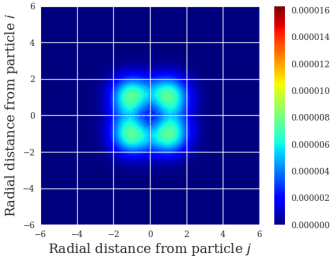
(b) RBM+SJ, 2P



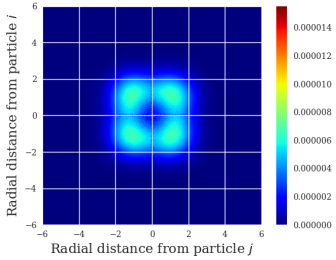
(c) RBM+PJ, 2P



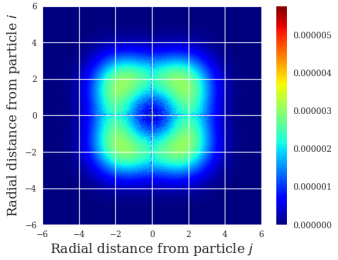
(d) VMC, 2P



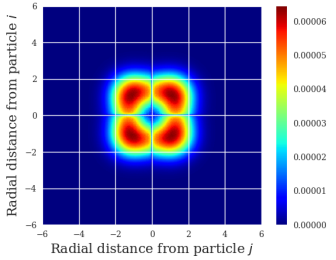
(e) RBM, 6P



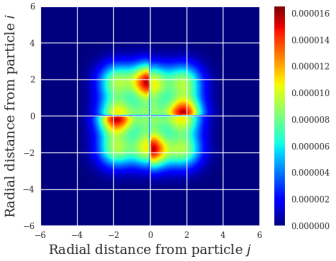
(f) RBM+SJ, 6P



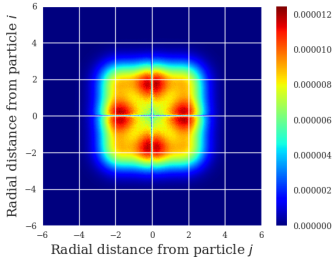
(g) RBM+PJ, 6P



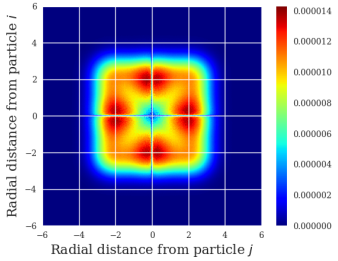
(h) VMC, 6P



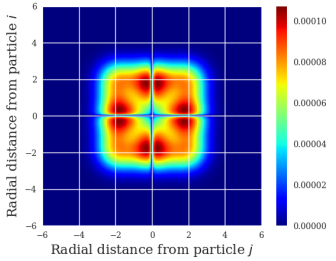
(i) RBM, 12P



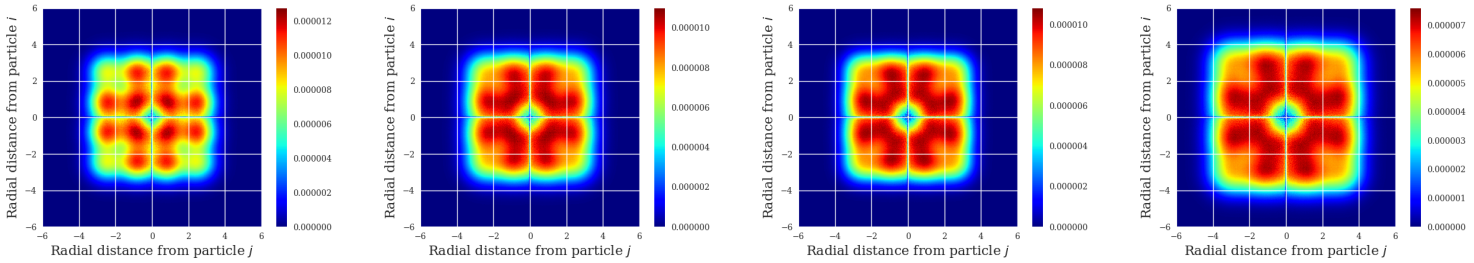
(j) RBM+SJ, 12P



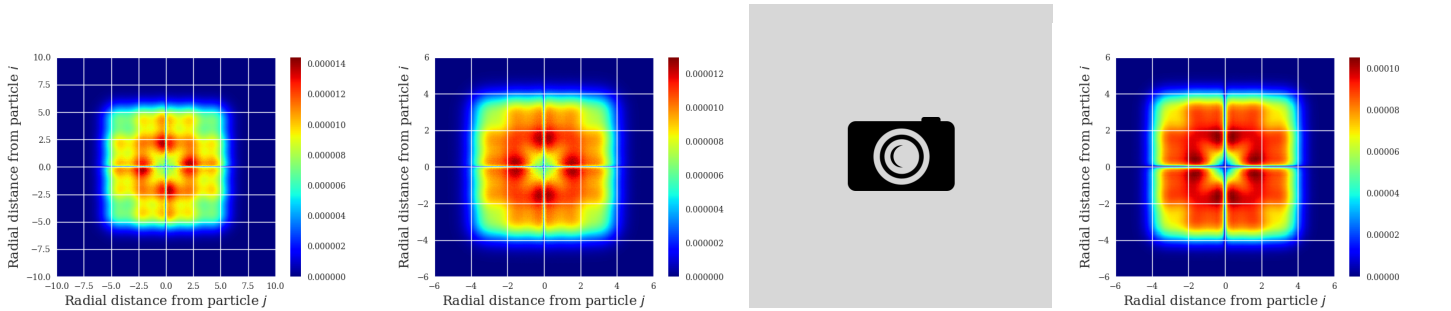
(k) RBM+PJ, 12P



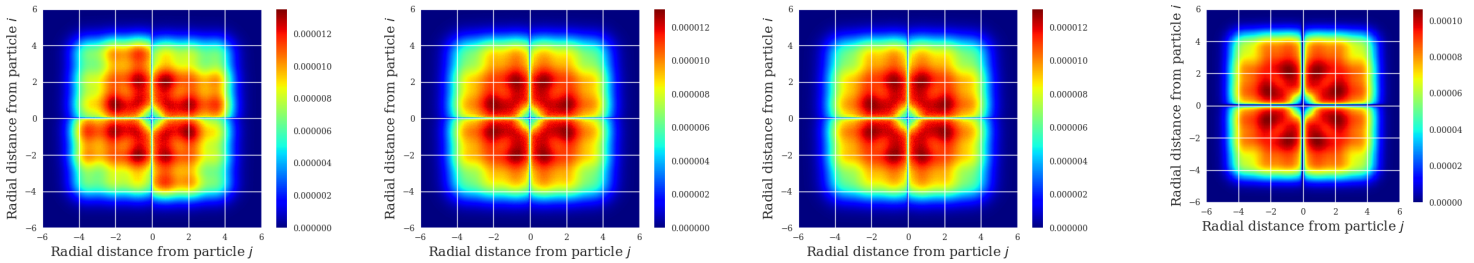
(l) VMC, 12P



(m) RBM, 20P (n) RBM+SJ, 20P (o) RBM+PJ, 20P (p) VMC, 20P



(q) RBM, 30P (r) RBM+SJ, 30P (s) RBM+PJ, 30P (t) VMC, 30P



(u) RBM, 42P (v) RBM+SJ, 42P (w) RBM+PJ, 42P (x) VMC, 42P

Figure 12.11: Two-body densities for interacting electrons in two dimensions for various oscillator frequencies produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM) and restricted Boltzmann machine with Padé-Jastrow factor (RBMPJ). ADAM optimizer was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268.435.456$.

Table 12.6: Double quantum dots. F is the number of functions used in the expansion.

N	ω	VMC (F=1)	RBM+PJ	VMC (F>1) (Ref.[**mariadason˙quantum˙2018**])	VMC+HF (Ref.[**mariadason˙quantum˙2018**])
2	0.1	0.41982(4)	0.41939(3)		
	0.28	0.9365(1)	0.926176(4)		
	0.5	1.4847(2)	1.44004(4)		
	1.0	2.6331(5)	2.38342(3)	2.42238(4)	2.36618(4)
4	0.1				
	0.28				
	0.5				
	1.0	-	-	7.95247(4)	7.90232(4)
6	0.1				
	0.28				
	0.5				
	1.0	-	-	16.61419(4)	16.55609(4)
8	0.1				
	0.28				
	0.5				
	1.0	-	-		

12.4.2 Double Quantum Dots

We also look at double quantum dots...

