Studies of Quantum Dots using Neural Networks and Coupled Cluster

by

Karl Henrik Fredly

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

October 2022

Abstract

Recent advances in the use of machine learning to solving the many-electron Schrödinger equation have shown that neural networks can compete with, and even outperform state of the art methods[17]. In addition to great performance, the use of neural networks as an ab-initio method, requiring no prior knowledge about the system, opens the possibility of solving systems that were previously unsolvable. The many-electron Schrödinger equation is notoriously difficult to solve, requiring approximate methods that obey Fermi-Dirac statistics. The variation Monte Carlo method(VMC) is a popular choice, requiring only a trial wave function that is flexible, fast to optimize and that obeys Fermi-Dirac statistics. Neural networks are a natural choice, but they do not obey Fermi-Dirac statistics on their own. One of the leading approaches called FermiNet solves this issue by merging neural networks into anti-symmetric Slater determinants. We instead build upon the work of others who have used a more simple approach of multiplying a neural network by an anti-symmetric Slater determinant.

We advance this work by showing that by using a Slater determinant optimized by the restricted Hartree-Fock method, and a neural network with a single hidden layer, we can approximate the ground state energy of 2 quantum dots in one dimension with a relative error of only 0.0004, compared to the coupled cluster doubles method, and the coupled cluster singles and doubles method which have relative errors of 0.0154 and 0.0002 respectively. This works as a proof of concept of utilizing the Hartree-Fock solution in this way, and of solving a multi-electronic system with no explicit use of the distance between particles. A central focus of our work has also been optimizing the algorithms used. We demonstrate an implementation of automatic differentiation that can compute double derivatives of a neural network 400 times faster than applying reverse mode twice, now offering speeds comparable to analytical formulas. This opens up the door for the use of wave functions which have no simple expressions for their double derivatives. We also present many other optimizations including specialized sorting, a very efficient method of evaluating the component functions of the Slater determinant, and many efficient uses of the Julia programming language that add up to a very significant speedup over comparable codes. Finally, in addition to this implementation of VMC for Slater determinants and neural networks, we have implemented the restricted Hartree-Fock, general Hartree-Fock, coupled cluster doubles, and coupled cluster singles doubles methods, to be used in conjunction with the VMC implementation and as benchmarks.

Acknowledgements

As my master studies comes to a close I would like to thank my supervisor Morten for helping me hone my skills in computational physics, and for helping me stay motivated and encouraging my efforts through the challenge of writing this thesis. I would also like to thank my co-supervisor Øyvind Schøyen and Even Nordhagen for tolerating my endless questions about many-body methods, and for being great people to hang around with.

I would also like to thank Tor Ole Odden for being an excellent mentor through my undergraduate and masters studies, offering great opportunities to apply myself and participate in exciting projects. Your help with finding my academic path has been invaluable, and I look forward to keep working with you in the years to come.

Finally, I would like to thank Ingrid for all the love and support you have given me these last months.

Contents

Chapter 1

Introduction

Quantum physics is the theory that describes the behavior of particles at scales much smaller than we can see, though the combined effects of quantum physics are very much visible. Every chemical reaction, every collision of objects, every breath you take is at the lowest level completely described by quantum mechanics. If we could solve any quantum mechanical problem then, we could solve (almost) any physics or chemistry problem. So why don't we then?

The problem is that quantum physics is *hard*. Every particle has positions it can be observed at, and the possible positions of one particle affect the possible positions of another particle, which in turn affects the possible positions of first particle again, and we are left with an impossible mess of particles interacting with where other particles *may* be observed and vice-versa. The function describing the probability of where a particle can be observed can even interact with *itself*.

Although these quantum many-body problems might sound impossible to solve, many of them can be solved efficiently by approximate many-body methods. We will present three of these methods in this thesis: The Hartree-Fock method, the coupled cluster method, and the variational Monte Carlo method. The field of many-body quantum mechanics is constantly evolving, in recent times especially the application of machine learning to solving these systems. I will attempt to contribute to this work as best I can, applying my skills and interest to this exciting field.

1.1 Goals and contributions

Our goals with this theses are threefold:

1. Implement a variational Monte Carlo code combining a Slater determinant using orbitals from Hartree-Fock with a neural network

2. Implement the restricted Hartree-Fock method, the general Hartree-Fock method, the coupled cluster doubles method, and the couple cluster singles and doubles method

3. Make the code really fast

We have achieved all of these goals, and the result is this thesis and the Julia package OrbitalNeuralMethods, which can be found at https://github.com/KarlHenrik/ComputationalPhysicsMaster. While plenty of implementations of latter four solvers certainly already exist, we believe there is no better way of understanding a method than implementing and bug-fixing it yourself. In addition to this, we have been able to find new optimizations for our specific type of VMC solver that makes our codes useful for both use and inspiration. One of the optimizations found was also implemented in the quantum_systems[21] python library by us.

In addition to these optimizations, our novel VMC solver showed great accuracy in solving the system of two quantum dots in 1D, motivating further studies.

1.2 Thesis structure

Our thesis is split into four parts: Theory, implementation, results and conclusion. The theory is written with the intention of being read in order, as all chapters refer to results and theory in a previous chapter. The theory begins by focusing on quantum mechanics and classical many-body methods. The first three chapters work as a build up and then explanation of the coupled cluster method in this way. The final four chapters are more focused on the theory behind the algorithms used in the VMC calculations. The implementation has a strong focus on code, as much of our efforts were spent writing efficient code. The results contain a mix of physics and computational results, as both are central to what we have tried to achieve in this thesis.

Finally, the conclusion contains a summary of the results from this thesis, and suggestions for future studies in the same vein.

Part I

Theory

Chapter 2

Quantum Mechanics

Quantum mechanics is the theory which describes how very small particles behave, such as electrons, atoms and subatomic particles. At this scale, classical physics fails to describe the strange behaviour of particles, and as such, the theory of quantum mechanics is needed. At large scales however, quantum mechanical effects are negligible, and the models of classical physics are much more practical. In this thesis, we will use the theory of quantum mechanics to study systems of electrons. This chapter will cover some preliminary quantum mechanics needed before we present the different methods we will use to solve the Schrödinger equation.

