More Accurate QMC Calculations using Machine Learning

by

Even Marius Nordhagen

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

March 13, 2019

Abstract

Abstract should be written here

Acknowledgements

First of all I would like to thank Christine for cooking delicious food and feeding me when I need it the most. She and her food have maintained my motivation in the best way, and they are the main reason why I've survived those two years of masters.

Secondly, my annoying cohabitants have forced me to spend more time at university. I'm so glad you behave the way you do, if you were more cozy I would certainly spend more time at home and this thesis would not be the same.

Acknowledgements should be written here

-One of my mottos used to be that everything has a reason, which I used to point out whenever people were talking about things that apparently could not be described immediately. It was first when I learned about quantum mechanics that I understood I was wrong, and that is one of the reasons why quantum mechanics catched me so hardly.

Contents

List of abbreviations

Letters		**Meaning**
RBM	-	Restricted Boltzmann Machine
MC	-	Monte Carlo
VMC	-	Variational Monte Carlo
DMC	-	Diffusion Monte Carlo
ML	-	Machine Learning
WF	-	Wave Function
SPF	-	Single Particle Function

Table 1: List of symbols used with explaination.

Source Code

The source code is given in `https://github.com/evenmn`

Chapter 1

Introduction

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, we do not have a good model for the three-body interaction, which is vital when it comes to strong correlations. 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

"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]

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.

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.

With machine learning, a function can be fitted to everything as long as we can define a cost function to minimize. The purpose of this thesis is to construct optimal wave functions for different systems by using machine learning.

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

1.1 Many-Body problem

Not possible to solve analytically

1.2 Machine learning

Branch of artificial intelligence

1.3 Goals and milestones

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

Chapter 2

Quantum Many-Body Physics

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

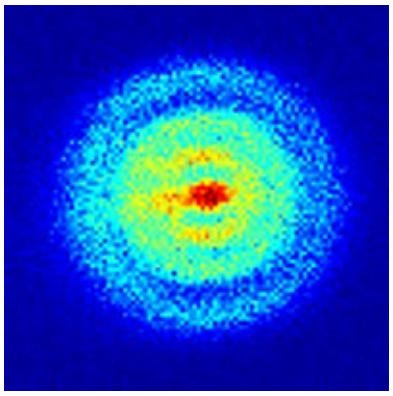


Figure 2.1: The first photograph of a Hydrogen atom was captured by an ultra sensitive camera in 2013. One can actually see the probability distribution $|\Psi(\boldsymbol{r})|^2$ with the naked eye. Published in Phys. rev. lett. 110, 213001 (2013), *Hydrogen atoms under magnification*. [13]

Around 1900, some physicists thought that there were nothing new to be discovered in physics and all that remained was more precise measurements, as Lord Kelvin famously pointed out. [11] He could not have been more wrong. In the following years, things were observed that could only be described by a quantized theory, led by Albert Einstein's explanation of the photoelectric effect in 1905.

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

2.1 Introductory Quantum Physics

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

Before we get started, we make a few assumptions in order to simplify our problem. They 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, that they lack spatial extension. This includes the nucleus in atomic systems, but it still makes sense since the distance from the nucleus to the electrons is known to be much larger then the nucleus extent.

- **Non-relativistic spacetime:** Second, we operate in the non-relativistic spacetime, which is valid 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. The forces acting are the weak Coulomb forces.

- For specific systems we might make new assumptions and approximations. For instance, for atomic systems we will assume that the nucleus is at rest. Those approximations will be discussed consecutively.

2.1.1 The Schrödinger Equation

The Schrödinger equation is a natural starting point, which gives the energy eigen states of a system defined by a Hamiltonian $\hat{\mathcal{H}}$ and its eigen functions, Ψ, which are the wave functions. The time-independent Schrödinger equation reads

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

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

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

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

The kinetic energy yields $T = p^2/2m$, 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 momentum operator is $\hat{\mathcal{P}} = -i\hbar\nabla$.

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

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

where r_{12} is the distance between the particles. The total Hamiltonian of a system of N identical particles takes the form

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

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

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

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

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

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

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

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

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

In many cases we do not know the exact wave function, and need to rely on a trial wave function guess. Henceforth, we will use Ψ as the exact total wave function, ψ as the exact single particle function (SPF), Ψ_T as the total trial wave function and ϕ as the trial SPF. [8]

2.1.2 The Variational Principle

In the equations above, the presented wave functions are assumed to be the exact eigen functions 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 variational principle, which states that only the exact ground state wave function is able to give the ground state energy. All other wave functions that fulfill the required properties (see section 2.2) give higher energies, and mathematically we can express the statement

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

Variational Monte-Carlo is a method based on (and named after) the variational principle, where we vary the trial wave function in order to obtain the lowest energy. It will be detailed in chaper (4).

2.1.3 Postulates of Quantum Mechanics

The quantum theory is built on a few fundamental postulates, which will always be true. Before we go further, we will take a quick look at some of them.

1. Associated with any particle moving in a conservative field of force is a wave function which determines everything that can be known about the system.

2. With every physical observable q there is associated an operator Q, which when operating upon the wave function associated with a definite value of that observable will yield that value times the wave function.

3. Any operator Q associated with a physically measurable property q will be Hermitian.

4. The set of eigen functions of operator Q will form a complete set of linearly independent functions.

5. For a system described by a given wave function, the expectation value of any property q can be found by performing the expectation value integral with respect to that wave function.

6. The time evolution of the wave function is given by the time-dependent Schrodinger equation.

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 to make physical sense. 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. More about this in the next section.

2.2.1 Bosons and fermions

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

$$\hat{P}(i \rightarrow j)\Psi_n(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_j, \ldots, \boldsymbol{x}_M) = p\Psi_n(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_j, \ldots, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_M), \quad (2.10)$$

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, $p = \pm 1$. [1]

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

It turns out that because of their antisymmetric wavefunction, two identical fermions cannot be found at the same position at the same time, known as the Pauli principle. This causes some difficulties when dealing with multiple fermions, because we always need to ensure that the total wavefunction becomes zero if two identical particles happen to be at the same position. To do this, we introduce a Slater determinant as described below. In this particular project, we are going to focus on electrons and therefore fermions. However, much of the theory applies for bosons as well.

2.2.2 Slater determinant

For a system of more particles we can define a total wavefunction, which is a composition of all the single particle wavefunctions (SPF) and contains all the information about the system. For fermions we need to compile the SPFs such that the Pauli principle is fulfilled at all times. One way to do this is by setting up the SPFs in a determinant, known as a Slater 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

$$\Psi_T(\boldsymbol{r}_1, \boldsymbol{r}_2) = \begin{vmatrix} \phi_1(\boldsymbol{r}_1) & \phi_2(\boldsymbol{r}_1) \\ \phi_1(\boldsymbol{r}_2) & \phi_2(\boldsymbol{r}_2) \end{vmatrix} = \phi_1(\boldsymbol{r}_1)\phi_2(\boldsymbol{r}_2) - \phi_2(\boldsymbol{r}_1)\phi_1(\boldsymbol{r}_2), \tag{2.11}$$

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

The Slater determinant is just an ansatz since it does not come from any analytical calculations, 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, the spin part needs to be included as well, giving

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N) = \begin{vmatrix} \psi_1(\boldsymbol{r}_1) & \psi_2(\boldsymbol{r}_1) & \ldots & \psi_N(\boldsymbol{r}_1) \\ \psi_1(\boldsymbol{r}_2) & \psi_2(\boldsymbol{r}_2) & \ldots & \psi_N(\boldsymbol{r}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(\boldsymbol{r}_N) & \psi_2(\boldsymbol{r}_N) & \ldots & \psi_N(\boldsymbol{r}_N) \end{vmatrix} \tag{2.12}$$

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

$$\psi = \phi \otimes \xi \tag{2.13}$$

with ξ as the spin part. How the spin can be factorized out for electronic systems will be shown below.

Similar to the Slater determinant, a Slater permanent might be included for bosonic systems. The permanent of a matrix is similar to the determinant, but all negative signs are replaced by positive signs.