12.4.3 Low frequency

Wigner crystals

Chapter 13

Conclusion and future work

See if machine learning is able to describe the three-body interaction, with nuclear physics applications.

Appendix A

Dirac notation

The Dirac notation, also called bracket notation, was suggested by Paul Dirac in a 1939 paper with the purpose of improving the reading ease. [**dirac˙1939**]

Appendix B

Scaling

B.1 Quantum dots - Natural units

The Hamiltonian is in one dimension given by

$$\hat{\mathcal{H}} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}m\omega^2 x^2 \tag{B.1}$$

which has corresponding wave functions

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \cdot \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} \exp\Big(-\frac{m\omega}{2\hbar}x^2\Big) H_n\Big(\sqrt{\frac{m\omega}{\hbar}}x\Big). \tag{B.2}$$

We want to get rid of $\hbar$ and m in equation (B.1), and we initially scale $\hat{\mathcal{H}}' \equiv \hat{\mathcal{H}}/\hbar$, such that the Hamiltonian reduces to

$$\hat{\mathcal{H}}' = -\frac{\hbar}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}\frac{m\omega^2}{\hbar}x^2. \tag{B.3}$$

One can now observe that the fraction $\hbar/m$ comes in both terms, which can be taken out by introducing a characteristic length $x' \equiv x \cdot \sqrt{m/\hbar}$. The final Hamiltonian is

$$\hat{\mathcal{H}} = \frac{1}{2}\frac{\partial^2}{\partial x^2} + \frac{1}{2}\omega^2 x^2 \tag{B.4}$$

which corresponds to setting $\hbar = m = 1$. In natural units, one often sets $\omega = 1$ as well by scaling $\hat{\mathcal{H}}' = \hat{\mathcal{H}}/\hbar\omega$, but since we want to keep the ω-dependency, we do it slightly different. This means that the exact wave functions for the one-particle-one-dimension case is

$$\psi_n(x) = \exp\Big(-\frac{\omega}{2}x^2\Big) H_n(\sqrt{\omega}x) \tag{B.5}$$

where we take advantage of the Metropolis algorithm and ignore the normalization constant.

B.2 Atomic systems - Atomic units

The atomic Hamiltonian in its simplest form is given by

$$\hat{\mathcal{H}} = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial r^2} - k\frac{Ze^2}{r} + \frac{\hbar^2 l(l+1)}{2mr^2}. \tag{B.6}$$

The first step is to divide all terms by $\hbar^2/m$,

$$\hat{\mathcal{H}} \cdot \frac{m}{\hbar^2} = -\frac{1}{2}\frac{\partial^2}{\partial r^2} + k\frac{m}{\hbar^2}\frac{Ze^2}{r} - \frac{1}{2}\frac{l(l+1)}{r^2} \tag{B.7}$$

and then define $a \equiv mke^2/\hbar^2$. If we then divide all terms by a^2, we can write the Hamiltonian as

$$\hat{\mathcal{H}} \cdot \frac{m}{a^2\hbar^2} = -\frac{1}{2a^2}\frac{\partial^2}{\partial r^2} + \frac{Z}{ar} - \frac{1}{2a^2}\frac{l(l+1)}{r^2} \tag{B.8}$$

and obtain a dimensionless equation by scaling $r' = ar$ and $\hat{\mathcal{H}}' = \hat{\mathcal{H}} \cdot m/a^2\hbar^2$. The final Hamiltonian is

$$\hat{\mathcal{H}} = -\frac{1}{2}\frac{\partial^2}{\partial r^2} - \frac{Z}{r} + \frac{l(l+1)}{2r^2}. \tag{B.9}$$

B.3 Comparison between natural and atomic units

As a summary, we will present how the observable are scaled nicely in a table, and how to convert them back to standard units.

Table B.1: Comparison the natural and atomic units presented above.

