Electronic structure studies using Machine Learning

by

Even Marius Nordhagen

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

July 29, 2019

Typesetting was done using the LaTeXdocument preparation system.

As long as nothing else is specified, the illustrations are creating using the TikZ package [46].

Plotting is done in Matplotlib [37] and PGFPlots [46].

Effort was made to follow the ISO 80000-2:2009 standard for mathematical signs and symbols [76], and the ISO 80000-9:2009 standard for quantities and units in physical chemistry[77].

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 my excellent high school teacher, Jens Otto Udnes, for getting me hooked on science

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 [5]

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. [57] V.Flugsrud went further and investigated circular quantum dots with the same method.[74] We will extend the work she did to larger quantum dots and double quantum dots.

1.1 The many-body problem

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.[52] Samsung already claim that they use this technology in their displays.[71]

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. [21][20] For that reason, our main focus in this work is two-dimensional dots, but also dots of three dimensions will be investigated.

There are two main problems we need to solve

1. The many-body energy expectation value is analytically infeasible
2. The correct many-body wave function is generally unavailable

1.2 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

1.3 Variational Monte-Carlo

Some great quantum many-body methods have been developed throughout the past century. The Hartree-Fock method is one of the most successful, which sets up a mean field and is thus relatively computational cheap to work with. It works also as an input to so-called post-Hartree-Fock methods, which includes configuration interaction, coupled cluster and quantum Monte-Carlo. The two former will be discussed in the next chapter, together with the Hartree-Fock method itself, and in this chapter we will dig into the quantum Monte-Carlo methods.

VMC typically yields excellent results.

The method has been used in studies of fermionic systems since the 1970's. [48] and

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.

1.4 Machine learning

Branch of artificial intelligence

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. (reproducibility)

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.

When it comes to machine learning, there are dozens of serious ethical aspects. (see introduction_minted.pdf).

1.6 Our goals and milestones

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

1.7 Our developed 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 [53], but our basis set is assumed to be given in Cartesian coordinates. The goal is not to compete with the performance of commercial software, but we will still put in significant effort to make the code fast.

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.

1.8 Structure of the 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

I do not like it, and I am sorry I ever had anything to do with it.

Erwin Schrödinger,[30]

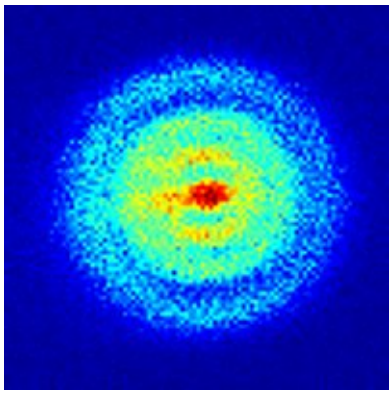


Figure 2.1: The first photograph of a Hydrogen atom was captured by an ultra sensitive camera in 2013. One can actually see the probability distribution $|\Psi(\boldsymbol{r})|^2$ with the naked eye. Published in Phys. rev. lett. 110, 213001 (2013), *Hydrogen atoms under magnification.* [45]

In the early 20th century, some scientists claimed that there were nothing new to be discovered in physics. They though that the deviation from experimental results were caused by uncertainties in the measurements and that the theory was complete. [38] They could not be more wrong.

Just a few years later in 1905, Albert Einstein published a paper advancing Max Planck's hypothesis that the energy of light is carried in discrete quantized packets to explain the photoelectric effect. This was a vital step in the development of quantum mechanics. [1] The years after this discovery, immense efforts were placed on completing the theory, and contributions from an array of scientists over a period of more than 20 years were necessary to complete what today is known as the quantum theory.

Key people include Niels Bohr, who developed a model to explain the spectral lines in atoms, Max Born who suggested the now-standard interpretation of the probability density function, Werner Heisenberg who formulated the matrix mechanics of quantum mechanics, Erwin

[1] The same year, Einstein presented his special theory of relativity, followed by the general theory of relativity in 1915. This theory superseded a 200-year-old theory of mechanics created primarily by Isaac Newton.

Schrödinger who formulated the wave mechanics of quantum mechanics and Paul Dirac who contributed to complete the theory. We will meet them again and again throughout this thesis.

In 1929, Paul Dirac formulated something similar to what the scientists had claimed in the early 20th century, but apparently more correctly. [5]

2.1 Preliminary quantum physics

In this section we will present the fundamentals of the quantum theory, which 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 is also applied on 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. In the quantum dots, this assumption hold even for large systems.
- 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 operator of total energy. 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 in 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

$$V_{\text{int}} = k_e \frac{q^2}{r_{12}} \tag{2.4}$$

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

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

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. For almost[2] all many-electron systems, this becomes analytically intractable due to the two-body interaction term.

As suggested by Max Born, we get the probability distribution function if we take the dot product between the complex conjugate wave function and the wave function itself,

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

so the denominator is basically the integral over all the probabilities. If the wave function is normalized correctly, this should always give 1.

2.1.2 Statistical interpretation

In equation (2.6), we found the expectation value of the energy using the Hamiltonian, which is the energy operator. However, this relation applies for an arbitrary operator $\hat{\mathcal{O}}$ related to an expectation value $\langle\hat{\mathcal{O}}\rangle$,

$$\langle\mathcal{O}\rangle = \frac{\int d\boldsymbol{r}\Psi_n^*(\boldsymbol{r})\hat{\mathcal{O}}\Psi_n(\boldsymbol{r})}{\int d\boldsymbol{r}\Psi_n^*(\boldsymbol{r})\Psi_n(\boldsymbol{r})}. \tag{2.8}$$

As a consequence, there is always an uncertainty associated with a measurement in quantum physics, which means that we can only tell the probability of measuring something. The variance of measurements is given by

$$\sigma^2 = \langle\mathcal{O}^2\rangle - \langle\mathcal{O}\rangle^2 \tag{2.9}$$

and if we take the square root of this we will get the standard deviation. From this, a variety of mathematical inequalities follows, where Heisenberg's uncertainty principle is the most famous. It states that the more precisely the position of some particle is determined, the less precisely its momentum can be known, and is mathematically presented as

$$\sigma_x\sigma_p \geq \frac{\hbar}{2} \tag{2.10}$$

where σ_x is the standard deviation of the position and σ_p is the standard deviation of the momentum. This standard deviation gives the statistical uncertainties.

[2]Exceptions include quantum dots of two electrons in two and three dimensions, where M.Taut have presented semi-analytical energies for a few oscillator frequencies [19, 22].

Before we end this section, we will introduce the Dirac formalism, which is a elegant and efficient way of describing quantum states. For instance, equation (2.8) can, using Dirac formalism, be written as

$$\langle \mathcal{O} \rangle = \frac{\langle \Psi | \hat{\mathcal{O}} | \Psi \rangle}{\langle \Psi | \Psi \rangle} \tag{2.11}$$

where $\langle \Psi |$ is called the *bra* and $| \Psi \rangle$ is called the *ket*. For that reason, the formalism is also called braket formalism.

Often, the inner product in the denominator is assumed to be 1, such that the expectation value can be simplified to

$$\langle \mathcal{O} \rangle = \langle \Psi | \hat{\mathcal{O}} | \Psi \rangle \,, \tag{2.12}$$

which we will henceforth assume. More information about the Dirac formalism can be found in Appendix A.

2.1.3 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 as

$$\epsilon_0 \leq \langle \Psi_T | \hat{\mathcal{H}} | \Psi_T \rangle \,. \tag{2.13}$$

Variational Monte-Carlo is a quantum many-body 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).

2.1.4 Quantum numbers

Unlike in the classical mechanics, all the observable in quantum mechanics are discrete or *quantized*, which means that the n associated with ε_n above cannot take any number. In fact, n can only take positive integers, and is named the principal quantum number.

Principal

The **principal** quantum number describes the electron shell, and can take the numbers $n \in [1, 2, 3, \ldots)$. As n increases, the electron excites to a higher shell such that also the energy increases. In general, $\varepsilon_1 < \varepsilon_2 < \varepsilon_3 \ldots$ as long as all other quantum numbers are fixed. The electron shells can again be split up in subshells, requiring more quantum numbers.

Angular

An electron shell can possibly have more than one subshell, described by the **angular** quantum number l. l can take the values $0, 1, \ldots n-1$, such that the degeneracy of subshells in a shell is simply n. In atoms, the angular quantum number describes the shape of the shell, where $l = 0$ gives a spherical shape, $l = 1$ gives a polar shape while $l = 2$ gives a cloverleaf shape.

Magnetic

We also have a **magnetic** quantum number m_l, which has the range $-l, -l+1, \ldots, l-1, l$. If l describes the shape of a shell, m_l specifies its orientation in space. This quantum number was first observed under presence of a magnetic field, hence the name.

Spin

The **spin** quantum number s gives the spin of a particle, which can just be seen as a particle's property. Particles are often divided in two groups dependent on the spin because of their different behavior: **bosons** have integer spin, while **fermions** have half-integer spin. Electrons and protons have spin $s = 1/2$, which makes them fermions.

Spin projection

The last number we will discuss is the **spin projection** quantum number m_s. It has the range $-s, -s+1, \ldots, s-1, s$, and is therefore related to the spin quantum number in the same way as m_l is related to the angular quantum number. Electrons can for that reason take the values $m_s = +1/2$ or $m_s = -1/2$, such that there are two groups of electrons. The consequences will be discussed in the section (2.2).

2.1.5 The Virial Theorem

The virial theorem relates the kinetic energy to the potential energy, and makes it possible to find the average kinetic energy even for complex systems. The classical statement of the theorem was formulated during the 19th century and named by Rudolf Clausius in 1870, [3], and is in the most general form given by

$$\langle \mathcal{T} \rangle = -\frac{1}{2} \sum_{i=1}^{N} \langle \boldsymbol{F}_i \cdot \boldsymbol{r}_i \rangle \tag{2.14}$$

where $\boldsymbol{F}_k$ represents the force on particle i at position $\boldsymbol{r}_i$. If we further assume that all the sources of potential are on the form $\mathcal{V}_i = ar^{n_i}$, we can use the relation $\boldsymbol{F}_i = -\nabla \mathcal{V}_i$ to express the virial theorem in a simpler fashion

$$2\langle \mathcal{T} \rangle = \sum_i n_i \langle \mathcal{V}_i \rangle. \tag{2.15}$$

Vladimir Fock proved that this holds for quantum mechanics in 1930. [6] The assumption raised above holds for our situation as the interaction energy does as r^{-1}. We will also later see that the potential used in quantum dots goes as r^2 and the potential used in atoms goes as r^{-1} as well.

The expectation value is in principle the time average of the operator. However, if the ergodic hypothesis hold for the system, an ensemble average can also be taken. Ergodicity means that the ensemble average is equal to the time average. [18]

2.1.6 Postulates of Quantum Mechanics

The quantum theory is built on six fundamental postulates, which should always hold. Some of them are already described, but here we present the complete list. Without mentioning them our section on quantum theory 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 $\hat{\mathcal{O}}$, the only values that will ever be observed are the eigenvalues o which satisfy $\hat{\mathcal{O}}\Psi = o\Psi$."*

4. *"The expectation value of the observable corresponding to operator $\hat{\mathcal{O}}$ is given by*

$$\langle \mathcal{O} \rangle = \frac{\int d\tau \Psi^* \hat{\mathcal{O}} \Psi}{\int d\tau \Psi^* \Psi}."$$

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 [29].

Since we will be looking at stationary systems only, the time-independent Schrödinger equation and then postulate no.5 will not be utilized, 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,

$$\hat{P}(i \to j)\Psi_n(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_j, \ldots, \boldsymbol{x}_M) = p\Psi_n(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_j, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_M), \tag{2.16}$$

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. [3]

The particles that have an antisymmetric (AS) wave function under exchange of two coordinates are called fermions, named after Enrico Fermi, and as discussed before they have half-integer spin. On the other hand, the particles that have a symmetric (S) wave function 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.

The probability of finding two identical particles at the same position at the same time should be zero due to the Pauli principle, 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 principle is not satisfied.

2.2.2 Slater determinant

For a system of many 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

$$\Psi_T(\boldsymbol{r}_1, \boldsymbol{r}_2) = \begin{vmatrix} \phi_1(\boldsymbol{r}_1) & \phi_2(\boldsymbol{r}_1) \\ \phi_1(\boldsymbol{r}_2) & \phi_2(\boldsymbol{r}_2) \end{vmatrix} = \phi_1(\boldsymbol{r}_1)\phi_2(\boldsymbol{r}_2) - \phi_2(\boldsymbol{r}_1)\phi_1(\boldsymbol{r}_2), \tag{2.17}$$

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

$$\Psi_T(\boldsymbol{r}) = \begin{vmatrix} \psi_1(\boldsymbol{r}_1) & \psi_2(\boldsymbol{r}_1) & \ldots & \psi_N(\boldsymbol{r}_1) \\ \psi_1(\boldsymbol{r}_2) & \psi_2(\boldsymbol{r}_2) & \ldots & \psi_N(\boldsymbol{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(\boldsymbol{r}_N) & \psi_2(\boldsymbol{r}_N) & \ldots & \psi_N(\boldsymbol{r}_N) \end{vmatrix} \tag{2.18}$$

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

$$\psi = \phi \otimes \xi \tag{2.19}$$

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.

[3] 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. [14]

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.

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 this makes it also possible to factorize out 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,

$$\Psi(\boldsymbol{r}) = |\hat{D}_\uparrow| \cdot |\hat{D}_\downarrow| \tag{2.20}$$

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 [39].

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 (5,6).

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 β_{ij}, and the factor becomes

$$J(\boldsymbol{r};\boldsymbol{\beta}) = \exp\left(\sum_{i=1}^{N}\sum_{j>i}^{N}\beta_{ij}r_{ij}\right). \tag{2.21}$$

All the β_{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

$$J(\boldsymbol{r};\beta) = \exp\left(\sum_{i=1}^{N}\sum_{j>i}^{N}\frac{a_{ij}r_{ij}}{1+\beta r_{ij}}\right). \tag{2.22}$$

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:

$$a_{ij} = \begin{cases} e^2/(D+1) & \text{if } i,j \text{ are particles of same spin} \\ e^2/(D-1) & \text{if } i,j \text{ are particles of opposite spin,} \end{cases} \tag{2.23}$$

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). [44, 63]

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,

$$\rho_i(\boldsymbol{r}) = \int_{-\infty}^{\infty} d\boldsymbol{r}_P \dots d\boldsymbol{r}_N |\Psi(\boldsymbol{r}_1, \dots \boldsymbol{r}_N)|^2. \tag{2.24}$$

where $P < N$.

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.

The one-body density integral can be solved by Monte-Carlo integration. We then divide the space into bins of equal sizes at different radii and count 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 the number of particles in each bin by its volume afterwards in order to get a correct distribution. In two dimensions, the area of bin i is

$$A_i = (2i + 1)\pi d^2 \tag{2.25}$$

and in three dimensions the volume of bin i is

$$V_i = 4(i(i + 1) + 1/3)\pi d^3. \tag{2.26}$$

where d is the radial width of a bin.

2.3.2 Two-body Density

The two-body density gives the probability of finding a pair of electrons at coordinates $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$.

For the one-body density, we integrate over all the particles but one, which corresponds to counting number of particles in each bin when doing Monte-Carlo integration. For the two-body density, we integrate over all particles but a *particle pair*, which means that we need to find the position of each particle pair in order to solve the integral by Monte-Carlo integration. See figure (**??**) for an illustration of this Monte-Carlo integration.

2.3.3 Wigner Crystals

A Wigner crystal is a solid phase where electrons form triangular lattices to minimize the potential energy. The phenomenon occurs only when the potential energy dominates the kinetic energy, since the electrons then are almost "at rest".

To minimize the potential energy, the distances between the electrons should be maximized. To achieve this, the electrons form the triangular lattice shape, not so unlike Norwegians on the metro trying to maximize the distance to anyone else. For that reason, the electron density should approach discrete radial values, and the phenomenon should be observable in one-body density plots.

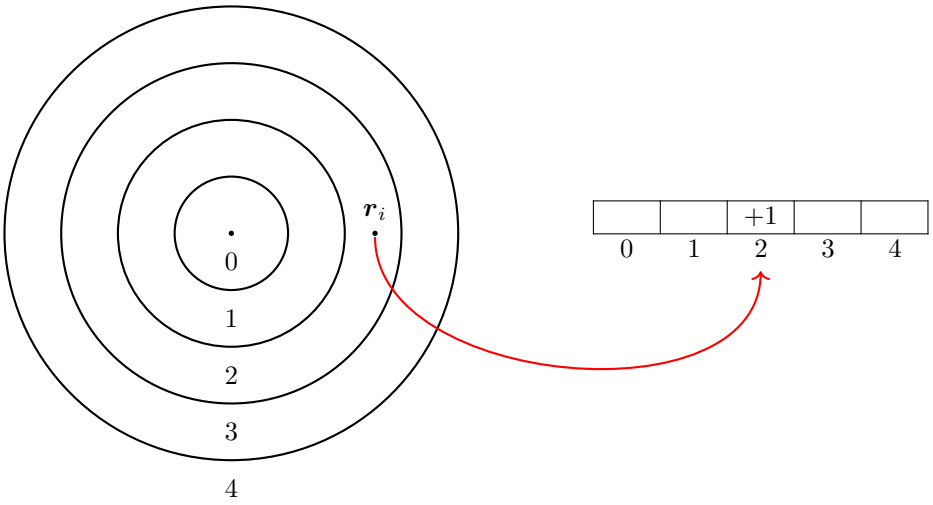


Figure 2.2: This figure is meant to illustrate how the one-body density is calculated using Monte-Carlo integration. One divides the space into n bins (here 5), and count the number of particles in each bin throughout the sampling. Afterwards, the bins need to be normalized.

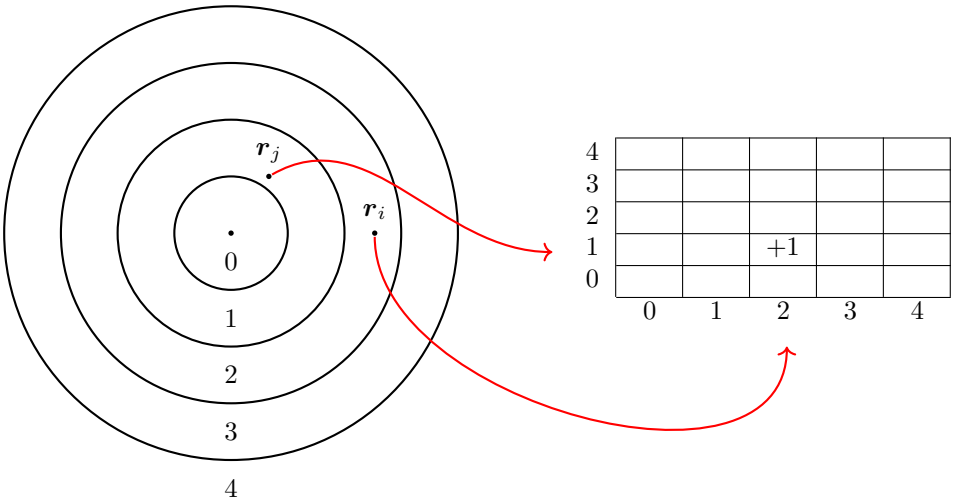


Figure 2.3: This figure is meant to illustrate how the two-body density is calculated using Monte-Carlo integration. One divides the space into n bins (here 5), and store the position of a pair of particles in a matrix throughout the sampling. Afterwards, the bins need to be normalized.

Chapter 3

Systems

> We must be clear that when it comes to atoms, language can be used only as in poetry.
>
> Niels Bohr, [13]

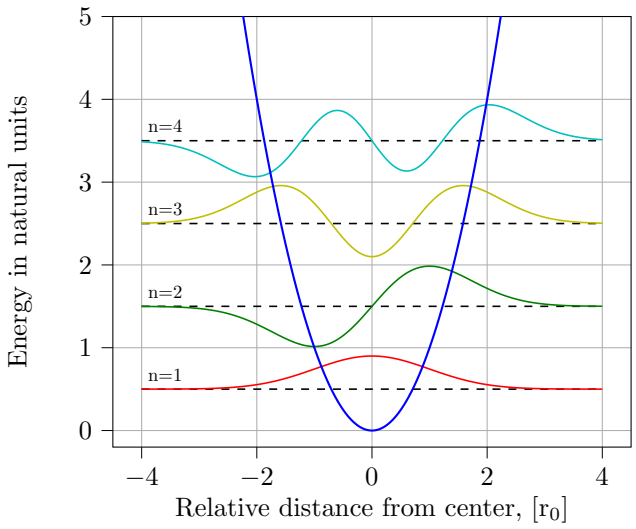


Figure 3.1: The quantum harmonic oscillator, with the Hermite functions represented up to 4th order. As in classical mechanics, the harmonic oscillator can describe various quantum systems, such as lattice vibration (phonons) and quantum fields.

Every system is associated with a Hamiltonian. When this Hamiltonian is found, the next thing we need to do is to find an appropriate basis.

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.

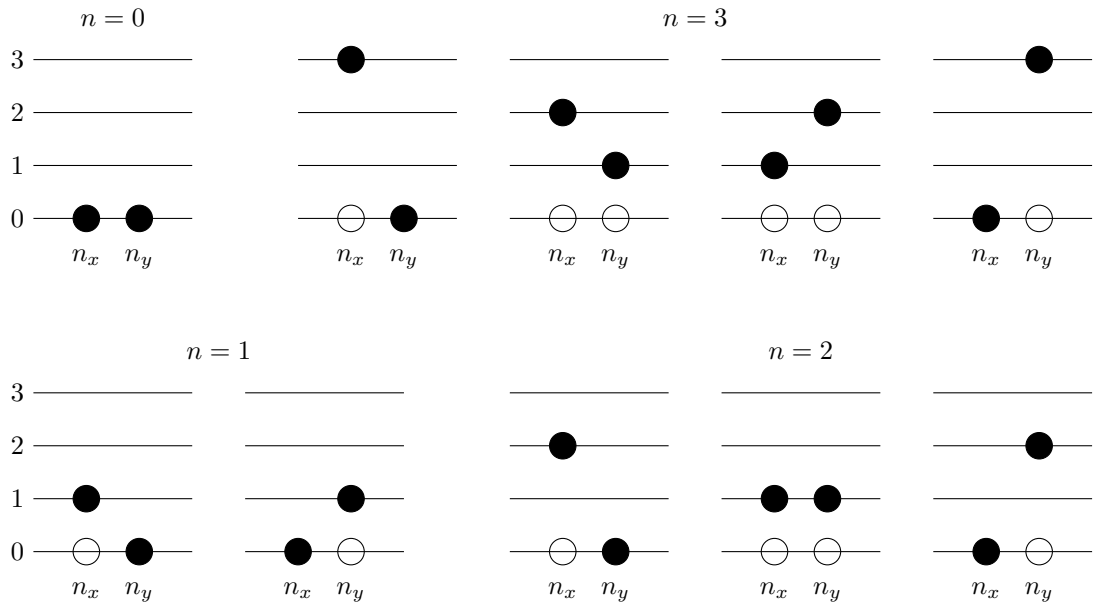


Figure 3.2: The possible states of a two-dimensional quantum dot for $n = n_x + n_y = 0, 1, 2, 3$. Recalling that the electrons can have spin ± 1, one can use this schematic to determine how many electrons there are in each closed shell and thus find the magic numbers.

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.

The potential of symmetrical quantum dots can in principle have a variety of different shapes, but the most used one-dimensional potentials can be derived from

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

with b as the distance between the wells and a as an arbitrary integer. [42] 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 of 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

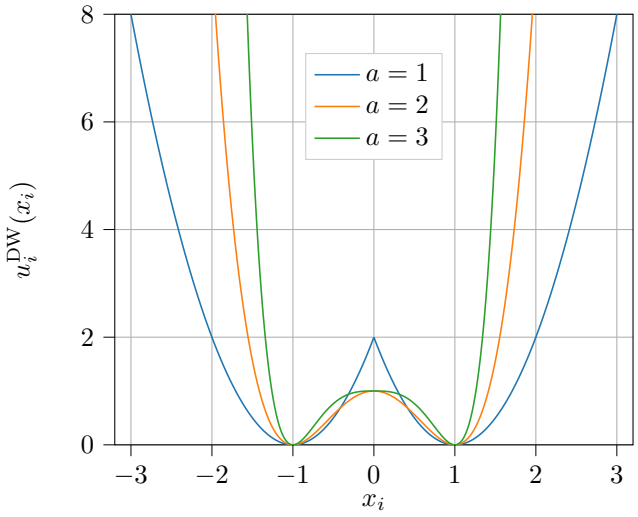


Figure 3.3: Double-well potentials plotted with $a = 1, 2$ and 3, and $b = 2$. For $a = 1$ the potential was multiplied with 2 to make it comparable to the others.

makes the double-dot Hamiltonian similar to the single-dot Hamiltonian,

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

What is remaining is to find an appropriate basis set, and based on the observations above, an expansion of Hermite functions 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

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

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

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

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

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

with $\hat{\mathcal{H}}_+ = (1/2)\omega\sum_{i=1}^{N}\Big((1/4)b^2 - b|x_i|\Big)$ as the extension. The former integrals are just the harmonic oscillator energies, presented in (3.4), while the latter integrals are trivial to calculate. However, since we need to calculate a relatively large number of matrix elements, we decided to do it using numerical integration on the computer.

3.3 Atoms

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 in the nucleus). The total Hamiltonian is given in (Hartree) atomic units,

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

which also is discussed in Appendix B. For the non-interacting case, the energy of a particle in shell n is given by the Bohr formula

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

which means that to find the energy of an atom, we need to summarize the energy og all the electrons.

Also for this system, we need to specify a basis set to use. For atoms where the electrons do not interact, the wave functions are given by the *Hydrogen-like orbitals*, which in spherical coordinates can be split in a radial part and an angular part,

$$\psi_{nlm_l}(r,\theta,\phi) = R_{nl}(r)Y_l^{m_l}(\theta,\phi). \tag{3.18}$$

where n again is the principal quantum number, l is the angular quantum number and m_l is the magnetic quantum number. Henceforth m_l will simply be displayed as m to make the expressions neater. Since we are physicists, we use θ as the polar angle and ϕ as the azimuthal angle.

The radial part can be presented as a function of the *associated Laguerre polynomials* or *generalized Laguerre polynomials*, $L_q^p(x)$, and reads

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

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 can be avoided by introducing the real solid harmonics instead

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

such that

$$\psi_{nlm}(r,\theta,\phi) = R_{nl}(r)S_l^m(\theta,\phi). \tag{3.22}$$

The real solid harmonics do not alter any physical quantities that are degenerate in the subspace consisting of opposite magnetic quantum numbers, and for that reason they will give the same energies as the spherical harmonics as long as there is no magnetic field present. [67]

For atomic systems it is also common to use a Hartree-Fock basis, based on for example Gaussian functions. Even though the Gaussian functions do not have the correct shape, an expansion can be fitted pretty well. [11]

As earlier, we will study closed shells only, but for atoms we will introduce subshells as well, which are dependent on l and m in addition to the principal quantum number n. 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: Degeneracy and naming conversations 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 hereafter*

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

Similar as for the principal quantum number n, we can use the rule of tumble that 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 periodic table, and observe that the first two rows agree 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. It must be something we have overlooked.

The reason is that the angular momentum contribution is not taken into account, i.e., we need to include the Hamiltonian term

$$V_L = \frac{l(l+1)}{2r^2} \tag{3.23}$$

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

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

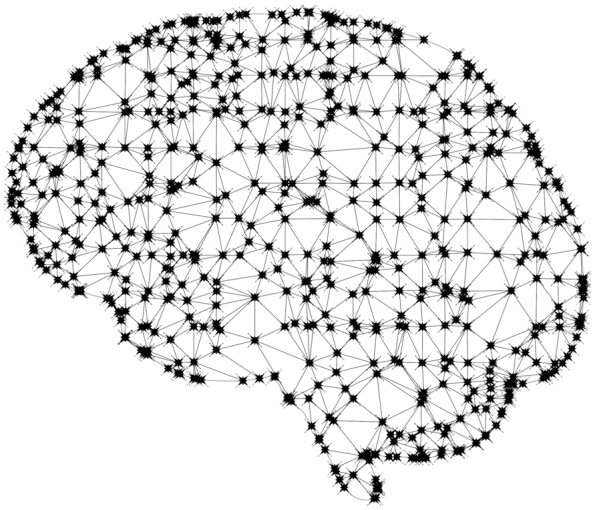


Figure 4.1: Artificial neural networks are inspired by neural networks in the brain.

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. Another important contributor to the booming popularity is the dramatically improvement of a majority of machine learning algorithms. We will come back to this, but before that we will make an attempt defining the term machine learning.

Unlike traditional algorithms, machine learning algorithms are not told what to do explicitly, but they use optimization tools to find patterns in a data set with or without prior knowledge. Based on this, we land at the following definition:

> “Machine learning is the science of getting computers to act without being explicitly programmed.”

As a consequence, we often do not know exactly what the algorithm does and why it behaves as is does. Because of this behavior and the fact that artificial neural networks are inspired by the human brain, 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.

Especially the artificial neural networks have experienced a significant progress over the past decade, which can be attributed to an array of innovations. Most notably, the convolutional neural network (CNN) **AlexNet** managed to increase the top-5 test error rate of image recognition with a remarkable 11.1% compared to the second best back in 2012! [43] Today, the CNNs have been further improved, and they are even able to beat humans in recognizing images! [59] 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** [55], Apple's **Siri** [54] and **Amazon Alexa** [73]. It is also interesting to see how machine learning has made computers eminent tacticians using reinforcement learning. The **Google DeepMind** developed program **AlphaGo** demonstrated this by beating the 9-dan professional L. Sedol in the board game Go [72], before an improved version, **AlphaZero**, beat the at that time highest rated chess computer, **StockFish**, plying chess. [78] Both these scenarios were unbelievable just a couple of decades 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

As hinted above, in machine learning we want to fit a model to a data set in the best possible way. In supervised learning, we have prior knowledge about what kind of results the model should give in some specific cases, which we can use to train our model. After training, we want the model to

1. be able to reproduce the *targets* (the prior known results)
2. be able to fit future observations.

In this section we will take a closer look at how to find a model which satisfies both these goals. If the first one is satisfied, the second is not necessarily satisfied. Let us first look at polynomial regression.

4.1.1 Polynomial regression

Polynomial 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 data set consists

of some n number of x- and y coordinates,

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$$
$$\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$$

which we for instance could try to fit to a second order polynomial,

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

where the parameters a, b and c are our estimators. The polynomial is now our model. By inserting the x-values into the polynomial, we obtain a set of equations

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

where $\tilde{y}_i = f(x_i)$ is the y-value of the polynomial at $x = x_i$. What we want to do is to determine the estimators a, b and c such that the mean squared error (MSE) of all these equations is minimized,

$$\min_{a,b,c} \frac{1}{n} \sum_{i=0}^{n-1} (y_i - f(x_i; a, b, c))^2. \tag{4.3}$$

There are several ways to do this, but they all are based on the *cost function* (also called the loss function), which can simply be defined as the MSE,

$$\mathcal{C}(a, b, c) = \frac{1}{n} \sum_{i=0}^{n-1} \Big(y_i - (ax_i^2 + bx_i + c) \Big)^2, \tag{4.4}$$

and which we seek to minimize. Before we start minimizing this, we will introduce a more general notation, where the estimators are collected in a column vector

$$\boldsymbol{\beta} = (a, b, c)^T$$

and the x_i^j's also are collected in a row vector

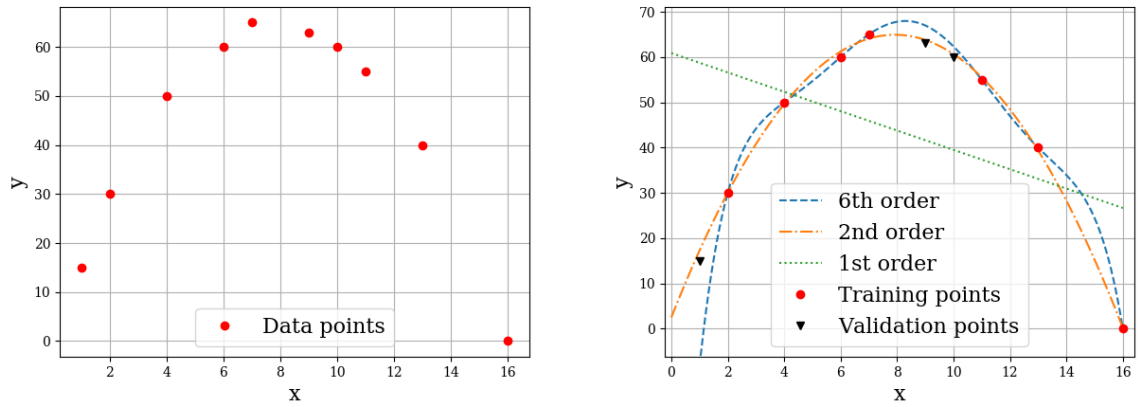
$$\boldsymbol{X}_i = (x_i^2, x_i, 1).$$

Using this, the cost function can be written as

$$\begin{aligned} \mathcal{C}(\boldsymbol{\beta}) &= \frac{1}{n} \sum_{i=0}^{n-1} \Big(y_i - \sum_{j=0}^{2} X_{ij} \beta_j \Big)^2 \\ &= \frac{1}{n} \sum_{i=0}^{n-1} \Big(y_i - \boldsymbol{X}_i \boldsymbol{\beta} \Big)^2 \\ &= \frac{1}{n} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) \end{aligned} \tag{4.5}$$

where we in the last step have collected all the vectors $\boldsymbol{X}_i$ in a matrix. As the minimum of the cost function with respect to a estimator β_j is found when the derivative is zero, we need to solve

$$\begin{aligned} \frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \beta_j} &= \frac{\partial}{\partial \beta_j} \left(\frac{1}{n} \sum_{i=0}^{n-1} \Big(y_i - \sum_{j=0}^{2} X_{ij} \beta_j \Big)^2 \right) \\ &= \frac{2}{n} \sum_{i=0}^{n-1} X_{ij} \Big(y_i - \sum_{j=0}^{2} X_{ij} \beta_j \Big) = 0. \end{aligned}$$



(a) Data set **(b)** Data set with fitted polynomials

Figure 4.2: Figure (a) presents the data points given in (4.7), while the figure (b) illustrates how a first-, second- and third order polynomial can be fitted to the training set in the best possible way.

We can go further and write it on matrix-vector form as

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = \frac{2}{n}\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) = 0$$

where differentiating with respect to a vector here means that each component is $\partial \mathcal{C}(\boldsymbol{\beta})/\partial \beta_j$. This is satisfied if

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

which is the equation we seek to solve to find the best fitting polynomial. Before we proceed to the general case, let us have a quick look at an example.

4.1.1.1 Example

In this example, we will see how we in practice fit a polynomial to a data set. Suppose we have a tiny data set consisting of 10 points on a plane

$$\begin{aligned} \boldsymbol{x} &= (1, 2, 4, 6, 7, 9, 10, 11, 13, 16) \\ \boldsymbol{y} &= (15, 30, 50, 60, 65, 63, 60, 55, 40, 0) \end{aligned} \tag{4.7}$$

to which we want to fit a polynomial of degree p. The data points can be seen in figure (4.2 a). The first thing we need to realize, is that in order to validate our models, we cannot use all points for the training. There is no strict rules on how much of the data set that should be used for training and validation, but at least the training data set should be larger than the validation data set. For this particular problem, we decide to leave out $\{(1, 15), (9, 63), (10, 60)\}$ from the training, which we later will use for validation.

Furthermore, we use equation (4.6) to find the best fitting first-, second- and sixth order polynomials, and obtain the functions presented in table (4.1) with the respective training and prediction errors. The polynomials are also plotted in figure (4.2 b) together with the actual data points.

What we immediately observe, is that the more complex model (higher degree polynomial) the lower training error. In fact, the polynomial of sixth order reproduces the points perfectly. The first order polynomial is quite bad, while the second order polynomial is intermediate.

Table 4.1: Best fitting polynomials of 1st, 2nd and 6th order degree to the data set in equation (4.7). $f(x)$ gives the actual form of the polynomial, the training error gives the MSE to the training data set and the prediction error gives the MSE to the validation set.

Order	$f(x)$	Training error	Prediction error
1st	$-2.14x + 60.87$	327.22	927.87
2nd	$-x^2 + 15.74x + 2.51$	0.47	2.04
6th	$-0.001x^6 + 0.04x^5 - 0.90x^4 + 9.04x^3 - 47.52x^2 + 129.74x - 98.67$	2.54E-11	187.53

However, what really makes sense is the prediction error, and for that we can see that the sixth order polynomial performs terribly. When a model can reproduce the training set very well, but is not able to reproduce the training set, we say that it overfits the data set. This means that the model is too complex for the purpose.