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

2.2.3 Simplification of Slater determinant for electronic systems

For our purpose we will study fermions of spin $\sigma = \pm 1/2$ only, i.e., electrons and protons. In this particular case, the SPFs can be arranged in spin-up and spin-down parts, such that the Slater determinant can be simplied to

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N) = \begin{vmatrix} \phi_1(\boldsymbol{r}_1)\xi_\uparrow & \phi_1(\boldsymbol{r}_1)\xi_\downarrow & \ldots & \phi_{N/2}(\boldsymbol{r}_1)\xi_\uparrow & \phi_{N/2}(\boldsymbol{r}_1)\xi_\downarrow \\ \phi_1(\boldsymbol{r}_2)\xi_\uparrow & \phi_1(\boldsymbol{r}_2)\xi_\downarrow & \ldots & \phi_{N/2}(\boldsymbol{r}_2)\xi_\uparrow & \phi_{N/2}(\boldsymbol{r}_2)\xi_\downarrow \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \phi_1(\boldsymbol{r}_{N-1})\xi_\uparrow & \phi_1(\boldsymbol{r}_{N-1})\xi_\downarrow & \ldots & \phi_{N/2}(\boldsymbol{r}_{N-1})\xi_\uparrow & \phi_{N/2}(\boldsymbol{r}_{N-1})\xi_\downarrow \\ \phi_1(\boldsymbol{r}_N)\xi_\uparrow & \phi_1(\boldsymbol{r}_N)\xi_\downarrow & \ldots & \phi_{N/2}(\boldsymbol{r}_N)\xi_\uparrow & \phi_{N/2}(\boldsymbol{r}_N)\xi_\downarrow \end{vmatrix}.$$

Here we assume that two particles of opposite spin are found at the same position, which makes the number of fermions with spin up equal to the number of fermions with spin down. This is obviously not a assumption that always will hold, but it eases the calculations.

Now recall that all particles with odd position indices ($\boldsymbol{r}_1$, $\boldsymbol{r}_3$, ...) have spin up, such that $\phi_j(\boldsymbol{r}_i)\xi_\downarrow = 0$ where $j = 1, \ldots, N/2$ and i spans over all the odd positions indices. The same applies for particles with even position indices and spin down, and we get

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N) = \begin{vmatrix} \phi_1(\boldsymbol{r}_1)\xi_\uparrow & 0 & \ldots & \phi_{N/2}(\boldsymbol{r}_1)\xi_\uparrow & 0 \\ 0 & \phi_1(\boldsymbol{r}_2)\xi_\downarrow & \ldots & 0 & \phi_{N/2}(\boldsymbol{r}_2)\xi_\downarrow \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \phi_1(\boldsymbol{r}_{N-1})\xi_\uparrow & 0 & \ldots & \phi_{N/2}(\boldsymbol{r}_{N-1})\xi_\uparrow & 0 \\ 0 & \phi_1(\boldsymbol{r}_N)\xi_\downarrow & \ldots & 0 & \phi_{N/2}(\boldsymbol{r}_N)\xi_\downarrow \end{vmatrix},$$

which by row operations can be found to be row equivalent with

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N) = \begin{vmatrix} \phi_1(\boldsymbol{r}_1)\xi_\uparrow & \phi_2(\boldsymbol{r}_1)\xi_\uparrow & \ldots & 0 & 0 \\ \phi_1(\boldsymbol{r}_3)\xi_\uparrow & \phi_2(\boldsymbol{r}_3)\xi_\uparrow & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & \phi_{N/2-1}(\boldsymbol{r}_{N-2})\xi_\downarrow & \phi_{N/2}(\boldsymbol{r}_{N-2})\xi_\downarrow \\ 0 & 0 & \ldots & \phi_{N/2-1}(\boldsymbol{r}_N)\xi_\downarrow & \phi_{N/2}(\boldsymbol{r}_N)\xi_\downarrow \end{vmatrix}.$$

This means that we can split the Slater determinant in a spin up part and a spin down part,

$$\Psi(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_N) = |\hat{D}_\uparrow| \cdot |\hat{D}_\downarrow| \tag{2.14}$$

which saves us from a lot of computations. For a detailed explanation of the splitting, appendix I of Daniel Nissenbaum's dissertation is an excellent read. [12] Szabo & Ostlund in [7] is also a good explanation.

2.2.4 Basis set

To go further, we need to define a basis set, $\phi_n(\boldsymbol{r})$ 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 (4-6).

2.2.5 Jastrow factor

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. Nevertheless, this does not fulfill the normalizability requirement of the trial wave function, but the same idea lies behind the Padé-Jastrow factor, which reads

$$J(\boldsymbol{r};\beta,\gamma) = \exp\left(\sum_{i=1}^{N}\sum_{j=1}^{i}\frac{\beta(i,j)r_{ij}}{1+\gamma r_{ij}}\right). \tag{2.15}$$

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. There are also plenty of other Jastrow factors, and in this project we will see if we can create a more flexible and general Jastrow factor using machine learning. Exactly how this is done is presented in chapter (7).

Chapter 3

Systems

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

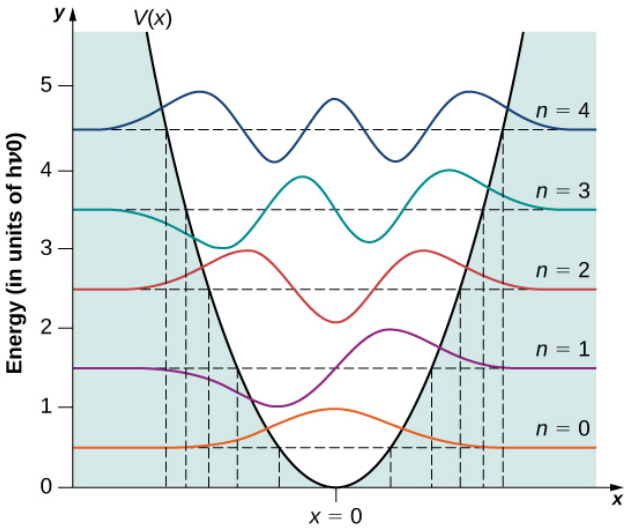


Figure 3.1: The quantum harmonic oscillator, with the Hermite functions represented up to 4th order. As in classical mechanics, the harmonic oscillator can describe various quantum systems, such as lattice vibration (phonons) and quantum fields.

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

3.1 Quantum dots

Quantum dots are very small particles, and contain fermions or bosons hold together by an external potential. Since these particles have discrete electronic states like an atom, they are often called artificial atoms.

In this thesis we will, among other systems, study electrons trapped in harmonic oscillators, where the external potential affecting particle i is given by

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

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

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

where the energy is scaled with respect to atomic 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 can be used as the basis. $H_n(x)$ 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 at energy level n in a d dimensional harmonic oscillator is given by

$$E_n = \omega\Big(n + \frac{d}{2}\Big). \tag{3.4}$$

We will study closed-shell systems only, since the Slater determinant in that case is unambiguous. For open shells, the total Slater determinant is a linear combination of all the possible Slater determinants. 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

$$N = s\binom{n+d}{d} \tag{3.5}$$

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 have have 4 particles with radial wave functions $\Phi_{n_x=1,n_y=0}$ and $\psi_{n_x=0,n_y=1}$ with degeneracy 2 and so on.

INSERT IMAGE WHICH SHOWS THE NUMBER OF ELECTRON IN EACH LEVEL

3.2 Atomic systems

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

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

$$u_i = -\frac{1}{2}k\frac{Ze^2}{r_i}, \tag{3.6}$$

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

$$\hat{\mathcal{H}} = \sum_{i=1}^{P} \Big(-\frac{1}{2}\nabla_i^2 - \frac{1}{2}\frac{Z}{r_i} + \frac{l(l+1)}{2r_i^2} \Big) + \sum_{i<j} \frac{1}{r_{ij}}, \tag{3.7}$$

which also is discussed in Appendix B.

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

$$\psi_{nlm}(r, \theta, \phi) = R_{nl}(r) Y_{lm}(\theta, \phi) \tag{3.8}$$

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

$$R_{nl}(r) \propto r^l e^{-Zr/n} \Big[L_{n-l-1}^{2l+1}\Big(\frac{2r}{n}Z\Big)\Big] \tag{3.9}$$

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

The angular part is given by the *spherical harmonics*

$$Y_{lm}(\theta, \phi) \propto P_l^m(\cos\theta) e^{im\phi} \tag{3.10}$$

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

$$S_l^m(r, \theta, \phi) \propto r^l P_l^{|m|}(\cos\theta) \begin{cases} \cos(m\phi) & \text{if} \quad m \geq 0 \\ \sin(|m|\phi) & \text{if} \quad m < 0. \end{cases} \tag{3.11}$$

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

Table 3.1: Table of the first subshells

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

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

Similar as for the principal quantum number n, we can use the tumble rule the lower l the lower energy, such that for Beryllium all four electrons are still in the s-subshell. Beryllium therefore has electron configuration $1s^2 2s^2$ or [He] $2s^2$. Since both subshells are fully occupied, Beryllium can be included in our closed-shell calculations.

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

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

$$V_L = \frac{l(l+1)}{2r^2} \tag{3.12}$$

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

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

3.3 Molecular systems

3.4 Bose-Einstein Condensation

3.5 Electron gas

3.6 Helium gas

He^3 and He^4

Chapter 4

Quantum Monte-Carlo Methods

Great quote.