2.1 The wave function

The behaviour of a quantum mechanical system is completely described by its wave function $|\Psi\rangle$. This function describes the probability of the particles having certain positions and velocities, among other values of interest. This is in contrast to classical physics, which instead just provides actual concrete values, as would be the case for a description the Earth's orbit around the sun.

The behaviour of particles are inherently non-deterministic. Particles don't *have* a position or velocity until they are measured. They only have *probabilities* of being measured with certain values for observables such as position and momentum. These probability distributions can be found however, and they let us make predictions of the likelihood of measurements. If we describe a single particle by the spatial wave function $\psi(x)$ for instance, the probability of observing the particle between positions a and b is

$$\int_a^b |\psi(x)|^2 dx. \tag{2.1}$$

The expectation value of its positions can similarly be found by

$$\int \psi(x)^* x \psi(x)\, dx. \tag{2.2}$$

If we instead of the spatial function $\psi(x)$ express the wave function as the quantum state $|\psi\rangle$ using bra-ket notation, we write expectation values as

$$\langle\psi|\, x \,|\psi\rangle\,. \tag{2.3}$$

The probability of finding a particle in a state $\psi(x)$ in another state $\varphi(x)$ can be written using spatial function and bra-ket notation as

$$\int \varphi^*(x)\psi(x)\, dx = \langle\varphi|\psi\rangle\,. \tag{2.4}$$

The wave function lets us calculate all such probabilities of measurements and expectation values. This is all the information there really *is* about the system, which is why we say that it is a complete description. We just saw that the expectation value of position is found by putting x inside the inner product, we therefore say that the position operator is $\hat{x} = x$. Likewise, we have the momentum operator

$$\hat{p} = -i\hbar \frac{\partial}{\partial x}, \tag{2.5}$$

where $i = \sqrt{-1}$ and $\hbar$ is Planck's constant over 2π. Another important operator is the energy operator $\hat{H}$, the Hamiltonian, whose form depends on the system at hand.

A measurement of a quantum state can be expressed as applying a suitable operator to the state. For instance, a measurement of energy can be expressed by the equation

$$\hat{H}\,|\Psi\rangle = E\,|\Psi\rangle\,, \tag{2.6}$$

where E is the measured energy of the state $|\Psi\rangle$. A given system described by $\hat{H}$ has eigenstates $|\Psi_i\rangle$ with eigenvalues E_i. Our state $|\Psi\rangle$, might not be one of these eigenstates, but it can always be written as a linear combination of them. When measuring the energy of our state, only the eigenvalues E_i of the eigenstates included in the linear combination have a chance of being observed.

The type of problem we will be solving in this thesis is one where we only have the Hamiltonian describing a system, and where we then wish to find the eigenstate of $\hat{H}$ with the lowest energy, the ground state. The ground state is often the most interesting because it is the state that we most often see in nature, and because it often is the starting point for finding the eigenstates with higher energies, the excited states.

2.2 The Electronic Schrödinger Equation

Given a physical system described by a Hamiltonian $\hat{H}$, we can find the eigenstates of the system by solving the Schrödinger equation:

$$i\hbar \frac{\partial}{\partial t} |\Psi\rangle = \hat{H} |\Psi\rangle . \tag{2.7}$$

If the Hamiltonian is time-independent, we can separate out the time-dependence, and instead solve the time-independent Schrödinger equation (note that we implicitly remove the time dependence from the state $|\Psi\rangle$):

$$\hat{H} |\Psi\rangle = E |\Psi\rangle , \tag{2.8}$$

where $\hat{H}$ is the Hamiltonian operator, $|\Psi\rangle$ is the eigenstate of the Hamiltonian, and E is its eigenvalue, the energy of the state. The Hamiltonian acts as a description of a physical system, and it is our goal to find the eigenstate $|\Psi\rangle$ with the lowest energy, the ground state.

We limit ourselves to time-independent electronic systems of N electrons where the Hamiltonian contains the kinetic energy of the electrons, the energy from a one-body potential V, and the Coulomb repulsion between electrons.

$$\hat{H} = \sum_{i}^{N} -\frac{1}{2}\nabla_i^2 + V(\mathbf{x}_i) + \sum_{i<j}^{N} \frac{1}{r_{ij}}, \tag{2.9}$$

where the restricted double sum $i < j$ ensures that we don't double count the repulsion between electrons.

The one-body potential $V(\mathbf{x}_i)$ will typically be a harmonic oscillator potential or the Coulomb repulsion from protons, but can be any one-body potential. If we include atomic nuclei in our systems of interest we will assume that they are stationary, since they generally move much slower than electrons (the Born-Oppenheimer approximation).

2.3 Atomic Units

The equations above, and the equations in the rest of this thesis, are written in atomic units[a.u.]. These units make the equations much more workable, since most physical constants simply vanish. The atomic unit for energy is the Hartree $E_a = 4.3498 \times 10^{-18}$J, the atomic unit of charge is the charge of a proton e, the atomic unit of length is the Bohr radius $a_0 = 5.2918 \times 10^{-11}$m, and the atomic unit of mass is the mass of an electron m_e. These are the important ones for our work.