On the other hand, we see that the first order polynomial has also a large prediction error, which means that it is not able to reproduce the validation set either. We say that it is underfitted, and we are in need of a more complex model.

Finally, we have the second order polynomial, which is miles ahead its competitors when it comes to the prediction error. It turns out that the second order model has an appropriate complexity, which we could have guessed just by looking at the data points.

The natural question now is *"How do we find a correct model complexity?"*. The answer is that one should try various complexities and calculate the prediction error for each model. To find the prediction error precisely, the standard is to use K cross-validation resampling, which tries K choices of validation set to make the most use of our data. More about resampling analysis can be found in section (8.2.2). A deeper understanding of the prediction error will hopefully be gained in the next section, on bias-variance decomposition.

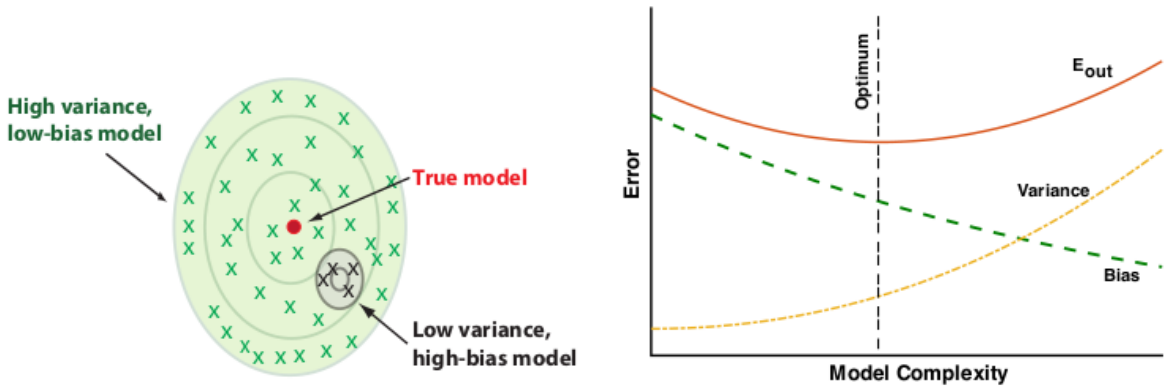
4.1.2 Bias-variance tradeoff

Up to this point, we have skipped some important terms in the statistics behind machine learning. First, we have the *bias*, which describes the best our model could do if we had an infinite amount of training data. We also have the *variance*, which is a measure on the fluctuations in the predictions. In figure (4.3 a), example on high variance low-bias and a low variance high bias models are presented. What we actually want is a low variance low-bias model, but this model is normally infeasible and we need to find the optimal tradeoff between bias and variance. This is known as the bias-variance tradeoff.

In figure (4.3 b), the bias-variance tradeoff is illustrated as a function of the model complexity. We observe that the prediction error if large when the model complexity is too low, which corresponds to a low variance. This substantiates what we discussed in the example in (4.1.1.1), where we claimed that a too low model complexity underfits the data set. Therefore, a too low variance is associated with underfitting.

On the other side of the plot, we can see that also a too complex model causes a large prediction error, which corresponds to a low bias. As discussed before, a too complex model overfits the model, which is associated with low bias.

To minimize the prediction error, we should therefore neither minimize the bias nor the variance. Instead, we should find the bias and variance which corresponds ot the lowest error.



(a) Illustration of bias and variance **(b)** Bias-variance trade-off

Figure 4.3: Examples of high variance, low-bias and low variance high-bias (a) and illustration of the bias-variance trade-off (b). Figures are taken from Mehta et.al., [66].

The prediction error can also be decomposed into bias and variance, given by

$$E[(y-\tilde{y})^2] = \text{bias}[\tilde{y}]^2 + \sigma^2[\tilde{y}] \tag{4.8}$$

where

$$\text{bias}[\tilde{y}] = E[\tilde{y}] - y \tag{4.9}$$

and

$$\sigma^2[\tilde{y}] = E[\tilde{y}^2] - E[\tilde{y}]^2. \tag{4.10}$$

4.1.3 Linear regression

Polynomial regression, as already discussed, is an example on a linear regression method, and was meant as motivation before we study linear regression in general. Instead of fitting a polynomials to a set of points, we can fit a general function on the form

$$f(x_i) = \sum_{j=0}^{p} X_{ij}(x_i)\beta_j \tag{4.11}$$

where we have p estimators β_j. The matrix $\boldsymbol{X}$ is called the *design matrix*, and the case where $X_{ij}(x_i) = x_i^j$ corresponds to polynomial regression, but it can in principle be an arbitrary function of x_i.

The cost function for *the ordinary least square regression* (OLS) case is already found in equation (4.5), and we can recall it as

$$\mathcal{C}(\boldsymbol{\beta}) = \sum_{i=1}^{n}\left(y_i - \sum_{j=0}^{p} X_{ij}\beta_j\right)^2, \qquad \text{OLS} \tag{4.12}$$

which is minimized when

$$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{4.13}$$

To solve this equation, we need to find the inverse of the matrix $\boldsymbol{X}^T\boldsymbol{X}$, which is typically done by *lower-upper* decomposition (LU) or *singular values decomposition* (SVD). Quite often when

we deal with large data sets, the matrix above is singular, which means that the determinant of the matrix is zero. In those cases, we cannot find the inverse, and LU decomposition does not work. Fortunately, SVD *always* works, and in cases where the matrix is singular, it turns out to be a good idea.

4.1.3.1 Singular value decomposition

Singular value decomposition is a method which decomposes a matrix into a product of three matrices, written as

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T. \tag{4.14}$$

This might sounds like a bad idea, but especially for singular matrices this often makes life easier. The reason for this, is that only Σ is singular after the decomposition. For our case, we can thus write the matrix $\boldsymbol{X}^T\boldsymbol{X}$ as

$$\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}\boldsymbol{V}^T = \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^T \tag{4.15}$$

which is non-singular. By multiplying with $\boldsymbol{V}$ on the right-hand-side, we obtain

$$(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{V} = \boldsymbol{V}\boldsymbol{D} \tag{4.16}$$

and similarly

$$(\boldsymbol{X}\boldsymbol{X}^T)\boldsymbol{U} = \boldsymbol{U}\boldsymbol{D} \tag{4.17}$$

when transposing the matrix. Using those expressions, one can show that

$$\boldsymbol{X}\boldsymbol{\beta} = \boldsymbol{U}\boldsymbol{U}^T\boldsymbol{y}. \tag{4.18}$$

4.1.3.2 Ridge regression

So, how can we avoid non-singular values in our matrix $\boldsymbol{X}^T\boldsymbol{X}$? We can remove them by 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. Doing this, all diagonal elements will get a non-zero value and the matrix is guaranteed to be non-singular. Still using the matrix-vector form, this can be written as

$$\boldsymbol{\beta} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{4.19}$$

where $\boldsymbol{I}$ is the identity matrix. The penalty λ is also a *hyper parameter*, which is a parameter that is set before the training begins, in contrast to the estimators which are determined throughout training. This method is called Ridge regression, and has a cost function given by

$$\mathcal{C}(\boldsymbol{\beta}) = \sum_{i=1}^{n}\left(y_i - \sum_{j=0}^{p} X_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j|^2 \qquad \text{Ridge} \tag{4.20}$$

where we in principle just add the L2-norm of the estimator vector to the OLS cost function. This can easiest be proven going from the cost function to the matrix-vector expression of $\boldsymbol{\beta}$, as we did for ordinary least squares.

4.1.3.3 LASSO regression

Finally, we introduce the *least absolute shrinkage and selection operator* (LASSO) regression, which in the same way as Ridge regression is based on a regularization. Instead of adding the L2-norm of the estimator matrix, we add the the L1-norm $|\beta_j|$, and the cost function expresses

$$\mathcal{C}(\boldsymbol{\beta}) = \sum_{i=1}^{n} \Big(y_i - \sum_{j=0}^{p} X_{ij}\beta_j \Big)^2 + \lambda \sum_{j=1}^{p} |\beta_j|. \qquad \text{Lasso} \tag{4.21}$$

For LASSO regression, we cannot set $\partial\mathcal{C}(\boldsymbol{\beta})/\partial\beta_j = 0$ and find a closed-form expression of $\boldsymbol{\beta}$, which means that we need to use an iterative optimization algorithm in order to obtain the optimal estimators. Such optimization methods are essential in non-linear problems such as deep neural networks and variational Monte-Carlo. They will therefore be familiar to the reader throughout this thesis.

In chapter (8), we will present various optimization schemes, but for now on we will stick to one of the most basic methods, *gradient descent*, which can be written as

$$\beta_j^+ = \beta_j - \eta \frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \beta_j}, \tag{4.22}$$

where β_j^+ is the updated version of β_j and $\mathcal{C}(\boldsymbol{\beta})$ is an arbitrary cost function. Here we are introduced to a new hyperparameter, η, known as the *learning rate*, which controls how much the estimators should be changed for each iteration. This should be set carefully, where a too large η will make the cost function diverge and a too small η will make the training too slow. Typically, to choose a $\eta \in 0.01 - 0.0001$ is a good choice. For ordinary least squares, the parameter update can be written as

$$\boldsymbol{\beta}^+ = \boldsymbol{\beta} - \eta \boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{\beta}) \tag{4.23}$$

4.1.4 Logistic regression

Up to this point, we have discussed regression with continuous outputs. But what do we do if we want a discrete output, for example in form of classification? This is what logistic regression is about, and we will now show how the cost function is defined.

Consider a system that can have two possible energies ε_0 and ε_1. From elementary statistical mechanics, we have that the probability of finding the system in the first state is given by

$$P(y_i = 0) = \frac{\exp(-\varepsilon_0/k_BT)}{\exp(-\varepsilon_0/k_BT) + \exp(-\varepsilon_1/k_BT)} \tag{4.24}$$

$$= \frac{1}{1 + \exp(-(\varepsilon_1 - \varepsilon_0)/k_BT)} \tag{4.25}$$

which is the *sigmoid function*, which in the most general form is given by

$$f(x) = \frac{1}{1 + \exp(-x)}. \tag{4.26}$$

The first denominator is known as the *partition function*,

$$Z = \sum_{i=0}^{1} \exp(-\varepsilon_i/k_BT) \tag{4.27}$$

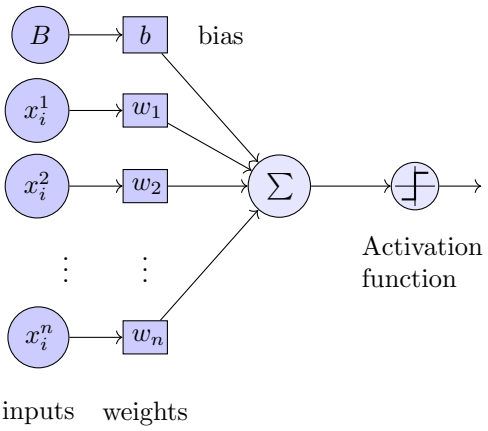


Figure 4.4: Logistic regression model with n inputs. Each input X_i^j is multiplied with a weight w_j, and the contribution from all elements is summarized. The output is obtained after the sum is activated by an activation function.

where k_B is Boltzmann's constant and T is the system temperature. The probability of finding the system in the second state is given by

$$P(y_i = 1) = 1 - P(y_i = 0) \tag{4.28}$$

$$= \frac{1}{1 + \exp(-(\varepsilon_0 - \varepsilon_1)/k_B T)}. \tag{4.29}$$

Notice that the only thing we need is the difference in energy between those two systems, not the energy itself. This is often the case in physics.

If we now assume that the difference in energy can be written as a function of the coordinates that specify the state i, $\boldsymbol{X}_i$ and a matrix of parameters, $\boldsymbol{w}$, known as the *weights*, the difference can be written as

$$\varepsilon_1 - \varepsilon_0 = \boldsymbol{X}_i^T \boldsymbol{w} \equiv \tilde{y}_i, \tag{4.30}$$

which gives the probability

$$P(\boldsymbol{X}_i, y_i | \boldsymbol{w}) = \big(f(\boldsymbol{X}_i^T \boldsymbol{w})\big)^{y_i} \big(1 - f(\boldsymbol{X}_i^T \boldsymbol{w})\big)^{1-y_i}. \tag{4.31}$$

given a set of weights $\boldsymbol{w}$. If we have a set of states $\mathcal{D} = \{(\boldsymbol{X}_i, y_i)\}$, the joint probability is given by

$$P(\mathcal{D}|\boldsymbol{w}) = \prod_{i=1}^{n} \big(f(\boldsymbol{X}_i^T \boldsymbol{w})\big)^{y_i} \big(1 - f(\boldsymbol{X}_i^T \boldsymbol{w})\big)^{1-y_i} \tag{4.32}$$

which is known as the *likelihood*. The *log-likelihood* function is simply the log of the likelihood, and is given by

$$l(\boldsymbol{w}) = \sum_{i=1}^{n} \Big[y_i \log f(\boldsymbol{X}_i^T \boldsymbol{w}) + (1 - y_i) \log\big(1 - f(\boldsymbol{X}_i^T \boldsymbol{w})\big) \Big]. \tag{4.33}$$

As in linear regression, we want to find a cost function which we can minimize in order to fit the model to the data set. Since the log-likelihood function is maximized where the highest probability is, a natural choice is to set

$$\mathcal{C}(\boldsymbol{w}) = -l(\boldsymbol{w}) = -\sum_{i=1}^{n} \Big[y_i \log f(\boldsymbol{X}_i^T \boldsymbol{w}) + (1 - y_i) \log\big(1 - f(\boldsymbol{X}_i^T \boldsymbol{w})\big) \Big], \tag{4.34}$$

which is known as the *cross entropy*. To make things clearer, we will try to illustrate how this works. In figure (4.4), we have a input set $\boldsymbol{X}_i$ where each element is multiplied with a parameter from $\boldsymbol{w}$ and summarized. This corresponds to the inner product $\boldsymbol{X}_i^T\boldsymbol{w}$. Further, the sum (or the inner product) is *activated* by an *activation function*, which we above have assumed to be the sigmoid function. The output is then given by

$$z_i = f(\boldsymbol{X}_i^T\boldsymbol{w}). \tag{4.35}$$

where we have assumed that the bias node is included in the $\boldsymbol{X}$'s and the bias weight is included in the $\boldsymbol{w}$'s.

In addition, a bias node is added, which allows us to shift the activation function to the left or right.

After the activation the output is sent into the cost function to calculate the cost. As before, the cost function is minimized in an iterative scheme, where for example the gradient descent method gives the weight update

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta\boldsymbol{X}^T[\boldsymbol{y} - f(\boldsymbol{X}^T\boldsymbol{w})]. \tag{4.36}$$

which is extremely similar to the estimator update for ordinary least square presented in equation (4.23).

4.1.5 Neural networks

Now we know enough to dive into the field of artificial neural networks. Neural networks can given either continuous or discrete outputs, and is therefore a competitor to both linear and logistic regression. The big strength of neural networks is that one can add multiple *layers*, which potentially makes the model extremely flexible. 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, and is known to give better results. Neural networks of more than one layer are called *deep* networks, and as more layers are added the network gets *deeper*.

In figure (4.5), a two-layer neural network (one hidden layer) is illustrated. It has some similarities with the logistic regression model in figure (4.4), but a hidden layer and multiple outputs are added. We have also dropped the representation of the weights, but each line corresponds to a weight. Each node represents a neuron in the brain.

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 4.1.5.3, where the most popular technique for that is discussed. Before that, we will generalize the forward phase presented in logistic regression.

4.1.5.1 Forward phase

In the previous section, we saw how the output is found for a single perceptron. For a neural network, the net output to the first layer is similar, and given by

$$z_j^{(1)} = \sum_{i=1}^{N_0} x_i w_{ij}^{(1)} = \boldsymbol{x}^T\boldsymbol{w}_j^{(1)}$$

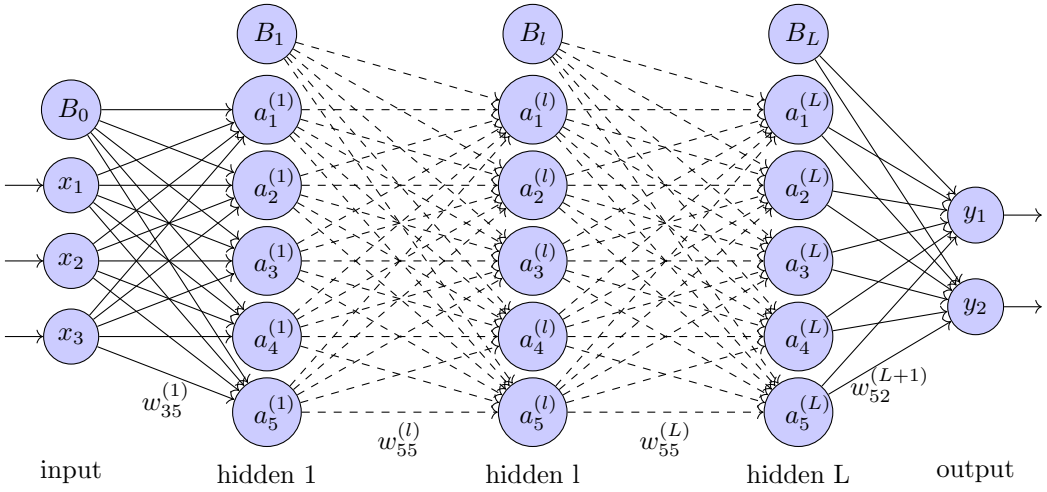


Figure 4.5: Neural network with 3 input nodes, L hidden layers with 5 hidden nodes each and two outputs. B_0, B_1, B_l and B_L are bias nodes for their respective layers, and the dashed lines indicate that it might be more layers between the two layers. We have labeled a few of the lines to relate them to the weights.

where N_0 is the number of nodes in layer 0 (the input layer) and we again have assumed that the bias node is included in $\boldsymbol{x}$ and the bias weight is included in $\boldsymbol{w}$. If we let the activation function, $f(x)$, act on the net output, we get the real output given by

$$a_j^{(1)} = f(z_j^{(1)}) = f\Big(\sum_{i=1}^{N_0} x_i w_{ij}^{(1)}\Big).$$

This is then again the input to the next layer with N_1 nodes, so the output from the second layer is simply

$$a_j^{(2)} = f\Big(\sum_{i=1}^{N_1} a_i^{(1)} w_{ij}^{(2)}\Big).$$

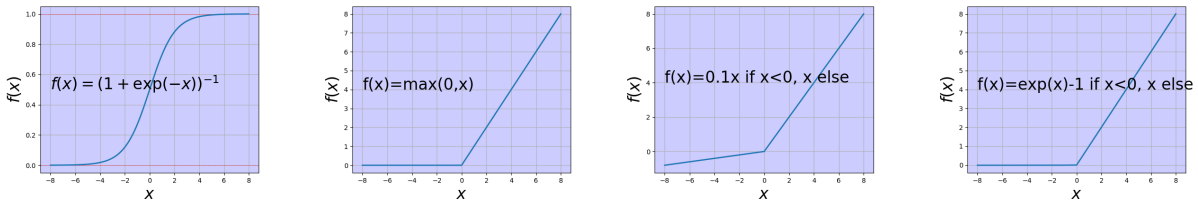
For a neural network of multiple layers, the same procedure applies to all the layers and we can find a general formula for the output at a layer l. The net output to a node $z_j^{(l)}$ in layer l can be found to be

$$z_j^{(l)} = \sum_{i=1}^{N_{l-1}} a_i^{(l-1)} w_{ij}^{(l)} \tag{4.37}$$

where layer $l-1$ has N_{l-1} nodes and we need to be aware that $a_j^{(0)} = x_j$. After activation, the output is obviously found to be

$$a_j^{(l)} = f\Big(\sum_{i=1}^{N_{l-1}} a_i^{(l-1)} w_{ij}^{(l)}\Big) \tag{4.38}$$

which is the only formula needed for the forward phase. The activation function $f(x)$ is not explicitly defined, because there is often expedient to be able to test multiple activation functions.



(a) Sigmoid **(b)** ReLU **(c)** Leaky ReLU (

Figure 4.6: Some well-known activation functions. The sigmoid function stands out from the others since it maps between 0 and 1, and it is not linear for positive numbers.

4.1.5.2 Activation function

Until now, we have mentioned the sigmoid function as the only activation function. However, there are plenty of other activation functions that one can use. In fact, the sigmoid function has lost its popularity, and is today superseded by the more modern functions based on *rectified linear units* (ReLU). Some popular choices are the *leaky* ReLU and *exponential linear units* (ELU), which are linear for positive numbers. The pure linear activation function is still widely used, especially on the output layer.

In figure (4.6), standard RELU, leaky RELU and ELU are plotted along with the sigmoid function.

4.1.5.3 Backward propagation

Backward propagation is the most robust technique for updating the weights in a neural network, and is actually again based on the weight update presented for linear and logistic regression. The algorithm for this was presented 1986, which made the deep neural networks able to solve relatively complicated problems for the first time. [17] To update the weights, one starts with the outputs and updates the weights layer-wise until one gets to the inputs, hence the name backward propagation.

As observed above, a node is dependent on all the nodes in the previous layers, and so are the weights. This means that the nodes are dependent on a large number of parameters, which makes the training scheme quite complex. Nevertheless, there is possible to generalize this to express the updating formulas on a relatively simple form.

From the linear and logistic regression, we know that we need the derivative of the cost function in order to implement the weight update regime. Again, we define the cost function as the mean square error,

$$\mathcal{C}(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_L}(y_i - a_i^{(L)})^2$$

where we have $L+1$ layers (L is the last layer) and N_L output nodes. The derivative of this with respect to one of the weights between the $L-1$'th and L'th layer can be written as a sum using the chain rule

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial w_{jk}^{(L)}} = \frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial a_j^{(L)}}\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}.$$

If we start with the first factor, it can easily be found to be

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial a_j^{(L)}} = (y_j - a_j^{(L)})$$

using the definition of the cost function. The second factor is the derivative of the activation function with respect to its argument, and is for the sigmoid function given by

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = a_j^{(L)}(1 - a_j^{(L)}).$$

Finally, the last factor is found from equation (4.37), and we obtain

$$\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} = a_k^{(L-1)}.$$

Collecting all the factors, the last set of weights can be found by

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial w_{jk}^{(L)}} = (y_j - a_j^{(L)})a_j^{(L)}(1 - a_j^{(L)})a_k^{(L-1)}$$

In the next step, we can define

$$\delta_j^{(L)} = a_j^{(L)}(1 - a_j^{(L)})(y_j - a_j^{(L)}) = f'(a_j^{(L)})\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial a_j^{(L)}} = \frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial z_j^{(L)}}$$

such that the weight update can be expressed on a neater form

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial w_{jk}^{(L)}} = \delta_j^{(L)} a_k^{(L-1)}.$$

For a general layer l, the derivative of the cost function with respect to a weight $w_{jk}^{(l)}$ is similar, and given by

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}. \tag{4.39}$$

Our goal is to find the general relation between layer l and $l+1$, and therefore we use the chain rule and sum over all the net outputs in layer $l+1$,

$$\delta_j^{(l)} = \frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}.$$

We now recognize that the first factor in the sum is just $\delta_k^{(l+1)}$ and the last factor can be found from equation (4.37). We obtain the final expression,

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} f'(z_j^{(l)}) \tag{4.40}$$

where we use the expression of $\delta_j^{(L)}$ as our initial condition. Similar to LASSO regression, a solution of the weight update does not exist in closed form and we need to rely on interative optimization methods. Using gradient descent, a new weight $w_{ij}^{(l)+}$ is found from

$$w_{ij}^{(l)+} = w_{ij}^{(l)} - \eta \frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial w_{jk}^{(l)}}. \tag{4.41}$$

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 weights in supervised learning can be adjusted using the backward propagation algorithm, but it does not work when we do not have prior known targets. Instead, the weights are controlled by a set of probabilities, and we let the cost function be defined by the log likelihood function. This is known as Bayesian statistics, and will be presented in the next section.

4.2.1 Statistical foundation

In this section, we will use Bayesian statistics to exploit the link between some data $\boldsymbol{x}$, called the *hypothesis*, and some other data $\boldsymbol{y}$, called the evidence. We will first do it in a general way, before we link it to machine learning in the next section.

Bayesian statistics appear in many field of science, as it is a basic and often useful probability theory. It is based on Bayes' theorem, which gives rise to some marginal and conditional distributions. The expressions can either be set up in the continuous space or the discrete space, but here we will stick to the latter as we in practice will deal with discrete data.

We start expressing the joint probability distribution of measuring both $\boldsymbol{x}$ and $\boldsymbol{y}$ using the general relation,

$$P(\boldsymbol{x}, \boldsymbol{y}) = P(\boldsymbol{x}|\boldsymbol{y})P(\boldsymbol{y}) = P(\boldsymbol{y}|\boldsymbol{x})P(\boldsymbol{x}), \tag{4.42}$$

which basically states that the probability of observing $\boldsymbol{x}$ *and* $\boldsymbol{y}$ is just the probability of observing $\boldsymbol{x}$ multiplied with the probability of observing $\boldsymbol{y}$ given $\boldsymbol{x}$. $p(\boldsymbol{x}|\boldsymbol{y})$ is the conditional distribution of $\boldsymbol{x}$ and gives the probability of $\boldsymbol{x}$ given that $\boldsymbol{y}$ is true. The opposite applies for $P(\boldsymbol{y}|\boldsymbol{x})$. $P(\boldsymbol{x})$ and $P(\boldsymbol{y})$ are called the marginal probabilities for $\boldsymbol{x}$ and $\boldsymbol{y}$, and by reordering equation (4.42), we obtain Bayes' theorem

$$P(\boldsymbol{x}|\boldsymbol{y}) = \frac{P(\boldsymbol{y}|\boldsymbol{x})P(\boldsymbol{x})}{P(\boldsymbol{y})}. \tag{4.43}$$

The marginal probability of $\boldsymbol{y}$, $P(\boldsymbol{y})$, is given by the sum over all the possible joint probabilities when $\boldsymbol{y}$ is fixed,

$$P(\boldsymbol{y}) = \sum_i P(x_i, \boldsymbol{y}) = \sum_i P(\boldsymbol{y}|x_i)P(x_i), \tag{4.44}$$

and from this we observe that Bayes' theorem gives us the *posterior* probability, $P(\boldsymbol{x}|\boldsymbol{y})$, given the *prior* probability, $P(\boldsymbol{x})$, and the *likelihood*, $P(\boldsymbol{y}|\boldsymbol{x})$, seen from

$$P(\boldsymbol{x}|\boldsymbol{y}) = \frac{P(\boldsymbol{y}|\boldsymbol{x})P(\boldsymbol{x})}{\sum_i P(\boldsymbol{y}|x_i)P(x_i)}. \tag{4.45}$$

However, the summation gets extremely expensive quickly, and is intractable even for small systems. This was a big problem for a long time, but with the advent of powerful computers, algorithms like Markov chain Monte-Carlo can be used to estimate the posterior without knowing the *normalization constant*, $P(\boldsymbol{y})$. More about that in chapter 5.

In the section on supervised learning, the cost function was an important concept, and so is the case in unsupervised learning. But how do we define a cost function when we do not have any targets? We find the answer by revealing the similarities between the logistic regression and the Bayesian statistics. In logistic regression, we find the probability that a system is in a particular

state, and define the cost function as the log-likelihood. We can do the same in unsupervised learning, and define the cost function as

$$\mathcal{C}(\boldsymbol{y}) = \ln \prod_{i=1}^{l} P(\boldsymbol{x}_i|\boldsymbol{y}) = \sum_{i=1}^{l} \ln P(\boldsymbol{x}_i|\boldsymbol{y}) \tag{4.46}$$

which is the log-likelihood. Maximizing the likelihood is the same as maximizing the log-likelihood, which again corresponds to minimizing the distance between the unknown distribution Q underlying $\boldsymbol{x}$ and the distribution P of the Markov random field $\boldsymbol{y}$. This distance is expressed in terms of the Kullback-Leibler divergence (KL divergence), which for a finite state space Ω is given by

$$\mathrm{KL}(Q||P) = \sum_{\boldsymbol{x}\in\Omega} Q(\boldsymbol{x}) \frac{Q(\boldsymbol{x})}{P(\boldsymbol{x})}. \tag{4.47}$$

The KL divergence is a measure of the difference between two *probability density functions* (PDFs), and is zero for two identical PDFs. The divergence is often called a distance, but that is an unsatisfying description as it is non-symmetric ($\mathrm{KL}(Q||P)) \neq \mathrm{KL}(P||Q))$ in general.

To proceed further, we will introduce latent variables in form of hidden nodes. Suppose we want to model an m-dimensional unknown probability distribution Q. Typically, not all the variables $\boldsymbol{s}$ are observed components, they can also be latent variables. If we split $\boldsymbol{s}$ into *visible* variables $\boldsymbol{x}$ and hidden variables $\boldsymbol{h}$, and under the assumption that $\boldsymbol{x}$ and $\boldsymbol{h}$ are variables in an energy function $E(\boldsymbol{x}, \boldsymbol{h})$, we can express the joint probability as the Boltzmann distribution

$$P(\boldsymbol{x}, \boldsymbol{h}) = \frac{\exp(-E(\boldsymbol{x}, \boldsymbol{h}))}{Z} \tag{4.48}$$

where Z is the partition function, which is the sum of the probability of all possible states, which was already introduced in equation (4.27). We have ignored the factor $k_B T$ by setting it to 1. Where the visible nodes correspond to components of an observation, the hidden nodes introduce the system to more degrees of freedom. This allows us to describe complex distributions over the visible variables by means of simple conditional distributions. [49] Those conditional distributions will be described later, but let us first take a look at the marginal distributions.

4.2.1.1 Marginal distributions

We have already used the term marginal distribution, which means that we get rid of a set of variables by integrating the joint probability over all of them. The marginal probability of $\boldsymbol{x}$ is given by

$$P(\boldsymbol{x}) = \sum_{\boldsymbol{h}} P(\boldsymbol{x}, \boldsymbol{h}) = \frac{1}{Z} \sum_{\boldsymbol{h}} \exp(-E(\boldsymbol{x}, \boldsymbol{h})). \tag{4.49}$$

The sum over the $\boldsymbol{h}$ vector is just a short-hand notation where we sum over all the possible values of all the variables in $\boldsymbol{h}$. Further, the marginal probability of $\boldsymbol{h}$ is expressed similarly, with

$$P(\boldsymbol{h}) = \sum_{\boldsymbol{x}} P(\boldsymbol{x}, \boldsymbol{h}) = \frac{1}{Z} \sum_{\boldsymbol{x}} \exp(-E(\boldsymbol{x}, \boldsymbol{h})). \tag{4.50}$$

$p(\boldsymbol{x})$ is important as it gives the probability of a particular set of visible nodes $\boldsymbol{x}$, while $p(\boldsymbol{h})$ will not be used in the same scope in this work.

4.2.1.2 Conditional distributions

The conditional distributions can be found from Bayes' theorem, and read

$$P(\boldsymbol{h}|\boldsymbol{x}) = \frac{P(\boldsymbol{x},\boldsymbol{h})}{P(\boldsymbol{x})} = \frac{\exp(-E(\boldsymbol{x},\boldsymbol{h}))}{\sum_{\boldsymbol{h}}\exp(-E(\boldsymbol{x},\boldsymbol{h}))} \tag{4.51}$$

and

$$P(\boldsymbol{x}|\boldsymbol{h}) = \frac{P(\boldsymbol{x},\boldsymbol{h})}{P(\boldsymbol{h})} = \frac{\exp(-E(\boldsymbol{x},\boldsymbol{h}))}{\sum_{\boldsymbol{x}}\exp(-E(\boldsymbol{x},\boldsymbol{h}))}. \tag{4.52}$$

The conditional probabilities are especially important in Gibbs sampling, where we want to update the $\boldsymbol{x}$'s given a $\boldsymbol{h}$ and vice versa.

4.2.1.3 Maximum log-likelihood estimate

Now suppose that the energy function also is a function of some parameters $\boldsymbol{\theta}$. We have already expressed the log-likelihood function,

$$\ln P(\boldsymbol{x}|\boldsymbol{\theta}) = \ln\left[\frac{1}{Z}\sum_{\boldsymbol{h}}\exp(-E(\boldsymbol{x},\boldsymbol{h}))\right] = \ln\sum_{\boldsymbol{h}}\exp(-E(\boldsymbol{x},\boldsymbol{h})) - \ln\sum_{\boldsymbol{x},\boldsymbol{h}}\exp(-E(\boldsymbol{x},\boldsymbol{h})) \tag{4.53}$$

and by maximizing this we find the maximum log-likelihood estimate. This estimate is important in neural networks since we always seek to maximize the likelihood in the training process. The function is maximized when

$$\begin{aligned}\frac{\partial \ln P(\boldsymbol{x}|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} &= \frac{\partial}{\partial\boldsymbol{\theta}}\left(\ln\sum_{\boldsymbol{h}}\exp(-E(\boldsymbol{x},\boldsymbol{h}))\right) - \frac{\partial}{\partial\boldsymbol{\theta}}\left(\ln\sum_{\boldsymbol{x},\boldsymbol{h}}\exp(-E(\boldsymbol{x},\boldsymbol{h}))\right)\\ &= -\sum_{\boldsymbol{h}}P(\boldsymbol{h}|\boldsymbol{x})\frac{\partial E(\boldsymbol{x},\boldsymbol{h})}{\partial\boldsymbol{\theta}} + \sum_{\boldsymbol{x},\boldsymbol{h}}P(\boldsymbol{x},\boldsymbol{h})\frac{\partial E(\boldsymbol{x},\boldsymbol{h})}{\partial\boldsymbol{\theta}} = 0.\end{aligned} \tag{4.54}$$

Similar to LASSO regression and neural networks, we cannot find a closed-form expression for this, and we need to solve it iteratively.

4.2.2 Boltzmann Machines

Boltzmann Machines are energy-based, generative neural networks based on the more primitive Hopfield network. They were invented in 1985 by Geoffrey Hinton [15], often referred to as "The Godfather of Deep Learning"[1], and the network is named after the Boltzmann distribution.

A Boltzmann machine consists of a set of nodes where a node is connected to all other nodes through weights, similar to the neural network already presented. It is also common to add bias nodes. In figure (4.7), a simple Boltzmann machine consisting of 6 nodes and 1 bias node is illustrated.

By multiplying each node with all the other nodes and the weight connecting them, one obtains the system energy. For the simplest case, the energy reads

$$E(\boldsymbol{s}) = -\sum_{i=1}^{N} s_i b_i - \sum_{i=1}^{N}\sum_{j=i}^{N} s_i w_{ij} s_j \tag{4.55}$$

[1]Hinton's contribution to machine learning can hardly be overstated. He was co-author of the paper popularizing the backpropagation algorithm, [17] supervisor of Alex Krizhevsky who designed AlexNet [43] and the main author of the paper introducing the regularization technique *dropout* [41].

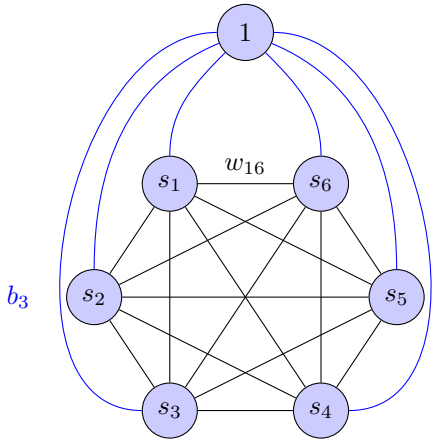


Figure 4.7: Unrestricted Boltzmann machine. Black lines are connections between all the nodes, where for instance the line between s_1 and s_6 is related to the weight w_{16}. The blue lines are related to the bias weights, and, for instance, the line going from the bias node to s_3 is related to b_3.

where $\boldsymbol{s}$ are the nodes and w_{ij} is the weight between node s_i and s_j. The bias node is fixed to 1, as always, and the weight between the bias node and the node s_i is denoted by b_i. In its most simple form, the nodes can only take binary values, and we therefore call it a binary-unit Boltzmann machine. Also other architectures are possible.