Author

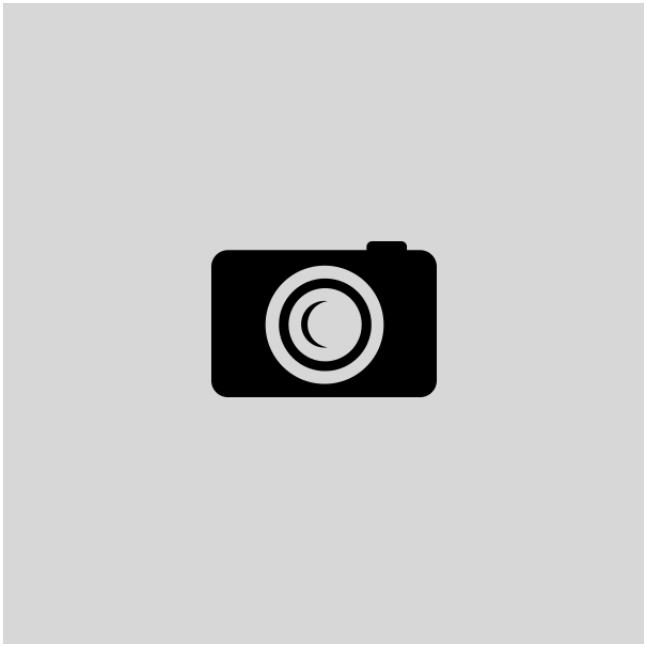


Figure 4.1: Caption

Some great methods are developed over the past decade. We will focus on the Hartree-Fock and variational Monte Carlo methods and give a detailed explanation of those methods. Additionally, the well-known methods configuration interaction and coupled cluster will be described briefly for some kind of completeness.

Monte Carlo methods in quantum mechanics are a bunch of methods that are built on diffusion processes, and includes Variational Monte Carlo (VMC), Diffusion Monte Carlo (DMC) and others. The common denominator is that we move particles in order to find the optimal configuration, usually where the energy is minimized. The particles can be moved isotropic, i.e., uniformly in all directions, or they can be affected by a drift force which makes the process anisotropic.

4.0.1 Isotropic processes

In isotropic processes, we have random walks where the particles move randomly in space which falls under the category Markov chains. If we assume constant timestep, the step index can be considered the time, thus we have a time-dependent probability density $P(x,t)$. This probability density needs to satisfy the isotropic diffusion equation,

$$\frac{\partial P(\boldsymbol{x},t)}{\partial t} = D\frac{\partial^2}{\partial \boldsymbol{x}^2}P(\boldsymbol{x},t). \tag{4.1}$$

4.0.2 Anisotropic processes

For anisotropic processes, we have a drift and the moves are no longer considered random but falls still under the category Markov chains. Because of the drift, we need to rewrite the diffusion equation and we end up with the Fokker-Planck equation,

$$\frac{\partial P(\boldsymbol{x},t)}{\partial t} = D\frac{\partial}{\partial \boldsymbol{x}}\left(\frac{\partial}{\partial \boldsymbol{x}} - F\right)P(\boldsymbol{x},t) \tag{4.2}$$

which needs to be satisfied. The new positions in coordinate space are given as solution of the Langevin equation

$$\frac{\partial \boldsymbol{x}(t)}{\partial t} = D\boldsymbol{F}(\boldsymbol{x}(t)) + \eta \tag{4.3}$$

4.0.3 Variational Monte Carlo

The Variational Monte Carlo (hereafter, VMC) method is today widely used when it comes to the study of ground state properties of quantum mechanical systems. It is a Monte Carlo method which makes use of Metropolis sampling, and has been used in studies of fermionic systems since the 1970's. [15] If we go back to the variational principle in equation (2.9), we see that by choosing a wave function which satisfies the criteria, we will get an energy larger or equal to the ground state energy.

There are two main problems we need to solve

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

Let us take the last problem first. The solution is to approximate the integral with a sum,

$$\begin{aligned} E &\leq \frac{\int \Psi_T(\boldsymbol{r})^* \hat{H} \Psi_T(\boldsymbol{r}) d\boldsymbol{r}}{\int \Psi_T(\boldsymbol{r})^* \Psi_T(\boldsymbol{r}) d\boldsymbol{r}} \\ &= \int P(\boldsymbol{r}) E_L(\boldsymbol{r}) d\boldsymbol{r} \\ &\approx \frac{1}{N}\sum_{i=1}^{N} E_L(\boldsymbol{r}_i) \end{aligned} \tag{4.4}$$

where the local energy is defined as

$$E_L(\boldsymbol{r}) \equiv \frac{1}{\Psi_T(\boldsymbol{r})}\hat{H}\Psi_T(\boldsymbol{r}) \tag{4.5}$$

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

$$P(\boldsymbol{r}) = \frac{|\Psi_T(\boldsymbol{r})|^2}{\int |\Psi_T(\boldsymbol{r})|^2 d\boldsymbol{r}}. \tag{4.6}$$

The energy found from equation (4.4) is an expectation value, and we therefore know that the true energy lies within the standard error. When increasing the number of Monte-Carlo cycles, the standard error decreases and we get a more accurate energy. In the limit N goes to infinity, the variance should approach zero,

$$\langle E \rangle = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} E_L(\boldsymbol{r}_i). \tag{4.7}$$

For more statistical details, see [15].

Given a wave function we are now able to estimate the corresponding energy, but how do we find a good wave function? In VMC, we define a wave function with variational parameters, which are adjusted in order to minimize the energy for every iteration. For every iteration, we run N Monte-Carlo cycles where we withdraw a new position $\boldsymbol{r}_i$. Whether or not the proposed move should be accepted is determined by the Metropolis algorithm.

4.0.3.1 The Metropolis Algorithm

The genius of the metropolis algorithm, is that the acceptance of a move is not based on the probabilities themselves, but the ratio between the new and the old probability
. In that way, we avoid calculating the sum over all probabilities, which often is expensive or even impossible to calculate.

In its simplest form, the move is proposed randomly, and it is accepted if the ratio is larger than a random number between 0 and 1. However, with this approach a lot of moves will be rejected, which wastes CPU time. A better method is **importance sampling**, which makes a educated guess of the best way to move based on diffusion processes, and move the particle in that direction.

A time-dependent probability density needs to satisfy the Fokker-Planck equation

Chapter 5

Hartree-Fock

Great quote.

Author

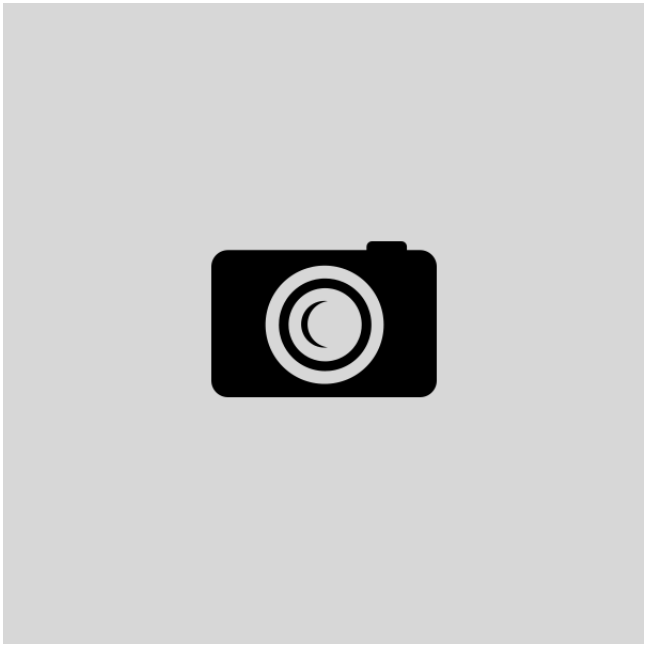


Figure 5.1: Caption

Hartree-Fock is like a "kitchen tool", ...

Chapter 6

Post Hartree-Fock Methods

Great quote.

Author

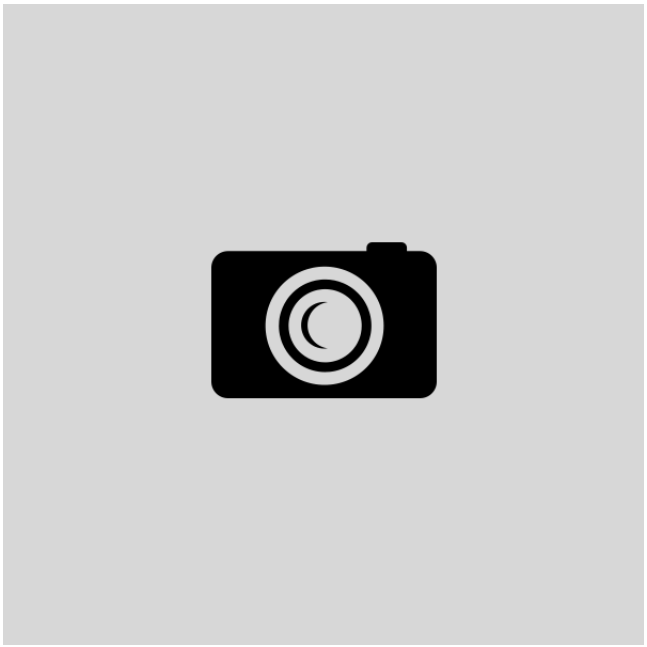


Figure 6.1: Caption

The term *ab initio* means from first principles, implying that only physical constants are put into the methods. The Monte-Carlo methods are not considered *ab initio* as non-physical hyper parameters are required. Some example methods are Hartree-Fock methods and post-Hartree-Fock methods like Configuration Interaction and Coupled Cluster, which will be discussed in the following.