Quantity	Symbol	Natural units	Atomic units
Energy	E	$1/\hbar$	$\hbar^2/m(ke^2)^2$
Length	r	$\sqrt{m/\hbar}$	$m(ke^2)/\hbar^2$
Reduced Planck's constant	$\hbar$	1	1
Elementary charge	e	1	$\sqrt{\alpha}$
Coulomb's constant	k_e	1	1
Boltzmann's constant	k_B	1	1
Electron rest mass	m_e	1	$511keV$

By a mix of classical and quantum mechanics, Niels Bohr found the quantized energy levels and radii in an atom to be

$$E_n = -\frac{Z^2(ke^2)^2 m}{2\hbar^2 n^2} \approx -\frac{Z^2}{n^2}13.6 \text{ eV} \tag{B.10}$$

and

$$r_n = \frac{n^2\hbar^2}{Zke^2 m} \approx \frac{n^2}{Z}5.29 \cdot 10^{-11} \text{ m} \tag{B.11}$$

respectively. What we observe, is that the energy in atomic units is scaled with respect to $2 \cdot E_1$ and r_1, which means that

$$1 \text{ a.u.} = 2 \cdot 13.6 \text{ eV} \qquad \text{and} \qquad 1 \text{ a.u.} = 5.29 \cdot 10^{-11} \text{ m} \tag{B.12}$$

Appendix C

Associated Laguerre Polynomials

Associated Laguerre polynomials $L^p_{q-p}(x)$ are solutions of the linear differential equation

$$xy'' + (p + 1 - x)y' + qy = 0, \tag{C.1}$$

which can be represented by the Rodriguez formula

$$L^p_{q-p}(x) = (-1)^p \Big(\frac{d}{dx}\Big)^p L_q(x) \tag{C.2}$$

where $L_q(x)$ are the Laguerre polynomials

$$L_q(x) = e^x \Big(\frac{d}{dx}\Big)^q e^{-x} x^q. \tag{C.3}$$

C.1 Recursive relation between polynomials

If one knows the elements $L^k_{n-1}(x)$ and $L^k_n(x)$, the next element can be calculated by

$$L^k_{n+1}(x) = \frac{(2n + k + 1 - x)L^k_n(x) - (n + k)L^k_{n-1}(x)}{n + 1}. \tag{C.4}$$

The first few elements are

$$L^k_0(x) = 1 \tag{C.5}$$

$$L^k_1(x) = 1 + k - x \tag{C.6}$$

$$L^k_2(x) = \frac{1}{2}\Big[x^2 - 2(k + 2)x + (k + 1)(k + 2)\Big] \tag{C.7}$$

$$L^k_3(x) = \frac{1}{6}\Big[- x^3 + 3(k + 3)x^2 - 3(k + 2)(k + 3)x + (k + 1)(k + 2)(k + 3)\Big] \tag{C.8}$$

Appendix D

General Gaussian-binary RBM wave function

D.1 Derive the wave function

We have seen that the probability of having a set of positions $\boldsymbol{x}$ with a set of hidden nodes $\boldsymbol{h}$ is given by

$$F(\boldsymbol{x}, \boldsymbol{h}) = \frac{1}{Z} \exp(-\beta E(\boldsymbol{x}, \boldsymbol{h})) \tag{D.1}$$

where we set $\beta = 1/kT = 1$, Z is the partition function and $E(\boldsymbol{x}, \boldsymbol{h})$ is the system energy

$$E(\boldsymbol{x}, \boldsymbol{h}) = \sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{j=1}^{H} b_j h_j - \sum_{i,j=1}^{F,H} \frac{x_i w_{ij} h_j}{\sigma_i^2} \tag{D.2}$$

such that

$$F_{\mathrm{RBM}}(\boldsymbol{x}, \boldsymbol{h}) = \exp\left(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\right) \exp\left(\sum_{j=1}^{H} \left(b_j h_j + \sum_{i=1}^{F} \frac{x_i w_{ij}}{\sigma^2}\right)\right). \tag{D.3}$$

We omit the partition function because it will not affect the results (it is just a normalization constant). The probability of a set of positions only is therefore the sum over all sets of $\boldsymbol{h}$, $\{\boldsymbol{h}\}$:

$$\begin{aligned}
F_{\mathrm{RBM}}(\boldsymbol{x}) &= \sum_{\{\boldsymbol{h}\}} \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \exp\Big(b_j h_j + \sum_{i=1}^{F} \frac{x_i w_{ij} h_j}{\sigma^2}\Big) \\
&= \sum_{h_1} \sum_{h_2} \cdots \sum_{h_N} \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \exp\Big(b_1 h_1 + \sum_{i=1}^{F} \frac{x_i w_{i1} h_1}{\sigma^2}\Big) \times \\
&\quad \exp\Big(b_2 h_2 + \sum_{i=1}^{F} \frac{x_i w_{i2} h_2}{\sigma^2}\Big) \ldots \exp\Big(b_H h_H + \sum_{i=1}^{F} \frac{x_i w_{iH} h_H}{\sigma^2}\Big) \\
&= \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \sum_{h_j=0}^{1} \exp\Big(b_j h_j + \sum_{i=1}^{F} \frac{x_i w_{ij} h_j}{\sigma^2}\Big) \\
&= \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \Big[1 + \exp\Big(b_j + \frac{\boldsymbol{x}^T \boldsymbol{w}_{*j}}{\sigma^2}\Big)\Big]
\end{aligned} \tag{D.4}$$