The energy formula is identical to the system energy of Hopfield networks, but what distinguish a Boltzmann machine from a Hopfield network is that the nodes are *stochastic*. This means that their values are randomly determined, introducing some randomness to the system. Also the energy of an Ising model takes the same form as equation (4.55)

You might already have foreseen the next step, which is to use the Boltzmann distribution to define the probability of finding the system in a particular state $E(\boldsymbol{s}; \boldsymbol{w}, \boldsymbol{b})$, as discussed in the previous section. The probability distribution function (PDF) is then given by

$$P(\boldsymbol{s}) = \frac{1}{Z} \exp(-E(\boldsymbol{s})), \tag{4.56}$$

where Z again is the partition function. The PDF contains weights, which can be adjusted to change the distribution. In a supervised scheme, one can update the parameters in order to minimize the Kullback-Leibler divergence to a prior known distribution and in that manner reproduce the known distribution. In unsupervised learning, we cannot do this, but we can hope that a reasonable distribution is obtained by minimizing the system energy.

A Boltzmann machine is also a Markov random field, as the stochastic processes satisfy the Markov property. Loosely speaking, this means that all the probabilities of going from one state to another are known, making it possible to predict future of the process based solely on its present state. It is also determined by a "memorylessness", meaning that the next state of the system depends only on the current state and not on the sequence of events that preceded it. [49] Markov chains is an important part of the sampling methods that will be discussed later.

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. Only the units in the first layer are the observable, while the nodes in the next layer are latent or hidden. In the same manner as in equation (4.55), we can look at

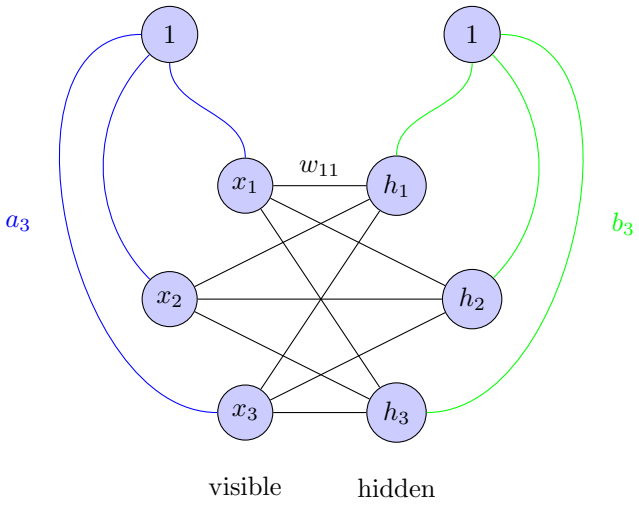


Figure 4.8: Restricted Boltzmann machine. Black lines are the inter-layer connections, where for instance the line between x_1 and h_1 is related to the weight w_{11}. The blue lines are related to the input bias weights, and, for instance, the line going from the bias node to x_3 is called a_3. Similarly, 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.

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 be 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

$$E(\boldsymbol{x}, \boldsymbol{h}) = -\sum_{i=1}^{F} x_i a_i - \sum_{j=1}^{H} h_j b_j - \sum_{i=1}^{F}\sum_{j=i}^{H} x_i w_{ij} h_j \tag{4.57}$$

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 (4.8), a restricted Boltzmann machine with three visible nodes and three hidden nodes is illustrated.

4.2.3.1 Gaussian-binary units

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:

$$E(\boldsymbol{x}, \boldsymbol{h}) = \sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma_i^2} - \sum_{j=1}^{H} h_j b_j - \sum_{i=1}^{F}\sum_{j=i}^{H} \frac{x_i w_{ij} h_j}{\sigma_i^2} \tag{4.58}$$

where σ_i is the width of the Gaussian distribution, which can be set to an arbitrary number. Inserting the energy expression into equation (4.56), we obtain the general expression

$$P(\boldsymbol{x}, \boldsymbol{h}) \propto \exp\Big(-\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \exp\Big(h_j b_j + \sum_{i=1}^{F} \frac{h_j w_{ij} x_i}{\sigma^2}\Big) \tag{4.59}$$

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 distribution of the visible nodes,

$$P(\boldsymbol{x}) \propto \exp\Big(-\sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma^2}\Big)\prod_{j=1}^{H}\Big(1+\exp\Big(b_j+\sum_{i=1}^{F}\frac{w_{ij}x_i}{\sigma^2}\Big)\Big), \tag{4.60}$$

which is shown thoroughly in appendix C. Since the visible nodes take continuous values, we need to integrate to find the marginal distribution of the hidden nodes, but since we never will use that distribution in this work, we will ignore the marginal distribution of the hidden units.

The conditional distributions are important in Gibbs sampling, and read

$$P(\boldsymbol{h}|\boldsymbol{x}) = \frac{P(\boldsymbol{x},\boldsymbol{h})}{P(\boldsymbol{x})} = \prod_{j=1}^{H}\frac{\exp\Big(h_j b_j + \sum_{i=1}^{F} x_i w_{ij} h_j/\sigma^2\Big)}{1+\exp\Big(b_j+\sum_{i=1}^{F} x_i w_{ij}/\sigma^2\Big)} \tag{4.61}$$

and

$$P(\boldsymbol{x}|\boldsymbol{h}) = \mathcal{N}(\boldsymbol{x};\boldsymbol{a}+\boldsymbol{w}^T\boldsymbol{h},\sigma^2) \tag{4.62}$$

where the latter is assumed to be given by a normal distribution in a Bayesian scheme. Note that the mean is $\boldsymbol{\mu} = \boldsymbol{a} + \boldsymbol{w}^T\boldsymbol{h}$, which is the vector obtained when going backwards in the restricted Boltzmann machine (multiplying the hidden nodes with the weights).

In Metropolis sampling, we only use the marginal distribution of the visible nodes use the weights to the hidden nodes are additional variational parameters. For completeness reasons, we will discuss the Gibbs sampling, but we will in practice stick to the Metropolis sampling. More about the different sampling tools can be found in chapter 3.

We also need the gradient of the log-likelihood function in order to train the network. The likelihood function is defined as the probability is $\boldsymbol{x}$ given a set of parameters $\boldsymbol{\theta}$, which relate to our problem as $P(\boldsymbol{x}|\boldsymbol{a},\boldsymbol{b},\boldsymbol{w})$. We therefore get three maximum log-likelihood estimates,

$$\frac{\partial \ln P(\boldsymbol{x}|\boldsymbol{a},\boldsymbol{b},\boldsymbol{w})}{\partial \boldsymbol{a}} = \frac{\boldsymbol{x}-\boldsymbol{a}}{\sigma^2} \tag{4.63}$$

$$\frac{\partial \ln P(\boldsymbol{x}|\boldsymbol{a},\boldsymbol{b},\boldsymbol{w})}{\partial \boldsymbol{b}} = \boldsymbol{n} \tag{4.64}$$

$$\frac{\partial \ln P(\boldsymbol{x}|\boldsymbol{a},\boldsymbol{b},\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\boldsymbol{x}\boldsymbol{n}^T}{\sigma^2} \tag{4.65}$$

which will be used later to maximize the likelihood with respect to the respective set of parameters.

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 (4.9).

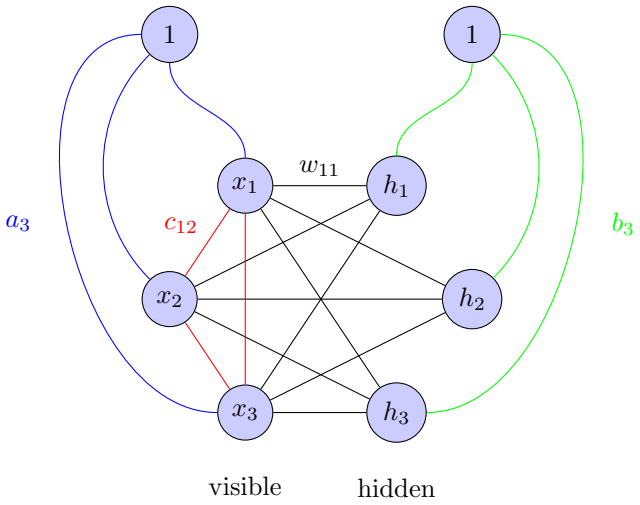


Figure 4.9: Partly restricted Boltzmann machine. Black lines are inter-layer connections, where for instance the line between x_1 and h_1 is related to the weight w_{11}. The blue lines are related to the input bias weights, and, for instance, the line going from the bias node to x_3 is related to a_3. 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

$$E(\boldsymbol{x},\boldsymbol{h}) = \sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma_i^2} - \sum_{i=1}^{F}\sum_{j>i}^{F} x_i c_{ij} x_j - \sum_{j=1}^{H} h_j b_j - \sum_{i=1}^{F}\sum_{j=i}^{H}\frac{x_i w_{ij} h_j}{\sigma_i^2} \tag{4.66}$$

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

$$P(\boldsymbol{x}) = \exp\Big(-\sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma^2} + \sum_{i=1}^{F}\sum_{j>i}^{F} x_i c_{ij} x_j\Big)\prod_{j=1}^{H}\Big(1+\exp\Big(b_j+\sum_{i=1}^{F}\frac{w_{ij}x_i}{\sigma^2}\Big)\Big). \tag{4.67}$$

In chapter 7, we utilize that this marginal distribution can be split in a Gaussian part, a *partly restricted* part and a product part. Then we see that the expression of the gradient of the log-likelihood function becomes the same with respect to $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{w}$ compared to the restricted Boltzmann machine, which means that we only need to calculate the expression of the gradient of the log-likelihood with respect to $\boldsymbol{c}$. This is given by the outer product

$$\frac{\partial \ln P(\boldsymbol{x}|\boldsymbol{a},\boldsymbol{b},\boldsymbol{c},\boldsymbol{w})}{\partial \boldsymbol{c}} = \boldsymbol{x}\boldsymbol{x}^T. \tag{4.68}$$

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 (4.10) 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.

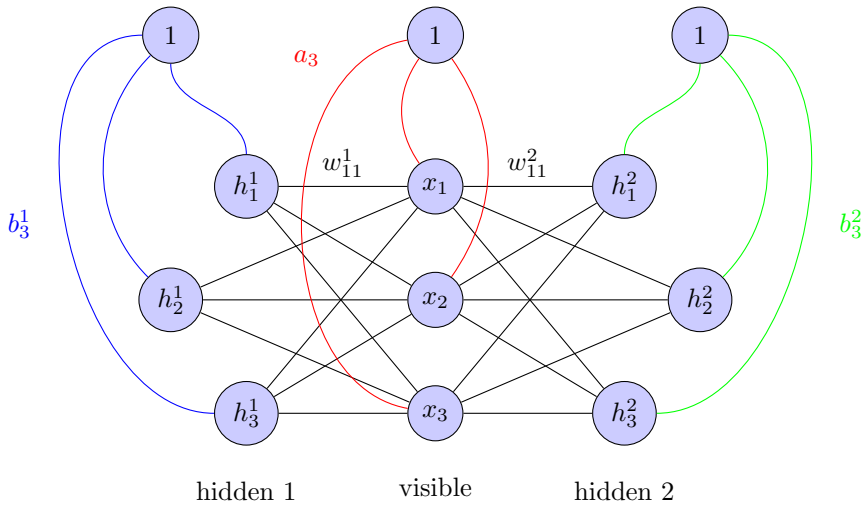


Figure 4.10: Deep restricted Boltzmann machine. Black lines the inter-layer connections, where for instance the line between x_1 and h_1 is related to the weight w_{11}. The blue lines are related to the input bias weights, and, for instance, the line going from the bias node to x_3 is related to a_3. 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

$$E(\boldsymbol{x},\boldsymbol{h}) = \sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma_i^2} - \sum_{l=1}^{L}\sum_{j=1}^{H_L} h_j^l b_j^l - \sum_{l=1}^{L}\sum_{i=1}^{F}\sum_{j=i}^{H_L}\frac{x_i w_{ij}^l h_j^l}{\sigma_i^2} \tag{4.69}$$

where H_L is the number of hidden nodes in layer L. The marginal probability distribution of the visible nodes read

$$P(\boldsymbol{x}) = \exp\Big(-\sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma^2}\Big)\prod_{l=1}^{L}\prod_{j=1}^{H_L}\Big(1+\exp\Big(b_j^l+\sum_{i=1}^{F}\frac{w_{ij}^l x_i}{\sigma^2}\Big)\Big). \tag{4.70}$$

which again can be obtained from the general expressions in appendix C.

Part II

Quantum Many-body Methods

Chapter 5

Quantum Monte-Carlo Methods

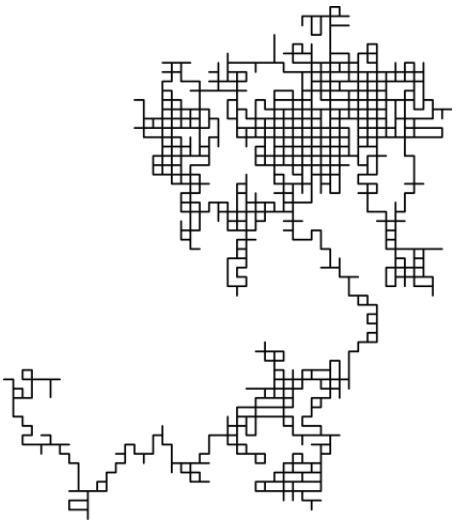


Figure 5.1: Random walk on a two-dimensional grid.
© Copyright wikipedia.org.

Quantum Monte-Carlo methods (QMC) is a bunch of *ab initio* methods that attempt to solve the Schrödinger equation using stochastic Monte-Carlo integration. *Ab initio* reads "from first principles", which implies that the methods constitute a fundamental approach to the problem. They all seek to evaluate the multi-dimensional integral arising from the Schrödinger equation,

$$E_0 = \langle \Psi_0 | \hat{\mathcal{H}} | \Psi_0 \rangle = \frac{\int d\boldsymbol{r} \Psi_0(\boldsymbol{r})^* \hat{\mathcal{H}} \Psi_0(\boldsymbol{r})}{\int d\boldsymbol{r} \Psi_0(\boldsymbol{r})^* \Psi_0(\boldsymbol{r})}, \tag{5.1}$$

which provides the ground state energy expectation value for the exact ground state wave function Ψ_0. As aforementioned, this integral is analytically infeasible for more or less all systems of interest, rising the need of numerical methods like QMC.

In Monte-Carlo integration, we use random numbers to evaluate integrals numerically. Typically, we want to estimate an expectation value $\langle \mathcal{O} \rangle$ by approximating the definition integral with a dense sum,

$$\langle \mathcal{O} \rangle \equiv \int_{-\infty}^{\infty} d\boldsymbol{r} P(\boldsymbol{r}) \hat{\mathcal{O}}(\boldsymbol{r}) \approx \frac{1}{M} \sum_{i=1}^{M} \hat{\mathcal{O}}(\boldsymbol{r}_i) \tag{5.2}$$

where M is the number of *Monte-Carlo cycles* and the coordinates $\boldsymbol{r}_i$ are drawn randomly from the probability density function $P(\boldsymbol{r})$.

A great advantage of the QMC methods, is that we obtain approximative ground state wave functions when solving equation (5.1), which by the fourth postulate of quantum mechanics let us estimate ground state expectation values associated with other operators as well.

Two widely popular QMC methods are the variational Monte-Carlo method (VMC) and the diffusion Monte-Carlo method (DMC), where the former is arguably the simplest of all the QMC methods. It attempts to directly solve the integrals in equation (5.1) by varying parameters, with the support of the variational principle presented in section 2.1.3. This makes VMC a comparably computationally cheap method, but the performance is not in the league of the best methods.

DMC, on the other hand, is computationally expensive, but is also potentially numerically exact, making it a preferred method when high accuracy is needed. At first glance it might seems like a tradeoff where VMC is used when computational time is more important than the accuracy and DMC is used when the opposite is true. However, DMC requires a wave function input which is close to the exact wave function, forcing us to first run VMC to obtain this wave function before the DMC machinery can be started.

As VMC is our main focus in this work, it will be explained thoroughly in this chapter together with common sampling techniques. In the end we will briefly explain the idea behind the DMC method, but since this method is not implemented it will not be our main priority.

5.1 Variational Monte-Carlo

The variational Monte-Carlo method (hereafter VMC) is today widely used when it comes to the study of ground state properties of quantum many-body systems. It makes use of Markov chain Monte-Carlo methods, often abbreviated MCMC, where the particles are assumed to be moving in Markov chains controlled by Monte-Carlo simulations. Going back to the variational principle in equation (2.13), one observes that by choosing an approved wave function, one gets an energy larger or equal to the ground state energy.

Before we present the mathematical framework of the method, we will restate the two big problems in many-body physics, mentioned in the introduction:

1. The many-body energy expectation value is analytically infeasible for most systems
2. The correct many-body wave function is generally unavailable

5.1.1 Approaching the first problem

Let us first address the first problem, which often is considered as the root of all evil in many-body physics. In chapter 2, we saw that the two-body interaction term makes the integral impossible to solve for many particles, such that we need to rely on numerical methods.

In VMC, we start with a trial wave function guess $\Psi_T(\boldsymbol{r};\boldsymbol{\theta})$ where the parameters $\boldsymbol{\theta}$ are varied to minimize the energy. According to the variational principle, the obtained energy will always be higher or equal to the true ground state energy, where the equality is the case if and only if the wave function is the exact ground state wave function. Denoting the exact ground state energy by E_0 and the obtained energy as E, we can summarize this by

$$E_0 \leq E = \frac{\int d\boldsymbol{r}\Psi_T(\boldsymbol{r})^*\hat{\mathcal{H}}\Psi_T(\boldsymbol{r})}{\int d\boldsymbol{r}\Psi_T(\boldsymbol{r})^*\Psi_T(\boldsymbol{r})}. \tag{5.3}$$

Furthermore, the integral can be written on the form of a general expectation value

$$E = \int d\boldsymbol{r}P(\boldsymbol{r})E_L(\boldsymbol{r}) \tag{5.4}$$

defining the local energy as

$$E_L(\boldsymbol{r}) \equiv \frac{1}{\Psi_T(\boldsymbol{r})}\hat{\mathcal{H}}\Psi_T(\boldsymbol{r}). \tag{5.5}$$

$P(\boldsymbol{r})$ is called the probability density function (PDF), was first introduced in equation (2.7) and in a more general scheme the PDF reads

$$P(\boldsymbol{r}) = \frac{|\Psi_T(\boldsymbol{r})|^2}{\int d\boldsymbol{r}|\Psi_T(\boldsymbol{r})|^2}. \tag{5.6}$$

Since the energy expectation value now is on the form of a general expectation value, we can use the approximation set up by Monte-Carlo integration in equation (5.2), yielding

$$E \approx \frac{1}{M}\sum_{i=1}^{M} E_L(\boldsymbol{r}_i). \tag{5.7}$$

We recall that the local energies $E_L(\boldsymbol{r}_i)$ are evaluated with particle configurations $\boldsymbol{r}_i$ drawn from $P(\boldsymbol{r})$. In this manner the obtained energy is guaranteed to approach the exact energy as the number of Monte-Carlo cycles, M, increases. Actually, the standard error goes as $\mathcal{O}(1/\sqrt{M})$, making the method pretty accurate for large numbers of cycles. In the limit $M \to \infty$, the error goes to zero,

$$\langle E_L \rangle = \lim_{M\to\infty} \frac{1}{M}\sum_{i=1}^{M} E_L(\boldsymbol{r}_i), \tag{5.8}$$

indicating the desire of largest number of cycles. This is associated with a zero-variance property governing the VMC method, stating that the variance in the true ground state should be zero. For more statistical details, see [48].

5.1.2 Approaching the second problem

So far so good, but how do we attack the second problem stated above? One of the big strengths of VMC, as aforesaid, is that we obtain an approximative wave function meanwhile the energy expectation value is calculated. If we go back to equation (5.5), we see that a PDF is required in order to estimate the energy, which again is based on a wave function.

In VMC, we need to "guess" an initial trial wave function, $\Psi_T(\boldsymbol{r}, \boldsymbol{\theta})$, which hopefully is close to the exact wave function. The parameters, θ, are then updated in order to minimize $\langle E_L \rangle$, such that the energy gets lower for each *iteration*. If the estimated energy approaches the ground state energy, we also assume that the wave function approaches the ground state wave function. Consequently, we will end up with a more or less accurate wave function approximation when the energy has converged.

As a notice, the term *iterations* should not be confused with the term *cycles*. For each iteration, we run M Monte-Carlo cycles and then update the parameters. As a simple sketch, the algorithm can be written as

Algorithm 1: Simple sketch of the VMC algorithm. The total algorithm is given in algorithm 6 in section 8.3.

Require : $\Psi_T(\boldsymbol{r}, \boldsymbol{\theta})$: Initial trial wave function guess

```
while not converged do
    for i ← 1 to M do
        sample;
    update parameters θ;
```

Result: Estimate of $\langle E_L \rangle$ and $\Psi(\boldsymbol{r}, \boldsymbol{\theta})$.

where the while loop runs over the iterations and the for loop runs over the cycles.

5.1.3 Common extensions

This finalizes the essential theory behind the VMC method. However, a sampling algorithm is needed to draw samples randomly from $P(\boldsymbol{r})$, and in section 5.2 some popular sampling techniques are described. Before we jump into this field, we will discuss some usual extensions and improvements to the VMC method.

Initially, the particle configuration $\boldsymbol{r}$ might not be representative for a configuration in equilibrium, making the first cycles a poor estimate of the expectation value. An easy fix to this problem is to basically ignore the first sampling period, known as the *burn-in period*. With this in mind, we implement an equilibriation fraction describing the fraction of the total number of cycles that is used in the burn-in period. When running multiple parallel processes, the burn-in period should be the same for all the processes.

We also have a technique called *thinning*, which means picking every n'th sample in the chain and ignore the rest. This is shown to give a more or less identical expectation value and makes the program less memory-requiring, but due to worse statistics, this should be avoided as long as there is no lack of memory.

5.2 The Metropolis algorithm

Metropolis sampling is a Markov chain Monte-Carlo method, which generates a Markov chain using a proposal density for new steps, and the method also rejects unsatisfying moves. In its most simple form, a particle is moved in a random direction, which was the original method invented by Metropolis et.al. back in 1953 [10]. Later, the method was improved by Hastings et.al., giving rise to the more general Metropolis-Hastings algorithm [12]. The genius of the Metropolis algorithms, 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 normalizing factor, which is often computational intractable.

We will first discuss Markov chains in general, before we connect them to the original Metropolis algorithm, henceforth called *brute-force sampling*, and then the Metropolis-Hastings algorithm, also called *importance sampling*.

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 opposite way. If we then assume that the rules satisfy *ergodicity* and *detailed balance*, we have the following relationship:

$$P(\boldsymbol{r}'|\boldsymbol{r})P(\boldsymbol{r}) = P(\boldsymbol{r}|\boldsymbol{r}')P(\boldsymbol{r}'), \tag{5.9}$$

which is actually a restatement of Bayes' theorem presented in section 4.2.1.

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

$$P(\boldsymbol{r}'|\boldsymbol{r}) = T(\boldsymbol{r}'|\boldsymbol{r})A(\boldsymbol{r}', \boldsymbol{r}). \tag{5.10}$$

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

$$A(\boldsymbol{r}', \boldsymbol{r}) = \min\left[1, \frac{T(\boldsymbol{r}|\boldsymbol{r}')P(\boldsymbol{r}')}{T(\boldsymbol{r}'|\boldsymbol{r})P(\boldsymbol{r})}\right], \tag{5.11}$$

since A cannot be larger than 1. We want to accept a move with probability $A(\boldsymbol{r}', \boldsymbol{r})$. One way to do that is to draw a number from a uniform distribution between 0 and 1. If this number is lower than the acceptance, the move should be accepted. Otherwise it will be rejected.

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

$$\boldsymbol{r}' = \boldsymbol{r} + \Delta x d\boldsymbol{r} \tag{5.12}$$

where Δx is the step length 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

$$A(\boldsymbol{r}', \boldsymbol{r}) = \min\left[1, \frac{P(\boldsymbol{r}')}{P(\boldsymbol{r})}\right]. \tag{5.13}$$

However, with this approach an unsatisfying number 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,

$$\frac{\partial P(\boldsymbol{r}, t)}{\partial t} = D\nabla\left(\nabla - \boldsymbol{F}\right) P(\boldsymbol{r}, t) \tag{5.14}$$

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 natural units.

The Langevin equation states that a diffusion particle tends to move parallel to the drift force in the coordinate space, with $\boldsymbol{\eta}$ introducing some random noise. The equation reads

$$\frac{\partial \boldsymbol{r}(t)}{\partial t} = D\boldsymbol{F}(\boldsymbol{r}(t)) + \boldsymbol{\eta}. \tag{5.15}$$

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

$$\boldsymbol{r}' = \boldsymbol{r} + D\Delta t\boldsymbol{F}(\boldsymbol{r}) + \boldsymbol{\xi}\sqrt{\Delta t} \tag{5.16}$$

where Δt is a fictive time step and $\boldsymbol{\xi}$ is a Gaussian random variable. Further, we want to find an expression of the drift force $\boldsymbol{F}$ which makes the system converge to a stationary state. A stationary state is found when the probability density function, $P(\boldsymbol{r})$, is constant in time, i.e, when the left-hand-side of Fokker-Planck is zero. In that case, we can rewrite the equation as

$$\nabla^2 P(\boldsymbol{r}) = P(\boldsymbol{r})\nabla\boldsymbol{F}(\boldsymbol{r}) + \boldsymbol{F}(\boldsymbol{r})\nabla P(\boldsymbol{r}). \tag{5.17}$$

Moreover, we assume that the drift force takes the form $\boldsymbol{F}(\boldsymbol{r}) = g(\boldsymbol{r})\nabla P(\boldsymbol{r})$ based on the fact that the force should point to a higher probability. We can then go ahead and write

$$\nabla^2 P(\boldsymbol{r})\big(1 - P(\boldsymbol{r})g(\boldsymbol{r})\big) = \nabla\big(g(\boldsymbol{r})P(\boldsymbol{r})\big)\nabla P(\boldsymbol{r}) \tag{5.18}$$

where the quantity $\nabla^2 P(\boldsymbol{r})$ is factorized out from two of the terms. The equation is satisfied when $g(\boldsymbol{r}) = 1/P(\boldsymbol{r})$, such that the drift force evolves to the well-known form

$$\boldsymbol{F}(\boldsymbol{r}) = \frac{\nabla P(\boldsymbol{r})}{P(\boldsymbol{r})} = 2\frac{\nabla\Psi_T(\boldsymbol{r})}{\Psi_T(\boldsymbol{r})} = 2\nabla \ln \Psi_T(\boldsymbol{r}), \tag{5.19}$$

which is also known as the *quantum force*.

The remaining part concerns 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 (5.11), which are just the solutions of the Fokker-Planck equation. The solutions read

$$G(\boldsymbol{r}', \boldsymbol{r}, \Delta t) \propto \exp\Big(- \big(\boldsymbol{r}' - \boldsymbol{r} - D\Delta t \boldsymbol{F}(\boldsymbol{r})\big)^2 / 4D\Delta t \Big) \tag{5.20}$$

which is categorized as a Green's function. In general, the essential property of any Green's function is that it provides a way to describe the response of an arbitrary differential equation solution. For our particular case, it correspond to the normal distribution centered around $\mu = \boldsymbol{r} + D\Delta t \boldsymbol{F}(\boldsymbol{r})$ with variance $2D\Delta t$, $\mathcal{N}(\boldsymbol{r}'|\boldsymbol{r} + D\Delta t \boldsymbol{F}(\boldsymbol{r}), 2D\Delta t)$. By utilizing this, the acceptance probability for the importance sampling can finally be written as

$$A(\boldsymbol{r}'|\boldsymbol{r}) = \min\ \left[1, \frac{G(\boldsymbol{r}, \boldsymbol{r}', \Delta t) P(\boldsymbol{r}')}{G(\boldsymbol{r}', \boldsymbol{r}, \Delta t) P(\boldsymbol{r})}\right], \tag{5.21}$$

where the marginal probabilities are still given by equation (5.6).

As a summary, we set up the actual algorithm, known as Metropolis-Hasting's algorithm, see algorithm (24). We write the position update in a general form, where we allow updating all the particles at the same time. However, often one ends up moving only a particle at each time step, as this make some advantageous optimization schemes possible. These optimization opportunities will be discussed in chapter 7.

The ratio between the new and old Green's function can be expressed in the elegant form

$$g(\boldsymbol{r}', \boldsymbol{r}, \Delta t) \equiv \frac{G(\boldsymbol{r}', \boldsymbol{r}, \Delta t)}{G(\boldsymbol{r}, \boldsymbol{r}', \Delta t)} = \exp\big((\boldsymbol{r}' - \boldsymbol{r}) \cdot (\boldsymbol{F}(\boldsymbol{r}) - \boldsymbol{F}(\boldsymbol{r}'))/2\big) \tag{5.22}$$

which may be evaluated efficiently using vector operations. Also the probability ratios can be computed in an efficient way for certain trial wave functions $\Psi_T(\boldsymbol{r})$, in particular when the wave function is on an exponential form. We express the probability ratio as

$$p(\boldsymbol{r}', \boldsymbol{r}) \equiv \frac{P(\boldsymbol{r}')}{P(\boldsymbol{r})} = \frac{|\Psi_T(\boldsymbol{r}')|^2}{|\Psi_T(\boldsymbol{r})|^2}, \tag{5.23}$$

and in chapter 7 we find closed-form expressions of all the wave function elements of interest, including the elements based on restricted Boltzmann machines. Additionally, closed-form expressions of $\nabla \ln \Psi_T(\boldsymbol{r})$ are found, which are vital in the quantum force computations as well as the kinetic energy computations.

5.2.3 Gibbs sampling

Gibbs sampling has throughout the years gained high popularity in the machine learning community when it comes to training Boltzmann machines. There are probably many factors that contribute to this, where the performance obviously is one of them. Another relevant motivation is the absent of tuning parameters, which makes the method easier to deal with compared to many of its competitors.

The method is an instance of the Metropolis-Hastings algorithm, and is therefore classified as another Markov chain Monte-Carlo method. It differs from the Metropolis methods discussed above by the fact that all the moves are accepted, such that we do not waste computational time on rejected moves. Nevertheless, we should not use this argument alone to motivate the use of Gibbs sampling, as less than 1% of the moves in importance sampling is usually rejected.

We will in the following briefly describe the mathematical foundation of the method, before we, for the sake of clarity, connect it to the restricted Boltzmann machines. The method is built

Algorithm 2: The Metropolis-Hastings algorithm. The positions are initialized randomly or was chosen by a previous sampling. The parameters are also usually initialized randomly or chosen by a parameter update. The Green's function ratio, g, can be evaluated efficiently using equation (5.22), and the probability ratio p can also often be found in a simple closed-form expression. The diffusion constant is $D = 1/2$ in natural units. $\boldsymbol{\xi}$ is a random Gaussian variable. For more information, see section 5.2.2

Parameter: Δt: Fictive time step
Data : $\boldsymbol{r}'$: Initial particle positions
Require : $\Psi_T(\boldsymbol{r})$: Initial trial wave function guess

$\boldsymbol{F}(\boldsymbol{r}') \leftarrow 2(\nabla\Psi_T(\boldsymbol{r}'))/\Psi_T(\boldsymbol{r}')$ (Initialize the quantum force) ;
$p(\boldsymbol{r}', \boldsymbol{r}) \leftarrow 1$ (Initialize the probability ratio) ;
$g(\boldsymbol{r}', \boldsymbol{r}, \Delta t) \leftarrow 1$ (Initialize Green's function ratio) ;
for $i \leftarrow 1$ **to** M **do**
  $\boldsymbol{r} \leftarrow \boldsymbol{r}'$ (Save current positions $\boldsymbol{r}'$ in a vector $\boldsymbol{r}$) ;
  $\boldsymbol{F}(\boldsymbol{r}) \leftarrow \boldsymbol{F}(\boldsymbol{r}')$ (Save the current quantum force, $\boldsymbol{F}(\boldsymbol{r}')$, in a vector $\boldsymbol{F}(\boldsymbol{r})$) ;
  $p(\boldsymbol{r}, \boldsymbol{r}') \leftarrow p(\boldsymbol{r}', \boldsymbol{r})$ (Save current probability ratio);
  $g(\boldsymbol{r}, \boldsymbol{r}', \Delta t) \leftarrow g(\boldsymbol{r}', \boldsymbol{r}, \Delta t)$ (Save the current Green's function ratio) ;
  $\boldsymbol{r}' \leftarrow \boldsymbol{r} + D\Delta t\boldsymbol{F}(\boldsymbol{r}) + \boldsymbol{\xi}\sqrt{\Delta t}$ (Update position based on the Langevin equation) ;
  $\boldsymbol{F}(\boldsymbol{r}') \leftarrow 2\nabla \ln \Psi_T(\boldsymbol{r}')$ (Update the quantum force) ;
  $p(\boldsymbol{r}', \boldsymbol{r}) = |\Psi_T(\boldsymbol{r}')|^2/|\Psi_T(\boldsymbol{r})|^2$ (Update the probability ratio) ;
  $g(\boldsymbol{r}', \boldsymbol{r}, \Delta t) = G(\boldsymbol{r}', \boldsymbol{r}, \Delta t)/G(\boldsymbol{r}, \boldsymbol{r}', \Delta t)$ (Update the Green's function ratio) ;
  $w \leftarrow p(\boldsymbol{r}', \boldsymbol{r})g(\boldsymbol{r}', \boldsymbol{r}, \Delta t)$ (Calculate acceptance probability) ;
  $q \leftarrow \mathcal{U}(0, 1)$ (Draw a random number between 0 and 1);
  if $w < q$ **then**
    $\boldsymbol{r}' \leftarrow \boldsymbol{r}$ (Reset positions);
    $\boldsymbol{F}(\boldsymbol{r}') \leftarrow \boldsymbol{F}(\boldsymbol{r})$ (Reset the quantum force) ;
    $p(\boldsymbol{r}', \boldsymbol{r}) \leftarrow p(\boldsymbol{r}, \boldsymbol{r}')$ (Reset the probability ratio);
    $g(\boldsymbol{r}', \boldsymbol{r}, \Delta t) \leftarrow g(\boldsymbol{r}, \boldsymbol{r}', \Delta t)$ (Reset the Green's function ratio) ;
  else
    keep going;
  end
end
Result: The optimized trial wave function.
.

on the concept that, given a multivariate distribution, it is simpler to sample from a conditional distribution than to marginalize by integrating over a joint distribution. This is the reason why we do not need the marginal distributions in Gibbs sampling, but rather the conditional distributions. In the most general, Gibbs sampling proposes a rule for going from a sample $\boldsymbol{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)})$ to another sample $\boldsymbol{x}^{(i+1)} = (x_1^{(i+1)}, x_2^{(i+1)}, \dots, x_n^{(i+1)})$, similar to the Metropolis algorithm. However, in Gibbs sampling the transition from $x_j^{(i)}$ to $x_j^{(i+1)}$ is performed according to the conditional distribution specified by

$$P\big(x_j^{(i+1)} | x_1^{(i+1)}, \dots, x_{j-1}^{(i+1)}, x_{j+1}^{(i)}, \dots, x_n^{(i)}\big). \tag{5.24}$$

The marginal distribution of a subset of variables can then be approximated by simply considering the samples for that subset of variables, ignoring the rest.

For a restricted Boltzmann machine, this becomes a two-step sampling process due to the latent variables, such that we use the conditional probabilities $P(\boldsymbol{r}|\boldsymbol{h})$ and $P(\boldsymbol{h}|\boldsymbol{r})$ to update the visible and hidden units respectively. For the restricted Boltzmann machine with Gaussian-binary units presented in section 4.2.3, the conditional probability of $h_j = 0$ given a set of visible nodes $\boldsymbol{x}$ is

$$P(h_j = 0|\boldsymbol{x}) = \frac{1}{1 + \exp\Big(b_j + \sum_{i=1}^{N} x_i w_{ij}/\sigma^2\Big)} \tag{5.25}$$

and the corresponding conditional probability of $h_j = 1$ is

$$P(h_j = 1|\boldsymbol{x}) = \frac{1}{1 + \exp\Big(-b_j - \sum_{i=1}^{N} x_i w_{ij}/\sigma^2\Big)}, \tag{5.26}$$

which is by the way the sigmoid function again. Note that $P(h_j = 0|\boldsymbol{x}) + P(h_j = 1|\boldsymbol{x}) = 1$, indicating that a hidden node h_j can only take 0 or 1, hence binary. When updating the hidden node, one typically calculate the sigmoid $P(h_j = 1|\boldsymbol{x})$ and set h_j to 1 if this probability is larger than $1/2$, and set it to 0 otherwise.

For the update of the visible nodes, we see from equation (4.62) that the visible nodes are chosen after the normal distribution,

$$P(x_i|\boldsymbol{h}) = \mathcal{N}\Big(a_i + \sum_{j=1}^{H} w_{ij} h_j, \sigma^2\Big) \tag{5.27}$$

which introduce some stochasticity to the system. By going back and forth in the Boltzmann machine multiple times (a round trip corresponds to an iteration), the hope is that the expectation values can be approximated by averaging over all the iterations.