6.1 Configuration Interaction

The completeness relation

6.2 Coupled Cluster

The coupled cluster method is the *de facto* standard wave function-based method for electronic structure calculations. [9] The method approximates the wave function with an exponential expansion,

$$|\Psi_{\mathrm{CC}}\rangle = e^{\hat{T}} |\Phi_0\rangle \tag{6.1}$$

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

$$\hat{T}_n = \left(\frac{1}{n!}\right)^2 \sum_{abc\dots} \sum_{ijk\dots} t_{ijk\dots}^{abc\dots} a_a^\dagger a_b^\dagger a_c^\dagger \cdots a_k a_j a_i. \tag{6.2}$$

We again want to solve the Schrödinger equation,

$$\hat{H} |\Psi\rangle = \hat{H} e^{\hat{T}} |\Phi_0\rangle = \epsilon e^{\hat{T}} |\Phi_0\rangle , \tag{6.3}$$

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

$$\bar{H} = e^{-\hat{T}} \hat{H} e^{\hat{T}}. \tag{6.4}$$

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

$$\langle \Phi_0 | \bar{H} | \Phi_0 \rangle = \epsilon \tag{6.5}$$

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

$$\left\langle \Phi_{ijk\dots}^{abc\dots} \middle| \bar{H} \middle| \Phi_0 \right\rangle = 0 \tag{6.6}$$

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

$$\begin{aligned} \bar{H} = \hat{H} &+ [\hat{H}, \hat{T}] \\ &+ \frac{1}{2}[[\hat{H}, \hat{T}], \hat{T}] \\ &+ \frac{1}{6}[[[\hat{H}, \hat{T}], \hat{T}], \hat{T}] \\ &+ \frac{1}{24}[[[[\hat{H}, \hat{T}], \hat{T}], \hat{T}], \hat{T}] \\ &+ \cdots \end{aligned} \tag{6.7}$$

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. [10]

Chapter 7

Machine Learning

Great quote.

Author

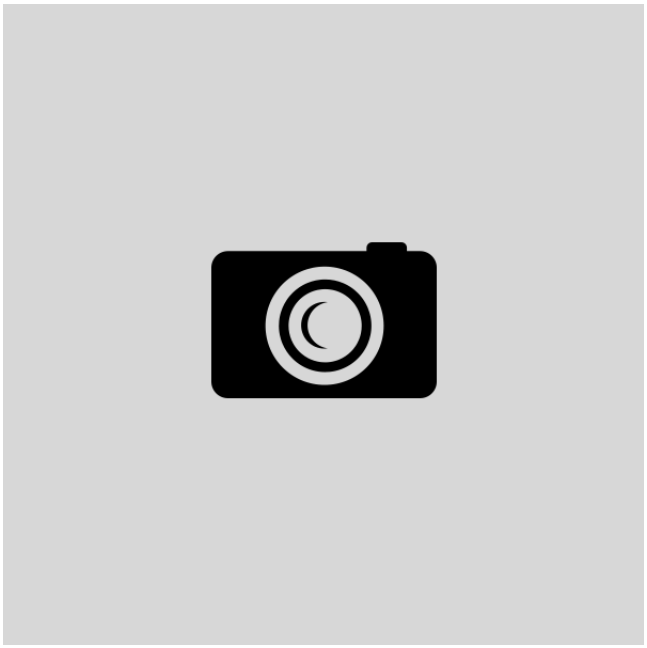


Figure 7.1: Caption

The use of the term *machine learning* has exploded over the past years, and sometimes it sounds like it is a totally new field. However, the truth is that many of the methods we use are quite 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 mentioned so often. As the UiO professor Anne Solberg pointed out during one of her lectures

"*In the early 1990's I was working a lot with machine learning, but at that time we called it pattern recognition and regression...*"

Another reason why machine learning has an increasingly popularity is that some of the algorithms have been significantly improved. *Convolutional Neural Networks* (CNNs)

are now as good as humans when it comes to object recognition in images, and *Long Short-Term Memory Recurrent Neural Networks* (LSTM RNNs) has improved voice recognition.

We can conclude that Machine learning includes all methods where we want to fit models to data sets, and pattern recognition and regression are clearly some of those methods. In both cases we have concrete targets for the training, and the training is therefore called supervised training. In other cases we do not have targets, but want to train our model based on a probability distribution or so, which is called unsupervised learning.

7.1 Supervised Learning

In supervised learning methods, the input data has known targets such that we can fit a model to give the correct outputs. Linear regression is perhaps the most intuitive example on this, where we want to find the line that fits some data points in the best possible way. This corresponds to a neural network without any hidden layer, but when we add more layers the model is no longer linear and it all gets more complex. For simple classification tasks, logistic regression can be used.

7.1.1 Linear regression

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

$$y_i = \sum_j X_{ij}\beta_j \tag{7.1}$$

where β_j's are the unknown parameters to be found. In principle, X_{ij} can be an arbitrary function of the arguments x_i, but often one wants a polynomial model which corresponds to $X_{ij} = x_i^j$.

The three most commonly used linear regression methods are Ordinary Least Square (OLS) regression, Ridge regression and Lasso regression, where the former has the cost function

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\right)^2 \qquad \text{OLS,} \tag{7.2}$$

which is minimized when

$$\vec{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\vec{y}. \tag{7.3}$$

Similarly, the Ridge cost function is

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 \qquad \text{Ridge} \tag{7.4}$$

where λ is called the penalty. This is minimized when

$$\vec{\beta} = (\hat{X}^T\hat{X} + \lambda I)^{-1}\hat{X}^T\vec{y}, \tag{7.5}$$

and finally the Lasso cost function is given by

$$c(\vec{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} X_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j \qquad \text{Lasso.} \tag{7.6}$$

7.1.2 Logistic regression

Despite its name, logistic regression is not a fitting tool, but rather a classification tool. Traditionally, the perceptron model was used for 'hard classification', which sets the outputs directly to binary values. However, often we are interested in the probability of a given category, which means that we need a continuous *activation function*. Logistic regression can, like linear regression, be considered as a function where coefficients are adjusted with the intention to minimize the error. Here, the coefficients are called *weights*. The process goes like this: The inputs are multiplied by several weights, and by adjusting those weights the model can classify every *linear classification problem*. A drawing of the perceptron is found in figure (7.2).

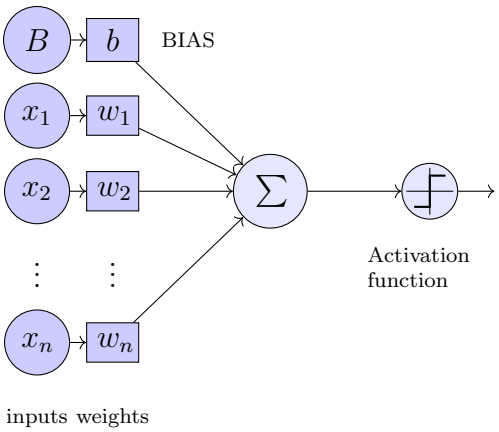


Figure 7.2: Logistic regression model with n inputs.

In logistic regression, we usually have one binary output node for each class, but for two categories one output node is sufficient, which can be fired or not fired.

Initially, one needs to train the perceptron such that it knows which outputs are correct, and for that one needs to know the outputs that correspond to the inputs. Every time the network is trained, the weights are adjusted such that the error is minimized.