2.4 The Variational Principle

An important theoretical result when trying to find the ground state of a system is the variational principle. We stated earlier that a system described by the Hamiltonian $\hat{H}$ has eigenstates $|\Psi_i\rangle$ with eigenvalues E_i. And that any state $|\Psi\rangle$ can be written as a linear combination of these eigenstates.

$$|\Psi\rangle = \sum_i c_i |\Psi_i\rangle . \tag{2.10}$$

The sum of c_i^2 must be 1 for the state to be normalized, meaning that the probability of a state being observed in itself $\langle\Psi|\Psi\rangle$ is 1. An important result is that these eigenstates are orthogonal, meaning a particle in only one eigenstate has a probability 0 of being observed in another. Using these results, we can show that the expectation value of the energy of a state Ψ can never be lower than the lowest eigenvalue E_i.

$$\langle\Psi| \hat{H} |\Psi\rangle = \sum_{ij} c_i c_j \langle\Psi_i| \hat{H} |\Psi_j\rangle = \sum_{ij} c_i c_j \langle\Psi_i| E_j |\Psi_j\rangle = \sum_i c_i^2 E_i \geqslant E_0, \tag{2.11}$$

where E_0 is the energy of the ground state. We used the fact that the other eigenstates of $\hat{H}$ have energies higher than E_0, along with the orthogonality and normalization (orthonormality) of the eigenstates. The variational principle ensures us that there is no state with a lower energy than the ground state, which means that if we find a state that minimizes the expected energy, that state is the ground state.

This result is the cornerstone of the variational method, where we have a parameterized wave function and then optimize the parameters to minimize the energy. This is a very powerful method, as when we use an approximate or iterative method we can be sure that as long as its wave function gets a low value for energy, it is a good approximation to the ground state. The Hartree-Fock method and Variational Monte-Carlo methods fall in this category.

Some methods for solving the Schrödinger equation are not variational, normally because they use a simplified Hamiltonian which makes the equation solvable, but not exactly correct. When using these methods, we cannot be sure whether the energy of a solution is low because it is close to the ground state, or because our simplified Hamiltonian allowed it. The coupled cluster method falls in this category, but we can still be very confident in its results because of the strong theoretical backing and proven track record of the method.

2.5 Many-electron systems

The systems we will be solving in this thesis are many-electron systems. These systems pose some unique challenges due to their inherent complexity. This complexity is due to two things: First, electrons interact via the Coulomb force. In classical physics, this is also a challenge, because many-body systems are expensive to simulate and are very chaotic. But the physics used to solve these systems are still relatively simple: Electrons at given positions are affected by easily computable forces which give them a certain acceleration which can be used to find where they move, and then repeat.

When we use quantum physics however, the Coulomb force between even just two electrons is not straightforward: Both electrons have a probability distribution of where they can be observed, so when computing the Coulomb force, the electrons cannot be treated as simple points. If electrons have a small probability of being observed close together and a large probability of being observed far apart, both of these possibilities affect the Coulomb force, and the Coulomb force in turn affects how the electrons behave. Every possible position of electron 1 is affected by every possible position of electron 2. This complexity of course increases with the number of electrons.

The second challenge when working with systems of electrons is that electrons are fermions, which means that they have one of two spin values, cannot occupy the same state, and that the total wave function must change sign when you interchange two electrons, or in other words, it must be anti-symmetric. We get around this challenge by only looking at systems with an even number of electrons, assuming that half have spin up, and half have spin down, as is the case for the ground state of spin-independent systems. These are called closed-shell systems. We also use a Slater determinant to ensure that the wave function is anti-symmetric. The next chapter will cover how the Slater determinant is an anti-symmetric wave function, and how the Hartree-Fock method simplifies the Coulomb interaction to find the optimal one.

Chapter 3

Hartree-Fock theory

In the previous chapter, we presented the electronic Schrödinger equation, and the two main problems faced when solving it: The anti-symmetry of electrons, and the correlations due to the Coulomb force. In this chapter, we will present the Slater determinant, which solves the first problem, and the Hartree-Fock method, which works as a good starting point for solving the second problem.

Later, we will use the coupled cluster method to improve upon the Hartree-Fock solution. And following that, we will attempt to improve upon the Hartree-Fock solution in yet another way: Using a neural network and the variational Monte-Carlo method to model the correlations between electrons. They both build on the Hartree-Fock solution though, an optimized Slater determinant.

3.1 The Slater Determinant

Imagine that you are looking at two electrons that are trapped together in some well potential. If one electron is observed in the center of the well, you know that the other electron is unlikely to be observed near the center, as the electrons repel each other and "tend to avoid each other". This simple example illustrates that you cannot give each electron its own one-electron function that ignores the other electrons, since the movement of electrons are correlated.

The electronic Hamiltionian of equation 2.9 has a two-body term, which means that one cannot separate equation 2.8 into one equation for each electron. If this was possible, we could solve the one-electron equations separately and then combine the resultant one-electron functions into a solution to the whole problem.

Despite this not being a good solution, as we have just discussed, we now choose to approximate our wave function as the product of one-electron functions ϕ_i anyway

$$\Psi^{HP} = \phi_1(\mathbf{x}_1)\phi_2(\mathbf{x}_2)...\phi_n(\mathbf{x}_n), \tag{3.1}$$

where the numbers 1...n indicate which electron goes with each function. This type of wavefunction is called a Hartree-Product.

The one-electron functions are called spin-orbitals, and contain a spatial and spin component.

$$|\phi\rangle = \phi(\mathbf{x}) = \phi_1(x)\alpha(m_s) + \phi_2(x)\beta(m_s), \tag{3.2}$$

where

$$\begin{aligned} \alpha(\uparrow) = 1, \quad \beta(\uparrow) = 0, \\ \alpha(\downarrow) = 0, \quad \beta(\downarrow) = 1. \end{aligned} \tag{3.3}$$