D.2 Find derivatives

A general Gaussian-binary restricted Boltzmann machine has the form

$$\Psi(\boldsymbol{x};\boldsymbol{a},\boldsymbol{b},\boldsymbol{w}) = \exp\left(-\sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma^2}\right)\prod_{j=1}^{H}\Big[1+\exp(f_j(\boldsymbol{x};\boldsymbol{b},\boldsymbol{w}))\Big] \tag{D.5}$$

where $f_j(\boldsymbol{x};\boldsymbol{b},\boldsymbol{w})$ is an arbitrary function of $\boldsymbol{x}$, $\boldsymbol{b}$ and $\boldsymbol{w}$. The Gaussian part is straight-forward to differentiate, so we will keep our attention on the product, which can be treated as a Jastrow factor,

$$J(\boldsymbol{x};\boldsymbol{b},\boldsymbol{w}) = \prod_{j=1}^{H}\Big[1+\exp(f_j(\boldsymbol{x};\boldsymbol{b},\boldsymbol{w}))\Big]. \tag{D.6}$$

Henceforth, we will omit the variable $\boldsymbol{x}$, $\boldsymbol{b}$ and $\boldsymbol{w}$. Introducing

$$p_j \equiv \frac{1}{1+\exp(+f_j)} \quad \wedge \quad n_j \equiv \frac{1}{1+\exp(-f_j)} \tag{D.7}$$

we find the gradient and Laplacian of $\ln\Psi$ to be

$$\nabla_k \ln J = \sum_{j=1}^{H} n_j \nabla_k(f_j) \tag{D.8}$$

and

$$\nabla_k^2 \ln J = \sum_{j=1}^{H} n_j\big[\nabla_k^2(f_j) + p_j\big(\nabla_k(f_j)\big)^2\big] \tag{D.9}$$

respectively. The parameter update can be found by

$$\frac{\partial}{\partial\alpha_i}\ln J = \sum_{j=1}^{H} n_j \frac{\partial}{\partial\alpha_i}(f_j). \tag{D.10}$$

and the ratio between wave functions can be found by

$$\frac{J^{\text{new}}}{J^{\text{old}}} = \prod_{j=1}^{H}\frac{p_j^{\text{old}}}{p_j^{\text{new}}}. \tag{D.11}$$

As a conclusion, what we actually need to calculate to find respective expressions for each wave function is $\nabla_k(f_j)$, $\nabla_k^2(f_j)$ and $\partial_{\alpha_i}(f_j)$. This does also apply for deep restricted Boltzmann machines with Gaussian-binary units.

Bibliography

[1] A.M. Legendre. "Nouvelles méthodes pour la détermination des orbites des comètes". In: (1805).

[2] C.F. Gauss. "Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientum". In: (1809).

[3] P. A. M. Dirac. "A new notation for quantum mechanics". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 35.3 (1939), pp. 416–418. DOI: 10.1017/S0305004100021162.

[4] W. Heisenberg. *Physics and Beyond: Encounters and Conversations*. Harper torchbooks. The Academy library. Allen and Unwin, 1971. URL: https://books.google.no/books?id=0-dEAAAAIAAJ.

[5] J.M. Leinaas and J. Myrheim. "One the theory of identical particles". In: *IL NUOVO CIMENTO* 37.1 (Aug. 1977).

[6] M. Taut. "Two electrons in an external oscillator potential: Particular analytic solutions of a Coulomb correlation problem". In: *Physical Review A* 48.5 (Nov. 1993), pp. 3561–3566. DOI: 10.1103/PhysRevA.48.3561. URL: https://link.aps.org/doi/10.1103/PhysRevA.48.3561 (visited on 03/13/2019).