As pointed out earlier, the Gibbs sampling will not be implemented in this work, but is described for completeness purposes. The reason for this, is that the method has not shown promising results for our specific problem, and we will instead rely on Metropolis sampling. We have already tested the Gibbs sampling in another similar project on small quantum dots [64], and so have others like Vilde Flugsrud [74]. The results are matching and show poor performance compared to the Metropolis-Hastings algorithm.

However, the Gibbs sampling method itself should not be underestimated. Carleo and Troyer showed its importance when solving the Ising model using a restricted Boltzmann machine and Gibbs sampling, and in traditional Boltzmann machines the Gibbs sampling is the preferred tool [57].

5.3 Similarities between restricted Boltzmann machines and variational Monte-Carlo

In chapter 3, we presented the relation between a quantum system and expressions of the Hamiltonian, which is the energy operator. In section 4.2, we described how Boltzmann machines are defined and how one can find the energy of a Boltzmann machine. Further, we looked at the updating schemes, which for both cases corresponds to minimizing the cost function and maximizing the log-likelihood function.

All in all, there are many similarities between the two methods. They are fundamentally related through Markov random fields, ...

5.4 Diffusion Monte-Carlo

Diffusion Monte-Carlo belongs to a class of projection and Green's function approaches. Consider the time-imaginary Schrödinger equation

$$-\frac{\partial \Psi(\boldsymbol{r},\tau)}{\partial \tau} = (\mathcal{H} - E_T)\Psi(\boldsymbol{r},\tau) \tag{5.28}$$

where τ is the imaginary time and E_T is an energy offset. If we first look at the equation with non-interacting particles, it contains basically only the kinetic energy term,

$$\frac{\partial \Psi(\boldsymbol{r},\tau)}{\partial \tau} = \frac{1}{2}\nabla^2\Psi(\boldsymbol{r},\tau) \tag{5.29}$$

which is really similar to the diffusion equation,

$$\frac{\partial \phi(\boldsymbol{r},t)}{\partial t} = D\nabla^2\phi(\boldsymbol{r},t) \tag{5.30}$$

where D is the diffusion constant set to $1/2$ in our calculations, no wonder why. The similarities between the imaginary time Schrödinger and the diffusion equation was noticed by Fermi already around 1945, and is what diffusion Monte-Carlo is based on [9, 16].

A solution of the

Chapter 6

Other Quantum Many-body Methods

> Hartree-Fock plays the same role in many-body quantum physics as a basic kitchen tool does in cooking; it can be used to prepare almost everything.
>
> Morten Hjorth-Jensen

QMC methods, as presented in the previous chapter, is just a fraction of the existing many-body methods, and we will in this chapter discuss some other popular methods. Again we start from the integral representation of the Schrödinger equation, but for these methods it is a clear advantage to use Dirac notation, presented in Appendix A. As noted in the theory, the energy eigenvalues are found from

$$E = \langle \Psi | \hat{\mathcal{H}} | \Psi \rangle \tag{6.1}$$

Hartree-Fock (HF) is (arguably) the most successive many-body method, as it provides relatively accurate results with an affordable computational cost. Just a few methods give more accurate results themselves (included QMC methods), but they are in return naturally more expensive. Additionally, HF is often used as an input to other methods, which then are categorized as *post Hartree-Fock methods*. Popular post Hartree-Fock methods cover the full configuration interaction method (FCI) and the coupled cluster method (CC), which both will be explained in this chapter. Also QMC methods commonly use the HF basis as inputs.

The methods HF, FCI and CC have all a fundamentally different approach to solving the Schrödinger equation compared to the QMC methods in the sense that they express the integral as expansions and require the one-body and two-body elements

$$\langle \phi_i | \hat{h} | \phi_i \rangle \quad \wedge \quad \langle \phi_i \phi_j | \hat{w} | \phi_i \phi_j \rangle \tag{6.2}$$

for a given basis $\{\phi_1, \phi_2, \ldots, \phi_n\}$ in advance. $\hat{h}$ is the one-particle Hamiltonian of a particle i, and $\hat{w}$ is the two-particle Hamiltonian of a couple of particles i, j. For commonly used basis sets, these matrix elements are freely available from numerous software programs.

The FCI method is by some considered as the most natural way of solving the Schrödinger equation, as it is a linear combination of Slater determinants yielding exact results for an infinity basis. Although this is an elegant method, it quickly becomes infeasible since it scales as $\mathcal{O}(N!)$.

On the other hand, the CC method truncates the Slater determinants using an exponential expansion, which for an infinity basis also yields exact result. Because of the exponential treatment, the method becomes more efficient than its brother, FCI, scaling as $\mathcal{O}(N^6)$.

We have implemented the FCI and the coupled cluster doubles (CCD) algorithms, and will therefore detail the methods in the following. However, since the HF algorithm is so important,

it will first be discussed thoroughly. It is implemented as well, though in its simplest form. All those methods will be applied on the atoms only as references for our VMC and RBM results.

6.1 The Hartree-Fock method

What distinguish the Hartree-Fock method from the aforementioned FCI and CC methods, is that it approximates the wave function with only *one* Slater determinant. This is naturally a poorer approximations than when multiple Slater determinants are used, but on the bright side the method becomes comparably computationally cheap. The method was long called a *self-consistent field method* (SCF) because electron-electron interactions are baked into the external field. However, this is not the only approach to the HF method, and it is therefore more appropriate to name the method after its inventors, Douglas Hartree and Vladimir Fock [4, 7].

The idea behind the Hartree-Fock method is to include the interaction in the one-body Hamiltonian as a mean field, such that we do not need to calculate the electron-electron interaction repetitively. By introducing the Hartree-Fock operator as

$$\hat{h}^{\mathrm{HF}} = \hat{t} + \hat{u}_{\mathrm{ext}} + \hat{u}^{\mathrm{HF}}. \tag{6.3}$$

with $\hat{u}^{\mathrm{HF}}$ being a single-particle potential to be determined by the HF algorithm, we can rewrite the Schrödinger equation as

$$\hat{h}^{\mathrm{HF}} \left|\psi_\alpha\right\rangle = \varepsilon_\alpha \left|\psi_\alpha\right\rangle \tag{6.4}$$

where ε_α is the Hartree-Fock energy and ψ_α is the single Slater determinant that provides the lowest energy.

6.1.1 The Hartree-Fock energy

The starting point of the method is that the Hamiltonian can be split in an one-body term, $\hat{\mathcal{H}}_0$, and an interaction term, $\hat{\mathcal{H}}_{\mathrm{I}}$, such that the energy can be expressed as

$$E = \langle\Psi|\hat{\mathcal{H}}_0|\Psi\rangle + \langle\Psi|\hat{\mathcal{H}}_{\mathrm{I}}|\Psi\rangle\,. \tag{6.5}$$

The one-body Hamiltonian is given by a sum over all the N single-particle one-body Hamiltonians $\hat{h}_i$,

$$\hat{\mathcal{H}}_0 = \sum_{i=1}^{N} \hat{h}_i = \sum_{i=1}^{N} \Big(\hat{t}_i + \hat{u}_i\Big) = \sum_{i=1}^{N} \Big(-\frac{1}{2}\nabla_i^2 + u_i\Big) \tag{6.6}$$

as described in the preliminary quantum theory, section 2.1. $\hat{t}_i$ is the kinetic energy operator of particle i and $\hat{u}_i$ is the corresponding external potential operator. The one-body energy contribution is therefore commonly written as

$$\langle\Psi|\hat{\mathcal{H}}_0|\Psi\rangle = \sum_{i=1}^{N} \langle\phi_i|\hat{h}|\phi_i\rangle \equiv \sum_{i=1}^{N} \langle i|\hat{h}|i\rangle \tag{6.7}$$

where we have utilized that we have one term for each i and in the last expression have defined $|i\rangle \equiv |\phi_i\rangle$.

Further, the interaction term reads

$$\hat{\mathcal{H}}_{\mathrm{I}} = \sum_{i=1}^{N}\sum_{j>i}^{N} \hat{w}_{ij} = \sum_{i=1}^{N}\sum_{j>i}^{N} \frac{1}{|\boldsymbol{r}_i - \boldsymbol{r}_j|}. \tag{6.8}$$

In order to fulfill the Pauli principle, the wave function needs to be anti-symmetric under exchange of two coordinates, and the iteraction term therefore needs to be

$$\langle\Psi|\hat{\mathcal{H}}_{\rm I}|\Psi\rangle = \sum_{i=1}^{N}\sum_{j>i}^{N}\Big(\langle\phi_i\phi_j|\hat{w}|\phi_i\phi_j\rangle - \langle\phi_i\phi_j|\hat{w}|\phi_j\phi_i\rangle\Big) \equiv \sum_{i=1}^{N}\sum_{j>i}^{N}\langle ij|\hat{w}|ij\rangle_{\rm AS} \tag{6.9}$$

where we have used the short-hand notation $|ij\rangle \equiv |\phi_i\phi_j\rangle$ and the subscript, AS, implies that the element is anti-symmetric with the properties

$$\begin{aligned}\langle ij|\hat{w}|ij\rangle_{\rm AS} &= -\langle ij|\hat{w}|ji\rangle_{\rm AS}\\ &= +\langle ji|\hat{w}|ji\rangle_{\rm AS}\\ &= -\langle ji|\hat{w}|ij\rangle_{\rm AS}\,.\end{aligned} \tag{6.10}$$

These relations also applies for a general matrix element $\langle ij|\hat{w}|kl\rangle_{\rm AS}$, which is also Hermitian such that $\langle ij|\hat{w}|kl\rangle_{\rm AS} = \langle kl|\hat{w}|ij\rangle_{\rm AS}$.

The total energy expression is thus given by

$$E = \sum_{i=1}^{N}\langle i|\hat{h}|i\rangle + \sum_{i=1}^{N}\sum_{j>i}^{N}\langle ij|\hat{w}|ij\rangle_{\rm AS} \tag{6.11}$$

which we name the Hartree-Fock energy and which in the following will be minimized by varying the spin-orbitals ϕ_k.

6.1.2 Minimization of the Hartree-Fock energy

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

$$\hat{h}^{\rm HF} = \hat{t} + \hat{u}_{\rm ext} + \hat{u}^{\rm HF}. \tag{6.12}$$

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

$$\hat{h}^{\rm HF}|p\rangle = \epsilon_\alpha|p\rangle \tag{6.13}$$

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

$$|p\rangle = \sum_{\lambda} C_{p\lambda}|\lambda\rangle\,, \tag{6.14}$$

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.

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,

$$|p\rangle = \sum_{\alpha} c_{p\alpha}|\alpha\rangle\,, \tag{6.15}$$

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

$$E = \sum_{p}^{N}\sum_{\alpha\beta} C^*_{p\alpha}C_{p\beta}\langle\alpha|\hat{h}_0|\beta\rangle + \frac{1}{2}\sum_{pq}^{N}\sum_{\alpha\beta\gamma\delta} C^*_{p\alpha}C^*_{q\beta}C_{p\gamma}C_{q\delta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle_{\rm AS}\,. \tag{6.16}$$

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

$$\langle p|q\rangle = \sum_{\alpha} c^*_{p\alpha}c_{q\alpha}\langle\alpha|\alpha\rangle = \sum_{\alpha} c^*_{p\alpha}c_{q\alpha} = \delta_{pq} \tag{6.17}$$

$$\Rightarrow \sum_{\alpha} c^*_{p\alpha}c_{q\alpha} - \delta_{pq} = 0 \tag{6.18}$$

We now have a function, E, that we want to minimize with respect to a constraint given in equation (6.18). This is a typical situation where Lagrange Multipliers is convenient to use, which in this case can be written as

$$\mathcal{L}(\{C_{p\alpha}\}) = E(\{C_{p\alpha}\}) - \sum_a \varepsilon_a\Big(\sum_{\alpha} c^*_{p\alpha}c_{q\alpha} - \delta_{pq}\Big). \tag{6.19}$$

The variation in reference energy is then find to be

$$\delta E = \sum_{k\alpha}\frac{\partial E}{\partial C^*_{k\alpha}}\delta C^*_{k\alpha} + \sum_{k\alpha}\frac{\partial E}{\partial C_{k\alpha}}\delta C_{k\alpha} - \sum_{k\alpha}\varepsilon_k(C_{k\alpha}\delta C^*_{k\alpha} + C^*_{k\alpha}\delta C_{k\alpha}) \tag{6.20}$$

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

$$\left(\frac{\partial E}{\partial C^*_{k\alpha}} - \varepsilon_k C_{k\alpha}\right)\delta C^*_{k\alpha} = 0, \tag{6.21}$$

which is satisfied if and only if

$$\frac{\partial E}{\partial C^*_{k\alpha}} - \varepsilon_k C_{k\alpha} = 0 \qquad \forall\ k,\alpha \tag{6.22}$$

The first term can be derived from (6.20), and reads

$$\frac{\partial E}{\partial C^*_{k\alpha}} = \sum_{\beta} C_{k\beta}\langle\alpha|\hat{h}_0|\beta\rangle + \sum_{p}^{N}\sum_{\beta\gamma\delta} C^*_{p\beta}C_{k\gamma}C_{p\delta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle_{\mathrm{AS}}\,. \tag{6.23}$$

This results in the equation

$$\sum_{\gamma}\hat{h}^{\mathrm{HF}}_{\alpha\gamma}C_{k\gamma} = \varepsilon_k C_{k\gamma} \tag{6.24}$$

where we have defined

$$\hat{h}^{\mathrm{HF}}_{\alpha\gamma} \equiv \langle\alpha|\hat{h}_0|\gamma\rangle + \sum_{p}^{N}\sum_{\beta\delta} C^*_{p\beta}C_{p\delta}\langle\alpha\beta|\hat{v}|\gamma\delta\rangle_{\mathrm{AS}}\,. \tag{6.25}$$

We recognize that (6.24) can be written as a matrix-vector product

$$\hat{h}^{\mathrm{HF}}C_k = \varepsilon^{\mathrm{HF}}_k C_k \tag{6.26}$$

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 (6.20).

$$\hat{h}^{\mathrm{HF}}(C^{i+1}_k)C^i_k = \varepsilon^{\mathrm{HF}}_k C^i_k \tag{6.27}$$

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

6.1.3 Restricted Hartree-Fock

6.1.4 Unrestricted Hartree-Fock

6.1.5 The Hartree-Fock limit

The Hartree-Fock limit is the energy provided by the Hartree-Fock algorithm for an infinity basis set. Doing calculations with an infinity data set is obviously not possible, but we can approach the Hartree-Fock limit by using a large basis set, and there are also methods for approximating the Hartree-Fock limit even more accurate, for instance by using interpolation.

6.2 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

$$\hat{H}_0 |\Phi_i\rangle = \varepsilon_i |\Phi_i\rangle , \qquad (\hat{H}_0 + \hat{H}_I) |\Phi_i\rangle \neq \varepsilon_i |\Phi_i\rangle . \tag{6.28}$$

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

$$\begin{array}{ccccccccc}
|\Psi_0\rangle & = & C_0^{(0)} |\Phi_0\rangle & + & C_1^{(0)} |\Phi_1\rangle & + & \dots & + & C_{N-1}^{(0)} |\Phi_{N-1}\rangle \\
|\Psi_1\rangle & = & C_0^{(1)} |\Phi_0\rangle & + & C_1^{(1)} |\Phi_1\rangle & + & \dots & + & C_{N-1}^{(1)} |\Phi_{N-1}\rangle \\
|\Psi_2\rangle & = & C_0^{(2)} |\Phi_0\rangle & + & C_1^{(2)} |\Phi_1\rangle & + & \dots & + & C_{N-1}^{(2)} |\Phi_{N-1}\rangle \\
\vdots & & \vdots & & \vdots & & & & \vdots \\
|\Psi_{N-1}\rangle & = & C_0^{(N-1)} |\Phi_0\rangle & + & C_1^{(N-1)} |\Phi_1\rangle & + & \dots & + & C_N^{(N-1)} |\Phi_{N-1}\rangle
\end{array} \tag{6.29}$$

such that

$$\hat{H} |\Psi_p\rangle = \varepsilon_p |\Psi_p\rangle . \tag{6.30}$$

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

$$\hat{H} = \sum_{ij} |\Phi_i\rangle \, \langle\Phi_i|\hat{H}|\Phi_j\rangle \, \langle\Phi_j| \tag{6.31}$$

such that the Schrödinger equation can be rewritten as

$$\begin{pmatrix}
\langle\Phi_0|\hat{H}|\Phi_0\rangle & \langle\Phi_0|\hat{H}|\Phi_1\rangle & \dots & \langle\Phi_0|\hat{H}|\Phi_{N-1}\rangle \\
\langle\Phi_1|\hat{H}|\Phi_0\rangle & \langle\Phi_1|\hat{H}|\Phi_1\rangle & \dots & \langle\Phi_1|\hat{H}|\Phi_{N-1}\rangle \\
\vdots & \vdots & \ddots & \vdots \\
\langle\Phi_{N-1}|\hat{H}|\Phi_0\rangle & \langle\Phi_{N-1}|\hat{H}|\Phi_1\rangle & \dots & \langle\Phi_{N-1}|\hat{H}|\Phi_{N-1}\rangle
\end{pmatrix}
\begin{pmatrix} c_0^{(p)} \\ c_1^{(p)} \\ \vdots \\ c_{N-1}^{(p)} \end{pmatrix}
= \varepsilon_p
\begin{pmatrix} c_0^{(p)} \\ c_1^{(p)} \\ \vdots \\ c_{N-1}^{(p)} \end{pmatrix} \tag{6.32}$$

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

$$N_{\text{FCI}} = \binom{N_{\text{single orbitals}}}{N_{\text{electrons}}} \tag{6.33}$$

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

6.3 Coupled Cluster

The coupled cluster method is the *de facto* standard wave function-based method for electronic structure calculations. [32] The method approximates the wave function with an exponential expansion,

$$|\Psi_{\mathrm{CC}}\rangle = e^{\hat{T}} |\Phi_0\rangle \tag{6.34}$$

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

$$\hat{T}_n = \left(\frac{1}{n!}\right)^2 \sum_{abc\dots} \sum_{ijk\dots} t_{ijk\dots}^{abc\dots} a_a^\dagger a_b^\dagger a_c^\dagger \cdots a_k a_j a_i. \tag{6.35}$$

We again want to solve the Schrödinger equation,

$$\hat{H} |\Psi\rangle = \hat{H} e^{\hat{T}} |\Phi_0\rangle = \epsilon e^{\hat{T}} |\Phi_0\rangle , \tag{6.36}$$

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

$$\bar{H} = e^{-\hat{T}} \hat{H} e^{\hat{T}}. \tag{6.37}$$

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

$$\langle \Phi_0 | \bar{H} | \Phi_0 \rangle = \epsilon \tag{6.38}$$

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

$$\left\langle \Phi_{ijk\dots}^{abc\dots} \middle| \bar{H} \middle| \Phi_0 \right\rangle = 0 \tag{6.39}$$

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

$$\begin{aligned} \bar{H} = \hat{H} &+ [\hat{H}, \hat{T}] \\ &+ \frac{1}{2}[[\hat{H}, \hat{T}], \hat{T}] \\ &+ \frac{1}{6}[[[\hat{H}, \hat{T}], \hat{T}], \hat{T}] \\ &+ \frac{1}{24}[[[[\hat{H}, \hat{T}], \hat{T}], \hat{T}], \hat{T}] \\ &+ \cdots \end{aligned} \tag{6.40}$$

and we are in principle set to solve the amplitude equations with respect to the amplitudes $t_{ijk\dots}^{abc\dots}$ 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. [35]

Part III

Implementation Preparatory

Chapter 7

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.

The trial wave function Ψ_T is a product of all the p wave function elements $\{\psi_1, \psi_2 \dots \psi_p\}$ that are involved in a calculation,

$$\Psi_T(\boldsymbol{r}) = \prod_{i=1}^{p} \psi_i(\boldsymbol{r}). \tag{7.1}$$

For instance, consider a quantum dot of six interacting electrons. In that case, a Slater determinant needs to be included in order to ensure that the wave function is anti-symmetric, but since the Gaussian function appears in all the single particle function, this can be factorized out. We can therefore split the Slater determinant up in a simple Gaussian element and a determinant consisting of the Hermite polynomials. In addition, a Jastrow factor is required to take care of the correlations, so the trial wave function should consist of a total of three elements for this particular system:

$$\Psi_T(\boldsymbol{r}) = \psi_{sg}(\boldsymbol{r})\psi_{sd}(\boldsymbol{r})\psi_{jf}(\boldsymbol{r})$$

where sg denotes the simple Gaussian, sd denotes the Slater determinant and jf denotes an arbitrary Jastrow factor.

We will first try to convince the reader that the local energy and the parameter update can be calculated separately for each element, and then move on to find closed-form expressions for the required term in the calculations for all the elements.

7.1 Kinetic energy computations

The local energy, defined in equation (**??**), is

$$\begin{aligned} E_L &= \frac{1}{\Psi_T(\boldsymbol{r})}\hat{\mathcal{H}}\Psi_T(\boldsymbol{r}) \\ &= \sum_{k=1}^{F}\Big[-\frac{1}{2}\Big(\frac{1}{\Psi_T(\boldsymbol{r})}\nabla_k^2\Psi_T(\boldsymbol{r})\Big)+\mathcal{V}\Big]. \end{aligned}$$

where we have $F = ND$ degrees of freedom. The first term, which is the kinetic energy term, is the only wave function-dependent one, and we will in this section split it up with respect to the elements. The potential energy term, $\mathcal{V}$, is not directly dependent on the wave function and will therefore not be further touched here.

From the definition of differentiation of a logarithm, we have that

$$\frac{1}{\Psi_T(\boldsymbol{r})}\nabla_k\Psi_T(\boldsymbol{r}) = \nabla_k \ln\Psi_T(\boldsymbol{r}), \tag{7.2}$$

which provides the following useful relation

$$\frac{1}{\Psi_T(\boldsymbol{r})}\nabla_k^2\Psi_T(\boldsymbol{r}) = \nabla_k^2 \ln\Psi_T(\boldsymbol{r}) + (\nabla_k \ln\Psi_T(\boldsymbol{r}))^2. \tag{7.3}$$

Using the fact that the trial wave function is a product of all the elements, the term above is calculated by

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

such that the total kinetic energy is given by

$$-\frac{1}{2}\frac{1}{\Psi_T(\boldsymbol{r})}\nabla^2\Psi_T(\boldsymbol{r}) = -\frac{1}{2}\Big[\sum_{i=1}^{p}\nabla^2 \ln\psi_i(\boldsymbol{r}) + \sum_{k=1}^{F}\Big(\sum_{i=1}^{p}\nabla_k \ln\psi_i(\boldsymbol{r})\Big)^2\Big]. \tag{7.4}$$

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

7.2 Parameter update

In gradient based optimization methods, as we use, one needs to know the gradient of the cost function with respect to all the parameters in order to update the parameters correctly. To update the specific parameter θ_j, we thus need to find

$$\nabla_{\theta_j}\mathcal{C}(\boldsymbol{\theta}) \equiv \frac{\partial\mathcal{C}(\boldsymbol{\theta})}{\partial\theta_j},$$

where $\mathcal{C}(\boldsymbol{\theta})$ is our cost function. For our problem, a natural choice is to define the cost function as the local energy, since that is the function that we want to minimize. We therefore set

$$\mathcal{C}(\boldsymbol{\theta}) = \langle E_L\rangle \tag{7.5}$$

since we get the expectation value of the local energy from the Metropolis sampling. Further, we use the definition of the gradient of an expectation value and obtain

$$\nabla_{\theta_j}\langle E_L\rangle = 2\Big(\langle E_L\nabla_{\theta_j}\ln\Psi_T\rangle - \langle E_L\rangle\langle\nabla_{\theta_j}\ln\Psi_T\rangle\Big) \tag{7.6}$$

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

$$\langle\nabla_{\theta_j}\ln\Psi_T\rangle = \int_{-\infty}^{\infty} d\boldsymbol{r}P(\boldsymbol{r})\nabla_{\theta_j}\ln\Psi_T(\boldsymbol{r})$$

and

$$\langle E_L\nabla_{\theta_j}\ln\Psi_T\rangle = \int_{-\infty}^{\infty} d\boldsymbol{r}P(\boldsymbol{r})E_L(\boldsymbol{r})\nabla_{\theta_j}\ln\Psi_T(\boldsymbol{r}),$$

which can be found by Monte-Carlo integration in the same way as the local energy:

$$\langle\nabla_{\theta_j}\ln\Psi_T\rangle \approx \frac{1}{M}\sum_{i=1}^{M}\nabla_{\theta_j}\ln\Psi_T(\boldsymbol{r}_i) \tag{7.7}$$

and

$$\langle E_L\nabla_{\theta_j}\ln\Psi_T\rangle \approx \frac{1}{M}\sum_{i=1}^{M}E_L(\boldsymbol{r}_i)\nabla_{\theta_j}\ln\Psi_T(\boldsymbol{r}_i). \tag{7.8}$$

By applying equation (7.1), we find that

$$\nabla_{\theta_j}\ln\Psi_T(\boldsymbol{r}) = \sum_{i=1}^{p}\nabla_{\theta_j}\ln\psi_i(\boldsymbol{r}), \tag{7.9}$$

which means that we need to find closed-form expressions of $\nabla_{\theta_j}\ln\psi_i(\boldsymbol{r})$ for all wave function elements $\psi_i(\boldsymbol{r})$ and all variational parameters θ_j.

We want to stress that the local energy is not the only possible choice of the cost function. By taking advantage of the zero-variance property of the expectation value of the local energy in the minimum, one can also minimize the variance. This requires the calculation of a few additional expectation values, but it is a fully manageable task to do. See for instance Ref.[40] for more information.

7.3 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 from each other by a factor, 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

$$\frac{\Psi_T(\boldsymbol{r}_{\text{new}})}{\Psi_T(\boldsymbol{r}_{\text{old}})} = \prod_{i=1}^{p}\frac{\psi_i(\boldsymbol{r}_{\text{new}})}{\psi_i(\boldsymbol{r}_{\text{old}})}, \tag{7.10}$$

and below we will calculate this ratio squared since we are going to use that directly in the sampling.

7.4 Derivatives

We will in this section go through the derivatives of the various wave function elements that are used in our work, in order to be able to compute the kinetic energy and the parameter update correctly. We will start with the elements used in a standard variational Monte-Carlo calculation, and thereafter move on to the elements inspired by Boltzmann machines. In the end, we will discuss the Hydrogen-like orbitals.

7.4.1 Simple Gaussian

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

$$\psi_{sg}(\boldsymbol{x};\alpha) = \exp\Big(-\frac{1}{2}\omega\alpha\sum_{j=1}^{N} r_j^2\Big) = \exp\Big(-\frac{1}{2}\omega\alpha\sum_{j=1}^{F} x_j^2\Big),$$

similarly to the function presented in section (3.1). 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.

When changing the coordinate x_i from x_i^{old} to x_i^{new}, the probability ratio can easily be found to be

$$\frac{|\psi_{sg}(\boldsymbol{x}_{\text{new}})|^2}{|\psi_{\text{sg}}(\boldsymbol{x}_{\text{old}})|^2} = \exp\Big(\omega\alpha\big((x_i^{\text{old}})^2 - (x_i^{\text{new}})^2\big)\Big), \tag{7.11}$$

and henceforth the index i will be reserved the changed coordinate. The gradient of $\ln\psi_{\text{sg}}$ with respect to the coordinate x_k is

$$\nabla_k \ln\psi_{sg} = -\omega\alpha x_k, \tag{7.12}$$

and similar to i, k will be reserved the coordinate we are differentiating with respect to. The corresponding Laplacian is

$$\nabla^2 \ln\psi_{sg} = -F\omega\alpha, \tag{7.13}$$

and finally, we will update α according to

$$\nabla_\alpha \ln\psi_{sg} = -\frac{1}{2}\omega\sum_{j=1}^{F} x_j^2. \tag{7.14}$$

Since this wave function element is quite simple, there is no special optimization available that will cause a noticeable speed-up.

7.4.2 Simple Jastrow factor

The Jastrow factor is introducted in order to take care of the corrolations. Recall the simple Jastrow factor from (2.21),

$$\psi_{sj}(\boldsymbol{r};\boldsymbol{\beta}) = \exp\Big(\sum_{i=1}^{N}\sum_{j>i}^{N}\beta_{ij}r_{ij}\Big). \tag{7.15}$$

with N as the number of particles, r_{ij} as the distance between particle i and j and β_{ij} as variational parameters.

This is relatively easy to work with, but one challenge is that we operate in Cartesian coordinates, while the expressed Jastrow factor obviously is easier to deal with in spherical coordinates. Since we need to differentiate this with respect to all free dimensions, we need to be attentive not confusing the particle indices with the coordinate indices. Let us define i as the coordinate index and i' as the index on the corresponding particle. The relationship between i and i' is *always* $i' = i \setminus D$, where the backslash denotes integer division. The other way around, we have $i = i' + d$ where d is the respective dimension of the coordinate i. With that notation, the probability ratio is given by

$$\frac{|\psi_{sj}(\boldsymbol{r}_{\text{new}})|^2}{|\psi_{\text{sj}}(\boldsymbol{r}_{\text{old}})|^2} = \exp\Big(2\sum_{j'=1}^{N}\beta_{i'j'}(r_{i'j'}^{\text{new}} - r_{i'j'}^{\text{old}})\Big) \tag{7.16}$$

The gradient reads

$$\nabla_k \ln\psi_{sj} = \sum_{j'=1}^{N}\frac{\beta_{k'j'}}{r_{k'j'}}(x_k - x_j) \tag{7.17}$$

where j is related to the same dimension as k. This also applies for the Laplacian,

$$\nabla^2 \ln\psi_{sj} = \sum_{k=1}^{F}\sum_{j'=1}^{N}\frac{\beta_{k'j'}}{r_{k'j'}}\Big(1 - \frac{(x_k - x_j)^2}{r_{k'j'}^2}\Big). \tag{7.18}$$

Finally, the parameter update is given by

$$\nabla_{\beta_{m'l'}} \ln\psi_{sj} = r_{m'l'}. \tag{7.19}$$

For this element, the most important thing we can do to keep the computational cost as low as possible is to reveal that only a row and a column of the distance matrix is changed as we change a coordinate. Updating the entire distance matrix means updating N^2 elements, while updating a row and a column means updating $2N$ elements, which is an essential difference for large systems.

7.4.3 The Padé-Jastrow factor

The Padé-Jastrow factor is a more complicated Jastrow factor, and was specified in equation (2.22),

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

where we want to emphasize that a_{ij} is *not* a variational parameter.

Similarly to the simple Jastrow, we also here need to distinguish between particle indices and coordinate indices because of the radial distances r_{ij}. We do the same trick as presented above, and obtain the gradient

$$\nabla_k \ln \psi_{pj} = \sum_{j' \neq k'=1}^{N} \frac{a_{k'j'}}{(1+\beta r_{k'j'})^2} \frac{x_k - x_j}{r_{k'j'}}$$

with respect to the coordinate x_k. By again differentiating this with respect to x_k, we obtain the Laplacian

$$\nabla^2 \ln \psi_{pj} = \sum_{k=1}^{F} \sum_{j' \neq k'=1}^{N} \frac{a_{k'j'}}{(1+\beta r_{k'j'})^2} \left[1 - \left(1 + 2 \frac{\beta r_{k'j'}}{1+\beta r_{k'j'}} \right) \frac{(x_k - x_j)^2}{r_{k'j'}^2} \right] \frac{1}{r_{k'j'}}.$$

The last expression we need is the one used to update the variational parameter β, which is found to be

$$\partial_\beta \ln \psi_{pj} = -\sum_{i'=1}^{N} \sum_{j'>i'}^{N} \frac{a_{ij} r_{ij}^2}{(1+\beta r_{ij})^2}.$$

Furthermore, we observe that some factors are found in multiple expressions. To simplify the expressions and as an beginning of the optimization, we introduce

$$f_{ij} = \frac{1}{1+\beta r_{ij}} \qquad g_{ij} = \frac{x_i - x_j}{r_{i'j'}} \qquad h_{ij} = \frac{r_{ij}}{1+\beta r_{ij}}.$$

The final expressions then read

$$\begin{aligned}
\frac{|\psi_{pj}(\boldsymbol{r}_{\text{new}})|^2}{|\psi_{pj}(\boldsymbol{r}_{\text{old}})|^2} &= \exp\left(2 \sum_{j'=1}^{N} a_{i'j'} (h_{i'j'}^{\text{new}} - h_{i'j'}^{\text{old}}) \right) \\
\nabla_k \ln \psi_{pj} &= \sum_{j' \neq k'=1}^{N} a_{k'j'} \cdot f_{k'j'}^2 \cdot g_{kj} \\
\nabla^2 \ln \psi_{pj} &= \sum_{k=1}^{F} \sum_{j' \neq k'=1}^{N} \frac{a_{k'j'}}{r_{k'j'}} f_{k'j'}^2 \left[1 - (1 + 2\beta h_{k'j'}) g_{kj}^2 \right] \\
\nabla_\beta \ln \psi_{pj} &= -\sum_{l'=1}^{N} \sum_{j>l}^{N} a_{l'j'} h_{l'j'}^2
\end{aligned} \tag{7.20}$$

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

In the same way as for the simple Jastrow, only a row and a column in the distance matrix should be updated for each step. Additionally, implementing the matrices f_{ij}, g_{ij} and h_{ij} will give a speed up in combination with vector-matrix operations.

7.4.4 Slater determinant

In general, the the Slater determinant contains all the single particle functions. However, in some cases all single particle functions have the same factor, and then this part can be factorized out. Therefore we will treat the Slater determinant as an ordinary wave function element in this chapter and in the code.

As discussed in section (2.2.2), it can be split up in a spin-up part and a spin-down part,

$$\psi_{sd}(\boldsymbol{r}) = |\hat{D}_{\uparrow}(\boldsymbol{r}_{\uparrow})| \cdot |\hat{D}_{\downarrow}(\boldsymbol{r}_{\downarrow})|.$$

$r_{\uparrow}$ are the coordinates of particles with spin up (defined as the first $N_{\uparrow}$ coordinates) and $r_{\downarrow}$ are the coordinates of particles with spin down (defined as the last $N_{\downarrow}$ coordinates).

We can now utilize the logarithmic scale, by using that the logarithm of a product corresponds to summarize the logarithm of each factor,

$$\ln \psi_{sd} = \ln |\hat{D}_{\uparrow}(\boldsymbol{r}_{\uparrow})| + \ln |\hat{D}_{\downarrow}(\boldsymbol{r}_{\downarrow})|$$

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:

$$\nabla_k \ln \psi_{sd} = \begin{cases} \nabla_k \ln |\hat{D}_{\uparrow}(\boldsymbol{r}_{\uparrow})| & \text{if} \quad k < N_{\uparrow} \\ \nabla_k \ln |\hat{D}_{\downarrow}(\boldsymbol{r}_{\downarrow})| & \text{if} \quad k \geq N_{\uparrow}. \end{cases}$$

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

$$\hat{D}(\boldsymbol{r}) \equiv \hat{D}_{m_s}(\boldsymbol{r}_{m_s})$$

where m_s is the spin projection. When summarizing, the sum is always over all relevant coordinates.

Furthermore, we have that

$$\nabla_k \ln |\hat{D}(\boldsymbol{r})| = \frac{\nabla_k |\hat{D}(\boldsymbol{r})|}{|\hat{D}(\boldsymbol{r})|}$$

and

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

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

$$\frac{\nabla_i |\hat{A}|}{|\hat{A}|} = \mathrm{tr}\Big(\hat{A}^{-1} \nabla_i \hat{A} \Big), \tag{7.21}$$

and the second derivative is then

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

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

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

where a_{ij} is element i, j of matrix $\hat{A}$.

Using all the general matrix operations presented above, for our specific case we end up with

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

and

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

7.4.4.1 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 goes as $\mathcal{O}(\mathrm{N}^3)$ for an N×N matrix. [33].