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

7.1.2.1 Forward phase

Let us look at it from a mathematical perspective, and calculate the net output. The net output seen from an output node is simply the sum of all the "arrows" that point towards the node, see figure (7.2), where each "arrow" is given by the left-hand node multiplied with its respective weight. For example, the contribution from input node 2 to the output node follows from $X_2 \cdot w_2$, and the total net output to the output O is therefore

$$net = \sum_{i=1}^{I} x_i \cdot w_i + b \cdot 1. \tag{7.7}$$

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

You might wonder why we talk about the net output all the time, do we have other outputs? If we look at the network mathematically, what we talk about as the net output

should be our only output. Anyway, it turns out to be convenient mapping the net output to a final output using an activation function, which is explained further in section 7.1.2.5. The activation function, f, takes in the net output and gives the output,

$$out = f(net). \tag{7.8}$$

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

7.1.2.2 BIAS

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

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

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

7.1.2.3 Learning rate

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

7.1.2.4 Cost function

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

$$c(\boldsymbol{W}) = -\sum_{i=1}^{n} \Big[y_i \log f(\boldsymbol{x}_i^T \boldsymbol{W}) + (1 - y_i) \log[1 - f(\boldsymbol{x}_i^T \boldsymbol{W})] \Big] \tag{7.9}$$

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

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

$$p(y|x) = \hat{y}^y \cdot (1 - \hat{y})^{1-y}. \tag{7.10}$$

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

$$\log \Big[p(y|x) \Big] = \log \Big[\hat{y}^y \cdot (1 - \hat{y})^{1-y} \Big] \tag{7.11}$$

$$= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \tag{7.12}$$

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

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

$$c(\boldsymbol{W})^+ = c(\boldsymbol{W}) + \lambda||\boldsymbol{W}||_2^2. \tag{7.13}$$

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

7.1.2.5 Activation function

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

$$f(x) = \frac{1}{1+e^{-x}}. \tag{7.14}$$

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

$$\frac{df(x)}{dx} = x(1-x). \tag{7.15}$$

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

7.1.2.6 Backward phase

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

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot \frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}}, \tag{7.16}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial c(\boldsymbol{w})}{\partial out} \cdot \frac{\partial out}{\partial net} \cdot \frac{\partial net}{\partial \boldsymbol{w}} \tag{7.17}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial out} = -\frac{y}{out} + \frac{1-y}{1-out}. \tag{7.18}$$

Further, the second derivative is the derivative of the activation function with respect to the output, which is given in (7.15)

$$\frac{\partial out}{\partial net} = out(1 - out). \tag{7.19}$$

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

$$\frac{\partial net}{\partial \boldsymbol{w}} = \boldsymbol{x}. \tag{7.20}$$

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

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]\boldsymbol{x} \tag{7.21}$$

and obtain a weight update algorithm

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]^T\boldsymbol{x}. \tag{7.22}$$

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

7.1.3 Neural network

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

In figure (7.3), a two-layer neural network (one hidden layer) is illustrated. It has some similarities with the logistic regression model in figure (7.2), but a hidden layer and multiple outputs are added. In addition, the output is no longer probabilities and can take any number, which means that we do not need to use the logistic function on the outputs anymore.

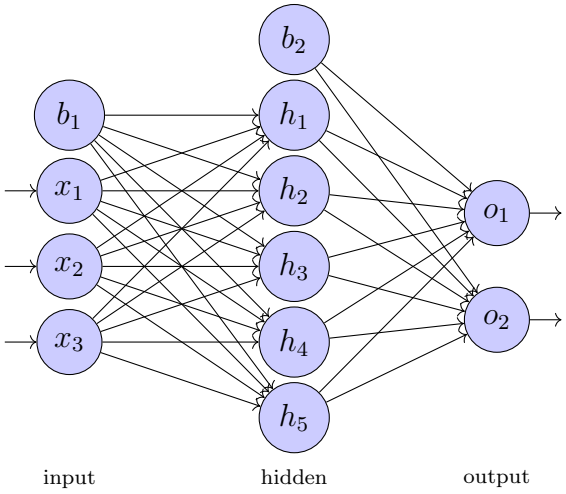


Figure 7.3: Neural network with 3 input nodes, 5 hidden nodes and 2 output nodes, in addition to the bias nodes.

Without a hidden layer, we have seen that the update of weights is quite straight forward. For a neural network consisting of multiple layers, the question is: how do we

update the weights when we do not know the values of the hidden nodes? And how do we know which layer causing the error? This will be explained in section 7.1.3.3, where one of the most popular techniques for that is discussed. Before that we will generalize the forward phase presented in logistic regression.

7.1.3.1 Forward phase

In section 7.1.2.1, we saw how the output is found for a single perceptron. Since we only had one output node, the weights could be stored in an array. Generally, it is more practical to store the weights in matrices, since they will have indices related to both the node on left-hand side and the node on the right-hand side. For instance, the weight between input node x_3 and hidden node h_5 in figure (7.3) is usually labeled as w_{35}. Since we have two layers, we also need to denote which weight set it belongs to, which we will do by a superscript ($w_{35} \Rightarrow w_{35}^{(1)}$). In the same way, $\boldsymbol{W}^1$ is the matrix containing all $w_{ij}^{(1)}$, $\boldsymbol{x}$ is the vector containing all x_i's and so on. We then find the net outputs at the hidden layer to be

$$net_{h,j} = \sum_{i=1}^{I} x_i \cdot w_{ij}^{(1)} = \boldsymbol{x}^T \boldsymbol{W}_j^{(1)} \tag{7.23}$$

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

$$h_j = f(net_{h,j}). \tag{7.24}$$

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

$$net_{o,j} = \sum_{i=1}^{H} h_i \cdot w_{ij}^{(2)} = \boldsymbol{h}^T \boldsymbol{W}_j^{(2)} \tag{7.25}$$

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

$$\boldsymbol{net}_{h_l} = \boldsymbol{h}^{(\boldsymbol{l-1})^T} \boldsymbol{W}^{(l)}. \tag{7.26}$$

7.1.3.2 Activation function

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

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

7.1.3.3 Backward Propagation

Backward propagation is probably the most used technique for updating the weights, and is actually again based on equation (7.16). What differs, is the differentiation of the net input with respect to the weight, which gets more complex as we add more layers. For one hidden layer, we have two sets of weights, where the last layer is updated in a similar way as for a network without hidden layer, but the inputs are replaced with the values of the hidden nodes:

$$w_{ij}^{(2)+} = w_{ij}^{(2)} - \eta \cdot [f(h_i^T w_{ij}) - y_j]^T h_i. \tag{7.27}$$

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

$$w_{ij}^{(1)+} = w_{ij}^{(1)} - \eta \cdot \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(2)} \cdot f'(out_{hj}) \cdot x_i \tag{7.28}$$

where we recall δ_{ok} as

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

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

7.1.3.4 Summary

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

$$\begin{aligned}
net_{hi} &= \sum_j w_{ji}^{(1)} \cdot x_j \\
out_{hi} &= \mathrm{f}(net_{hi}) \\
\\
net_{ki} &= \sum_j w_{ji}^{(2)} \cdot out_{hj} \\
out_{ki} &= \mathrm{f}(net_{ki}) \\
\\
net_{oi} &= \sum_j w_{ji}^{(3)} \cdot out_{kj} \\
out_{oi} &= \mathrm{f}(net_{oi})
\end{aligned} \tag{7.29}$$

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

$$\begin{aligned}
w_{ij}^{(3)} &= w_{ij}^{(3)} - \eta \cdot \delta_{oj} \cdot out_{ki} \\
\\
w_{ij}^{(2)} &= w_{ij}^{(2)} - \eta \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(3)} \cdot f'(out_{kj}) \cdot out_{hi} \\
\\
w_{ij}^{(1)} &= w_{ij}^{(1)} - \eta \sum_{k=1}^{O} \sum_{l=1}^{K} \delta_{ok} \cdot w_{lk}^{(3)} \cdot f'(out_{kl}) \cdot w_{jl}^{(2)} f'(out_{hj}) \cdot x_i
\end{aligned}$$

where we again use the short hand

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

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

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

7.2 Unsupervised Learning

How can we train using unsupervised learning?

7.2.1 Statistical foundation

Bayesian statistics

7.2.2 Boltzmann Machines

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

$$P = \exp(-E/kT) \tag{7.30}$$

which is the Boltzmann distribution, hence the name Boltzmann machine. k is known as Boltzmann's constant and T is the system temperature, but henceforth they both will be avoided by scaling $E' = E/kT$.

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

INSERT FIGURE

In the same manner as in a Feed-forward Neural Network, we can directly multiply each node s_i with its respective inner weights w_{ij} and then with the opposite node s_j, which defines the system energy together with the bias contributions. This gives the energy

$$E = -\sum_{i=1}^{N}\sum_{j=i}^{N} s_i w_{ij} s_j - \sum_{i=1}^{N} s_i b_i \tag{7.31}$$

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

7.2.3 Restricted Boltzmann Machines

When there is an unrestricted guy, a restricted guy must exist as well.

Chapter 8

Optimization

Great quote.

Author

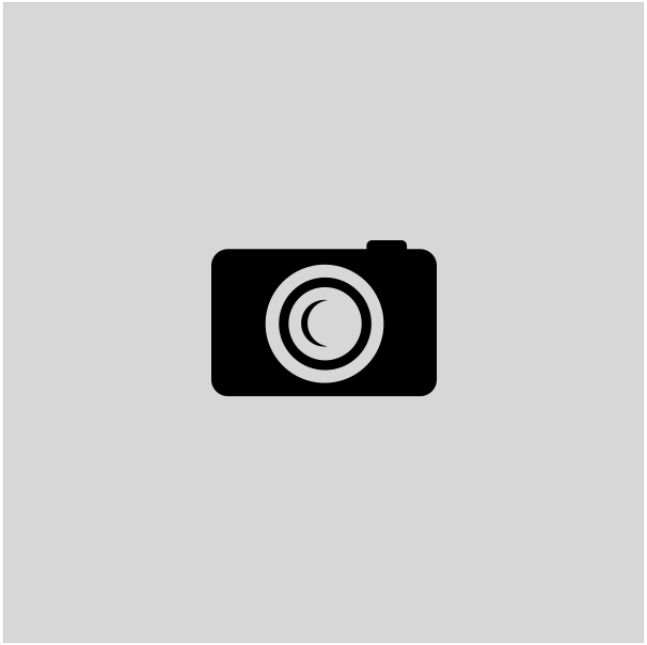


Figure 8.1: Caption

Optimization is a wide term..