The Hartree-Product wavefunction has the problem that it is not-antisymmetric: Swapping electron 1 and 2 does not change the sign of the wavefunction. This can be solved by adding together all possible orderings of the one-electron functions with altering signs depending on the number of permutations. One must also normalize the function depending on the number of terms in the sum. For two electrons this will look like:

$$\begin{aligned} |\mathbf{\Phi}_0\rangle &= \frac{1}{\sqrt{2}} \begin{vmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) \end{vmatrix}, \\ &= \frac{1}{\sqrt{2}}(|\phi_1(\mathbf{x}_1)\phi_2(\mathbf{x}_2)\rangle - |\phi_2(\mathbf{x}_1)\phi_1(\mathbf{x}_2)\rangle), \\ &= |\phi_1\phi_2\rangle\,, \end{aligned} \tag{3.4}$$

where the final notation implies both the anti-symmetrization, $\mathbf{x}$-ordering and normalization. This type of wavefunction is called a Slater determinant, and for n electrons it can be written as

$$\begin{aligned} |\mathbf{\Phi}_0\rangle &= \frac{1}{\sqrt{n!}} \begin{vmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \dots & \phi_n(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \dots & \phi_n(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{x}_n) & \phi_2(\mathbf{x}_n) & \dots & \phi_n(\mathbf{x}_n) \end{vmatrix}, \\ &= |\phi_1\phi_2\dots\phi_n\rangle\,. \end{aligned} \tag{3.5}$$

3.2 The Hartree-Fock equation

Minimizing the energy of equation 2.8 with a single Slater determinant wave function is the same as solving the Hartree-Fock equation [24]

$$
\begin{aligned}
(-\frac{1}{2}\nabla_1^2 + V(\mathbf{x}_1))\phi_i(\mathbf{x}_1) + \sum_{j\neq i}\left[\int d\mathbf{x}_2|\phi_j(\mathbf{x}_2)|^2\frac{1}{r_{12}}\right]\phi_i(\mathbf{x}_1)& \\
-\sum_{j\neq i}\left[\int d\mathbf{x}_2\phi_j^*(\mathbf{x}_2)\frac{1}{r_{12}}\phi_i(\mathbf{x}_2)\right]\phi_j(\mathbf{x}_1) = \epsilon_i\phi_i(\mathbf{x}_1).&
\end{aligned}
\tag{3.6}
$$

The first term in equation 3.6 is the kinetic energy and one-body potential from the electronic Hamiltonian of equation 2.9. We define

$$\hat{h}(1) = -\frac{1}{2}\nabla_1^2 + V(\mathbf{x}_1). \tag{3.7}$$

The second term is the mean-field coulomb potential at position $\mathbf{x_1}$, set up by electrons in all states ϕ_j different from the chosen state ϕ_i for which this is the eigenvalue equation. By taking the average interaction from all other particles, weighted by their probability density, we have simplified the Coulomb interaction into a one-body potential. This is an approximation, as there is no "average-field" in the actual Coulomb interaction, each pair of possible positions has a single interaction strength. The third term is an exchange term which appears due to the wave function being a Slater determinant, which introduces terms with the orbitals permuted. It has no simple physical interpretation.

We can combine the second and third terms using the permutation operator $P(ij)$ which has the effect

$$P(ij)g(i,j) = g(i,j) - g(j,i). \tag{3.8}$$

We can also remove the restrictions on both sums, as the two extra terms will cancel out. We define the Hartree-Fock potential

$$\hat{v}^{HF}(1) = \sum_j \int d\mathbf{x}_2\phi_j^*(\mathbf{x}_2)\frac{1}{r_{12}}P(1,2)\phi_j(\mathbf{x}_2). \tag{3.9}$$

and the Fock operator

$$\hat{f}(1) = \hat{h}(1) + \hat{v}^{HF}(1). \tag{3.10}$$

The Hartree-Fock equations can now be written as

$$\hat{f}\,|\phi_i\rangle = \epsilon_i\,|\phi_i\rangle\,. \tag{3.11}$$

Now we need to find the molecular orbitals $|\phi_i\rangle$ that satisfy equation 3.11. Since the operator f depends on the orbitals $|\phi_i\rangle$, this is a non-linear equation, which we will choose to solve iteratively.

3.3 The Self-Consistent field procedure

Equation 3.11 is solved by first writing the orbitals $|\phi_i\rangle$ as a linear combination of the orthonormal basis functions (using a non-orthogonal basis makes things more complicated)

$$|\phi_i\rangle = \sum_{\mu}^{L} C_{\mu i} |\psi_\mu\rangle . \tag{3.12}$$

The equation can be solved exactly with a complete basis, but in practice one has to choose a limited set of L basis function which hopefully fits the system at hand well. The choice and number of basis functions are central to both the speed of the calculation and the accuracy of the result. A normal way to construct the basis is to use a set of L spatial orbitals $\xi(x)$ and then make a spin-up and a spin-down version of each, to get 2L basis functions that also include spin

$$\begin{aligned} \psi_i(\mathbf{x}) &= \xi_i(x) \otimes \alpha(m_s) \quad \text{for } i = 1,2,3,...,N/2 \\ \psi_{\frac{N}{2}+i}(\mathbf{x}) &= \xi_i(x) \otimes \beta(m_s) \quad \text{for } i = 1,2,3,...,N/2, \end{aligned} \tag{3.13}$$

where

$$\begin{aligned} \alpha(\uparrow) = 1, \quad \beta(\uparrow) = 0, \\ \alpha(\downarrow) = 0, \quad \beta(\downarrow) = 1. \end{aligned} \tag{3.14}$$

Using this basis the Fock-operator's matrix elements can be written as

$$\begin{aligned} \hat{f}_{\mu\nu} &= \langle\psi_\mu| \hat{f} |\psi_\nu\rangle , \\ &= \langle\psi_\mu| \hat{h} |\psi_\nu\rangle + \sum_{i=1}^{n} \langle\psi_\mu\phi_i| \frac{1}{r_{12}} |\psi_\nu\phi_i\rangle - \langle\psi_\mu\phi_i| \frac{1}{r_{12}} |\phi_i\psi_\nu\rangle , \\ &= \langle\psi_\mu| \hat{h} |\psi_\nu\rangle + \sum_{i=1}^{n} \langle\psi_\mu\phi_i| \frac{1}{r_{12}} |\psi_\nu\phi_i\rangle_{AS} , \\ &= \hat{h}_{\mu\nu} + \sum_{i=1}^{n} \sum_{\kappa,\lambda=1}^{l} C^*_{\kappa i} C_{\lambda i} \langle\psi_\mu\psi_\kappa| \frac{1}{r_{12}} |\psi_\nu\psi_\lambda\rangle_{AS} , \\ &= \hat{h}_{\mu\nu} + \sum_{\kappa,\lambda=1}^{l} D_{\lambda\kappa} \langle\psi_\mu\psi_\kappa| \frac{1}{r_{12}} |\psi_\nu\psi_\lambda\rangle_{AS} , \\ &= \hat{h}_{\mu\nu} + \sum_{\kappa,\lambda=1}^{l} D_{\lambda\kappa} \langle\mu\kappa\|\nu\lambda\rangle, \end{aligned} \tag{3.15}$$