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

$$\hat{A}^{-1} = \frac{\hat{C}^T}{|\hat{A}|} \tag{7.22}$$

where the comatrix is defined by the inner determinants of the matrix. [81] As a consequence, the determinant can be written as

$$|\hat{A}| = \sum_{i,j} a_{ij}c_{ij}. \tag{7.23}$$

where c_{ij} is the element i, j of the comatrix. As always, we are interested in the ratio between the old and the new wave functions, and since only a row in the matrix $\hat{D}$ is updated every time we move a particle, the ratio between the determinants can be expressed as

$$R \equiv \frac{|\hat{D}(\boldsymbol{r}_{\text{new}})|}{|\hat{D}(\boldsymbol{r}_{\text{old}})|} = \frac{\sum_j d_{ij}(\boldsymbol{r}_{\text{new}})c_{ij}(\boldsymbol{r}_{\text{new}})}{\sum_j d_{ij}(\boldsymbol{r}_{\text{old}})c_{ij}(\boldsymbol{r}_{\text{old}})} \tag{7.24}$$

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 $c_{ij}^{\text{new}} = c_{ij}^{\text{old}}$. From equation (7.23), we can see that $c_{ij} = |\hat{A}|a_{ji}^{-1}$, such that the ratio can be further expressed as

$$R = \frac{\sum_j d_{ij}(\boldsymbol{r}_{\text{new}})d_{ji}^{-1}(\boldsymbol{r}_{\text{old}})}{\sum_j d_{ij}(\boldsymbol{r}_{\text{old}})d_{ji}^{-1}(\boldsymbol{r}_{\text{old}})} = \sum_j d_{ij}(\boldsymbol{r}_{\text{new}})d_{ji}^{-1}(\boldsymbol{r}_{\text{old}}) \tag{7.25}$$

where we have used that fact that $a_{ij}a_{ji}^{-1} = 1$.

To calculate the inverse of matrix, $\hat{D}^{-1}$, efficiently, we need to calculate

$$S_j = \sum_{l=1}^{N} d_{il}(\boldsymbol{r}_{\text{new}})d_{lj}^{-1}(\boldsymbol{r}_{\text{old}}) \tag{7.26}$$

for all columns but the one associated with the moved particle, i. The $j'th$ column of $\hat{D}^{-1}$ is then given by

$$d_{kj}^{-1}(\boldsymbol{r}_{\text{new}}) = d_{kj}^{-1}(\boldsymbol{r}_{\text{old}}) - \frac{S_j}{R}d_{ki}^{-1}(\boldsymbol{r}_{\text{old}}) \tag{7.27}$$

while the remaining column, i, can simply be updated as

$$d_{ki}^{-1}(\boldsymbol{r}_{\text{new}}) = d_{ki}^{-1}(\boldsymbol{r}_{\text{old}}). \tag{7.28}$$

Those procedures makes the inverting go as $\mathcal{O}(\mathrm{N}^2)$ instead of $\mathcal{O}(\mathrm{N}^3)$, which is largely beneficial for large systems. [67]

We assume that we do not have any variational parameter in the Slater determinant, and obtain three expressions of the case when a particle with spin up is moved and three of the case when a particle with spin down is moved.

$$\begin{aligned} &\text{if} \quad k < N_\uparrow : \\ &\frac{|\psi_{sd}(\boldsymbol{r}_{\text{new}})|^2}{|\psi_{sd}(\boldsymbol{r}_{\text{old}})|^2} = \frac{|\hat{D}_\uparrow(\boldsymbol{r}_\uparrow^{\text{new}})|^2}{|\hat{D}_\uparrow(\boldsymbol{r}_\uparrow^{\text{old}})|^2} \\ &\nabla_k \ln|\hat{D}_\uparrow(\boldsymbol{r}_\uparrow)| = \sum_{j=1}^{N_\uparrow} \nabla_k d_{jk}(\boldsymbol{r}_\uparrow) d_{kj}^{-1}(\boldsymbol{r}_\uparrow) \end{aligned} \tag{7.29}$$

$$\begin{aligned} &\text{if} \quad k \geq N_\uparrow : \\ &\frac{|\psi_{sd}(\boldsymbol{r}_{\text{new}})|^2}{|\psi_{sd}(\boldsymbol{r}_{\text{old}})|^2} = \frac{|\hat{D}_\downarrow(\boldsymbol{r}_\downarrow^{\text{new}})|^2}{|\hat{D}_\downarrow(\boldsymbol{r}_\downarrow^{\text{old}})|^2} \\ &\nabla_k \ln|\hat{D}_\downarrow(\boldsymbol{r}_\downarrow)| = \sum_{j=N_\uparrow}^{F} \nabla_k d_{jk}(\boldsymbol{r}_\downarrow) d_{kj}^{-1}(\boldsymbol{r}_\downarrow) \end{aligned} \tag{7.30}$$

$$\nabla^2 \ln|\hat{D}(\boldsymbol{r})| = \sum_{k=1}^{F} \left[\sum_{j=1}^{F} \nabla_k^2 d_{jk}(\boldsymbol{r}) d_{kj}^{-1}(\boldsymbol{r}) - \left(\sum_{j=1}^{F} \nabla_k d_{ik}(\boldsymbol{r}) d_{ki}^{-1}(\boldsymbol{r}) \right)^2 \right] \tag{7.31}$$

7.4.5 Restricted Boltzmann machine

Now over to the real deal; the wave function elements inspired by machine learning. The marginal distribution of the visible nodes of a restricted Boltzmann machine was presented in equation (4.60). By defining the wave function as this marginal distribution, we have

$$\psi_{rbm}(\boldsymbol{x}; \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{w}) = \exp\Big(-\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2} \Big) \prod_{j=1}^{H} \Big(1 + \exp\Big(b_j + \sum_{i=1}^{F} \frac{w_{ij} x_i}{\sigma^2} \Big) \Big), \tag{7.32}$$

which contains a Gaussian part and a product part. In order to minimize the complexity of each wave function element, we decided to split it up in the code and they will therefore be presented separately below. The first part will henceforth be denoted as the RBM-Gaussian, while the last part will be denoted as the RBM-Product.

7.4.5.1 RBM-Gaussian

The RBM-Gaussian reads

$$\psi_{rg}(\boldsymbol{x}; \boldsymbol{a}) = \exp\left(-\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2} \right) \tag{7.33}$$

The derivatives of the RBM-Gaussian are similar to those of the simple Gaussian, they are therefore just listed in equation (7.34).

$$\begin{aligned}
\frac{|\psi_{rg}(\boldsymbol{x}_{\text{new}})|^2}{|\psi_{rg}(\boldsymbol{x}_{\text{old}})|^2} &= \exp\Big((x_i^{\text{old}} + x_i^{\text{new}} - 2a_i)(x_i^{\text{old}} - x_i^{\text{new}})\Big) \\
\nabla_k \ln \psi_{\text{rg}} &= -\frac{x_k - a_k}{\sigma^2} \\
\nabla_k^2 \ln \psi_{\text{rg}} &= -\frac{1}{\sigma^2} \\
\nabla_{\alpha_l} \ln \psi_{\text{rg}} &= \frac{x_l - a_l}{\sigma^2}
\end{aligned} \tag{7.34}$$

7.4.5.2 RBM-product

The RBM product is the last part of (4.60), and is thus given by

$$\psi_{rp}(\boldsymbol{x}; \boldsymbol{b}, \boldsymbol{w}) = \prod_{j=1}^{H} \left[1 + \exp\left(b_j + \sum_{i=1}^{F} \frac{w_{ij} x_i}{\sigma^2}\right)\right].$$

In appendix C, section (C.2), a general Gaussian-binary RBM product on the form

$$\psi(\boldsymbol{x}; \boldsymbol{\theta}) = \prod_{j=1}^{H} \left[1 + \exp\Big(f_j(\boldsymbol{x}; \boldsymbol{\theta})\Big)\right]$$

is differentiated, which for this element corresponds to setting $f_j = b_j + \boldsymbol{w}_j^T \boldsymbol{x}/\sigma^2$. As we further claim, the only expressions that need to be calculated are $\nabla_k(f_j)$, $\nabla_k^2(f_j)$ and $\partial_{\theta_i}(f_j)$ for all the coordinates k and all the parameters θ_i. They can easily be found to be

$$\begin{aligned}
\nabla_k(f_j) &= \frac{w_{kj}}{\sigma^2} \\
\nabla_k^2(f_j) &= 0 \\
\partial_{b_l}(f_j) &= \delta_{lj} \\
\partial_{w_{ml}}(f_j) &= \frac{x_m}{\sigma^2}\delta_{lj}
\end{aligned}$$

for our specific function. δ_{lj} is the Kronecker delta. By reintroducing the sigmoid function and the counterpart

$$n_j(x) = \frac{1}{1 + \exp(-x)} \quad \wedge \quad p_j(x) = n_j(-x) = \frac{1}{1 + \exp(x)}$$

we can express the required derivatives in the following fashion

$$\begin{aligned}
\frac{|\psi_{rp}(\boldsymbol{x}_{\text{new}})|^2}{|\psi_{\text{rp}}(\boldsymbol{x}_{\text{old}})|^2} &= \prod_{j=1}^{H} \frac{p_j(\boldsymbol{x}_{\text{old}})^2}{p_j(\boldsymbol{x}_{\text{new}})^2} \\
\nabla_k \ln \psi_{rp} &= \sum_{j=1}^{H} \frac{w_{kj}}{\sigma^2} n_j \\
\nabla_k^2 \ln \psi_{rp} &= \sum_{j=1}^{H} \frac{w_{kj}^2}{\sigma^4} p_j n_j \\
\nabla_{b_l} \ln \psi_{rp} &= n_l \\
\nabla_{w_{ml}} \ln \psi_{rp} &= \frac{x_m n_l}{\sigma^2}.
\end{aligned} \tag{7.35}$$

This is the same result as obtained for the gradient of the log-likelihood function presented in chapter 4. In this element, there are plenty of optimization possibilities. By revealing that some sums are vector products, we can get a significant speed-up. Instead of presenting the vectorized expressions, we will present how the elements actually are implemented.

```
double prod =  m_pOld.prod() / m_p.prod();
m_probabilityRatio  = prod * prod;

m_gradient = double(m_W.row(k) * m_n) / m_sigmaSqrd;

m_laplacian = (m_w.cwiseAbs2() * m_p.cwiseProd(m_n)).sum() /
    ↪ (m_sigmaSqrd*m_sigmaSqrd);

Eigen::MatrixXd out = m_positions * m_n.transpose();
m_gradients.segment(m_numberOfHiddenNodes, out.size()) = flatten(out);
m_gradients.head(m_numberOfHiddenNodes) = m_n;
```

There the linear algebra package Eigen is used for the matrix-vector operations, and `m_gradients` consists of all the derivatives with respect to the parameters.

7.4.6 Partly restricted Boltzmann machine

For the partly restricted Boltzmann machine given in equation (4.67), we observe that the only difference from a standard Boltzmann machine is the factor

$$\psi_{pr} = \exp\Big(\sum_{i=1}^{F}\sum_{j=1}^{F} x_i c_{ij} x_j\Big) \tag{7.36}$$

which we can threaten separately. We end up with the expressions

$$\begin{aligned}
\frac{|\psi_{pr}(\boldsymbol{x}_{\text{new}})|^2}{|\psi_{pr}(\boldsymbol{x}_{\text{old}})|^2} &= \exp\Big(2\sum_{j=1}^{F} c_{ij} x_j (x_i^{\text{new}} - x_i^{\text{old}})\Big) \\
\nabla_k \ln \psi_{pr} &= 2\sum_{j=1}^{F} c_{kj} x_j \\
\nabla_k^2 \ln \psi_{pr} &= 2c_{kk} \\
\nabla_{c_{ml}} \ln \psi_{pr} &= x_m x_l
\end{aligned} \tag{7.37}$$

where x_i is the changed coordinated. Also here can we use vectorization to speed-up the computations.

7.4.7 Hydrogen-like orbitals

The Hydrogen-like orbitals were presented in (3.18), but as we discussed earlier they cause some problems for atoms of the size of Neon and larger due to complex numbers. Instead, we decided to look at hydrogen-like orbitals with solid harmonics. Even though they do not have problems with complex numbers, they are quite complicated to differentiate, and the closed form will therefore be found by symbolic differentiating on the computer. However, we will do the exercise for the simplest case, which is sufficient to find the Hydrogen and Helium ground states. This reads

$$\psi_{hl}(\boldsymbol{r};\alpha) = \exp\Big(-Z\alpha\sum_{j=1}^{N} r_j\Big) \tag{7.38}$$

where r_j is the distance from particle j to the center. We then differentiate with respect to coordinate x_k, and obtain

$$\nabla_k \ln\psi_{hl} = -Z\alpha\frac{x_k}{r_{k'}} \tag{7.39}$$

The Laplacian is then given by

$$\nabla_k^2 \ln\psi_{hl} = -Z\alpha\Big(1-\frac{x_k^2}{r_{k'}^2}\Big)\frac{1}{r_{k'}} \tag{7.40}$$

and the differentiation with respect to the variational parameter α is

$$\partial_\alpha \ln\psi_{\mathrm{hl}} = -Z\sum_{j=1}^{N} r_j. \tag{7.41}$$

For close-form expressions for higher order wave functions, please run the script `generateHydrogenOrbitals.py`

Chapter 8

Optimization and Resampling Algorithms

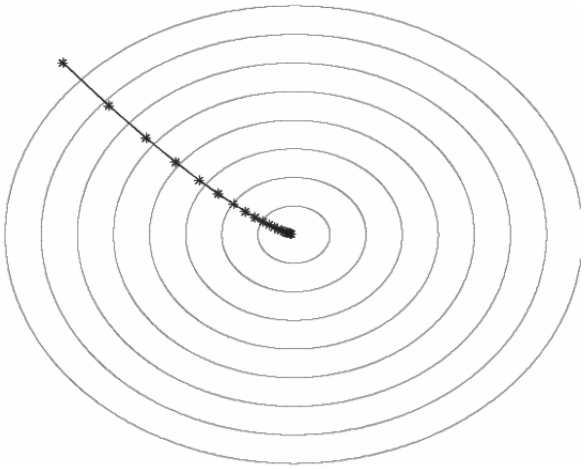


Figure 8.1: An iterative optimization algorithm is always approaching an extremum in a possibly multi-dimensional space. Here illustrated in a two-dimensional space where the equipotential curves are drawn.

In this chapter, we will cover the remaining algorithms we need before we can move on to the implementation. First of all, a good optimization algorithm is a must in order to obtain the correct minimum of the cost function. For our purpose, we potentially have thousands of parameters to optimize, and therefore a robust optimizer is required. The ADAM optimizer has proven its ability to solve similar problems, and we will therefore have a thoroughly discussion of the algorithm.

Secondly, we need an algorithm to determine the statistical errors with high precision. This is the task of the resampling methods, where we keep our attention on the blocking method. For long, the blocking method suffered from the need of tailored hyper-parameters, but the recent years automated blocking methods have been developed, in particular by Marius Jonsson [62]. We will explain the idea behind the traditional blocking method, and in the code we use the automated blocking algorithm.

8.1 Optimization algorithms

In chapter 4, we discussed the gradient descent optimization algorithm, which is among the most basic methods available. That method is based on the gradient, which is the slope of the cost function, but many methods are also in need of the Hessian matrix, which gives the curvature of the cost function. We will barely scratch the surface of this field, limiting us to the gradient methods.

To have the method fresh in mind, we will start with reintroducing the gradient descent method before we move on the its stochastic brother. We will then have a look at how momentum can be added, and finally we examine the stochastic and momentum based ADAM optimizer.

8.1.1 Gradient descent

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

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla_\theta \mathcal{C}(\boldsymbol{\theta}_{t-1}) \tag{8.1}$$

where $\boldsymbol{\theta}_t$ is the parameter vector at time step (iteration) t and η is the learning rate. $\nabla_\theta \mathcal{C}(\boldsymbol{\theta}_{t-1})$ is the gradient of the cost function with respect to all the parameters θ at time $t-1$.

The idea is to find the direction where the cost function $\mathcal{C}(\boldsymbol{\theta})$ has the steepest slope, and move in the direction which minimizes the cost function. For every time step, the cost function is thus minimized, and when the gradient approaches zero the minimum is found. A possible, but basic, stop criterion is

$$\nabla_\theta \mathcal{C}(\boldsymbol{\theta}_t) < \varepsilon. \tag{8.2}$$

where ε is a tolerance. More robust methods are based on comparing the value of the cost function for several past iterations.

In cases where the cost function is not strictly 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.

8.1.2 Stochastic gradient descent

Stochastic gradient descent is closely related to the gradient descent method, but the method uses randomly selected batches to evaluate the gradients, hence the stochasticity. By introducing this randomness, the parameters will not always be updated in order to minimize the energy, which makes us less likely to be stuck in a local minimum.

In practice, one splits the data set in n batches, and select one of them to be used in the parameter update. Our hope is that this batch is representative for the entire data set, such that the new parameters gives a lower cost function. If that is the case, we have reduced the cost of an iteration significantly, since we only need to care about a batch. After each batch in the data set has had an opportunity to update the internal parameters, we say that we have went through an *epoch*.

We are not guaranteed that updating the parameters with respect to a batch gives a lower cost function, and when it is not, we need to run more batches in order to minimize the cost function. Since each iteration is faster than for standard gradient descent, this is acceptable. As long as the batch is slightly representative for the entire data set, the cost function will be minimized in the end.

Mathematically, the method can be expressed as

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla_\theta \mathcal{C}_i(\boldsymbol{\theta}_{t-1}) \tag{8.3}$$

where we use the i'th batch in the parameter update. Standard gradient descent is actually just a special case of this, where we only have one batch (i includes the whole data set). If we still get stuck in local minima after adding the stochasticity, it might be a good idea to add momentum as well.

8.1.3 Adding momentum

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. If it has enough momentum, it will 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.

Momentum can be added to most optimization algorithms, also gradient descent and stochastic gradient descent. The way we do it is to save the direction we were moving in the previous iteration, and use it as a contribution to the next gradient update. A typical implementation of the first-order momentum looks like

$$\begin{aligned} \boldsymbol{m}_t &= \gamma \boldsymbol{m}_{t-1} + \eta \nabla_\theta \mathcal{C}_i(\boldsymbol{\theta}_{t-1}) \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \boldsymbol{m}_t \end{aligned} \tag{8.4}$$

where γ is the momentum parameter, which is just another hyper-parameter usually initialized to a small number. $\boldsymbol{m}_t$ is the momentum vector, and can be initialized as the zero vector.

The optimization algorithm can be modified further in unlimited ways. A common improvement is to add higher order momentum, another is to make the learning rate adaptive. We have implemented the most basic version of this, with monotonic adaptivity. Many algorithms, such as the conjugate gradient method, also make use of the Hessian as discussed in the introduction, but that is another level of complexity.

We will end this section with setting up the algorithm of a stochastic gradient descent optimization with momentum and monotonic adaptivity. The algorithm is found in algorithm (3).

8.1.4 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 25000 citations! [50] So what makes this method so popular?

The main reason why it is widely used, is obviously that it performs good. The fact that it only requires the gradient makes it efficient, and the way the momentum is implemented still makes able to handle a large number of parameters.

The optimization algorithm can be expressed as a set of equations

$$\begin{aligned} \boldsymbol{g}_t &= \nabla_\theta \mathcal{C}_t(\boldsymbol{\theta}_{t-1}) \\ \boldsymbol{m}_t &= \gamma_1 \boldsymbol{m}_{t-1} + (1-\gamma_1)\boldsymbol{g}_t \\ \boldsymbol{v}_t &= \gamma_2 \boldsymbol{v}_{t-1} + (1-\gamma_2)\boldsymbol{g}_t^2 \\ \hat{\boldsymbol{m}}_t &= \boldsymbol{m}_t/(1-\gamma_1^t) \\ \hat{\boldsymbol{v}}_t &= \boldsymbol{v}_t/(1-\gamma_2^t) \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \eta \hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon) \end{aligned} \tag{8.5}$$

Algorithm 3: Adaptive stochastic gradient descent with momentum. See sections (8.1.2-8.1.3) for details. Robust default settings for the hyper-parameters are $\eta = 0.001$, $\gamma = 0.01$ and $\lambda = 0.1$. All the operations are element-wise.

Parameter: η: Learning rate
Parameter: γ: Momentum parameter
Parameter: λ: Monotonic decay rate
Require : $\mathcal{C}(\boldsymbol{\theta})$: Cost function
Data : $\boldsymbol{\theta}_0$: Initial parameters
$\boldsymbol{m}_0 \leftarrow 0$ (Initialize momentum vector);
$t \leftarrow 0$ (Initialize time step);
while $\boldsymbol{\theta}_t$ *not converged* **do**
  $t \leftarrow t + 1$ (Increase time for each iteration);
  $\boldsymbol{g}_t \leftarrow \nabla_\theta \mathcal{C}_t(\boldsymbol{\theta}_{t-1})$ (Get gradients from a given batch at time t);
  $\boldsymbol{m}_t \leftarrow \gamma \boldsymbol{m}_{t-1} + \eta \cdot \boldsymbol{g}_t$ (Update first momentum estimate);
  $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \cdot \boldsymbol{m}_t / \lambda^t$ (Update parameters);
end
Result: Updated parameters $\boldsymbol{\theta}_t$ after convergence

where $\boldsymbol{m}_t$ is the biased first momentum estimate of the parameter vector $\boldsymbol{\theta}$ and $\boldsymbol{v}_t$ is the biased second raw moment estimate. The momentum parameters need to be in the range $\gamma_1, \gamma_2 \in [0, 1)$, and are often set to values close to 1. This makes the optimization adaptive: as time goes, the factors $1 - \gamma_1^t$ and $1 - \gamma_2^t$ approach 1 from below. η corresponds to the learning rate, and should be a small number. Finally, the parameter ε is added to avoid division by zero.

We can set up the algorithm in a similar manner to the adaptive stochastic gradient descent algorithm from above, which gives the algorithm (4).

8.2 Variance estimation

In experiments, we have two main classes of errors, systematical errors and statistical errors. The former is a result of external factors such as uncertainties in the apparatus or a person constantly takes an incorrect measurement, which is an error that is hard to estimate. The latter, however, can be found by estimating the variance of the sample mean, which we want to find accurate and efficient. Monte-Carlo simulations can be treated as computer experiments, and therefore we can use the same analyzing tools as we do for real experiments.

There are several ways to estimate the variance, where the cheapest ones also are the less accurate ones. To make the most use of our data, we use resampling methods to estimate the statistical errors. Some well-known resampling algorithms are blocking, bootstrap and jackknife. We will cover the blocking method only, since that is the only one we use in our particular implementations. To save computational time, we resample the final iteration only, for the others we use the simple estimation method.

8.2.1 Basic concepts of statistics

Before we go through the methods, we will give a brief introduction to some useful statistical quantities. We start with the *moments*, which are given by

$$\langle x^n \rangle = \int dx p(x) x^n$$

Algorithm 4: ADAM optimizer. Robust default settings for the hyper-parameters are $\eta = 0.001$, $\gamma = 0.01$ and $\lambda = 0.1$. All the operations are element-wise, and for in-depth information see the original paper, [50].

Parameter: η: Learning rate
Parameter: $\gamma_1, \gamma_2 \in [0, 1)$: Momentum parameters
Parameter: ε: Division parameter
Require : $\mathcal{C}(\boldsymbol{\theta})$: Cost function
Data : $\boldsymbol{\theta}_0$: Initial parameters
$\boldsymbol{m}_0 \leftarrow 0$ (Initialize 1st momentum vector);
$\boldsymbol{v}_0 \leftarrow 0$ (Initialize 2st momentum vector);
$t \leftarrow 0$ (Initialize time step);
while $\boldsymbol{\theta}_t$ *not converged* **do**
  $t \leftarrow t + 1$ (Increase time for each iteration);
  $\boldsymbol{g}_t \leftarrow \nabla_\theta \mathcal{C}_t(\boldsymbol{\theta}_{t-1})$ (Get gradients from a given batch at time t);
  $\boldsymbol{m}_t \leftarrow \gamma_1 \boldsymbol{m}_{t-1} + (1 - \gamma_1) \cdot \boldsymbol{g}_t$ (Update first momentum estimate);
  $\boldsymbol{v}_t \leftarrow \gamma_2 \boldsymbol{v}_{t-1} + (1 - \gamma_2) \cdot \boldsymbol{g}_t^2$ (Update second raw momentum estimate);
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \gamma_1^t)$ (Bias-corrected first momentum estimate);
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \gamma_2^t)$ (Bias-corrected second momentum estimate) ;
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta \cdot \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \boldsymbol{\varepsilon})$ (Update parameters) ;
end
Result: Updated parameters $\boldsymbol{\theta}_t$ after convergence

where $p(x)$ is the true *probability density function*. In order to make physical sense, this function needs to normalized such that the integral over all possible outcomes gives a total probability of 1. This is the zero'th moment, $\int dx p(x) = 1$. The first moment is the *mean* of $p(x)$, and is often denoted by the letter μ

$$\langle x \rangle = \mu = \int dx p(x) x \tag{8.6}$$

Furthermore, we can define the *central moments* given by

$$\langle (x - \langle x \rangle)^n \rangle = \int dx (x - \langle x \rangle)^n p(x), \tag{8.7}$$

which is centered around the mean. With $n = 0$ and $n = 1$, this is easy to find, but what is the central moment with $n = 2$? The central moment with $n = 2$ is what we call the *variance*, and is often denoted as σ^2 as we did in the equation (2.9). One can show that

$$\sigma^2 = \langle (x - \langle x \rangle)^2 \rangle = \langle x^2 \rangle - \langle x \rangle^2 \tag{8.8}$$

which was already stated in the theory chapter. The *standard deviation* is given by the square-root of the variance, σ, and is the quantity we will present as the measure of the uncertainty in the last digit of a numerical result.

Using the expressions above, we can calculate the mean and the variance of a probability density function $p(x)$, but what do we do if we have a set of data drawn from an unknown $p(x)$?

If the probability density function is unavailable, we cannot find the exact sample mean and sample mean variance. However, we can make an estimate, writing

$$\langle X \rangle \approx \frac{1}{n} \sum_{i=1}^{n} X_i \equiv \overline{X}, \tag{8.9}$$

for the sample mean assuming that X is our sample containing n points. Further, the variance of the sample mean, assuming that the measurements are *independent*, can be found from

$$\mathrm{var}(\overline{X}) \equiv \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2 = \frac{\sigma^2}{n} \tag{8.10}$$

where σ^2 is the variance of the probability density function. The law of large numbers states that the estimated sample mean approaches the true sample mean as n goes to infinity. This is closely related to the central limit theorem, which says that the probability distribution of X, $p_X(x)$, can be approximated as a normal distribution with $\mu = \overline{X}_n$ and $\sigma = \mathrm{var}(\overline{X})$ as the number of sampling points n goes to infinity.

On the other hand, if the samples are *correlated*, equation (8.10) becomes an underestimation of the *sample error* err_X^2, and is thus more a guideline for the size of the uncertainty, more than an actual estimate of it. In that case, we need to calculate the more general *covariance*, given by

$$\mathrm{cov}(X_i, X_j) = (X_i - \overline{X})(X_j - \overline{X}) \tag{8.11}$$

which gives a measure on the correlations between X_i and X_j. Note that the variance is just the special case where X_i and X_j are independent. We define the sample error as

$$\mathrm{err}_X^2 = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\mathrm{cov}(X_i, X_j) \tag{8.12}$$

which can be further expressed as a function of the *autocovariance*, defined as

$$\gamma(h) = \mathrm{cov}(X_i, X_{i+h}) \tag{8.13}$$

and is thus just a measure of the correlation of a part of a time series with another part of the same time series. If $h = 0$, this is just the variance, $\gamma(h) = \sigma^2$. Using this, the sample error reads

$$\mathrm{err}_X^2 = \frac{\sigma^2}{n} + \frac{2}{n}\sum_{h=1}^{n-1}\Big(1 - \frac{h}{n}\Big)\gamma(h). \tag{8.14}$$

A problem with this definition of the sample error, is that it turns out to be very expensive to calculate as the number of samples gets large. We are in need of a cheaper method, which is the task of the resampling methods. We will in the following discuss the blocking algorithm, which is the resampling method of choice in this work.

8.2.2 Blocking

Above, we have described the need of a proper estimation of the uncertainty in computational simulations, where the covariance was included in the calculation of σ^2. A quick and easy way to get a proper estimate of the uncertainty, is by using the blocking method.

When the blocking method was made popular by Flyvberg and Peterson in 1989, the method required hyper-parameters which had to be carefully adjusted for each particular data set. [18] In 2018, Marius Jonsson reinvented the algorithm and made it automated, with no need for external parameters. Despite this, no compromise was made on performance. The method scales as $12n + \mathcal{O}(\log_2 n)$ for small data sets, but reduces to $n + \mathcal{O}(1)$ for large data sets, which makes it preferred over bootstrap and jackknife for large data sets. [62] We will now go through the idea behind the blocking method.

Consider a time series $\{x_1, x_2, \ldots, x_n\}$ with $n = 2^d$ data point for some integer $d > 1$. For this series, an autocovariance function $\gamma(h)$ is guaranteed to exist. We arrange the data in a vector

$$X = (x_i, x_2, \ldots, x_n), \tag{8.15}$$

which we assume to be asymptotically uncorrelated. The idea is to take the mean of subsequent pair of elements from X, and form a new vector X_1. We then repeat the operation on X_1 and form a new vector X_2 and so on. This is the reason why we require $n = 2^d$. If k denotes an element in vector X_i, we can write the procedure recursively as

$$\begin{aligned} (X_0)_k &\equiv (X)_k \\ (X_{i+1})_k &\equiv \frac{1}{2}\Big((X_i)_{2k-1} + (X_i)_{2k}\Big) \end{aligned} \tag{8.16}$$

where $1 \leq i \leq d-1$ and $1 \leq k \leq n/2^i$, which are known as the *blocking transformations*. According to equation (8.14), we can express the sample mean variance of X_k as

$$\mathrm{var}(\overline{X}_k) = \frac{\sigma_k^2}{n_k} + \frac{2}{n_k}\sum_{h=1}^{n_k-1}\Big(1 - \frac{h}{n_k}\Big)\gamma_k(h), \tag{8.17}$$

where we define the last term as the *truncation error*, e_k, as it is intractable. It can be shown that the sample variance of all pairs $(X_i)_k$ and $(X_j)_k$ after a while will be identical [18],

$$\mathrm{var}(\overline{X}_i) = \mathrm{var}(\overline{X}_j) \tag{8.18}$$

with the consequence that the sample mean variance of the entire set of samples if also given by

$$\mathrm{var}(\overline{X}) = \frac{\sigma_k^2}{n_k} + e_k \qquad \forall \quad 0 \leq k \leq d-1. \tag{8.19}$$

In the original (manual) blocking method, we had to know exactly where to stop the procedure in order to equation (8.18) to hold. If we do not continue long enough, the sample variance has not converged, while if we keep on going for too long, the standard error of $\mathrm{var}(\hat{\sigma}_k^2/n_k)$ get very large. If one plots the sample variance as a function of the iterations, one will see that the curve forms a plateau before it gets very noisy, but with Jonsson's automated method, we do not need to worry about this.

We will end this section by scratching the most basic blocking algorithm

In our work, we will consequently use Marius Jonsson's automated blocking algorithm and code, available on https://github.com/computative/block.

8.3 Summary of the algorithms

Up to this point we have presented several algorithms that contribute to the variational Monte-Carlo algorithm, and to give the reader a complete description of the simulating procedure, we will here summarize them and place them in the correct order. First, Metropolis-Hastings algorithm was given, which is responsible for the sampling. Thereafter, we presented a few optimization algorithms, before the blocking algorithm was given.

The optimization loop forms the outer environment of the algorithm, which means that for every time we update the parameters, we run a set of Monte-Carlo cycles. The expectation values found throughout those cycles are used to update the parameters. Further, the blocking algorithm is used to approximate the uncertainties of the expectation values. In algorithm 6, a simple variational Monte-Carlo algorithm is presented in its entirety. For simplicity, we present the Brute-Force Metropolis algorithm and standard gradient descent.

Algorithm 5: Sketch of the blocking method. Here we find the sample mean variance of the data set $\boldsymbol{X}$ containing n samples. See section 8.2 for details.

Data : $\boldsymbol{X}$: Initial samples

$\boldsymbol{X}_0 \leftarrow \boldsymbol{X}$ (Redefine initial data set);
$i \leftarrow 0$ (Initialize step);
while $var(\boldsymbol{X}_{i+1}) \neq var(\boldsymbol{X}_i)$ **do**
  $n_i \leftarrow n/2^i$ (Number of elements in $\boldsymbol{X}_i$);
  $\text{var}(\boldsymbol{X}_i) \leftarrow \sigma_i^2/n_i$ (Estimate the variance of $\boldsymbol{X}_i$);
  for $k \leftarrow 1$ **to** n_i **do**
    $(\boldsymbol{X}_{i+1})_k \leftarrow 0.5\big((\boldsymbol{X}_i)_{2k-1} + (\boldsymbol{X}_i)_{2k}\big)$ (The blocking transformations);
  end
  $i \leftarrow i + 1$ (Increase i for each iteration);
end

Result: The sample mean variance of the initial data set $\boldsymbol{X}$.

Algorithm 6: The variational Monte-Carlo algorithm in its simplest form. The outer loop is for the parameter update. The first inner loop is the sampling where M is the number of Monte-Carlo cycles, and the last inner loop is the resampling loop. Note that we present the brute-force Metropolis algorithm, while the more robust Metropolis-Hasting algorithm, found in algorithm 24, is preferred. Furthermore, we also limit us to the simple gradient descent method. $\mathcal{N}(\mu, \sigma)$ denotes a normal distribution with mean μ and variance σ, while $\mathcal{U}(0, 1)$ denotes a uniform distribution between 0 and 1.

Parameter: η: Learning rate
Parameter: Δx: Step length
Require : $\Psi_T(\boldsymbol{r}, \boldsymbol{\theta})$: Trial wave function

$\boldsymbol{r} \leftarrow \mathcal{N}(0, \sigma^2)$ (Initialize positions randomly);
$\boldsymbol{\theta} \leftarrow \mathcal{U}(0, 1)$ (Initialize parameters, could be randomly);
while *not converged* **do**
  $\boldsymbol{E} \leftarrow [-]$ (Declare an empty local energy list);
  $\boldsymbol{d}_\theta \leftarrow [-]$ (Declare an empty parameter gradient list);
  $\boldsymbol{Ed}_\theta \leftarrow [-]$ (Declare an empty list containing parameter gradient $\cdot$ local energy);
  for $i \leftarrow 1$ **to** M **do**
    $\boldsymbol{r} \leftarrow \boldsymbol{r} + \Delta x$ (Update position);
    if $\Psi_T(\boldsymbol{r}_{new})/\Psi_T(\boldsymbol{r}_{old}) < \mathcal{U}(0, 1)$ **then**
      reject move ;
    end
    $E^i \leftarrow \big(\hat{\mathcal{H}}\Psi_T(\boldsymbol{r}, \boldsymbol{\theta})\big)/\Psi_T(\boldsymbol{r}, \boldsymbol{\theta})$ (Fill $\boldsymbol{E}$-list with local energies) ;
    $d^i_\theta \leftarrow \nabla_\theta \ln \Psi_T(\boldsymbol{r})$ (Fill $\boldsymbol{d}_\theta$-list with parameter gradients) ;
    $Ed^i_\theta \leftarrow \boldsymbol{E}^i \cdot \boldsymbol{d}^i_\theta$ (Fill $\boldsymbol{Ed}_\theta$-list) ;
  end
  $\boldsymbol{E}_0 \leftarrow \boldsymbol{E}$;
  $i \leftarrow 0$ (Initial blocking step);
  while $var(\boldsymbol{E}_{i+1}) \neq var(\boldsymbol{E}_i)$ **do**
    $n_i \leftarrow n/2^i$ (Number of elements in $\boldsymbol{E}_i$);
    $\mathrm{var}(\boldsymbol{E}_i) \leftarrow \sigma_i^2/n_i$ (Estimate the variance of $\boldsymbol{E}_i$);
    for $k \leftarrow 1$ **to** n_i **do**
      $(\boldsymbol{E}_{i+1})_k \leftarrow 0.5\big((\boldsymbol{E}_i)_{2k-1} + (\boldsymbol{E}_i)_{2k}\big)$ (The blocking transformations);
    end
    $i \leftarrow i + 1$ (Increase i for each iteration);
  end
  $\overline{\boldsymbol{E}} \leftarrow \sum_{j=1}^{M} E^j/M$ (Compute the local energy sample mean);
  $\overline{\boldsymbol{d}}_\theta \leftarrow \sum_{j=1}^{M} d^j_\theta/M$ (Compute the sample mean of $\boldsymbol{d}_\theta$);
  $\overline{\boldsymbol{Ed}}_\theta \leftarrow \sum_{j=1}^{M} Ed^j_\theta/M$ (Compute the sample mean of $\boldsymbol{Ed}_\theta$);
  $G_\theta \leftarrow 2\big(\overline{\boldsymbol{Ed}}_\theta - \overline{\boldsymbol{E}} \cdot \overline{\boldsymbol{d}}_\theta\big)$;
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \cdot G_\theta$;
end

Result: An estimate of the many-body wave function $\Psi(\boldsymbol{r})$ and the local energy $\langle E_L \rangle \approx \overline{\boldsymbol{E}}$ with its uncertainty $\mathrm{var}(\overline{\boldsymbol{E}})$.