8.1 Minimization Algorithms

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

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

8.1.1 Gradient Descent

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

$$\alpha_i^{\text{new}} = \alpha_i - \eta \cdot \frac{\partial Q(\alpha_i)}{\partial \alpha_i} \tag{8.1}$$

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

$$\frac{\partial Q(\alpha_i)}{\partial \alpha_i} < \varepsilon. \tag{8.2}$$

where ε is a tolerance.

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

8.1.2 Stochastic Gradient Descent

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

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

$$\alpha_i^{\text{new}} = \alpha_i - \eta \cdot \frac{\partial Q_j(\alpha_i)}{\partial \alpha_i} \tag{8.3}$$

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

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

8.1.3 ADAM

8.1.4 Adding momentum

8.2 Optimization of Trial Wave Function

8.2.1 Energy Minimization

8.2.2 Variance Minimization

8.3 Computational optimization

Write about how we calculate Slater determinants efficiently and things like that

8.4 Random number generators

Mersenne Twister needs to be mentioned here

Chapter 9

Scientific Programming

Great quote.

Author

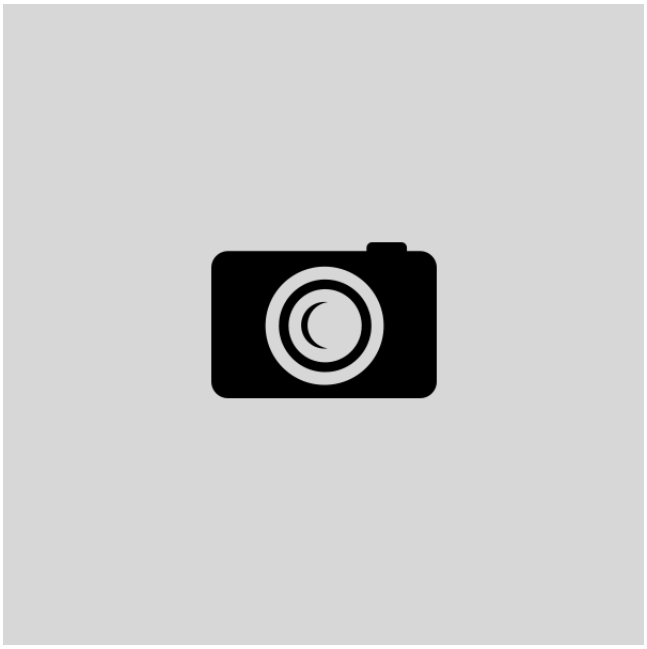


Figure 9.1: Caption

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

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

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

9.1 Object Orientated Programming

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

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

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

where...

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

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

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

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

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

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

INCLUDE C++ IMPLEMENTATION

As one can see,

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

9.1.1 Inheritance

This is also called parent and child, respectively.

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

1. Single inheritance
2. Multiple inheritances

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

9.1.2 Pointers

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

9.1.3 Virtual Functions

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

Chapter 10

Implementation

There are only two hard things in Computer Science: cache invalidation and naming things.
`https://martinfowler.com/bliki/TwoHardThings.html`

Phil Karlton

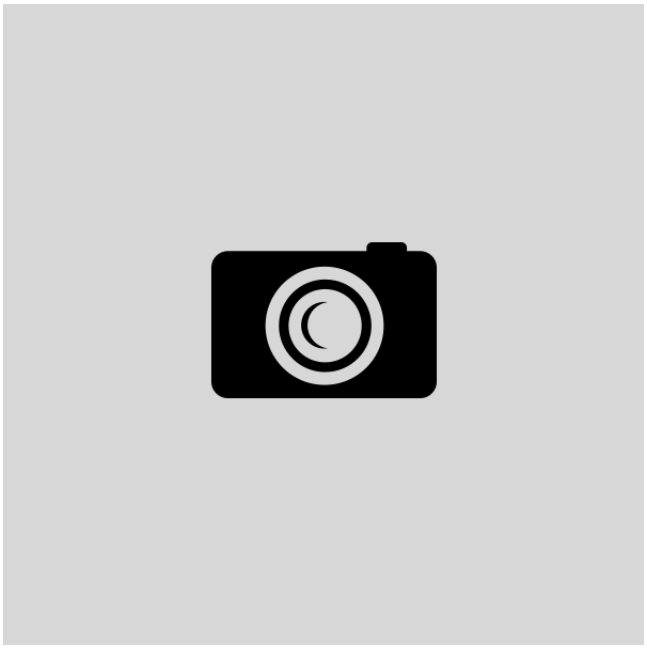


Figure 10.1: Caption

For many projects, planning is half the job, and so is true for a good VMC implementation. In fact, the program was restructured three times before we landed on this final version. It is based on Morten Ledum's VMC framework found at [16], which was meant as an example implementation in the course *FYS4411 - Computational Physics II: Quantum Mechanical Systems*. As our development environment, we use **QT creator**, which offers an elegant platform for writing, compiling and profiling. In addition, one can easily construct a *graphical user interface* (GUI) based on **QT Widgets**. For all matrix and array operations, the **Eigen** library was used. All source code can be found at `https://github.com/evenmn/VMC/`.

We had three main areas of focus when developing the code. It should be

- efficient
- flexible
- legible.

The efficiency is mostly based on recursive computations, such that we do not need to calculate everything over again when it is enough to calculate the fraction between the old the new number. This is highly relevant when it comes to the Slater determinant, but we also do this for other wave function elements. See chapter (8) for more details.

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

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

Listing 10.1: `main.cpp`

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

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

To maximize the legibility, 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 `evaluateRatio`, `computeFirstDerivative`, `computeSecondDerivative`, `computeFirstEnergyDerivative` and `computeSecondEnergyDerivative` among others. To ensure that all wave function elements have all the necessary properties, the super class `WaveFunctions` is equipped with the corresponding virtual functions

Listing 10.2: Simplification of `wavefunction.h`

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

class WaveFunction {
public:
        WaveFunction(class System *system);
        virtual double evaluateRatio() = 0;
        virtual double computeFirstDerivative(int k) = 0;
        virtual double computeSecondDerivative() = 0;
        virtual Eigen::VectorXd computeFirstEnergyDerivative(int k) = 0;
        virtual Eigen::VectorXd computeSecondEnergyDerivative() = 0;
        virtual ~WaveFunction() = 0;