where we defined the density matrix $D_{\lambda\kappa} = \sum_{i=1}^{n} C^*_{\kappa i} C_{\lambda i}$, and introduced a shorthand for anti-symmetric two-body integrals (AS), and an even shorter shorthand for anti-symmetric two-body integrals between states from an established basis. With these matrix elements of $\hat{f}$, we can project equation 3.11 onto an element of our basis and get

$$\begin{aligned} \langle\psi_\mu|\,\hat{f}\,|\phi_p\rangle &= \varepsilon_p \langle\psi_\mu|\phi_p\rangle . \\ \sum_{\nu=1}^{l} C_{\nu p} f_{\mu\nu} &= \varepsilon_p \sum_{\nu=1}^{l} C_{\nu p}\delta_{\mu\nu}. \\ \implies \mathbf{FC} &= \mathbf{C}\boldsymbol{\epsilon}. \end{aligned} \tag{3.16}$$

where the latter eigenvalue equations are known as the Roothan-Hall equations. This looks like a simple eigenvalue problem, the columns of **C** will be the eigenvectors of **F**, and $\boldsymbol{\epsilon}$ will be the eigenvalues. However, **F** depends on **C**, so solving the equation is not so simple.

An iterative scheme is employed to optimize the coefficients C. First, we make an initial guess for C, typically the identity matrix, then we compute the density matrix D and the Fock-matrix F. Using any eigen-vector solver, we can then compute the eigenvectors of F and use them as the columns of our next guess for C. If the new C is the same as the old C, it means that our guess for the optimal orbitals $|\phi_i\rangle$ are eigenstates of the Fock-matrix they set up, and the equations are solved. If not, we need repeat the process, recomputing F, finding the eigenvectors to set up the new C, and so on until the algorithm converges. At that point, we say that we have a self-consistent field, as the states are eigenstates of the potentials they set up. This process is not guaranteed to converge, but convergence can be improved by using a convergence acceleration method, as described in section 10.2.

What we get from this solution are the L orbitals $|\phi_i\rangle$ that satisfy equation 3.11. The N orbitals with the lowest energies ϵ_i then form the single Slater determinant with the lowest possible energy, the Hartree-Fock wave function. The expected energy of this state is given by

$$E_0 = \sum_{\mu}\sum_{\nu} D_{\nu\mu}(\hat{h}_{\mu\nu} + \frac{1}{2}\hat{f}_{\mu\nu}). \tag{3.17}$$

3.4 Restricted Closed-Shell Hartree-Fock

Say we are working with a system of an even number of electrons whose Hamiltonian has no spin-dependence. If we solve for the ground state of the system and find the spatial orbitals that the electrons will occupy, the orbital with the lowest energy will be occupied by two electrons with opposite spin. If there are more electrons left, the orbital with the next lowest energy will also be doubly occupied, and so on. This is because this is the distribution of electrons in orbitals that gives the lowest energy. No more than two electrons can occupy the same spatial orbital, because two electrons cannot occupy the same state, and there are only two possible spin states to go with the single spatial state with the given energy.

This is all to say that for the ground state of a system with no spin-dependence (no magnetic field or other things that interact with spin), N electrons will doubly occupy $N/2$ spatial orbitals. But if you recall how we defined the Hartree-Fock orbitals:

$$|\phi_i\rangle = \sum_{\mu}^{L} C_{\mu i} |\psi_\mu\rangle , \tag{3.18}$$

each one has its own unique coefficients $C_{\mu i}$, which allows the electrons to all occupy unique spatial orbitals. This could become a problem, as the Hartree-Fock determinant might not satisfy the above condition of the true ground state.

We remedy this by forcing the occupied spatial orbitals to all be doubly occupied. This restriction gives the restricted closed-shell Hartree-Fock method. If we had an odd number of electrons, one spatial orbital would only be singly occupied, and we could use the open-shell restricted Hartree-Fock method. We will limit ourselves to closed-shell systems however, and thus only use the restricted Closed-Shell Hartree-Fock method, from here on called simply the restricted Hartree-Fock method. We will also implement the more general formulation of Hartree-Fock theory described above, but as we will see, the resulting orbitals end up with un-physical spin, which makes them unsuited for our purposes.

The restricted Hartree-Fock method (RHF) uses a basis of L spatial orbitals $\xi(x)$, and gives L optimized spatial orbitals $\phi(x)$, of which we use the $N/2$ with the lowest energy to form the optimal restricted spin-orbitals

$$\begin{aligned} \phi_i(\mathbf{x}) &= \phi_i(x) \otimes \alpha(m_s) \quad \text{for } i = 1, 2, 3, ..., N/2. \\ \phi_{\frac{N}{2}+i}(\mathbf{x}) &= \phi_i(x) \otimes \beta(m_s) \quad \text{for } i = 1, 2, 3, ..., N/2. \end{aligned} \tag{3.19}$$