Part IV

Implementation and Results

Chapter 9

Scientific Programming

> Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.
>
> John F. Woods, [82]

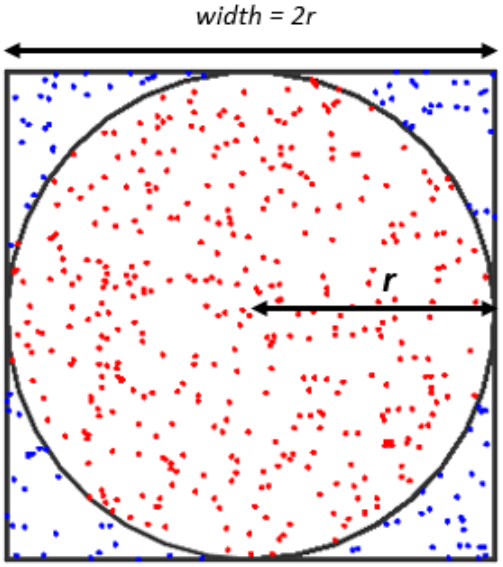


Figure 9.1: One can estimate the value of π using Monte-Carlo integration. Here we approximate the ratio between the area of the circle and the area of the square by counting the number of random points occurring inside the circle and inside the square. The estimate of π is found from $\pi = 4 \cdot A_{\text{circle}}/A_{\text{square}}$.

Since this scientific work is based on solving equations numerically on the computer, software development is a major part of the work. We will in this chapter go through the main concepts of scientific programming, and directly relate them to our code in the next chapter on implementation. As a motivation, we will provide a brief recap of some historical milestones in scientific computing.

Computers have long been used for solving complex scientific problems. Actually, for a long time science was the main motivation of developing computers. Already in 1929, Egil Hylleraas

used his mechanical desk calculator to calculate the ground state energy of the Helium atom, which remains a milestone in computational quantum chemistry as well as quantum chemistry in general [26]. Later, pioneers like Alan Turing and John von Neumann contributed to the invention of the first electronic computer, which made numerical solutions of ordinary differential equations possible [60].

Also on the software side, big breakthroughs ... Since Fortran saw the light in 1957, then considered as a higher-level language, MENTION Ole Johan Dahl and Kristen Nygaard

The computer used today have become possible due to heroic effort of an array of scientists, engineers and mechanics over a long period. The computers are stronger than ever, have more memory than ever, are compacter than ever and, last but not least, they are more user-friendly than ever. This has made advanced simulations possible even for young, inexperienced students like the author, and we should we aware that it has not always been like that.

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 more similar to the binary language than a high-level language. The result is that low-level languages in general are more demanding to deal with, but they are typically faster than high-level languages. In our work, we have used the low-level language C++ for the main code, but for scripting, including plotting and solving less expensive problems, the high-level language Python has been used.

In this chapter, we will explain the essentials behind scientific programming, using examples from both C++ and Python. We will begin from the very basic, so if the reader is an experienced programmer, this chapter is probably old news.

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.

9.1 Object orientated programming in Python

In the everyday life, we are surrounded by objects all the time which we can place in different categories. For instance, a *circle* is an object with properties like area, circumference and so on, and can be placed in the class *shapes*. Other members of the class shapes might be square and triangle. In object orientated programming, we create similar objects, or classes, to make the program flow more intuitive for the reader. The main class is called the parent class, while the sub-classes are called the children. The reason for this analogy, is that the sub-classes always inherit from their main class, but not the other way around.

The example with the shapes can therefore be implemented with `Shapes` being the parent, and `Circle`, `Square` and `Triangle` being its children. In the high-level language Python, this can be implemented as

```
import numpy as np

class Shapes:
    def __init__(self, r):
        self.r = r

    def getArea(self):
        raise NotImplementedError("Class {} has no instance 'getArea'."
                .format(self.__class__.__name__))


class Circle(Shapes):
    def getArea(self):
```

```
        return np.pi*self.r**2

    def getCircumference(self):
        return 2*np.pi*self.r

    def getExtent(self, x, y):
        if np.linalg.norm([x,y]) < self.r:
            return True
        return False

class Square(Shapes):
    def getArea(self):
        return 4*self.r**2

    def getCircumference(self):
        return 8*self.r

    def getExtent(self, x, y):
        if abs(x) < self.r and abs(y) < self.r:
            return True
        return False

class Triangle(Shapes):
    def getCircumference(self):
        return 6*self.r
```

where we give the children the properties area, circumference and extent. The length `r` means the radius of a circle, half the side length of a square and also half a side length of the equilateral triangle, see figure (9.1). We can implement a circle, square and triangle of `r`=1, respectively named "circ", "squr" and "tria" by these three lines

```
circ = Circle(1)
squr = Square(1)
tria = Triangle(1)
```

Moreover, we can easily get their areas by calling the `getArea()` function,

```
>>> circ.getArea()
3.141592653589793

>>> squr.getArea()
4

>>> tria.getArea()
NotImplementedError: Class Triangle has no instance 'getArea'.
```

As we can see, the area of the circle and the square were calculated successfully, giving πr^2 and $4r$ respectively. On the other hand, the area of the triangle raised an error, which is obviously because we do not have defined the area of the triangle! This example illustrates that properties of the children overwrites the properties of the parent, but if the children does not have the called property, it will instead return the parent's property.

9.1.0.1 An example based on Monte-Carlo simulations

Up to this point, we have talked a lot about Monte-Carlo simulations without giving an illustrating example on what it is. Methods where we draw numbers randomly from a function to reveal properties of the function are in general categorized as Monte-Carlo simulations.

Suppose we did not know the value of π, what could we do to approximate its value? There are many ways to do this, where the most intuitive might be to measure the ratio between the diameter and the circumference in a circle. It is not possible to do this very accurate manually.

A more accurate and robust solution would be to use Monte-Carlo simulations, where we for instance can take the starting point on the relation

$$\pi = 4\frac{A_{\text{circle}}}{A_{\text{square}}} \tag{9.1}$$

which is derived from the ratio between the circle area and the square area. By drawing random numbers from a uniform distribution and count the number of point which are in the square and in the circle, we can approximate $A_{\text{circle}}/A_{\text{square}}$. We can do this by using the `getExtent()` function of the square class and circle class,

```
for i in range(1,10):                    # Iteration loop
    M = 10**i
    A_square = 0
    A_circle = 0
    for _ in range(int(M)):              # Monte-Carlo loop
        x = np.random.uniform(-1,1)
        y = np.random.uniform(-1,1)
        if circ.getExtent(x,y):
            A_circle += 1
        if squr.getExtent(x,y):
            A_square += 1
    print("Pi: ", 4 * A_circle/A_square)
```

which gives a output similar to

```
Pi:  2.4
Pi:  3.32
Pi:  3.156
Pi:  3.1296
Pi:  3.14168
Pi:  3.140044
Pi:  3.1420708
Pi:  3.1416844
Pi:  3.141615628
```

Considering the fact that $\pi = 3.141592653589793...$, the obtained result is quite acceptable. However, to run the program with $M = 10^9$ Monte-Carlo cycles, the program is quite slow. Is it possible to do this faster? The answer is yes, by switching entirely or partly to a low-level language.

9.2 Low-level programming with C++

Above, we looked at how inheritance can be implemented in the high-level language Python, and we observed the main weakness of high-level languages: the computational time. In this section we will repeat the implementation and example above using C++. This will, hopefully, provide a significantly speed-up. A similar class structure in C++ looks like

```
class Shapes {
    public:
        Shapes();
        virtual double getArea() = 0;
        virtual double getCircumference() = 0;
        virtual bool getExtent(double x, double y) = 0;
        virtual ~Shapes();
};

class Circle : public Shapes {
    public:
```

```
        Circle(double r);
        double getArea() {
            return M_PI * m_r * m_r;
        }
        double getCircumference() {
            return 2 * M_PI * m_r;
        }
        bool getExtent(double x, double y) {
            if(sqrt(x*x+y*y) < m_r) {
                return true;
            }
            return false;
        }
    private:
        double m_r = 1;
};

Circle::Circle(double r) : Shapes() {
    m_r = r;
}

class Square: public Shapes {
    public:
        Square(double r);
        double getArea() {
            return 4 * m_r * m_r;
        }
        double getCircumference() {
            return 8 * m_r;
        }
        bool getExtent(double x, double y) {
            if(fabs(x) < m_r && fabs(y) < m_r) {
                return true;
            }
            return false;
        }
    private:
        double m_r = 1;
};
Square::Square(double r) : Shapes() {
m_r = r;
}
```

which looks quite different from the implementation in Python. We will go thoroughly through the different part of the code.

One of the first thing we observe, is that we here need to *declare* all the variables, which means that we need to specify what kind of variable it is. In the following we will have a look at some standard *data types*.

9.2.1 Built-in data types

In low-level languages we need to specify the data types manually, giving us more control and flexibility. The most fundamental data types are `int` representing an integer number, `float` representing a floating number, `bool` representing a Boolean number and a `char` representing a character.

Often, these four data types are not sufficient for the task we want to implement and we need to introduce more data types. An especially common error is *overflow*, meaning that the number computed is out of the range of our data type. To avoid this, we can switch to data types of longer range, replacing `int` with a `long long int` (or just a `long long`), and replacing `float` with a `double`.

Some languages also deal with `long int` and `long double`, but in C++ an `int` and a `long int` is the same. Additionally, a `long double` is the same as a `double`. As lack of memory

Table 9.1: Built-in data types in C++, with their memory occupation and range. Parenthesis () means that the the extension is optional. Numbers are taken from [56].

Data types	Size [bytes]	Range
(`signed`) `short` (`int`)	2	-2^{15} to $2^{15}-1$
(`signed`) `int` / `long` (`int`)	4	-2^{31} to $2^{31}-1$
(`signed`) `long long` (`int`)	8	-2^{63} to $2^{63}-1$
`unsigned short` (`int`)	2	0 to 2^{16}
`unsigned int` / `unsigned long` (`int`)	4	0 to 2^{32}
`unsigned long long` (`int`)	8	0 to 2^{64}
`float`	4	$\sim \pm$3.4E38 ($\sim$ 7 digits)
(`long`) `double`	8	$\sim \pm$1.7E308 ($\sim$ 15 digits)
(`signed`) `char`	1	-2^{7} to $2^{7}-1$
`unsigned char`	1	0 to 2^{8}

seldomly is a problem on modern computers, it is common to declare all floating numbers as `double`s. In table (9.1), the most common data types are listed with their range and memory occupation in C++.

For integers it is also possible to "move" the range from its zero-centered default to only include positive numbers. This is done by declaring a `unsigned int` or `unsigned long`. This doubles the range in the positive direction, but should not be done if one, by any chance, can get negative numbers, as one will get *underflow*.

9.2.2 Access modifiers

Another thing that we observe from the code example above, is that we define the functions as `private` or `public`, which are called access modifiers. The access modifiers specify how much access the environment should have to the function: `private` means that the function can only be called from the particular class, while `public` means that also sub-classes (children) and other classes have access to the function.

We also have a third access modifier in C++, called `protected`. `protected` members of a class A are not accessible outside of A's code, but any class that inherit from A has access to the protected member.

9.2.3 Virtual functions

In Python, we saw how a parent's function was overwritten by a function of the child. This work as long as the child's function has identical name and identical arguments as the parent's function, otherwise it all will fail.

In C++, a much safer method exists based on virtual functions, where a child is *forced* to have the same virtual functions as its parent. The virtual functions of the parent therefore serves as a template of its children, such that all the children have the same features.

This is often very useful, as the children tend to have the same features. Taking it back to the `Shapes` class, all the shapes ought to have features area and circumference, which can be forced by using virtual functions, as done in the example above.

9.2.4 Constructors and destructors

9.2.5 Pointers

In the example above, we have not directly used any pointer. However, pointers are important in low-level languages

9.2.6 Back to the Monte-Carlo example

We can now call the class above from the `main` function. We start defining the children `circ` and `squr`, and use the `getExtent()` function to approximate the area ratio between them. The code looks like

```
int main() {
    class Shapes* circ = new Circle(1);
    class Shapes* squr = new Square(1);

    for(int i=1; i<10; i++) {
        int M = int(pow(10,i));
        long A_square = 0;
        long A_circle = 0;
        for(int m=0; i<M; i++) {
            double x = uniform(gen);
            double y = uniform(gen);
            if(circ->getExtent(x, y)) {
                A_circle ++;
            }
            if(squr->getExtent(x, y)) {
                A_square ++;
            }
        }
        std::cout << std::fixed;
        std::cout << std::setprecision(10);
        std::cout << "Pi: " << 4 * double(A_circle)/A_square << std::endl;
    }
    return 0;
}
```

which gives the same output as the Python script, but is much faster.

Chapter 10

Implementation

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> Phil Karlton, [75]

In this chapter de will describe the implemented VMC code, which was developed from scratch in C++. As the code itself is around 7000 significant[1] lines of code, we will just go through selected and often not obvious parts. As often said, *good planning is half the battle*, which largely relates to writing VMC code. The code was rewritten and restructured several times before we ended on the final version. As a starting point, we used Morten Ledum's VMC framework [53], which was meant as an example implementation in the course *FYS4411 - Computational Physics II: Quantum Mechanical Systems*. The entire source code can be found on the authors github, [69].

For all matrix operations, the open source template library for linear algebra Eigen was used throughout the code. Eigen provides an elegant interface, with support for all the needed matrix and vector operations. In addition, Eigen is built on the standard software libraries for numerical linear algebra, BLAS and LAPACK, which are incredibly fast. These contribute greatly to the performance of the code.

The code was developed in regards to three principal aims:

- flexible,
- fast,
- readable.

It needs to be flexible in order to support the Boltzmann machines as our trial wave function guess, and since we will try out various Jastrow factors it should be easy to add and remove wave function elements. Since quantum mechanical simulations in general are very expensive, it is important to develop efficient code to be able to study systems of some size. Lastly, we aim to write readable code such that others can reuse the code in its entirety or parts of it later.

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.

[1] Significant lines of code in this sense means lines that are not blank or commented. Counted by the cloc program [65].

10.1 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 Slater determinant with the Gaussian part factorized out and the Padé-Jastrow factor were implemented as three separate elements, but they all can easily be combined. The way one does this in practice, is to append multiple wave function elements to a vector `WaveFunctionElements`. A trial wave function for a quantum dot combining a Slater determinant with the Gaussian part factorized out and the Padé-Jastrow factor can be specified by a system `quantumDot` is the following way

```
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);
```

Listing 10.1: The code was implemented in a flexible way such that all the wave function elements can be combined with other wave function elements by appending all the wanted elements to a vector, here named `waveFunctionElements`, which again is set as the trial wave function using the function `setWaveFunctionElements`.

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 will be very useful when dealing with the Boltzmann machines, which will be tested with various Jastrow factors, and also with extensions like the partly restricted element. Additionally, 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). The exact derivation of all the wave function elements, with all the required derivatives, can be read in chapter 7. The con is that the program becomes slightly slower, since even canceling cross terms are calculated, but it will most likely not be noticeable. In the next section, we will look at what really matters when dealing with the efficiency of the code.

10.2 Efficiency

Even though the computers have become much faster the past decades, they are still far from being fast enough for large scale quantum computations. To be able to study interesting systems, which often are relatively large, we therefore need to focus on the efficiency in all terms. In our particular code, a majority of the run time is spent on the sampling such that this is the part we should strive to optimize.

In general, there are several points that one should consider when optimizing the code. One should

- find closed-form expressions if possible,
- never repeat calculations,
- find iterative expressions for the update of scalars, vectors and matrices.

We have already done the first point for all our wave function elements. The explicit expressions are presented in chapter 7, where we also write about possible optimization schemes for each

specific wave function element. In the next section, 10.2.1, we will go through how the wave function class is implemented with respect to the optimizations.

Furthermore, we will describe how the distance matrix is updated in section 10.2.2, which is an illustrating example on how to avoid repeating calculations. This might sounds obvious, but in complex calculations not recalculate anything is really an art.

Lastly, we will show how vectors and matrices can be updated iteratively to gain efficiency. We do this in more os less all the wave function elements, but as the update of the Slater determinant is the most comprehensive, it will be used as the example in section

10.2.1 The wave function class

The wave function class is the heart of the code, and in order to make the code fast, it needs to be implemented in a clever way. In chapter 9, we looked at how virtual functions can be used to force all the sub-classes of a class to have the same properties, which we will utilize here. Each wave function element should carry the properties `evaluateRatio` (which returns the probability ratio), `computeGradient` (which returns $\nabla_k \ln \psi_i(\boldsymbol{r})$), `computeLaplacian` (which returns $\nabla^2 \ln \psi_i(\boldsymbol{r})$) and `computeParameterGradient` (which returns $\partial_\theta \ln \psi_i(\boldsymbol{r})$). All these four expressions are given in chapter 7 for all the wave function elements. In addition, the wave function class includes a bunch of functions that are used in the update of vectors, parameters and so on. We will first present the virtual functions in the header file `wavefunction.h`, and thereafter discuss how the various functions are used in the optimization.

```
public:
    WaveFunction(class System *system);
    virtual int getNumberOfParameters() = 0;
    virtual int getGlobalArrayNeed() = 0;
    virtual std::string getLabel() = 0;

    virtual void updateParameters(const Eigen::MatrixXd parameters) = 0;
    virtual void initializeArrays(const Eigen::VectorXd positions,
                                  const Eigen::VectorXd radialVector,
                                  const Eigen::MatrixXd distanceMatrix)
        = 0;
    virtual void updateArrays(const Eigen::VectorXd positions,
                              const Eigen::VectorXd radialVector,
                              const Eigen::MatrixXd distanceMatrix,
                              const int changedCoord)
        = 0;
    virtual void setConstants(const int elementNumber) = 0;
    virtual void setArrays() = 0;
    virtual void resetArrays() = 0;
    virtual double evaluateRatio() = 0;
    virtual double computeGradient(const int k) = 0;
    virtual double computeLaplacian() = 0;
    virtual Eigen::VectorXd computeParameterGradient() = 0;

    virtual ~WaveFunction() = 0;
```

Listing 10.2: The code is taken from `wavefunction.h`, presenting all the virtual functions that are used in the wave function class. Clang formating is used.

The three former functions are getting the properties "number of parameters", "array need" and "label" respectively for each wave function element. The first one, `getNumberOfParameters`, is important when declaring the global parameter matrix, which contains all the parameters in all the elements. The `getGlobalArrayNeed` function is used to chart the need of the distance matrix, containing all the relative distances, and the radial vector, containing all the radial coordinates, $r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$. An element returns 1 if it requires the distance matrix, 2 if it requires the radial vector, 3 if it requires both and 0 if it requires none. This makes it possible to calculate

the global arrays only when they are needed. The `getLabel` function is returning the label of the respective element, used to collect a composition of elements under a common label (ex. "RBM" contains the elements "slaterdeterminant", "rbmgaussian" and "rbmproduct").

Further, the `updateParameters` is called to update the parameters in each wave function element after the global parameter matrix is updated. `initializeArrays` is used to initialize all the arrays in an element, which is important when the arrays are updated iteratively, since they will only be set once. This function will therefore only be called in the very beginning of a run. After this, all the arrays are updated using the `updateArrays` function, where all the iterative magic happens, further detailed in section 10.2.3. As this function takes the changed coordinate, `changedCoord`, as argument, it can update only the affected parts of the arrays.

We always need to store the old arrays in case a move is rejected, since we then need to go back to the old state. The function `setArrays` replaces the old arrays with the new arrays, $\boldsymbol{X}_{\text{old}} = \boldsymbol{X}_{\text{new}}$, which is done to store the arrays. When a move is rejected, we need to reset the arrays going the other way, $\boldsymbol{X}_{\text{new}} = \boldsymbol{X}_{\text{old}}$, which naturally is done by the function `resetArrays`. Finally, the function `setConstants` is called once, and gives each element a unique element number and defines the size of the global parameter matrix.

Before we proceed further, we will give an example from the code on how a specific element is implemented. We choose the

10.2.2 Updating the distance matrix

The distance matrix, which is used in the Jastrow factors, gives an illustrating example on this. The matrix, henceforth named M, contains the relative distances between all the particles, for three particles given by

$$M = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} = \begin{pmatrix} 0 & r_{12} & r_{13} \\ r_{12} & 0 & r_{23} \\ r_{13} & r_{23} & 0 \end{pmatrix} \tag{10.1}$$

where r_{12} means the distance between particles 1 and 2. Since $r_{12} = r_{21}$ and $r_{ii} = 0$, the matrix becomes symmetric with zeros on the diagonal, which means that we only need to calculate $N(N-1)/2$ elements instead of N^2. Further, we can exploit that only a particle is moved at a time, which means that only a row and a column are changed when a particle is moved. For instance, if particle 1 is moved, the upper row and the left-hand-side column in matrix M need to be updated.

In our program, we have implemented this in the following way

```
double Metropolis::calculateDistanceMatrixElement(const int i, const int j) {
    double dist = 0;
    int parti   = m_numberOfDimensions*i;
    int partj   = m_numberOfDimensions*j;
    for(int d=0; d<m_numberOfDimensions; d++) {
        double diff = m_positions(parti+d)-m_positions(partj+d);
        dist += diff*diff;
    }
    return sqrt(dist);
}

void Metropolis::calculateDistanceMatrixCross(const int particle) {
    for(int i=0; i<m_numberOfParticles; i++) {
        m_distanceMatrix(particle, i) = calculateDistanceMatrixElement(particle,
            ↪ i);
        m_distanceMatrix(i, particle) = m_distanceMatrix(particle, i);
    }
}
```

Listing 10.3: When a particle is moved, we update a row and a column in the distance matrix. The functions are parts of the Metropolis class, as the distance matrix is updated for every step in the Metropolis sampling. Taken from `metropolis.cpp`.

where the function `calculateDistanceMatrixElement(i,j)` returns element `i,j` of the matrix, which is called from the function `calculateDistanceMatrixCross(particle)`. The latter takes the moved particle index as input, and updates the necessary row and column of the matrix.

For systems of non-interacting particles, the distance matrix is redundant, and should therefore not be calculated. We have solved this by giving all the wave function elements and the Hamiltonians a number which indicated whether they require the distance matrix or not. If no part of the code needs the distance matrix, it is never calculated.

We also calculate the radial position, $r_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$, when it is required by any part of the code. There components are stored in a vector named `radialVector`, applying the same optimization ideas as the distance matrix.

10.2.3 Iterative update of the Slater determinant

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.

```
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)));
}
```

Listing 10.4: from `gaussian.cpp`

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

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

```
    m_probabilityRatio      = m_probabilityRatioOld;
}
```

Listing 10.5: from `gaussian.cpp`

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.

```
NEED TO ADD EXACT IMPLEMENTATION
```

Listing 10.6: from `system.cpp`

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

10.3 Readability

Whenever writing code, the code should be not only be understandable for the computer, but also for humans. ClangFormat...

To maximize the readability, we developed a highly object oriented code based on the theory in chapter (9). 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

```
#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;
};
```

Listing 10.7: `wavefunction.h`

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 Ref.[68].

10.3.1 Energy calculation

The way we calculate the total kinetic energy then is based on the theory presented in chapter 7, where we explain that

$$T = -\frac{1}{2}\frac{1}{\Psi_T}\nabla_k^2\Psi_T = -\frac{1}{2}\left[\sum_{i=1}^{p}\nabla_k^2 \ln\phi_i + \left(\sum_{i=1}^{p}\nabla_k \ln\phi_i\right)^2\right]. \tag{10.2}$$

The corresponding implementation thus reads

```
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;
}
```

Listing 10.8: from `system.cpp`

10.3.2 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

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

Listing 10.9: from `system.cpp`

10.3.3 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`:

```
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;
}
```

Listing 10.10: from `system.cpp`

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 7.

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.

10.4 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.

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.

10.5 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

10.6 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.

10.6.1 Super classes

10.6.1.1 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.

10.6.1.2 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.

10.6.1.3 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()`.

10.6.1.4 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`.

10.6.1.5 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.

10.6.1.6 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 (8) for details on how the optimization methods work.

10.6.1.7 The `Plotter` class

Not sure if I will keep this as a class

10.6.1.8 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 (10.5).

10.6.1.9 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.

10.6.2 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.

10.7 Running the code

The software is freely available on github.com/evenmn/VMC under MIT license, summarized with "do whatever you want". Officially, the MIT license says [79]

"Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software"

We encourage reuse of the code.

10.7.1 Download

The code can be downloaded from github by cloning the repository github.com/evenmn/VMC. Prerequisites are MPI, Eigen and the automatized blocking code developed by Marius Jonsson.

- In Linux, MPI can be installed by
  `sudo apt-get install libopenmpi-dev`
  `sudo apt-get install openmpi-bin`
- To install Eigen, see http://eigen.tuxfamily.org/.
- The automatized blocking code can be cloned from https://github.com/computative/block.

10.7.2 Build

The code can either be built using CMake or QMake in QT-creator.

10.7.2.1 CMake

Go to the VMC prefix, and run the four commands

```
mkdir build
cd build
cmake ../
make -j4
```

or one can simply run `./CompileVMC`. The executable is then found in the `build` folder.

10.7.2.2 QMake (QT-Creator)

- Download QT-Creator from https://www.qt.io/download.
- Configure `QMC.pro`.

After the configuration, the executable can be found in the building folder, by default named `build-VMC-...`.

10.7.3 Run

Before the code is run, the system of interest should be specified. There are multiple ways of doing this: one can either use the provided `main.cpp` with all possible settings, create a new more minimalistic `main.cpp` or use a configuration to specify the system.

10.7.3.1 Create `main.cpp`

The simplest possible main file one can write, is probably

```
int numberOfDimensions =  2;
int numberOfParticles = 6;
int numberOfSteps = int(pow(2, 20));
int maxNumberOfIterations = 1000;
double learningrate = 0.1;
double omega = 1.0;

System *QD = new System();

QD->setNumberOfParticles(numberOfParticles);
QD->setNumberOfDimensions(numberOfDimensions);
QD->setNumberOfMetropolisSteps(numberOfSteps);
QD->setFrequency(omega);
QD->setLearningRate(learningRate);

QD->setBasis(new Hermite(QD));

std::vector<class WaveFunction *> waveFunctionElements;
waveFunctionElements.push_back(new Gaussian(QD));
waveFunctionElements.push_back(new SlaterDeterminant(QD));
waveFunctionElements.push_back(new PadeJastrow(QD));

QD->setWaveFunctionElements(waveFunctionElements);
QD->setHamiltonian(new HarmonicOscillator(QD));
QD->runIterations(maxNumberOfIterations);
```

where we look at a two-dimensional circular quantum dot containing 6 electrons using a standard variational Monte-Carlo calculation. For more inspiration, see the provided `main.cpp`. When the main file is implemented, the code can be run using

```
mpirun -n 4 build/dotnet
```

where `dotnet` is the executable and we run 4 parallel processes.

10.7.3.2 Write configuration file

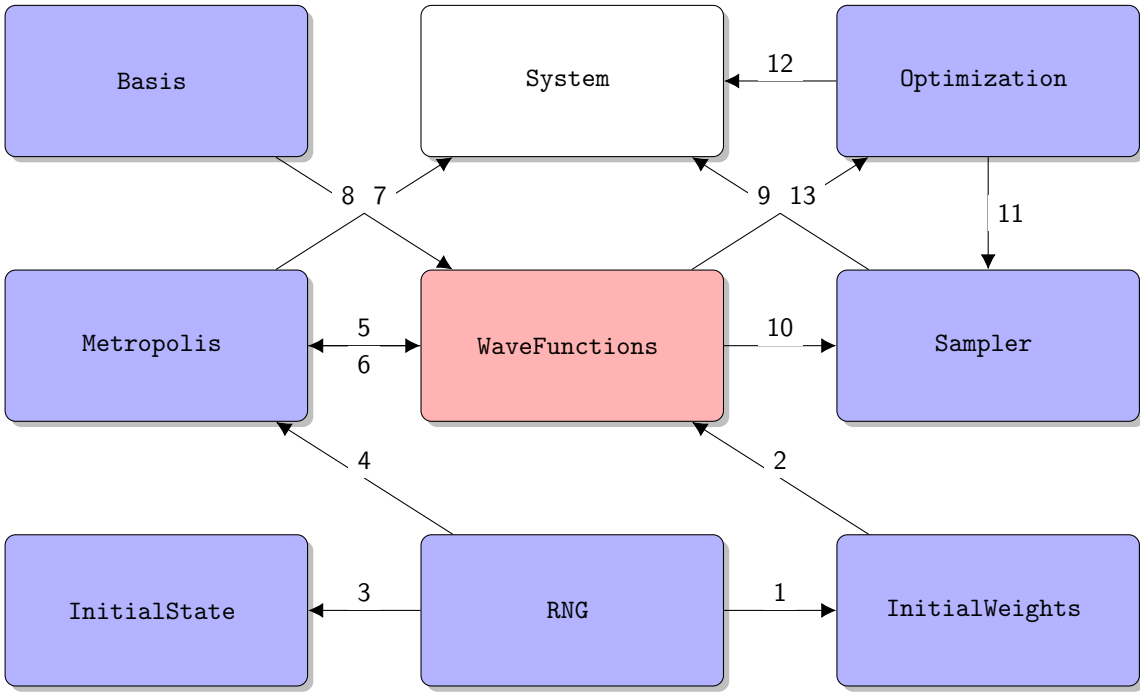
Second, one can specify the system partly or entirely by a configuration file. The config file overwrites the settings in main. We have provided two config files, `config` and `config_extended`, where the first one is limited and is dependent on settings in main, while the second contains all possible settings. We can specify a corresponding system to the one above, using the configuration file

```
numDimensions: 2
numParticles: 6
numSteps: 1048576
numIterations: 1000
learningRate: 1.0
omega: 1.0
basis: hermite
waveFunction: VMC
hamiltonian: harmonicOscillator
```

and run the code using

```
mpirun -n 4 build/dotnet config
```

where we run 4 parallel processes, `dotnet` is the executable and `config` is the configuration file.



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 10.1: 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.

Chapter 11

Results

Great quote.

Author

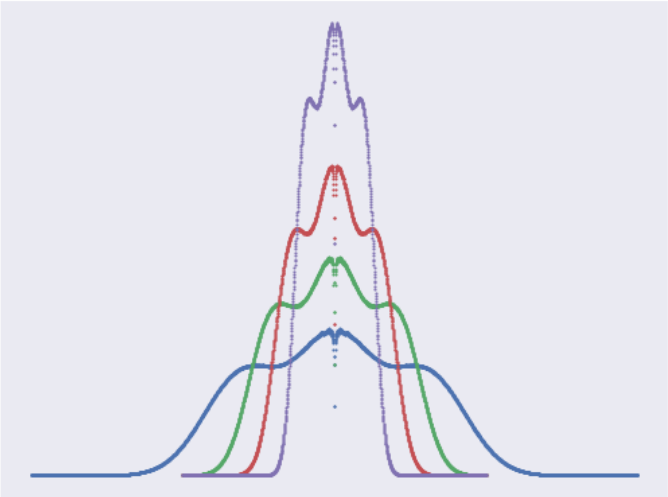
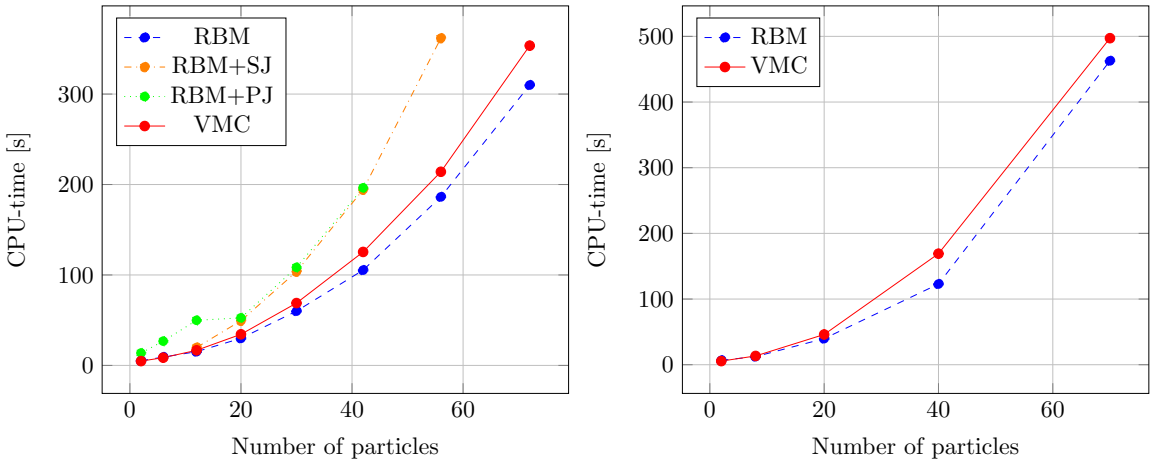


Figure 11.1: One-body density plots for a two-dimensional single quantum dot containing 12 electrons, popularly called an artificial Magnesium atom. The four graphs correspond to four different oscillator frequencies, where the weakest oscillator gives the broadest density distribution. It's quite artistic, 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.



(a) 2D **(b)** 3D

Figure 11.2: CPU-time per iteration as a function of number of particles for two dimensions (a) and three dimensions (b). The solid line is associated with standard variational Monte-Carlo (VMC), while the dashed lines are associated with restricted Boltzmann machines without Jastrow factor (RBM), with simple Jastrow factor (RBM+SJ) and with Padé-Jastrow factor (RBM+PJ).

11.1 Computational cost

One of the major problems when performing quantum many-body simulations is the computational cost, which explodes as the system size increases. In figure (11.2) the CPU-time is plotted as a function of the 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 in the RBM we have $N \cdot D \cdot (1 + H) + H$ 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 was found to give the lowest energies when testing on small systems. [64]

With our choice of hidden nodes, we end up with 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, and the very efficient LAPACK and BLAS packages lie behind the matrix operations. For RBM+SJ and RBM+PJ, the cost is significantly higher because of the update of 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

Table 11.1: 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).

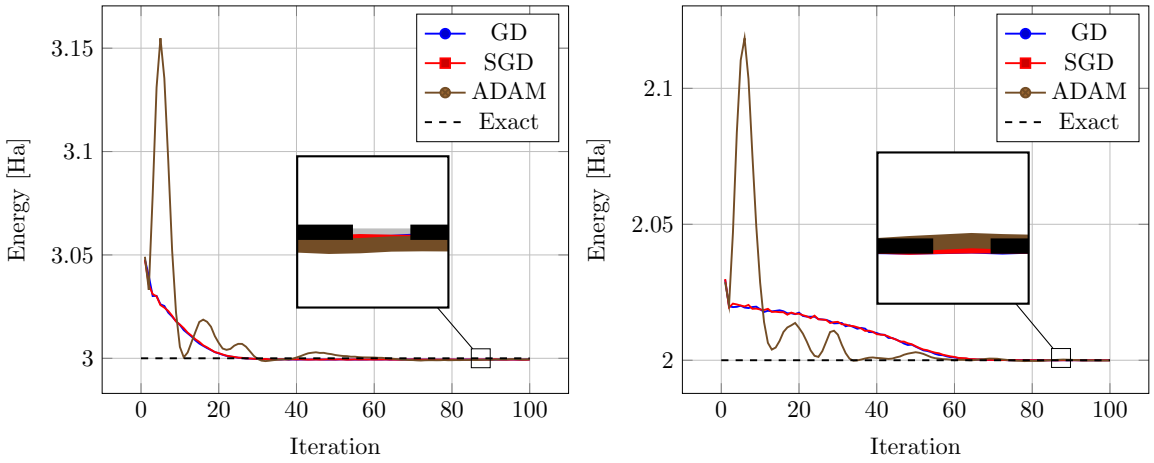
	2D		3D	
	α	β	α	β
RBM	0.0840	1.92	0.0302	2.268
RBM+SJ	-	-	-	-
RBM+PJ	-	-	-	-
VMC	0.111	1.88	0.148	1.91

the unknown parameters to be found. From the limited number of points, we have found the parameters and presented them in table (11.1). 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.