protected:
        class System* m_system = nullptr;
};
```

which serves a template for all the sub classes (wave function elements). As you might notice, we use the **lowerCamelCase** naming convention for function and variable names, which means that all words in start 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 of 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 convention can be read here `https://en.wikipedia.org/wiki/Naming_convention_(programming)`.

10.1 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.

10.2 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.

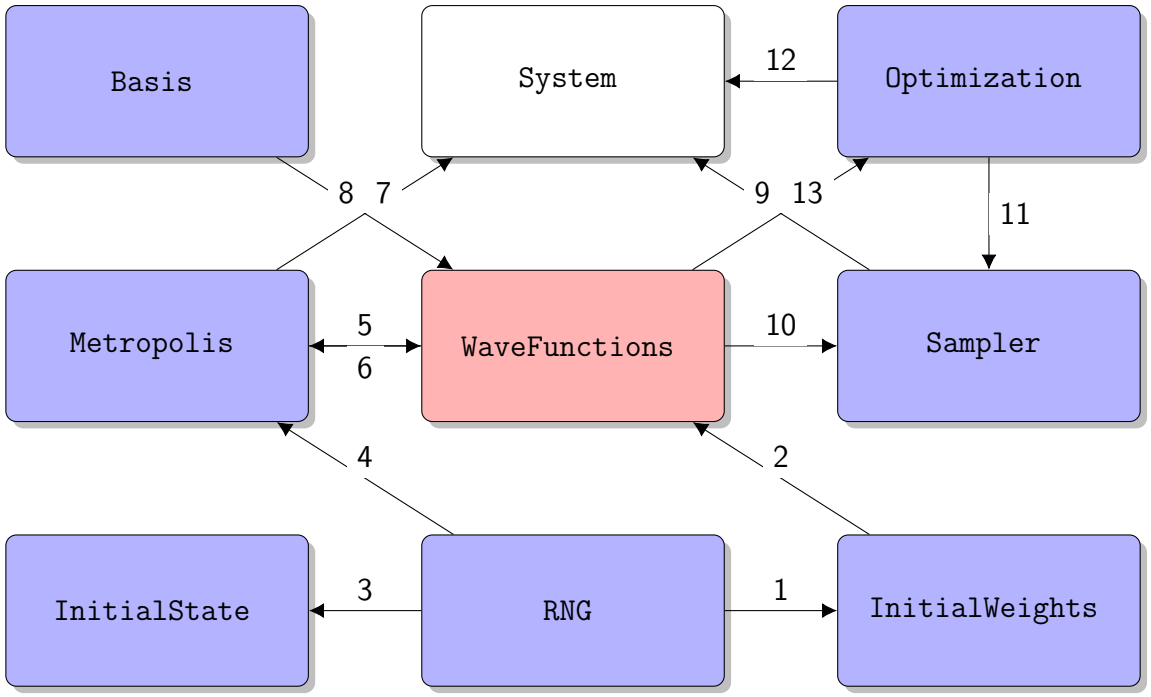
10.2.1 Super classes

10.2.1.1 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.



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

10.2.1.2 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.

10.2.1.3 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()`.

10.2.1.4 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`.

10.2.1.5 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.

10.2.1.6 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.

10.2.1.7 The `Plotter` class

Not sure if I will keep this as a class

10.2.1.8 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 (8.4).

10.2.1.9 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.

10.2.2 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.

10.3 Graphical User Interface (GUI)

Chapter 11

Results

Great quote.

Author

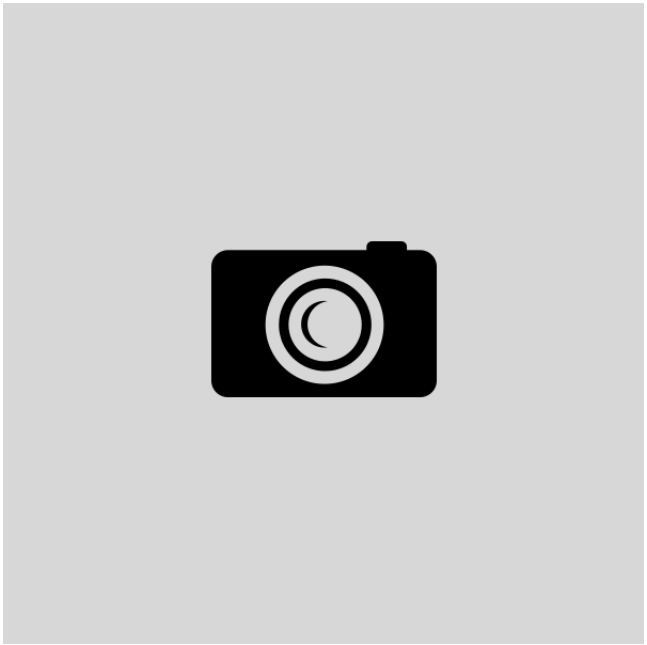


Figure 11.1

Now over to the heart of this thesis: the results...

11.1 No repulsive interaction

We start with the non-interacting case in order to prove the flexibility of the implemented code.

11.1.1 Ground state energy

11.1.1.1 Quantum dots

We will start studying systems with no interaction between particles. The number of closed-shell particles are given by equation (3.5) and the exact energies are found from formula (3.4). The first 5 closed-shell energies for $\omega = 0.5$ and $\omega = 1.0$ are presented in table (11.1).

Table 11.1

	ω	0.5			1.0		
	N	E_{NQS}	E_{VMC}	E_{exact}	E_{NQS}	E_{VMC}	E_{exact}
2D	2	1.0	1.0	1	2.0	2.0	2
	6	5.0	5.0	5	10.0	10.0	10
	12	14.0	14.0	14	28.0	28.0	28
	20	30.0	30.0	30	60.0	60.0	60
	30	55.0	55.0	55	110.0	110.0	110
3D	2	1.5	1.5	1.5	3.0	3.0	3
	8	9.0	9.0	9	18.0	18.0	18
	20	27.0	27.0	27	54.0	54.0	54
	40	67.0	67.0	67	134.0	134.0	134
	70	149.5	149.5	149.5	299.0	299.0	299

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

11.1.1.2 Atoms

In the next we will

Table 11.2

Atom	N	E_{NQS}	E_{VMC}	E_{exact}
H	1	0.0	0.0	-0.5
He	2	0.0	0.0	-4
Be	4	0.0	0.0	-20
Ne	10	0.0	0.0	-200

11.1.2 One-body density

11.2 With repulsive interaction

Table 11.3: This table presents the energies of N electrons trapped in a three-dimensional oscillator well with frequency ω. The exact energies are calculated analytically by M.Taut, see [6]. The reference is to J. Høgberget CITE HIM (DMC).

N	ω	E_{NQS}	E_{VMC}	E_{ref}	E_{exact}
2	0.0365	0.0	0.0		0.054806
	0.1	0.0	0.0	0.499997(3)	0.5
	0.5	0.0	0.0	2.000000(2)	2.0
	1.0	0.0	0.0	3.730123(3)	
8	0.0365	0.0	0.0	5.7028(1)	
	0.1	0.0	0.0	12.1927(1)	
	0.5	0.0	0.0	18.9611(1)	
	1.0	0.0	0.0	32.6680(1)	
20	0.0365	0.0	0.0	27.2717(2)	
	0.1	0.0	0.0	56.3868(2)	
	0.5	0.0	0.0	85.6555(2)	
	1.0	0.0	0.0	142.8875(2)	
40	0.0365	0.0	0.0		
	0.1	0.0	0.0		
	0.5	0.0	0.0		
	1.0	0.0	0.0		
70	0.0365	0.0	0.0		
	0.1	0.0	0.0		
	0.5	0.0	0.0		
	1.0	0.0	0.0		

Chapter 12

Conclusion and future work

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

Appendix A

Dirac notation

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

Appendix B

Scaling

B.1 Harmonic Oscillator - 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 start with scaling $H' \equiv H/\hbar$, 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 we can take out by scaling the length $x' \equiv x \cdot \sqrt{m/\hbar}$. The final Hamiltonian is

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

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

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

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

B.2 Atomic systems - Atomic units

The atomic Hamiltonian in its simplest form is given by

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

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

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

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

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

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

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

B.3 Comparison between natural and atomic units

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

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

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

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

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

and

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

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

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

Appendix C

Associated Laguerre Polynomials

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

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

which can be represented by the Rodriguez formula

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

where $L_q(x)$ are the Laguerre polynomials

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

C.1 Recursive relation between polynomials

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

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

The first few elements are

$$L^k_0(x) = 1 \tag{C.5}$$
$$L^k_1(x) = 1 + k - x \tag{C.6}$$
$$L^k_2(x) = \frac{1}{2}\Big[x^2 - 2(k + 2)x + (k + 1)(k + 2)\Big] \tag{C.7}$$
$$L^k_3(x) = \frac{1}{6}\Big[- x^3 + 3(k + 3)x^2 - 3(k + 2)(k + 3)x + (k + 1)(k + 2)(k + 3)\Big] \tag{C.8}$$

Appendix D

Machine Learning Wave Function

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

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

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

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

such that

$$F_{\mathrm{RBM}}(\boldsymbol{X}, \boldsymbol{h}) = e^{\sum_{i=1}^{M} \frac{(X_i - a_i)^2}{2\sigma^2}} e^{\sum_{j=1}^{N} \left(b_j h_j + \sum_{i=1}^{M} \frac{X_i W_{ij}}{\sigma^2} \right)}. \tag{D.3}$$

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

$$\begin{aligned}
F_{\mathrm{RBM}}(\boldsymbol{X}) &= \sum_{\{\boldsymbol{h}\}} e^{\sum_{i=1}^{M} \frac{(X_i - a_i)^2}{2\sigma^2}} \prod_{j=1}^{N} e^{b_j h_j + \sum_{i=1}^{M} \frac{X_i W_{ij} h_j}{\sigma^2}} \\
&= \sum_{h_1} \sum_{h_2} \cdots \sum_{h_N} e^{\sum_{i=1}^{M} \frac{(X_i - a_i)^2}{2\sigma^2}} \times \\
&\quad e^{b_1 h_1 + \sum_{i=1}^{M} \frac{X_i W_{i1} h_1}{\sigma^2}} e^{b_2 h_2 + \sum_{i=1}^{M} \frac{X_i W_{i2} h_2}{\sigma^2}} \cdots e^{b_N h_N + \sum_{i=1}^{M} \frac{X_i W_{iN} h_N}{\sigma^2}} \\