Because all spatial orbitals are included twice, and we use each column in $\mathbf{C}$ for two spin orbitals, the expressions used in the self-consistent field procedure are slightly different. For a full treatment of these equations, see [24]. The two most important ones are the Fock-matrix for RHF:

$$\begin{aligned} \hat{f}_{\mu\nu} &= \langle\psi_\mu| \hat{f} |\psi_\nu\rangle , \\ &= \langle\psi_\mu| \hat{h} |\psi_\nu\rangle + \sum_{i=1}^{n} \langle\psi_\mu\phi_i| \frac{1}{r_{12}} |\psi_\nu\phi_i\rangle - \frac{1}{2} \langle\psi_\mu\phi_i| \frac{1}{r_{12}} |\phi_i\psi_\nu\rangle , \\ &= \hat{h}_{\mu\nu} + \sum_{\kappa,\lambda=1}^{l} D_{\lambda\kappa} \left(\langle\psi_\mu\psi_\kappa| \frac{1}{r_{12}} |\psi_\nu\psi_\lambda\rangle - \frac{1}{2} \langle\psi_\mu\psi_\kappa| \frac{1}{r_{12}} |\psi_\lambda\psi_\nu\rangle \right) . \end{aligned} \tag{3.20}$$

And the expectation value of energy for RHF

$$E_0 = \frac{1}{2} \sum_{\mu} \sum_{\nu} D_{\nu\mu} (\hat{h}_{\mu\nu} + \hat{f}_{\mu\nu}). \tag{3.21}$$

3.5 The one- and two-body integrals

Some useful integrals for both the Hartree-Fock procedure and Coupled Cluster are the one-electron integrals

$$\hat{h}_{ai} \equiv \langle \psi_a | \hat{h} | \psi_i \rangle, \tag{3.22}$$

the two-electron integrals

$$\begin{aligned} \langle ab \| ij \rangle \equiv \langle \psi_a \psi_b \| \psi_i \psi_j \rangle &\equiv \langle \psi_a \psi_b | \frac{1}{r_{12}} | \psi_i \psi_j \rangle - \langle \psi_a \psi_b | \frac{1}{r_{12}} | \psi_j \psi_i \rangle, \\ &= \int \int d\mathbf{x}_1 d\mathbf{x}_2 \psi_a^*(\mathbf{x}_1) \psi_b^*(\mathbf{x}_2) \frac{1}{r_{12}} \psi_i(\mathbf{x}_1) \psi_j(\mathbf{x}_2), \\ &\quad - \int \int d\mathbf{x}_1 d\mathbf{x}_2 \psi_a^*(\mathbf{x}_1) \psi_b^*(\mathbf{x}_2) \frac{1}{r_{12}} \psi_j(\mathbf{x}_1) \psi_i(\mathbf{x}_2), \end{aligned} \tag{3.23}$$

which come straight from the Coulomb interaction and exchange term. And finally the Fock-Matrix elements

$$f_{ai} \equiv \hat{h}_{ai} + \sum_{j=1}^{n} \langle \psi_a \phi_j \| \psi_i \phi_j \rangle. \tag{3.24}$$

Both the Hartree-Fock equation and the Coupled Cluster equations which we will solve later are written in terms of these integrals. There are L^2 one-body integrals and L^4 two-body integrals. The basis vectors used to compute these integrals are chosen to somewhat fit the system of interest, sometimes being the eigenstates of the system when interactions between particles are ignored, other times being Gaussian- or Slater-Type orbitals which have been optimized for the system. When Hartree-Fock has been used to find optimized orbitals from the chosen basis, we transform the integrals to instead use the Hartree-Fock orbitals as a basis.

The Hartree-Fock method has now given us a single optimized Slater determinant of N spin-orbitals, for a given system and a given basis set of size L. We also have access to $L - N$ unused spin-orbitals and integrals between all L spin-orbitals. The next step is to now produce other Slater-determinants using our starting determinant and our $L - N$ unused spin-orbitals.

Chapter 4

Coupled Cluster theory

This theory chapter regarding coupled cluster has been adapted from an assignment for a special curriculum regarding many-body methods, with a few revisions. The original assignment pdf can be found here: https://github.com/KarlHenrik/ComputationalPhysicsMaster/blob/master/Thesis/CC_Method.pdf.

The coupled cluster method is one of the best methods for finding approximate solutions to the electronic Schödinger equation. It is one of many post Hartree-Fock methods which improve upon the single Slater determinant from the Hartree-Fock method. The coupled cluster method uses the exponential cluster operator to describe a large number number of Slater determinants, which together can accurately account for electron correlations. This approach uses only using a small number of parameters, leading to fast computations. The method was introduced into quantum chemistry in the 1960s by íek and Paldus [6] [4, 5, 3], and has since become one of the most used methods for solving many-electron systems . Many variants of the method have been developed, including variational formulations as well as a formulation using perturbation theory to efficiently include an estimate for triple excitations (CCSD(T)) [6]. We will focus on the simple variants of coupled cluster doubles (CCD) and coupled cluster singles doubles (CCSD).

We will use the coupled cluster method as a benchmark to compare with our own approach of solving the Schrodinger equation, combining the Hartree-Fock solution with a neural network. This proved quite useful, as we found few such accurate benchmarks for 1D systems of electrons, to which we are constrained. The coupled cluster method is all about creating many Slater determinants in an efficient way, and finding the combination of them that minimizes the energy. The theory of the method is quite involved, and we make no attempt to provide a thorough treatment of it here. For the interested reader, we strongly recommend An Introduction to Coupled Cluster Theory for Computational Chemists by T. Daniel Crawford and Henry F. Schaefer. It proved an invaluable resource when studying the method.

We now begin our overview of the method by presenting the tool used to alter Slater determinants, second quantization.