11.2 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 (11.3) we compare standard gradient descent to stochastic gradient descent and ADAM for two interacting electrons in a two- and three dimensional well. The frequency $\omega = 1.0$ is used for the two-dimensional case since we know that the exact energy is $E = 3.0$ for this case. [19] Similarly, we use the frequency $\omega = 0.5$ for the three-dimensional case since the exact energy is $E = 2.0$. [22]

The first thing we observe, is that all the three optimization tools manage to converge to the exact energy. The stochastic and non-stochastic gradient descent methods are hard to distinguish, they both converge smoothly. The ADAM optimizer, on the other hand, fluctuates much more. This behavior can be described by the momentum, as discussed in section 8.1.3. It is also important to remember that this is the case when we use standard variational Monte-Carlo wave function. The ADAM optimizer is known to be good at machine learning problems, so we will stick to it even though gradient descent seems like a clever choice seen from the figure.



(a) 2D, $\omega = 1.0$ **(b)** 3D, $\omega = 0.5$

Figure 11.3: Energy convergence for circular quantum dots where we use the optimization tools gradient descent (GD), stochastic gradient descent with 10 batches (SGD) and the ADAM optimizer. The plot in (a) shows a two-dimensional quantum dot of frequency $\omega = 1.0$ containing two interacting electrons with exact energy $E = 3.0$. The plot in (b) shows a three-dimensional quantum dot of frequency $\omega = 0.5$ containing two interacting electrons with exact energy $E = 2.0$. We use a 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 $M = 2^{24} = 16,777,216$. All energies are given in units of $\hbar$ (Hartree units).

11.3 No repulsive interaction

We start with the non-interacting case in order to validate the implemented code. For this case, we know the exact energy and the exact one-body density for both quantum dots and atoms, which makes it a good test for both systems. The physical significance is though limited, such systems do not appear in the real world. We will first present the quantum dots, before we move on to atoms.

11.3.1 Quantum dots

11.3.1.1 Ground-state energy

The single quantum dot has analytical ground-state energies given by equation (3.4) and a number particles given by the magic numbers in equation (3.5). For some selected number of electrons and for frequencies $\omega = 0.5$ and $\omega = 1.0$ the obtained energies are presented in table (11.2).

We observe that both the standard VMC, with $\alpha = 1$, and standard RBM, with all parameters set to zero, are able to reproduce the analytical expression. This is as expected, since the exact wave functions are found when the parameters have these particular values.

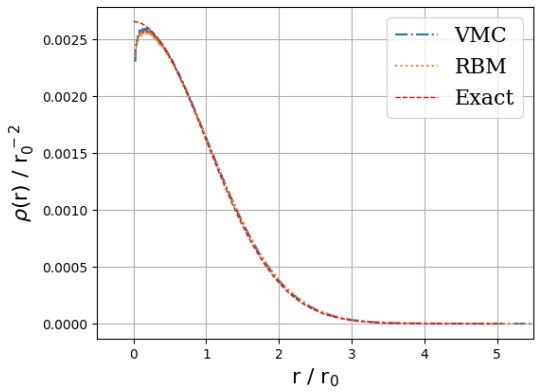
11.3.1.2 One-body density

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 (11.4) the one-body densities are plotted for quantum dots of two non-interacting electrons in two and three dimensions and frequencies $\omega = 0.5$ and $\omega = 1.0$. The analytical one-body densities are found from the definition of one-body density in equation (2.24).

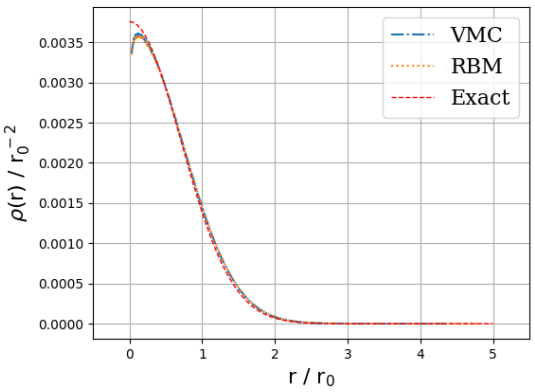
Table 11.2: Energy of circular quantum dots of frequency $\omega = 0.5$ and $\omega = 1.0$ consisting of N non-interacting particles. RBM is a single Slater determinant with a plain Boltzmann machine baked in, while VMC is a standard variational Monte-Carlo Slater determinant. Exact values are obtained by $E = \omega(n + 1/2)$, and all values are given in units of $\hbar$.

		$\omega = 0.5$			$\omega = 1.0$		
	N	RBM	VMC	Exact	RBM	VMC	Exact
2D	2	1.0	1.0	1	2.0	2.0	2
	12	14.0	14.0	14	28.0	28.0	28
	30	55.0	55.0	55	110.0	110.0	110
3D	2	1.5	1.5	1.5	3.0	3.0	3
	20	30.0	30.0	30	60.0	60.0	60
	70	157.5	157.5	157.5	315.0	315.0	315

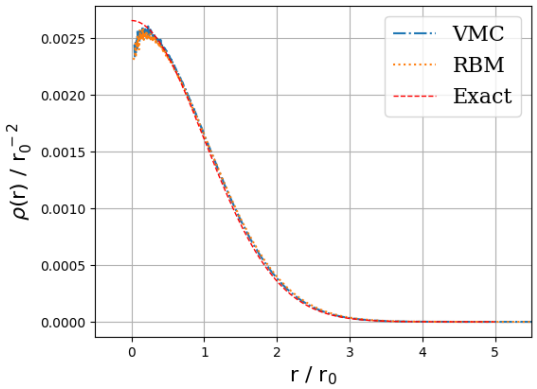
We observe that both the standard variational Monte-Carlo wave function and the restricted Boltzmann machine reproduce the analytical one-body density. The distribution gets narrower as the frequency is increased, and we also observe that the distributions are identical for two- and three dimensions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



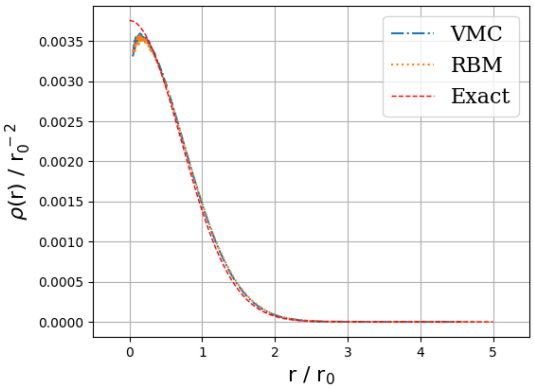
(a) 2D, $\omega = 0.5$



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



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



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

Figure 11.4: One-body densities of two non-interacting electrons in two- and three dimensions for $\omega = 0.5$ and $\omega = 1.0$.

11.3.2 Double quantum dots

For the double quantum dots, we can also find analytical energies without interaction, as explained in section 3.2. We first find the matrix

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

for a satisfying number of harmonic oscillator functions. All the symbols are detailed in section 3.2. The energies are the eigenvalues of the matrix, which were obtained by the script `doublewell_functions.py`. For one particle in one-dimensional potentials, we get the energy $E = 0.3095$ for $\omega = 1.0$ and $b = 2$, and $E = 0.1916$ for $\omega = 0.5$ and $b = 4$. This gives the analytical energies presented in table (11.3).

Table 11.3: Energy of double quantum dots of N non-interacting electrons. RBM is a single Slater determinant with a plain Boltzmann machine baked in, while VMC is a standard variational Monte-Carlo Slater determinant.

D	ω	b	RBM	VMC	Exact
2D	0.5	4.0	-	-	0.8832
	1.0	2.0	-	-	1.619
3D	0.5	4.0	-	-	1.3832
	1.0	2.0	-	-	2.619

11.3.3 Atoms

The second system we will use for validation is atoms. The energy of an atoms containing non-interacting electrons is given by the Bohr formula presented in equation (3.17). In table (11.4), the lowest closed-shell atoms He, Be and Ne are listed with their exact energy and the obtained energy from the code. In addition, we added calculations of the Hydrogen ground-state energy as another test.

Table 11.4: Energy of atoms of N non-interacting electrons. RBM is a single Slater determinant with a plain Boltzmann machine baked in, while VMC is a standard variational Monte-Carlo Slater determinant. The variance is zero to machine precision for all listed results.

Atom	N	RBM	VMC	Exact
H	1	-	-0.5	-0.5
He	2	-	-4.0	-4
Be	4	-	-20.0	-20
Ne	10	-	-200.0	-200

11.4 Quantum Dots

We now move on to the more 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.

Initially we look at standard quantum dots of size up to $N = 56$ electrons in 2D and $N = 40$ electrons in 3D and frequencies between $\omega = 1.0$ and $\omega = 0.1$. For those systems, we will compute the ground state energy, the one-body density and the two-body density.

After that, we move on to some special cases where the dots have low frequency ($\omega = 0.01$), the total spin projection $S = \sum_i m_s$ is unlike zero, the quantum dots are in a medium or the number of electrons is large. For those systems, we will typically focus on either the ground-state energy or the one-body density dependent on what we want to investigate. For instance, we will focus on the one-body density for low frequency dots because of the search for Wigner crystals.

11.4.1 Ground-state energy

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. [19][22]

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. [44]. 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. [63]

Ground state energy computations of two- and three dimensional quantum dots are found in tables (11.5) and (11.6) 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. This is as expected, but considering that no Jastrow factor is added to take care of the interactions, the result is not bad. It is overall lower than the Hartree-Fock limit.

When we add more intuition in form of a simple Jastrow factor, the energy drops significantly. The RBM+SJ is not very far from reproducing the reference, and the RBM+PJ is even closer and on the same level as VMC. The RBM+PJ and VMC also give the lowest variances.

In three dimensions, all the methods give smaller errors with respect to the references compared to two dimensions, which is because we have more degrees of freedom.

We made effort in trying to get the deep Boltzmann machine and the partly restricted Boltzmann machines to work, but they did not behave nicely.

Table 11.5: 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), the Hartree-Fock limit (HF), standard variational Monte-Carlo with Hartree-Fock basis (VMC+HF), standard variational Monte-Carlo with Hermite basis (VMC), diffusion Monte-Carlo (DMC) and semi-analytical results (Exact). Hartree-Fock results are taken from Ref.[63], DMC results are taken from [44] and semi-analytical results are taken from [22]. 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	HF (Ref.[63])	VMC+HF	VMC	DMC (Ref.[44])	Exact (Ref.[22])
2	0.1	0.4728(1)	0.44858(6)	0.440975(8)	0.525635	-	0.44129(1)	0.44079(1)	
	1/6	0.7036(1)	0.67684(7)	0.66715(6)	0.768675	-	0.66710(1)	-	2/3
	0.28	1.0707(2)	1.03470(7)	1.021668(7)	1.14171	-	1.02192(1)	1.02164(1)	
	0.5	1.7234(2)	1.67739(9)	1.659637(6)	1.79974	-	1.65974(1)	1.65977(1)	
	1.0	3.0829(2)	3.0259(1)	2.999587(5)	3.16190	-	2.99936(1)	3.00000(1)	3.0
6	0.1	3.697(1)	3.6584(4)	3.5700(2)	3.85238	-	3.5695(1)	3.55385(5)	
	0.28	7.9273(9)	7.7503(4)	7.6203(2)	8.01957	-	7.6219(1)	7.60019(6)	
	0.5	12.241(1)	11.9659(5)	11.8074(2)	12.2713	-	11.8104(2)	11.78484(6)	
	1.0	20.716(1)	20.4061(7)	20.1832(2)	20.7192	-	20.1918(2)	20.15932(8)	
12	0.1	12.705(2)	12.566(2)	12.3416(4)	12.9247	-	12.3196(3)	12.26984(8)	
	0.28	26.389(2)	26.083(1)	25.7331(5)	26.5500	-	25.7049(4)	25.63577(9)	
	0.5	40.440(3)	39.694(1)	39.2743(6)	40.2161	-	39.2421(5)	39.1596(1)	
	1.0	67.632(3)	66.378(2)	65.7911(7)	66.9113	-	65.7928(5)	65.7001(1)	

20	0.1	30.824(2)	30.567(3)	30.1553(9)	31.1902	-	30.086(1)	29.9779(1)
	0.28	63.746(4)	62.811(3)	62.148(1)	63.5390	-	62.0755(7)	61.9268(1)
	0.5	97.166(5)	94.920(4)	94.104(1)	95.7328	-	94.0433(9)	93.8752(1)
	1.0	159.640(5)	157.209(4)	156.104(1)	158.004	-	156.102(1)	155.8822(1)
30	0.1	61.829(5)	61.351(4)	60.774(2)		-	60.585(1)	60.4205(2)
	0.28	126.958(6)	126.067(5)	124.437(2)		-	124.195(2)	123.9683(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)

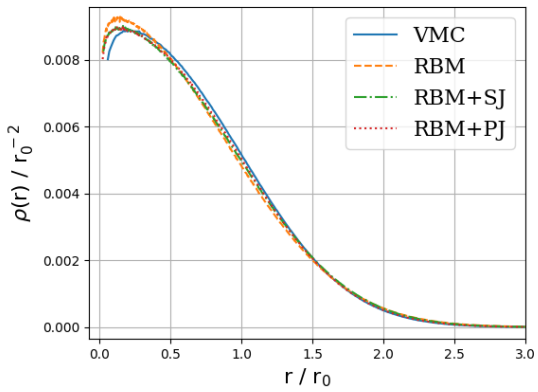
Table 11.6: The ground state energy of three-dimensional circular quantum dots of frequency ω obtained by various methods. The column on the left-hand-side represents restricted Boltzmann machine (RBM), followed by restricted Boltzmann machine with simple Jastrow factor (RBM+SJ), restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ), the Hartree-Fock limit (HF), standard variational Monte-Carlo with Hartree-Fock basis (VMC+HF), standard variational Monte-Carlo with Hermite basis (VMC), diffusion Monte-Carlo (DMC) and semi-analytical results (Exact). Hartree-Fock results are taken from Ref.[63], DMC results are taken from [44] and semi-analytical results are taken from [19]. 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	HF (Ref.[63])	VMC+HF	VMC	DMC (Ref.[44])	Exact (Ref.[19])
2	0.1	0.5178(1)	0.50214(3)	0.500080(6)	0.529065	-	0.500083(7)	0.499997(3)	0.5
	0.28	1.2259(1)	1.20475(4)	1.201717(6)	1.23722	-	1.201752(6)	1.201725(2)	
	0.5	2.0269(1)	2.00371(4)	1.999912(5)	2.03851	-	1.999977(5)	2.000000(2)	2.0
	1.0	3.7571(1)	3.73543(4)	3.729827(5)	3.77157	-	3.730030(5)	3.730123(3)	
8	0.1	6.549(7)	5.7498(4)	5.8448(7)	5.86255	-	5.7126(1)	5.7028(1)	
	0.28	13.098(2)	12.2492(4)	12.2056(2)	12.3987	-	12.2050(2)	12.1927(1)	
	0.5	19.487(2)	19.0241(4)	18.9747(2)	19.1916	-	18.9759(1)	18.9611(1)	
	1.0	33.302(1)	32.739(4)	32.6820(2)	32.9246	-	32.6863(2)	32.6680(1)	
20	0.1	27.813(2)	27.470(1)	29.425(9)	-	-	27.3144(5)	27.2717(2)	
	0.28	57.700(4)	56.600(1)	-	-	-	56.4297(5)	56.3868(2)	
	0.5	87.840(4)	85.893(1)	-	-	-	85.7161(5)	85.6555(2)	
	1.0	146.292(4)	143.209(2)	-	-	-	142.9560(7)	142.8875(2)	
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)		

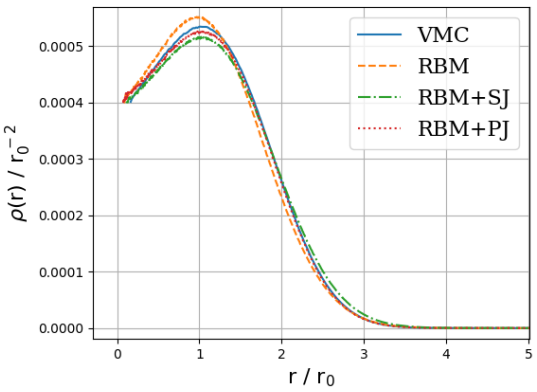
11.4.2 One-body density

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 simple Jastrow factor (RBM+SJ), a restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ) and standard variational Monte-Carlo (VMC) for frequencies $\omega = [1.0, 0.5, 0.1]$ and up to 42 electrons in two dimensions and 40 electrons in three dimensions. The plots for two- and three-dimensional dots can be found in figures (11.5-11.7) and **??** respectively.

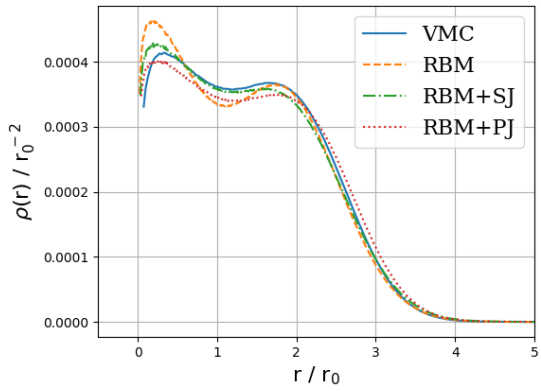
We observe that the density gets a wave shape with more peaks as the number of particles increases. The plain RBM tends to exaggerate the wave form, especially when the frequency gets lower, which is due to the lack of a Jastrow factor. The interaction gets more dominating as the frequency decreases, and the presence of Jastrow factor to handle the interactions is then important. We also see that the Padé-Jastrow factor works better than the simple Jastrow factor in all cases as expected.



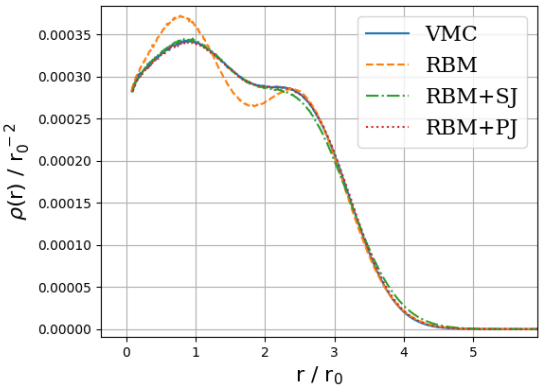
(a) 2P, $\omega = 1.0$



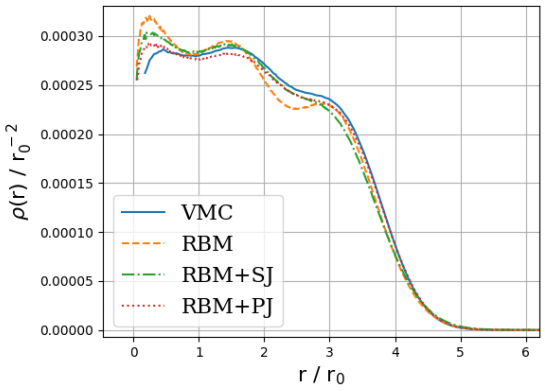
(b) 6P, $\omega = 1.0$



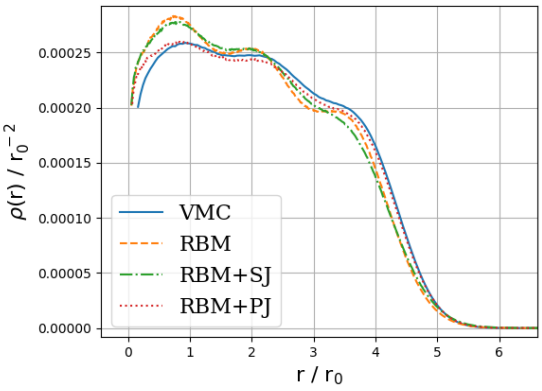
(c) 12P, $\omega = 1.0$



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

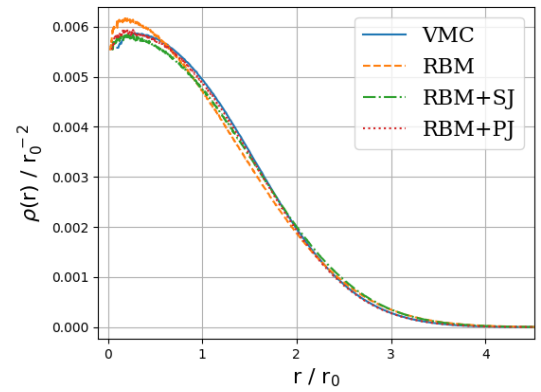


(e) 30P, $\omega = 1.0$

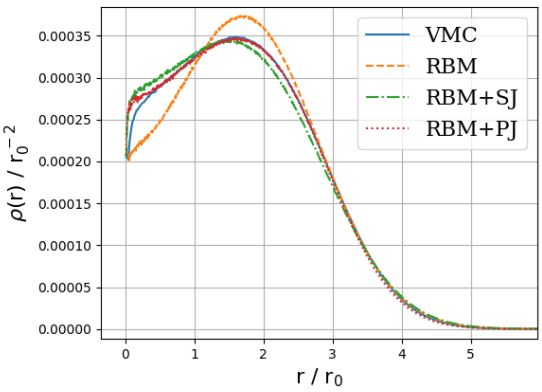


(f) 42P, $\omega = 1.0$

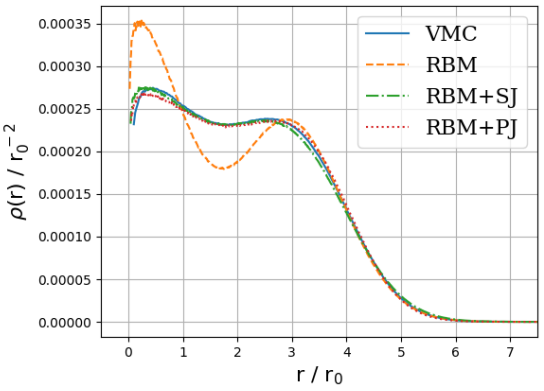
Figure 11.5: One-body density plots for two-dimensional circular quantum dots of 2, 6, 12, 20, 30 and 42 interacting electrons for frequencies $\omega = 1.0$ produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM), restricted Boltzmann machine with simple Jastrow factor (RBM+SJ) and restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ). ADAM optimizer was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268,435,456$.



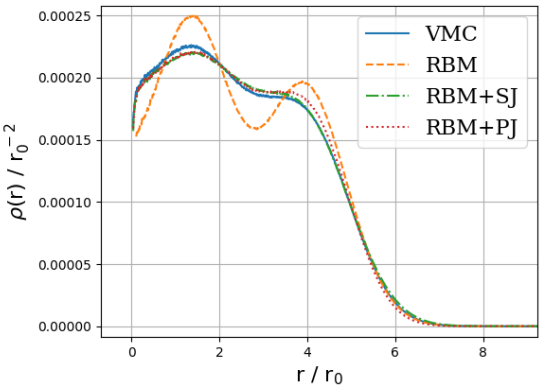
(a) 2P, $\omega = 0.5$



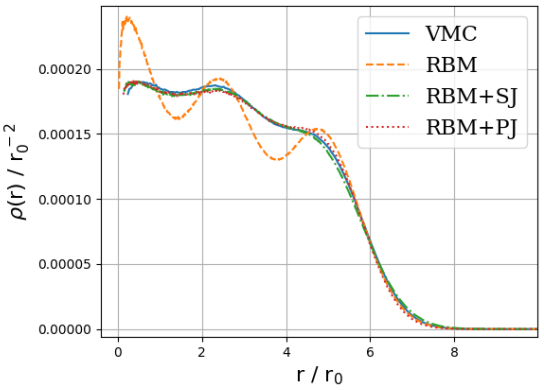
(b) 6P, $\omega = 0.5$



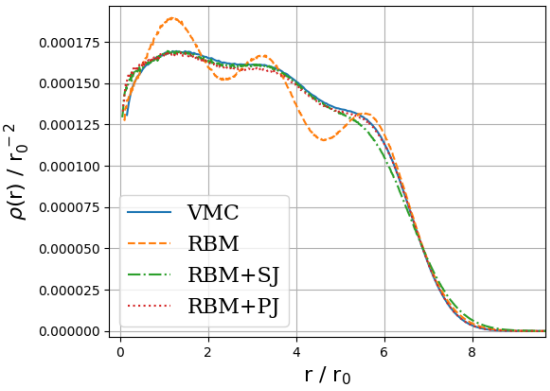
(c) 12P, $\omega = 0.5$



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

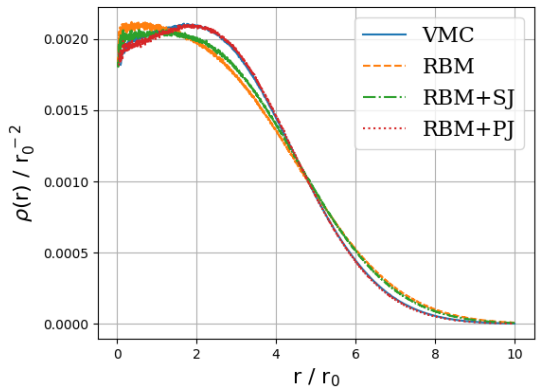


(e) 30P, $\omega = 0.5$

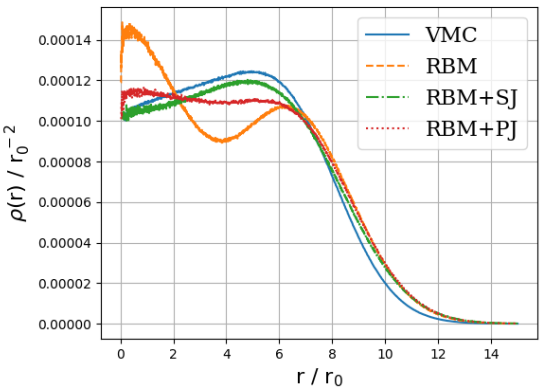


(f) 42P, $\omega = 0.5$

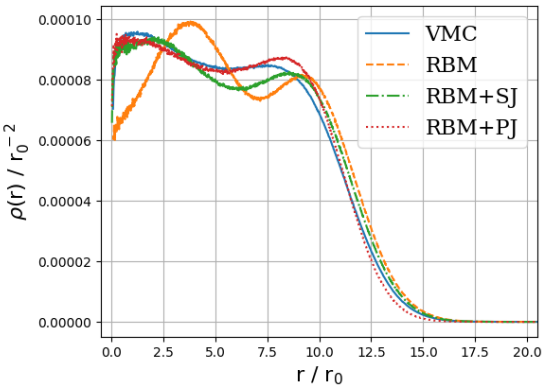
Figure 11.6: One-body density plots for two-dimensional circular quantum dots of 2, 6, 12, 20, 30 and 42 interacting electrons for frequencies $\omega = 0.5$ produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM), restricted Boltzmann machine with simple Jastrow factor (RBM+SJ) and restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ). ADAM optimizer was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268,435,456$.



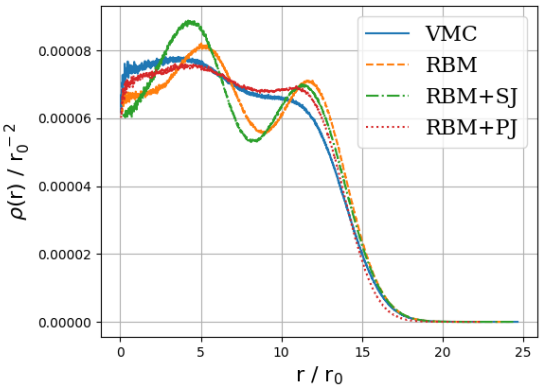
(a) 2P, $\omega = 0.1$



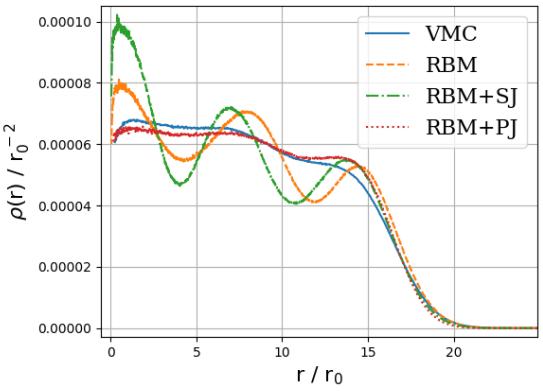
(b) 6P, $\omega = 0.1$



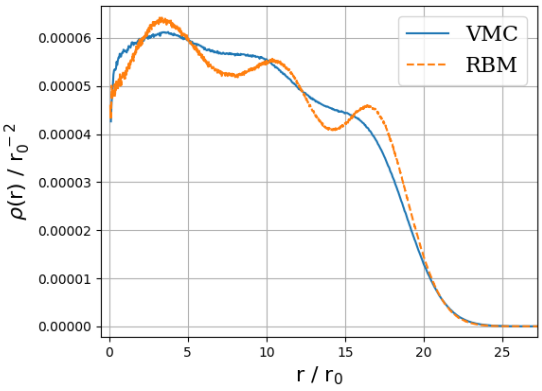
(c) 12P, $\omega = 0.1$



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



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

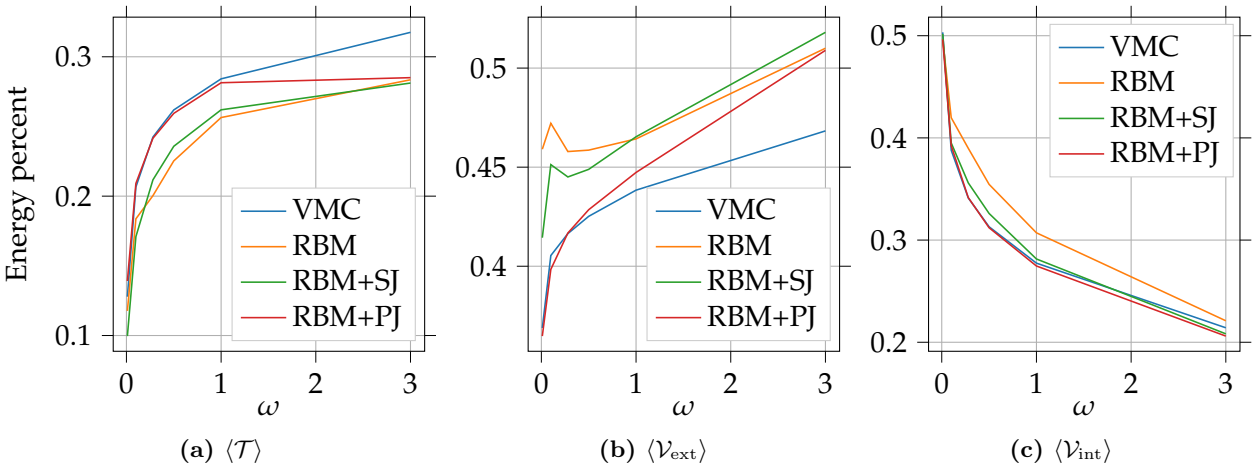


(f) 42P, $\omega = 0.1$

Figure 11.7: One-body density plots for two-dimensional circular quantum dots of 2, 6, 12, 20, 30 and 42 interacting electrons for frequencies $\omega = 0.1$ produced by standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM), restricted Boltzmann machine with simple Jastrow factor (RBM+SJ) and restricted Boltzmann machine with Padé-Jastrow factor (RBM+PJ). ADAM optimizer was used, and after convergence the number of Monte-Carlo cycles was $MC = 2^{28} = 268,435,456$.

11.4.3 Energy distribution

It is also interesting to investigate how the energy is distributed between kinetic energy, external energy and interaction energy for the different methods and various oscillator frequencies. This makes us able to check if the results agree with the virial theorem presented in section 2.1.5, and it is also interesting to see if the different methods have different energy distribution. In figure (11.8), the kinetic energy, external potential energy and interaction energy are plotted as a function of oscillator frequency for a two-dimensional quantum dot containing two electrons.



(a) $\langle \mathcal{T} \rangle$ **(b)** $\langle \mathcal{V}_{\text{ext}} \rangle$ **(c)** $\langle \mathcal{V}_{\text{int}} \rangle$

Figure 11.8: In figure (a), the kinetic energy over total energy, $\langle \mathcal{T} \rangle / \langle \mathcal{H} \rangle$, of two-dimensional quantum dots containing two electrons are plotted as a function of the oscillator frequency ω. Similar plots for the external potential energy and interaction energy are found in figure (b) and (c) respectively. The methods used are standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM), restricted Boltzmann machine with simple Jastrow factor (RBM+SJ) and restricted Boltzmann machine with Padé-Jastrow factor.

One can observe that the kinetic energy and potential energy are dominant for high frequencies, but as the frequency is decreased the interaction energy gets ever more important. At frequency $\omega = 0.01$, the interaction energy actually accounts for 50% of the energy, while the kinetic energy accounts for about 10% of the energy. We predicted this already in section 2.3.3.

We also see a significant difference between the methods, where the plain restricted Boltzmann machine is the one that really stands out. This is not very surprising, considering that it does not have any Jastrow factor to take care of the interactions. This apparently gives a higher interaction energy and a lower kinetic energy.

If we now recall the virial theorem in section 2.1.5, it states that the kinetic energy is related to the potential energies in a certain way. For the interaction quantum dot system, the virial theorem reads

$$2\langle \mathcal{T} \rangle = 2\langle \mathcal{V}_{\text{ext}} \rangle - \langle \mathcal{V}_{\text{int}} \rangle \tag{11.1}$$

which can be verified from the energy splitting. To do this accurate, we better list up the exact energies, which is done in tables (11.7),(11.8) for some selected frequencies for quantum dots containing two electrons in two- and three dimensions respectively. By doing the math, we see that the virial theorem is satisfied for low frequencies, but as the frequency increases the theorem gets more and more off. This is the case for all the methods.

Table 11.7: This table shows how the total energy ($\langle \mathcal{H} \rangle$) is distributed between kinetic energy ($\langle \mathcal{T} \rangle$), external potential energy ($\langle \mathcal{V}_{\text{ext}} \rangle$) and interaction energy ($\langle \mathcal{V}_{\text{int}} \rangle$) of two-dimensional circular quantum dots at a wide range of frequencies ω and two interacting electrons. The methods used are standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM), restricted Boltzmann machine with a simple Jastrow factor (RBM+SJ) and restricted Boltzmann machine with Padé-Jastrow factor. The energy is given in units of $\hbar$, and the numbers in parenthesis are the statistical uncertainties in the last digit.

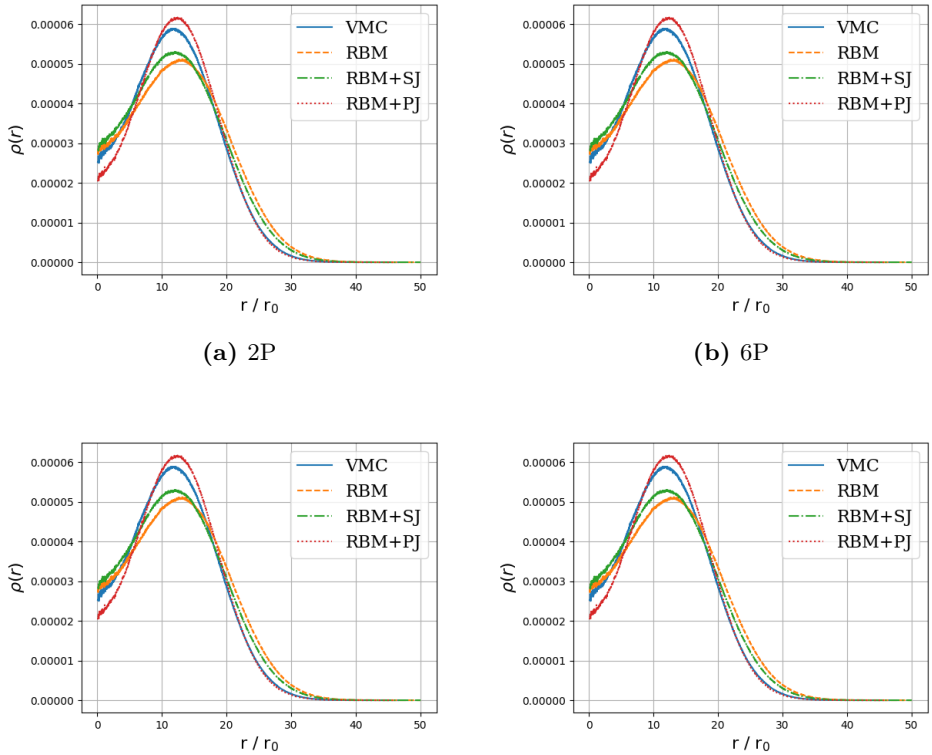
	ω	$\langle \mathcal{H} \rangle$	$\langle \mathcal{T} \rangle$	$\langle \mathcal{V}_{\text{ext}} \rangle$	$\langle \mathcal{V}_{\text{int}} \rangle$
RBM	0.01	0.07954(7)	0.00872(2)	0.03402(9)	0.0368(1)
	0.5	1.7234(2)	0.3739(2)	0.7611(3)	0.5884(3)
	3.0	7.9968(4)	2.2346(5)	4.0201(7)	1.7422(4)
RBM+SJ	0.01	0.075267(3)	0.00738(2)	0.03071(8)	0.03718(8)
	0.5	1.67739(9)	0.3913(2)	0.7450(3)	0.5411(2)
	3.0	7.9409(1)	2.2162(5)	4.0834(7)	1.6413(3)
RBM+PJ	0.01	0.074107(8)	0.01031(3)	0.02703(4)	0.03677(3)
	0.5	1.659637(6)	0.4305(2)	0.7112(2)	0.5179(2)
	3.0	7.882355(8)	2.2466(6)	4.0117(6)	1.6240(3)
VMC	0.01	0.074070(8)	0.00947(3)	0.02732(5)	0.03728(4)
	0.5	1.65974(1)	0.4346(2)	0.7057(2)	0.5195(2)
	3.0	7.881906(9)	2.5028(5)	3.6911(6)	1.6880(3)

Table 11.8: This table shows how the total energy ($\langle\mathcal{H}\rangle$) is distributed between kinetic energy ($\langle\mathcal{T}\rangle$), external potential energy ($\langle\mathcal{V}_{\text{ext}}\rangle$) and interaction energy ($\langle\mathcal{V}_{\text{int}}\rangle$) of three-dimensional circular quantum dots at a wide range of frequencies ω and two interacting electrons. The methods used are standard variational Monte-Carlo (VMC), plain restricted Boltzmann machine (RBM), restricted Boltzmann machine with a simple Jastrow factor (RBM+SJ) and restricted Boltzmann machine with Padé-Jastrow factor. The energy is given in units of $\hbar$, and the numbers in parenthesis are the statistical uncertainties in the last digit.