&= e^{\sum_{i=1}^{M} \frac{(X_i - a_i)^2}{2\sigma^2}} \prod_{j=1}^{N} \sum_{h_j=0}^{1} e^{b_j h_j + \sum_{i=1}^{M} \frac{X_i W_{ij} h_j}{\sigma^2}} \\
&= e^{\sum_{i=1}^{M} \frac{(X_i - a_i)^2}{2\sigma^2}} \prod_{j}^{N} (1 + e^{b_j + \frac{\boldsymbol{x}^T \boldsymbol{W}_{*j}}{\sigma^2}})
\end{aligned} \tag{D.4}$$

Appendix E

Wave Function Elements

E.1 Kinetic Energy Calculations

The local energy, defined in equation (4.5), is

$$E_L = \frac{1}{\Psi_T}\hat{\mathrm{H}}\Psi_T \tag{E.1}$$

$$= \sum_{k=1}^{M}\Big[-\frac{1}{2\Psi_T}\nabla_k^2\Psi_T + U_k + V_k\Big]. \tag{E.2}$$

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

$$\frac{1}{\Psi_T}\nabla_k\Psi_T = \nabla_k \ln\Psi_T, \tag{E.3}$$

which provides the following useful relation

$$\frac{1}{\Psi_T}\nabla_k^2\Psi_T = \nabla_k^2 \ln\Psi_T + (\nabla_k \ln\Psi_T)^2. \tag{E.4}$$

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

$$\Psi_T = \prod_{i=1}^{p}\phi_i. \tag{E.5}$$

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

$$\frac{1}{\Psi_T}\nabla_k^2\Psi_T = \sum_{i=1}^{p}\nabla_k^2 \ln\phi_i + \Big(\sum_{i=1}^{p}\nabla_k \ln\phi_i\Big)^2, \tag{E.6}$$

which can be computed given all local derivatives $\nabla_k^2 \ln\phi_i$ and $\nabla_k \ln\phi_i$. 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.

E.2 Parameter Update

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

$$\partial_{\alpha_i} E_L \equiv \frac{\partial E_L(\alpha_i)}{\partial \alpha_i}. \tag{E.7}$$

If we assume that each parameter, α, only exists in a wave function element,

$$\Psi_T(\alpha) = \phi_1(\alpha) \prod_{i=2}^{p} \phi_i \tag{E.8}$$

the derivative of the entire local energy is reduced to the derivative of the kinetic energy term given the wave function element,

$$\partial_\alpha E_L = -\frac{1}{2}\partial_\alpha \Big(\nabla_k^2 \ln \phi_1(\alpha) + \sum_{i=2}^{p} \nabla_k^2 \ln \phi_i + \Big(\nabla_k \ln \phi_1(\alpha) + \sum_{i=2}^{p} \nabla_k \ln \phi_i \Big)^2 \Big) \tag{E.9}$$

$$= -\frac{1}{2}\partial_\alpha \nabla_k^2 \ln \phi_1(\alpha) - \Big(\sum_{i=1}^{p} \nabla_k \ln \phi_i \Big) \cdot \partial_\alpha \nabla_k \ln \phi_1(\alpha). \tag{E.10}$$

The sum is already evaluated in the kinetic energy calculations, which means that to calculate the gradients, it is sufficient to calculate $\partial_\alpha \nabla_k \ln \phi_1(\alpha)$ and $\partial_\alpha \nabla_k^2 \ln \phi_1(\alpha)$ for all wave function elements with respect to their variational parameters.

E.3 Derivatives

E.3.1 Simple Gaussian

A natural starting point is the Gaussian function, since it appears in all harmonic oscillator calculations. For N particles in D dimensions, the function is given by

$$\Psi(\alpha, \boldsymbol{r}) = \exp\Big[-\frac{1}{2}\alpha \sum_{i=1}^{N} r_i^2 \Big] \tag{E.11}$$

where the derivative with respect to coordinate r_k is

$$\nabla_k \ln \Psi(\alpha) = -\alpha r_k \tag{E.12}$$

and the second derivative is

$$\nabla_k^2 \ln \Psi(\alpha) = -\alpha N D. \tag{E.13}$$

The gradients for those derivatives are

$$\partial_\alpha \nabla_k \ln \Psi(\alpha) = -r_k \tag{E.14}$$

and

$$\partial_\alpha \nabla_k^2 \ln \Psi(\alpha) = -ND \tag{E.15}$$

respectively.

E.3.2 Padé-Jastrow Factor

The Padé-Jastrow factor is introduced in order to take care of the correlations. It is specified in equation (2.15),

$$J(\boldsymbol{r};\beta,\gamma) = \exp\left(\sum_{i=1}^{N}\sum_{j=1}^{i}\frac{\beta(i,j)r_{ij}}{1+\gamma r_{ij}}\right) \tag{E.16}$$

, which gives the first and second derivatives

$$\nabla_k \ln J = \sum_{j=1}^{k}\left(\frac{\beta_{kj}}{(1+\gamma r_{kj})^2}\right) \tag{E.17}$$

and

$$\nabla_k^2 \ln J = \sum_{j=1}^{k}\left(\frac{\beta_{kj}}{(1+\gamma r_{kj})^2}\left[\frac{1}{r_k} - \frac{\gamma}{1+\gamma r_{kj}}\right]\right) \tag{E.18}$$

respectively.

The derivative of those again, with respect to the parameters γ and β_{kl} are

$$\partial_\gamma \nabla_k \ln J = -2\sum_{j=1}^{k}\left(\frac{\beta_{kj}}{(1+\gamma r_{kj})^3}\right) \tag{E.19}$$

$$\partial_\gamma \nabla_k^2 \ln J = -2\sum_{j=1}^{k}\left(\frac{\beta_{kj}}{(1+\gamma r_{kj})^3}\left[\frac{1}{r_{kj}} + \frac{2}{r_k} - \frac{3\gamma}{1+\gamma r_{kj}}\right]\right) \tag{E.20}$$

$$\partial_{\beta_{kl}} \nabla_k \ln J = \frac{1}{(1+\gamma r_{kl})^2} \tag{E.21}$$

$$\partial_{\beta_{kl}} \nabla_k^2 \ln J = \frac{2}{(1+\gamma r_{kl})^2}\left[\frac{1}{r_l} - \frac{\gamma}{1+\gamma r_{kl}}\right] \tag{E.22}$$

E.3.3 Slater Determinant

We need to introduce the Slater Determinant to study more fermions.

E.3.4 NQS-Gaussian

Now over to the real deal; the machine learning inspired wave function elements.

E.3.5 NQS-Jastrow Factor

E.3.6 Partly Restricted Element

E.3.7 Hydrogen-Like Orbitals

$$\Psi(\alpha,\boldsymbol{r}) = \exp\Big[-\frac{1}{2}\alpha\sum_{i=1}^{N} r_i\Big] \tag{E.23}$$

where the derivative with respect to coordinate r_k is

$$\nabla_k \ln \Psi(\alpha) = -\alpha \tag{E.24}$$

and the second derivative is

$$\nabla_k^2 \ln \Psi(\alpha) = 0 \tag{E.25}$$

The gradients for those derivatives are

$$\partial_\alpha \nabla_k \ln \Psi(\alpha) = -1 \tag{E.26}$$

and

$$\partial_\alpha \nabla_k^2 \ln \Psi(\alpha) = 0 \tag{E.27}$$

respectively.

Bibliography

[1] A.M. Legendre. "Nouvelles méthodes pour la détermination des orbites des comètes". In: (1805).

[2] C.F. Gauss. "Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientum". In: (1809).

[3] P. A. M. Dirac. "A new notation for quantum mechanics". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 35.3 (1939), pp. 416–418. DOI: 10.1017/S0305004100021162.

[4] W. Heisenberg. *Physics and Beyond: Encounters and Conversations*. Harper torchbooks. The Academy library. Allen and Unwin, 1971. URL: https://books.google.no/books?id=0-dEAAAAIAAJ.

[5] J.M. Leinaas and J. Myrheim. "One the theory of identical particles". In: *IL NUOVO CIMENTO* 37.1 (Aug. 1977).

[6] M. Taut. "Two electrons in an external oscillator potential: Particular analytic solutions of a Coulomb correlation problem". In: *Physical Review A* 48.5 (Nov. 1993), pp. 3561–3566. DOI: 10.1103/PhysRevA.48.3561. URL: https://link.aps.org/doi/10.1103/PhysRevA.48.3561 (visited on 03/13/2019).

[7] 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.

[8] D.J. Griffiths. *Introduction to quantum mechanics*. 2nd Edition. Pearson PH, 2005. ISBN: 0-13-191175-9.

[9] 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.

[10] 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.

[11] E.W Weisstein. *Kelvin, Lord William Thomson (1824-1907)*. 2007. URL: http://scienceworld.wolfram.com/biography/Kelvin.html.

[12] D.A. Nissenbaum. "The stochastic gradient approximation: an application to li nanoclusters". In: (2008).

[13] 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.

[14] 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).

[15] 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.

[16] Morten Ledum. *Simple Variational Monte Carlo solve for FYS4411*. 2016. URL: https://github.com/mortele/variational-monte-carlo-fys4411.

[17] 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).