4.1 Second Quantization

Given a Slater determinant, we can define a new determinant with an extra electron and orbital using a creation operator

$$a_p^\dagger |\phi_q \dots \phi_s\rangle = |\phi_p \phi_q ... \phi_s\rangle . \tag{4.1}$$

Likewise we can remove an electron and orbital using an annihilation operator

$$a_p |\phi_p \phi_q \dots \phi_s\rangle = |\phi_q \dots \phi_s\rangle . \tag{4.2}$$

If you add an orbital which is already present, the determinant will evaluate to zero due to its anti-symmetry

$$a_q^\dagger |\phi_q \dots \phi_s\rangle = 0. \tag{4.3}$$

If you remove an orbital which is not included in the determinant, you also get zero.

$$a_p |\phi_q \dots \phi_s\rangle = 0. \tag{4.4}$$

A state with no orbitals or electrons is the true vacuum state $|\rangle$. With these rules we can write any Slater determinant as a combination of creation operators acting on the true vacuum state.

$$a_q^\dagger \dots a_s^\dagger |\rangle = |\phi_q \dots \phi_s\rangle . \tag{4.5}$$

As the interchange of two columns in a determinant changes the sign, the interchange of two creation or annihilation operators also changes the sign. They anticommute.

$$\begin{aligned} a_p^\dagger a_q^\dagger + a_q^\dagger a_p^\dagger &= 0. \\ a_p a_q + a_q a_p &= 0. \end{aligned} \tag{4.6}$$

The anticommutator relation of creation and annihilation operators is

$$a_p^\dagger a_q + a_q a_p^\dagger = \delta_{pq}, \tag{4.7}$$

where δ_{pq} is the Kronecker delta, which equals 1 when $p = q$ and 0 when $p \neq q$. This means that when we swap two neighbouring creation and annihilation operators we end up with an extra term in our expression.

4.2 Cluster Operators

The Hartree-Fock method gave us a single optimized Slater determinant. Since it only contains single-electron functions, it cannot properly describe electron-correlations: Electron-correlations can only be properly described by many-electron functions. All is not lost however, as we can construct many-electron functions using only combinations of products of our single-electron functions. For instance, any 2-variable function can be written as a linear combination of 1-variable function products like this

$$f(\mathbf{x}_1, \mathbf{x}_2) = \sum_{p<q} \phi_p(\mathbf{x}_1)\phi_q(\mathbf{x}_2), \tag{4.8}$$

where we sum over a complete basis of functions. Using this principle, we can use a linear combination of determinants (each having a different combination of orbitals from a complete basis) to describe any N-variable function.

These Slater determinants with the different ϕ_i's can be written as different excitations of the Hartree-Fock wavefunction, our reference determinant. An excitation of the referecne determinant is when we replace the occupied orbitals already present in the determinant with unoccupied orbitals from the basis of orbitals. An example of such a determinant where we replace the occupied oribtal ϕ_i with the unoccupied orbital ϕ_a can be written as

$$a_a^\dagger a_i \left|\phi_i\phi_j \dots \phi_k\right\rangle = \left|\phi_a\phi_j \dots \phi_k\right\rangle. \tag{4.9}$$

We now introduce a naming convention for the orbitals, which will be very important later. Occupied orbitals are called holes, hole states or hole orbitals and are labeled $i, j, k, \dots$. Unoccupied orbitals are called particles, particle states or particle orbitals and are labeled $a, b, c, \dots$.

Any determinant can be written as excitations of the reference determinant, and it is convenient to do so, since it is the determinant with the lowest energy. There are many possible excited determinants however, so we use cluster operators to create them in a compact and organized way.

We define a single-orbital cluster operator, with its corresponding weights t_i^a

$$\hat{t}_i = \sum_a t_i^a a_a^\dagger a_i, \tag{4.10}$$

where we loop over all unoccupied orbitals a. Applying this operator to the reference determinant creates a linear combination of the $L - N$ singly excited determinants where orbital ϕ_i has been exchanged with each of the $L - N$ unoccupied orbitals.

$$\hat{t}_i \left|\phi_i\phi_j \dots\right\rangle = \sum_a t_i^a \left|\phi_a\phi_j \dots\right\rangle. \tag{4.11}$$

Likewise we define a two-orbital cluster operator

$$\hat{t}_{ij} = \sum_{a>b} t_{ij}^{ab} a_a^\dagger a_b^\dagger a_j a_i. \tag{4.12}$$

The one-orbital cluster operator only exchanges the orbital ϕ_i for unoccupied orbitals. It is convenient to define a total one-orbital cluster operator

$$\hat{T}_1 = \sum_i \hat{t}_i. \tag{4.13}$$

Applying the total one-orbital cluster operator to the reference determinant produces a linear combination of all possible single excitations. Not only those where orbital ϕ_i has been exchanged.

The two-orbital cluster operator becomes

$$\hat{T}_2 = \frac{1}{2} \sum_{ij} \hat{t}_{ij}. \tag{4.14}$$

And in general the n-orbital cluster operator is

$$\hat{T}_n = \sum_{ij\ldots ab\ldots} t_{ij\ldots}^{ab\ldots} a_a^\dagger a_b^\dagger \ldots a_j a_i. \tag{4.15}$$

4.3 The Coupled Cluster Equations

4.3.1 The wavefunction ansatz

Recall that we wish to create new determinants from our reference determinant, and that we wish to find the combination of these that best satisfy the Schrödinger equation. Using the cluster operators we can write the linear combination of all possible excited determinants as

$$|\Psi_{CI}\rangle = (1 + \hat{T}_1 + \hat{T}_2 + \hat{T}_3 + \cdots + \hat{T}_N) |\Phi_0\rangle = \hat{T} |\Phi_0\rangle , \tag{4.16}$$

where N is the number of electrons. This wavefunction ansatz, called the configuration interaction(CI) ansatz, gives the exact solution to the Schödinger equation, within the space of our basis. It has the problem however, of being too difficult to optimize for large systems. The number of parameters that must be found is simply too large.

A way around this problem is to model higher excitations as combinations of lower excitations. We don't "need" $\hat{T}_4$ for instance, when we could use $\hat{T}_2^2$ to give us quadruple excitations. This works surprisingly well in practice, as a quadruple excitation are often just two unrelated double excitations anyway.