	ω	$\langle\mathcal{H}\rangle$	$\langle\mathcal{T}\rangle$	$\langle\mathcal{V}_{\text{ext}}\rangle$	$\langle\mathcal{V}_{\text{int}}\rangle$
RBM	0.01	-	-	-	-
	0.5	2.0269(1)	0.6595(3)	0.8778(4)	0.4896(2)
	3.0	-	-	-	-
RBM+SJ	0.01	0.07994(2)	0.01069(3)	0.03190(8)	0.03735(8)
	0.5	2.00371(4)	0.6340(3)	0.9201(4)	0.4496(2)
	3.0	10.32289(5)	3.5281(8)	5.6147(9)	1.1801(2)
RBM+PJ	0.01	0.079312(6)	0.01283(4)	0.02987(6)	0.03661(4)
	0.5	1.999912(5)	0.6515(3)	0.9040(3)	0.4444(1)
	3.0	10.31271(4)	3.5410(8)	5.5989(9)	1.1728(2)
VMC	0.01	0.079284(6)	0.01221(4)	0.039757(6)	0.036319(4)
	0.5	1.999977(5)	0.6517(3)	0.9032(3)	0.4451(1)
	3.0	10.31717(1)	3.8365(7)	5.2770(8)	1.2037(2)

11.4.4 Low frequency dots

As we pointed out in section 2.3.3, and as shown in the previous section, the potential energy dominates the kinetic energy as low frequencies. This means that the electrons have a lower average momentum, which makes Wigner crystallization possible. As already discussed in the theory part, this phenomenon can be observed by some distinct peaks in the one-body density plots. In figure (11.9), the one-body density is plotted for low frequency dots containing 2, 6, 12 and 20 particles.



(a) 2P **(b)** 6P

(c) 12P **(d)** 20P

Figure 11.9: One-body density of two-dimensional circular quantum dots. $\omega = 0.01$.

For two particles, we can see that the one-body density has a peak at a distance from origin, and a local minimum in origin. This means that the probability of finding the two electrons close to each other is low, which is as expected due to the low momentum.

11.4.5 Large dots

In order to test the code, we also decided to run for systems with $N \geq 56$. This has no scientific significance, other than testing how far a VMC code can go. We look at weakly interacting particles, so we will not get any relativistic effects even when adding many particles.

Table 11.9: Energy of large circular quantum dots, $\omega = 1.0$. All energies are given in units of $\hbar$, and the numbers in parenthesis are the statistical uncertainties in the last digit.

N	RBM	VMC	N	RBM	VMC
72	-	1340.520(7)	70	1129.40(2)	1108.950(4)
90	-	-	112	-	
110	-	-			

11.5 Double quantum dots

Double quantum dots are systems that are very interesting when it comes to testing the Boltzmann machines. Even though we know the wave functions for the non-interacting case, they are expansions that are computational expensive to work with.

11.5.1 Ground state energy

For the interacting case, we will use Alocias Mariadason's VMC calculations with and without a Hartree-Fock basis as a reference. In addition, Jørgen Høgberget provides a DMC energy for the case with two particles and $\omega = 1.0$. All the results are given in table (11.10). What we observe, is that the Boltzmann machine with a Padé-Jastrow factor outperforms a Hermite expansion of 10 functions, which is surprising. The reason might be that the Boltzmann machine consists of a sum over all hidden nodes, which makes it behave similar to an expansion. However, the Boltzmann machine was then again beat by the Hermite expansion with a Hartree-Fock basis.

Table 11.10: Double quantum dots. F is the number of functions used in the expansion.

N	ω	VMC (F=1)	RBM+PJ	VMC (F>1) (Ref.[63])	VMC+HF (Ref.[63])	DMC (Ref.[44])
2	0.1	0.41982(4)	0.41939(3)			
	0.28	0.9365(1)	0.92618(4)			
	0.5	1.4847(2)	1.44004(4)			
	1.0	2.6331(5)	2.38342(3)	2.42238(4)	2.36618(4)	2.3496(1)
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	-	-			

Table 11.11: Energy of neutral atoms of atomic number Z. RBM is a single Slater determinant with a plain Boltzmann machine baked in, VMC is a standard variational Monte-Carlo Slater determinant, FCI is full configuration interaction and CCD means coupled cluster doubles. a is reference to [25], b is to [24] and c is to [44]. The energy is in units of $\hbar$.

Atom	Z	RBM+PJ	VMC	FCI	CCD	Expt.
He	2	-	-2.84824(9)	-2.838648	-2.839144	-2.903694^a
Be	4	-	-	-14.3621	-14.5129	-14.6674^b
Ne	10	-	-	-	-	-128.9383^c
Mg	12	-	-	-	-	-200.054^c
Ar	18	-	-	-	-	-527.544^c
Zn	30	-	-	-	-	-
Kr	36	-	-	-	-	-2752.054976^c

11.6 Atoms

The next set of systems we will address are the real Atoms, which have been investigated by physicists since the childhood of quantum mechanics, and for that reason we have an array of computational and experimental references to use.

11.6.1 Ground state energy

We will still stick to closed shells, and as discussed in section 3.3, this includes the noble gases and a few others. In table (11.11), the ground state energies are presented.

11.6.2 One-body density

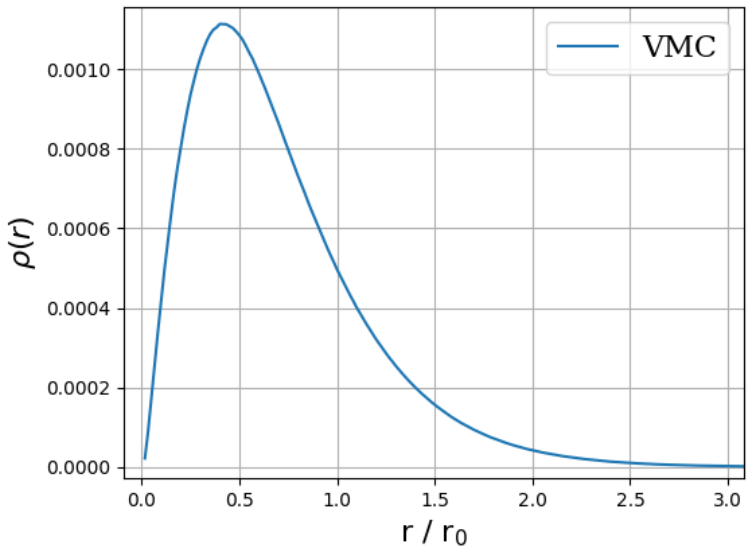


Figure 11.10: One-body density of Helium.

Chapter 12

Conclusion and future work

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

Run linear algebra operations on GPU, we only tried CPU.

In-medium

Appendix A

Dirac Formalism

The Dirac formalism, also called braket notation, was suggested by Paul Dirac in a 1939 paper with the purpose of improving the reading ease. [8] The notation unites the integral representation and the matrix representation in an elegant fashion by representing all elements in the *Hilbert-space*.

The Hilbert-space is a linear vector space.. A column vector in the space is denoted by $|\psi\rangle$, which is called the *ket*, and by taking the Hermitian conjugate of it we obtain the corresponding row vector, $(|\psi\rangle)^+ = \langle\psi|$ called the *bra*.

As the Hilbert-space is characterized by linearity, it requires that the sum of two element in the space is also an element in the space,

$$|\psi\rangle + |\phi\rangle = |\psi + \phi\rangle \,. \tag{A.1}$$

and that two elements always *commute*,

$$|\psi\rangle + |\phi\rangle = |\phi\rangle + |\psi\rangle \,. \tag{A.2}$$

Another important property is that an element of the space multiplied with a complex number is also an element of the space,

$$c\,|\psi\rangle = |c\psi\rangle \qquad \forall c \in \mathbb{C}. \tag{A.3}$$

To fully utilize the notation, orthogonality properties need to be taken into account. Assume that we have a orthogonal basis set

$$\{\psi_1, \psi_2, \ldots, \psi_n\}. \tag{A.4}$$

The inner product between two basis elements is then given by

$$\langle\psi_i|\psi_j\rangle = \begin{cases} \neq 0 & \text{if} \quad i = j \\ = 0 & \text{otherwise} \end{cases} \equiv \delta_{ij} \tag{A.5}$$

where we have introduced the Kronecker delta δ_{ij}. If we further require that our basis is *orthonormal*, the inner product is 1, and we can prove the *completeness relation*. Assume we want to expand a vector $|\phi\rangle$ in our orthonormal basis set,

$$|\phi\rangle = \sum_{i=1}^{n} c_i\,|\psi_i\rangle \,. \tag{A.6}$$

By multiplying with one of the basis vectors on the left hand side, the sum collapses, and we are just left with one of the coefficients c_j,

$$\langle \psi_j | \phi \rangle = \sum_{i=1}^{n} c_i \langle \psi_j | \psi_i \rangle = c_j. \tag{A.7}$$

If we now again insert this into the expansion, we obtain

$$|\phi\rangle = \sum_{i=1}^{n} \langle \psi_i | \phi \rangle |\psi_i\rangle = \left[\sum_{i=1}^{n} |\psi_i\rangle \langle \psi_i| \right] |\phi\rangle \tag{A.8}$$

which implies that the outer product

$$\sum_{i=1}^{n} |\psi_i\rangle \langle \psi_i| = 1. \tag{A.9}$$

We can use these properties to demonstrate how the normalization constant can be obtained without explicitly solving any integrals. Consider two spin-1/2 particles, for example the electron and the proton in the ground state of Hydrogen. There are then four possible states: the *triplet* with $s = 1$ and the *singlet* with $s = 0$. The latter reads [31]

$$|00\rangle = A\big(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle \big) \tag{A.10}$$

which is associated with the bra

$$\langle 00| = \big(|00\rangle \big)^{+} = A\big(\langle\uparrow\downarrow| - \langle\downarrow\uparrow| \big). \tag{A.11}$$

The inner product is then given by

$$\begin{aligned} \langle 00|00\rangle &= A\big(\langle\uparrow\downarrow| - \langle\downarrow\uparrow| \big) A\big(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle \big) \\ &= A^2\big(\langle\uparrow\downarrow|\uparrow\downarrow\rangle - \langle\uparrow\downarrow|\downarrow\uparrow\rangle - \langle\downarrow\uparrow|\uparrow\downarrow\rangle + \langle\downarrow\uparrow|\downarrow\uparrow\rangle \big) \\ &= A^2\big(1 - 0 - 0 + 1\big) \\ &= 2A^2 = 1 \end{aligned} \tag{A.12}$$

where we have assumed that also the states $|\uparrow\downarrow\rangle$ and $|\downarrow\uparrow\rangle$ are orthonormal. From this, we can see that the normalization constant A must be equal to $1/\sqrt{2}$.

Further, a ket consisting of multiple single particle wave function, $|\psi_1\psi_2, \ldots, \psi_n\rangle$, is assumed to be a short-hand notation of the Slater determinant,

$$|\psi_1(\boldsymbol{r}_1)\psi_2(\boldsymbol{r}_2), \ldots, \psi_n(\boldsymbol{r}_n)\rangle \equiv \begin{vmatrix} \psi_1(\boldsymbol{r}_1) & \psi_2(\boldsymbol{r}_1) & \ldots & \psi_n(\boldsymbol{r}_1) \\ \psi_1(\boldsymbol{r}_2) & \psi_2(\boldsymbol{r}_2) & \ldots & \psi_n(\boldsymbol{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(\boldsymbol{r}_n) & \psi_2(\boldsymbol{r}_n) & \ldots & \psi_n(\boldsymbol{r}_n) \end{vmatrix} \tag{A.13}$$

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

Calculations of a general Gaussian-binary RBM wave function

In this appendix, we start from a general Gaussian-binary restricted Boltzmann machine and set up the system energy and joint probability distribution.

The most basic Gaussian-binary restricted Boltzmann machine is the one on the form

$$\begin{aligned} E(\boldsymbol{x},\boldsymbol{h}) &= \sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma_i^2} - \sum_{j=1}^{H} b_j h_j - \sum_{i,j=1}^{F,H}\frac{x_i w_{ij} h_j}{\sigma_i^2} \\ &= \sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma_i^2} - \sum_{j=1}^{H} h_j\Big(b_j + \sum_{i=1}^{F}\frac{x_i w_{ij}}{\sigma_i^2}\Big) \end{aligned} \tag{C.1}$$

as discussed in chapter 4. If we now denote the expression in the last parenthesis by $f_j(\boldsymbol{x};\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ includes all the parameters, we end up with the expression of the general Gaussian-binary restricted Boltzmann machine,

$$E(\boldsymbol{x},\boldsymbol{h}) = \sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma_i^2} - \sum_{j=1}^{H} h_j f_j(\boldsymbol{x};\boldsymbol{\theta}) \tag{C.2}$$

where $f_j(\boldsymbol{x};\boldsymbol{\theta})$ in principle can be any function. From this expression, we obtain the joint probability distribution

$$\begin{aligned} P(\boldsymbol{x},\boldsymbol{h}) &= \frac{1}{Z}\exp(-\beta E(\boldsymbol{x},\boldsymbol{h})) \\ &\propto \exp\left(\sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma^2}\right)\exp\left(\sum_{j=1}^{H}\Big(h_j f_j(\boldsymbol{x};\boldsymbol{\theta})\Big)\right) \\ &\propto \exp\left(\sum_{i=1}^{F}\frac{(x_i-a_i)^2}{2\sigma^2}\right)\prod_{j=1}^{H}\exp\Big(h_j f_j(\boldsymbol{x};\boldsymbol{\theta})\Big) \end{aligned} \tag{C.3}$$

where we set $\beta = 1/k_B T = 1$ and we ignore the partition function Z. Our main interest is the marginal distribution, which we will derive in detail.

C.1 Marginal probability

In chapter 4, we presented the marginal distribution of the visible nodes as

$$P(\boldsymbol{x}) = \sum_{\boldsymbol{h}} P(\boldsymbol{x},\boldsymbol{h}) \tag{C.4}$$

which for our function is

$$\begin{aligned}
P(\boldsymbol{x}) &\propto \sum_{\{\boldsymbol{h}\}} \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \exp\Big(h_j f_j(\boldsymbol{x}, \boldsymbol{\theta})\Big) \\
&= \sum_{h_1=0}^{1} \sum_{h_2=0}^{1} \dots \sum_{h_H=0}^{1} \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \exp\Big(h_1 f_1\Big) \exp\Big(h_2 f_2\Big) \dots \exp\Big(h_H f_H\Big) \\
&= \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \sum_{h_1=0}^{1} \sum_{h_2=0}^{1} \dots \sum_{h_H=0}^{1} \exp\Big(h_1 f_1\Big) \exp\Big(h_2 f_2\Big) \dots \exp\Big(h_H f_H\Big) \\
&= \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \sum_{h_1=0}^{1} \exp\Big(h_1 f_1\Big) \sum_{h_2=0}^{1} \exp\Big(h_2 f_2\Big) \dots \sum_{h_H=0}^{1} \exp\Big(h_H f_H\Big) \\
&= \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \sum_{h_j=0}^{1} \exp\Big(h_j f_j\Big) \\
&= \exp\Big(\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\Big) \prod_{j=1}^{H} \Big[1 + \exp\Big(f_j(\boldsymbol{x}; \boldsymbol{\theta})\Big)\Big]
\end{aligned} \tag{C.5}$$

where we utilize that only one element in the product contains a certain hidden node h_j.

C.2 Gradient of wave function and log-likelihood function

Defining the wave function as the marginal probability, we have seen that wave function of a general Gaussian-binary restricted Boltzmann machine has the form

$$\psi(\boldsymbol{x}; \boldsymbol{a}, \boldsymbol{\theta}) = \exp\left(-\sum_{i=1}^{F} \frac{(x_i - a_i)^2}{2\sigma^2}\right) \prod_{j=1}^{H} \Big[1 + \exp(f_j(\boldsymbol{x}; \boldsymbol{\theta}))\Big] \tag{C.6}$$

where $f_j(\boldsymbol{x}; \boldsymbol{\theta})$ is an arbitrary function of the coordinates $\boldsymbol{x}$ and the weights $\boldsymbol{\theta}$. The Gaussian part is straight-forward to differentiate, so we will keep our attention on the product,

$$\psi_{\mathrm{p}}(\boldsymbol{x}; \boldsymbol{\theta}) = \prod_{j=1}^{H} \Big[1 + \exp(f_j(\boldsymbol{x}; \boldsymbol{\theta}))\Big]. \tag{C.7}$$

Henceforth, we will omit the variable $\boldsymbol{x}$ and $\boldsymbol{\theta}$. By introducing the functions

$$p_j \equiv \frac{1}{1 + \exp(+f_j)} \quad \wedge \quad n_j \equiv \frac{1}{1 + \exp(-f_j)}, \tag{C.8}$$

where the last one is the sigmoid function, we find the gradient and Laplacian of $\ln \psi_{\mathrm{p}}$ to be

$$\nabla_k \ln \psi_{\mathrm{p}} = \sum_{j=1}^{H} n_j \nabla_k(f_j) \tag{C.9}$$

and

$$\nabla_k^2 \ln \psi_{\mathrm{p}} = \sum_{j=1}^{H} n_j \big[\nabla_k^2(f_j) + p_j \big(\nabla_k(f_j)\big)^2\big] \tag{C.10}$$

respectively. The parameter θ_i can be updated according to the following rule

$$\frac{\partial}{\partial \theta_i} \ln \psi_{\mathrm{p}} = \sum_{j=1}^{H} n_j \frac{\partial}{\partial \theta_i}(f_j). \tag{C.11}$$

and the ratio between wave functions can be found by

$$\frac{\psi_{\mathrm{p}}^{\mathrm{new}}}{\psi_{\mathrm{p}}^{\mathrm{old}}} = \prod_{j=1}^{H} \frac{p_j^{\mathrm{old}}}{p_j^{\mathrm{new}}}. \tag{C.12}$$

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 $\partial_{\theta_i}(f_j)$.This applies for a general Gaussian-binary restricted Boltzmann machine, also for a deep Boltzmann machine if all the units are Gaussian-binary.

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] R. Clausius. “XVI. On a mechanical theorem applicable to heat”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 40.265 (Aug. 1870), pp. 122–127. ISSN: 1941-5982. DOI: 10.1080/14786447008640370. URL: https://doi.org/10.1080/14786447008640370 (visited on 07/04/2019).

[4] D. R. Hartree. “The Wave Mechanics of an Atom with a Non-Coulomb Central Field. Part II. Some Results and Discussion”. en. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 24.1 (Jan. 1928), pp. 111–132. ISSN: 1469-8064, 0305-0041. DOI: 10.1017/S0305004100011920. URL: https://www.cambridge.org/core/journals/mathematical-proceedings-of-the-cambridge-philosophical-society/article/wave-mechanics-of-an-atom-with-a-noncoulomb-central-field-part-ii-some-results-and-discussion/5916E7A0DEC0A051B435688BE2 (visited on 07/28/2019).

[5] Dirac Paul Adrien Maurice and Fowler Ralph Howard. “Quantum mechanics of many-electron systems”. In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 123.792 (Apr. 1929), pp. 714–733. DOI: 10.1098/rspa.1929.0094. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1929.0094 (visited on 07/03/2019).

[6] V. Fock. “Bemerkung zum Virialsatz”. de. In: *Zeitschrift für Physik* 63.11 (Nov. 1930), pp. 855–858. ISSN: 0044-3328. DOI: 10.1007/BF01339281. URL: https://doi.org/10.1007/BF01339281 (visited on 07/03/2019).

[7] V. Fock. “„Selfconsistent field“ mit Austausch für Natrium”. de. In: *Zeitschrift für Physik* 62.11 (Nov. 1930), pp. 795–805. ISSN: 0044-3328. DOI: 10.1007/BF01330439. URL: https://doi.org/10.1007/BF01330439 (visited on 07/28/2019).

[8] P. a. M. Dirac. “A new notation for quantum mechanics”. en. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 35.3 (July 1939), pp. 416–418. ISSN: 1469-8064, 0305-0041. DOI: 10.1017/S0305004100021162. URL: https://www.cambridge.org/core/journals/mathematical-proceedings-of-the-cambridge-philosophical-society/article/new-notation-for-quantum-mechanics/4631DB9213D680D6332BA11799D76AFB (visited on 07/21/2019).

[9] Nicholas Metropolis and S. Ulam. “The Monte Carlo Method”. In: *Journal of the American Statistical Association* 44.247 (Sept. 1949), pp. 335–341. ISSN: 0162-1459. DOI: 10.1080/01621459.1949.10483310. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1949.10483310 (visited on 07/23/2019).

[10] Nicholas Metropolis et al. “Equation of State Calculations by Fast Computing Machines”. In: *The Journal of Chemical Physics* 21.6 (June 1953), pp. 1087–1092. ISSN: 0021-9606. DOI: 10.1063/1.1699114. URL: https://aip.scitation.org/doi/10.1063/1.1699114 (visited on 07/22/2019).

[11] W. J. Hehre, R. F. Stewart, and J. A. Pople. “Self-Consistent Molecular-Orbital Methods. I. Use of Gaussian Expansions of Slater-Type Atomic Orbitals”. In: *The Journal of Chemical Physics* 51.6 (Sept. 1969), pp. 2657–2664. ISSN: 0021-9606. DOI: 10.1063/1.1672392. URL: https://aip.scitation.org/doi/10.1063/1.1672392 (visited on 07/04/2019).

[12] W. K. Hastings. “Monte Carlo sampling methods using Markov chains and their applications”. en. In: *Biometrika* 57.1 (Apr. 1970), pp. 97–109. ISSN: 0006-3444. DOI: 10.1093/biomet/57.1.97. URL: https://academic.oup.com/biomet/article/57/1/97/284580 (visited on 07/22/2019).

[13] W. Heisenberg. *Physics and Beyond: Encounters and Conversations*. Harper torchbooks. The Academy library. Allen and Unwin, 1971. URL: https://books.google.no/books?id=O-dEAAAAIAAJ.

[14] J.M. Leinaas and J. Myrheim. “One the theory of identical particles”. In: *IL NUOVO CIMENTO* 37.1 (Aug. 1977).

[15] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. “A Learning Algorithm for Boltzmann Machines*”. en. In: *Cognitive Science* 9.1 (1985), pp. 147–169. ISSN: 1551-6709. DOI: 10.1207/s15516709cog0901_7. URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog0901_7 (visited on 07/21/2019).

[16] David Ceperley and Berni Alder. “Quantum Monte Carlo”. en. In: *Science* 231.4738 (Feb. 1986), pp. 555–560. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.231.4738.555. URL: https://science.sciencemag.org/content/231/4738/555 (visited on 07/23/2019).

[17] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. En. In: *Nature* 323.6088 (Oct. 1986), p. 533. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 07/09/2019).

[18] H. Flyvbjerg and H. G. Petersen. “Error estimates on averages of correlated data”. In: *The Journal of Chemical Physics* 91.1 (July 1989), pp. 461–466. ISSN: 0021-9606. DOI: 10.1063/1.457480. URL: https://aip.scitation.org/doi/abs/10.1063/1.457480 (visited on 07/18/2019).

[19] 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).

[20] K. Brunner et al. “Sharp-Line Photoluminescence and Two-Photon Absorption of Zero-Dimensional Biexcitons in a GaAs/AlGaAs Structure”. In: *Physical Review Letters* 73.8 (Aug. 1994), pp. 1138–1141. DOI: 10.1103/PhysRevLett.73.1138. URL: https://link.aps.org/doi/10.1103/PhysRevLett.73.1138 (visited on 06/18/2019).

[21] J. Y. Marzin et al. “Photoluminescence of Single InAs Quantum Dots Obtained by Self-Organized Growth on GaAs”. In: *Physical Review Letters* 73.5 (Aug. 1994), pp. 716–719. DOI: 10.1103/PhysRevLett.73.716. URL: https://link.aps.org/doi/10.1103/PhysRevLett.73.716 (visited on 06/18/2019).

[22] 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).

[23] 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.

[24] A. Kramida and W. C. Martin. "A Compilation of Energy Levels and Wavelengths for the Spectrum of Neutral Beryllium (Be I)". In: *Journal of Physical and Chemical Reference Data* 26.5 (Sept. 1997), pp. 1185–1194. ISSN: 0047-2689. DOI: 10.1063/1.555999. URL: https://aip.scitation.org/doi/10.1063/1.555999 (visited on 06/27/2019).

[25] S. Bergeson et al. "Measurement of the He Ground State Lamb Shift Via the Two-Photon 11S - 21S transition". en. In: *Physical Review Letters* 80.No. 16 (Apr. 1998). URL: https://www.nist.gov/publications/measurement-he-ground-state-lamb-shift-two-photon-11s-21s-transition-0 (visited on 06/27/2019).

[26] Trygve Helgaker and Wim Klopper. "Perspective on "Neue Berechnung der Energie des Heliums im Grundzustande, sowie des tiefsten Terms von Ortho-Helium"". en. In: *Theoretical Chemistry Accounts* 103.3 (Feb. 2000), pp. 180–181. ISSN: 1432-2234. DOI: 10.1007/s002149900051. URL: https://doi.org/10.1007/s002149900051 (visited on 07/23/2019).

[27] F. Pederiva, C. J. Umrigar, and E. Lipparini. "Diffusion Monte Carlo study of circular quantum dots". In: *Physical Review B* 62.12 (Sept. 2000). arXiv: cond-mat/9912166, pp. 8120–8125. ISSN: 0163-1829, 1095-3795. DOI: 10.1103/PhysRevB.62.8120. URL: http://arxiv.org/abs/cond-mat/9912166 (visited on 06/20/2019).

[28] Roland Assaraf and Michel Caffarel. "Zero-Variance Zero-Bias Principle for Observables in quantum Monte Carlo: Application to Forces". In: *The Journal of Chemical Physics* 119.20 (Nov. 2003). arXiv: physics/0310035, pp. 10536–10552. ISSN: 0021-9606, 1089-7690. DOI: 10.1063/1.1621615. URL: http://arxiv.org/abs/physics/0310035 (visited on 07/22/2019).

[29] Sherrill, David. *Postulates of Quantum Mechanics*. July 2003. URL: http://vergil.chemistry.gatech.edu/notes/intro_estruc/node4.html (visited on 06/20/2019).

[30] "A Quantum Sampler". en-US. In: *The New York Times* (Dec. 2005). ISSN: 0362-4331. URL: https://www.nytimes.com/2005/12/26/science/a-quantum-sampler.html (visited on 07/03/2019).

[31] D.J. Griffiths. *Introduction to quantum mechanics*. 2nd Edition. Pearson PH, 2005. ISBN: 0-13-191175-9.

[32] 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.

[33] 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).

[34] Richard Cohen et al. *Quantities, Units and Symbols in Physical Chemistry*. en. Aug. 2007. ISBN: 978-0-85404-433-7. DOI: 10.1039/9781847557889. URL: https://pubs.rsc.org/en/content/ebook/978-0-85404-433-7 (visited on 07/23/2019).

[35] 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.

[36] Amit Ghosal et al. "Incipient Wigner localization in circular quantum dots". In: *Physical Review B* 76.8 (Aug. 2007), p. 085341. DOI: 10.1103/PhysRevB.76.085341. URL: https://link.aps.org/doi/10.1103/PhysRevB.76.085341 (visited on 06/20/2019).

[37] J. D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3 (May 2007), pp. 90–95. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55.

[38] E.W Weisstein. *Kelvin, Lord William Thomson (1824-1907)*. 2007. URL: http://scienceworld.wolfram.com/biography/Kelvin.html.

[39] Daniel Andres Nissenbaum. "The stochastic gradient approximation: an application to li nanoclusters." In: (2008). URL: https://repository.library.northeastern.edu/files/neu:1790 (visited on 06/20/2019).

[40] Michal Bajdich and Lubos Mitas. "Electronic structure quantum Monte Carlo". In: *arXiv:1008.2369 [cond-mat, physics:physics]* (Aug. 2010). arXiv: 1008.2369. DOI: 10.2478/v10155.009.0002.3. URL: http://arxiv.org/abs/1008.2369 (visited on 06/20/2019).

[41] Geoffrey E. Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv:1207.0580 [cs]* (July 2012). arXiv: 1207.0580. URL: http://arxiv.org/abs/1207.0580 (visited on 07/21/2019).

[42] V. Jelic and F. Marsiglio. "The double well potential in quantum mechanics: a simple, numerically exact formulation". In: *European Journal of Physics* 33.6 (Nov. 2012). arXiv: 1209.2521, pp. 1651–1666. ISSN: 0143-0807, 1361-6404. DOI: 10.1088/0143-0807/33/6/1651. URL: http://arxiv.org/abs/1209.2521 (visited on 05/24/2019).

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (visited on 06/20/2019).

[44] 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).

[45] 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.

[46] Till Tantau. "Graph Drawing in TikZ". en. In: *Graph Drawing*. Ed. by Walter Didimo and Maurizio Patrignani. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 517–528. ISBN: 978-3-642-36763-2.

[47] 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).

[48] 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.

[49] Asja Fischer and Christian Igel. "Training restricted Boltzmann machines: An introduction". In: *Pattern Recognition* 47.1 (Jan. 2014), pp. 25–39. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2013.05.025. URL: http://www.sciencedirect.com/science/article/pii/S0031320313002495 (visited on 07/21/2019).

[50] 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).

[51] 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 06/20/2019).

[52] Jesse R. Manders et al. "8.3: Distinguished Paper: Next-Generation Display Technology: Quantum-Dot LEDs". en. In: *SID Symposium Digest of Technical Papers* 46.1 (2015), pp. 73–75. ISSN: 2168-0159. DOI: 10.1002/sdtp.10276. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sdtp.10276 (visited on 06/18/2019).

[53] Morten Ledum. *Simple Variational Monte Carlo solve for FYS4411*. 2016. URL: https://github.com/mortele/variational-monte-carlo-fys4411.

[54] Chris Smith. *iOS 10: Siri now works in third-party apps, comes with extra AI features*. en. June 2016. URL: https://bgr.com/2016/06/13/ios-10-siri-third-party-apps/ (visited on 06/27/2019).

[55] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *arXiv:1609.08144 [cs]* (Sept. 2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144 (visited on 06/27/2019).

[56] *C++ Data Types*. en-US. May 2017. URL: https://www.geeksforgeeks.org/c-data-types/ (visited on 07/26/2019).

[57] 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).

[58] Giuseppe Carleo. *Neural-Network Quantum States*. en. June 2017. URL: https://gitlab.com/nqs/ucas_workshop/blob/master/lecture_notes.pdf (visited on 07/05/2019).

[59] Md Zahangir Alom et al. "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches". In: *arXiv:1803.01164 [cs]* (Mar. 2018). arXiv: 1803.01164. URL: http://arxiv.org/abs/1803.01164 (visited on 07/09/2019).

[60] Bertil Gustafsson. *Scientific Computing: A Historical Perspective*. en. Texts in Computational Science and Engineering. Springer International Publishing, 2018. ISBN: 978-3-319-69846-5. URL: https://www.springer.com/gp/book/9783319698465 (visited on 07/28/2019).

[61] Marius Jonsson. "Standard error estimation by an automated blocking method". nob. In: (2018). URL: https://www.duo.uio.no/handle/10852/63557 (visited on 07/19/2019).

[62] Marius Jonsson. "Standard error estimation by an automated blocking method". In: *Physical Review E* 98.4 (Oct. 2018), p. 043304. DOI: 10.1103/PhysRevE.98.043304. URL: https://link.aps.org/doi/10.1103/PhysRevE.98.043304 (visited on 07/18/2019).

[63] Alfred Alocias Mariadason. "Quantum many-Body Simulations of Double Dot System". eng. In: (2018). URL: https://www.duo.uio.no/handle/10852/64577 (visited on 06/20/2019).

[64] Even Marius Nordhagen. *Computational physics II: Quantum mechanical systems: evenmn/FYS4411*. original-date: 2018-01-25T16:21:07Z. Nov. 2018. URL: https://github.com/evenmn/FYS4411 (visited on 07/16/2019).

[65] AlDanial. *cloc counts blank lines, comment lines, and physical lines of source code in many programming languages.: AlDanial/cloc*. original-date: 2015-09-07T03:30:43Z. July 2019. URL: https://github.com/AlDanial/cloc (visited on 07/27/2019).

[66] Pankaj Mehta et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *Physics Reports* 810 (May 2019). arXiv: 1803.08823, pp. 1–124. ISSN: 03701573. DOI: 10.1016/j.physrep.2019.03.001. URL: http://arxiv.org/abs/1803.08823 (visited on 06/20/2019).

[67] Morten Hjorth-Jensen. *Computational Physics 2: Variational Monte Carlo methods*. English. 2019. URL: http://compphysics.github.io/ComputationalPhysics2/doc/pub/vmc/html/vmc-reveal.html (visited on 07/04/2019).

[68] *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).

[69] 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).

[70] *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).

[71] *2019 QLED Technology - Innovation behind TV | Samsung Singapore*. en-SG. URL: //www.samsung.com/sg/tvs/qled-tv/technology/ (visited on 06/18/2019).

[72] *AlphaGo Movie*. en. URL: https://www.alphagomovie.com/ (visited on 06/27/2019).

[73] *Bringing the Magic of Amazon AI and Alexa to Apps on AWS. - All Things Distributed*. URL: https://www.allthingsdistributed.com/2016/11/amazon-ai-and-alexa-for-all-aws-apps.html (visited on 06/27/2019).

[74] Flugsrud, Vilde Moe. "Solving Quantum Mechanical Problems with Machine Learning". In: ().

[75] Martin Fowler. *bliki: TwoHardThings*. URL: https://martinfowler.com/bliki/TwoHardThings.html (visited on 03/14/2019).

[76] ISO/TC 12. *ISO 80000-2:2009*. en. URL: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/18/31887.html (visited on 07/23/2019).

[77] ISO/TC 12. *ISO 80000-9:2009*. en. URL: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/18/31894.html (visited on 07/23/2019).

[78] Mike Klein (MikeKlein). *Google's AlphaZero Destroys Stockfish In 100-Game Match*. en-US. URL: https://www.chess.com/news/view/google-s-alphazero-destroys-stockfish-in-100-game-match (visited on 06/27/2019).

[79] *MIT License*. en. URL: https://choosealicense.com/licenses/mit/ (visited on 07/29/2019).

[80] *Training Restricted Boltzmann Machines: An Introduction**. URL: https://webcache.googleusercontent.com/search?q=cache:JHzm9N4wEaoJ:https://christian-igel.github.io/paper/TRBMAI.pdf+&cd=1&hl=en&ct=clnk&gl=no&client=ubuntu (visited on 07/20/2019).

[81] Eric W. Weisstein. *Matrix Inverse*. en. Text. URL: http://mathworld.wolfram.com/MatrixInverse.html (visited on 05/24/2019).

[82] John F. Woods. *Usage of comma operator - Google Groups*. URL: https://groups.google.com/forum/#!msg/comp.lang.c++/rYCO5yn4lXw/oITtSkZOtoUJ (visited on 07/26/2019).