[7] M. Taut. "Two electrons in a homogeneous magnetic field: particular analytical solutions". en. In: *Journal of Physics A: Mathematical and General* 27.3 (Feb. 1994), pp. 1045–1055. ISSN: 0305-4470. DOI: 10.1088/0305-4470/27/3/040. URL: https://doi.org/10.1088%2F0305-4470%2F27%2F3%2F040 (visited on 04/09/2019).

[8] Attila Szabo and Neil S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. English. Revised ed. edition. Mineola, N.Y: Dover Publications, July 1996. ISBN: 978-0-486-69186-2.

[9] D.J. Griffiths. *Introduction to quantum mechanics*. 2nd Edition. Pearson PH, 2005. ISBN: 0-13-191175-9.

[10] Josef Paldus. *The beginnings of coupled-cluster theory: An eyewitness account*. Amsterdam: Elsevier, 2005. ISBN: 978-0-444-51719-7. DOI: 10.1016/B978-044451719-7/50050-0.

[11] Jamie Trahan, Autar Kaw, and Kevin Martin. "Computational time for finding the inverse of a matrix: LU decomposition vs. naive gaussian elimination". In: *University of South Florida* (2006).

[12] T Daniel Crawford and Henry F. Schaefer III. "An Introduction to Coupled Cluster Theory for Computational Chemists". In: *Rev Comp Chem*. Vol. 14. 2007, pp. 33 –136. ISBN: 978-0-470-12591-5. DOI: 10.1002/9780470125915.ch2.

[13] E.W Weisstein. *Kelvin, Lord William Thomson (1824-1907)*. 2007. URL: http://scienceworld.wolfram.com/biography/Kelvin.html.

[14] D.A. Nissenbaum. "The stochastic gradient approximation: an application to li nanoclusters". In: (2008).

[15] Jørgen Høgberget. "Quantum Monte-Carlo Studies of Generalized Many-body Systems". eng. In: (2013). URL: https://www.duo.uio.no/handle/10852/37167 (visited on 03/22/2019).

[16] A. S. Stodolna et al. "Hydrogen Atoms under Magnification: Direct Observation of the Nodal Structure of Stark States". In: *Phys. Rev. Lett.* 110.21 (May 2013), p. 213001. DOI: 10.1103/PhysRevLett.110.213001. URL: https://link.aps.org/doi/10.1103/PhysRevLett.110.213001.

[17] Francesco Calcavecchia et al. "On the Sign Problem of the Fermionic Shadow Wave Function". In: *Physical Review E* 90.5 (Nov. 2014). arXiv: 1404.6944. ISSN: 1539-3755, 1550-2376. DOI: 10.1103/PhysRevE.90.053304. URL: http://arxiv.org/abs/1404.6944 (visited on 03/13/2019).

[18] Sukanta Deb. "Variational Monte Carlo technique". In: *Resonance* 19.8 (Aug. 2014), pp. 713–739. ISSN: 0973-712X. DOI: 10.1007/s12045-014-0079-x. URL: https://doi.org/10.1007/s12045-014-0079-x.

[19] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Dec. 2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 03/14/2019).

[20] Morten Ledum. *Simple Variational Monte Carlo solve for FYS4411*. 2016. URL: https://github.com/mortele/variational-monte-carlo-fys4411.

[21] Giuseppe Carleo and Matthias Troyer. "Solving the Quantum Many-Body Problem with Artificial Neural Networks". In: *Science* 355.6325 (Feb. 2017). arXiv: 1606.02318, pp. 602–606. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aag2302. URL: http://arxiv.org/abs/1606.02318 (visited on 03/13/2019).

[22] *Naming convention (programming)*. en. Page Version ID: 882668262. Feb. 2019. URL: https://en.wikipedia.org/w/index.php?title=Naming_convention_(programming)&oldid=882668262 (visited on 03/14/2019).

[23] Even Marius Nordhagen. *General variational Monte-Carlo solver written in C++: evenmn/VMC*. original-date: 2019-02-21T19:10:19Z. Mar. 2019. URL: https://github.com/evenmn/VMC (visited on 03/14/2019).

[24] *Qt Creator 4.8.2 released*. en. Mar. 2019. URL: https://blog.qt.io/blog/2019/03/01/qt-creator-4-8-2-released/ (visited on 03/14/2019).

[25] Martin Fowler. *bliki: TwoHardThings*. URL: https://martinfowler.com/bliki/TwoHardThings.html (visited on 03/14/2019).