We will however need to optimize the $\hat{T}_2$ amplitudes for describing both double excitations and quadruple excitations (through $\hat{T}_2$) in order to get the best results. Sextuple and octuple excitations and so on can be modeled in the same way using only $\hat{T}_2$. In this way all possible excitations can be modeled with only combinations of $\hat{T}_1$ and $\hat{T}_2$.

A normal approach is therefore to include only single and double excitations, $\hat{T} = \hat{T}_1 + \hat{T}_2$. This is called coupled cluster singles and doubles (CCSD). The wavefunction ansatz using this method can be written as

$$|\Psi_{\text{CCSD}}\rangle = (1 + \hat{T}_1 + \frac{1}{2!}\hat{T}_1^2 + \hat{T}_2 + \hat{T}_2\hat{T}_1 + \frac{1}{3!}\hat{T}_1^3 + \frac{1}{4!}\hat{T}_1^4 + \dots)\, |\Phi_0\rangle , \tag{4.17}$$

where the terms inside the parentheses can be written as the power series expansion of an exponential function, since $\hat{T}_1$ and $\hat{T}_2$ commute:

$$|\Psi_{\text{CCSD}}\rangle = e^{\hat{T}_1 + \hat{T}_2} |\Phi_0\rangle = e^{\hat{T}} |\Phi_0\rangle . \tag{4.18}$$

Another, less common approach is coupled cluster doubles (CCD) which only includes double excitations.

$$|\Psi_{\text{CCD}}\rangle = (1 + \hat{T}_2 + \hat{T}_2^2 + \hat{T}_2^3 + \dots)\, |\Phi_0\rangle = e^{\hat{T}_2} |\Phi_0\rangle = e^{\hat{T}} |\Phi_0\rangle . \tag{4.19}$$

It might seem strange to only use double excitations instead of only using single excitations, but when using a Hartree-Fock determinant, Brillouin's theorem tells us that we cannot reach a lower energy using only single excitations [24].

Both these approaches both suffer from not being well suited to modeling triple excitations. Including $\hat{T}_3$ is often not worth the increased computational complexity however, but there are methods which include a triples (T) correction that are very accurate and still quite fast [6].

4.3.2 The Equations to be solved

We wish to solve the Schrödinger equation

$$\begin{aligned} \hat{H} |\Psi\rangle &= E |\Psi\rangle , \\ \hat{H} e^{\hat{T}} |\Phi_0\rangle &= E e^{\hat{T}} |\Phi_0\rangle . \end{aligned} \tag{4.20}$$

From here we can left multiply with some carefully chosen states to reach an equation which we can solve for the energy:

$$\begin{aligned} &\langle\Phi_0| e^{-\hat{T}} \hat{H} e^{\hat{T}} |\Phi_0\rangle = \langle\Phi_0| e^{-\hat{T}} E e^{\hat{T}} |\Phi_0\rangle , \\ &\langle\Phi_0| e^{-\hat{T}} \hat{H} e^{\hat{T}} |\Phi_0\rangle = E, \end{aligned} \tag{4.21}$$

where the reader must note that $e^{-\hat{T}} \neq (e^{\hat{T}})^\dagger$, making this equation different from the actual expectation value of the energy, which is $\langle\Psi|\hat{H}|\Psi\rangle$. This method is therefore not variational, as we are not working with the proper expression for the energy. When $\hat{T}$ is not truncated however, the equation gives the correct energy, which we can use as a kind of theoretical backing that this will give somewhat good results.

Furter, we get an equation we can solve for the $\hat{T}_1$ amplitudes (only applicable for CCSD):

$$\begin{aligned} \langle\Phi_i^a| e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi_0\rangle &= \langle\Phi_i^a| e^{-\hat{T}}Ee^{\hat{T}}|\Phi_0\rangle, \\ \langle\Phi_i^a| e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi_0\rangle &= 0, \end{aligned} \tag{4.22}$$

and finally an equation we can solve for the $\hat{T}_2$ amplitudes:

$$\begin{aligned} \left\langle\Phi_{ij}^{ab}\right| e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi_0\rangle &= \left\langle\Phi_{ij}^{ab}\right| e^{-\hat{T}}Ee^{\hat{T}}|\Phi_0\rangle, \\ \left\langle\Phi_{ij}^{ab}\right| e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi_0\rangle &= 0. \end{aligned} \tag{4.23}$$

The steps following this are quite involved, and more suited for a textbook. We will therefore present only a short description of the mathematical tools used and the steps taken to simplify the equations further in the next sections.

4.4 Simplifying the equations

4.4.1 Normal ordering

We have already written the cluster operators $\hat{T}_1$ and $\hat{T}_2$ in terms of creation and annihilation operators, and we will later show how $e^{-\hat{T}}\hat{H}e^{\hat{T}}$ can be written using creation and annihilation operators. We can also write the Hamiltonian in terms of these operators

$$\hat{H} = \sum_{pq} \langle p|\hat{h}|q\rangle a_p^\dagger a_q - \frac{1}{4}\langle pq||rs\rangle a_p^\dagger a_q^\dagger a_s a_r. \tag{4.24}$$

This is quite helpful, as we can now work on equation 4.21, 4.22, and 4.23 using only the tools of second quantization.

We can write all of these equations as expectation values of the reference determinant. The $\hat{T}_1$ equation, for instance, can be written as

$$\langle\Phi_0|(a_i^\dagger a_a)e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi_0\rangle = 0. \tag{4.25}$$

This will be the key to turn these equations into a workable form, writing them as expectation values of the reference determinant. To evaluate these expectation values, we use normal ordering and Wick's theorem.

A string of creation and annihilation operators is normal ordered if all operators that destroy hole and particle states are to the right of the operators that create them. A reminder: $a_i^\dagger$ destroys a hole, as it fills it with orbital i. a_i creates a hole, as it removes orbital i and leaves a hole. $a_a^\dagger$ creates a particle state, as it adds the "particle" a. And finally, a_a destroys a particles state, as it removes "particle" a. The ordering scheme in normal ordering then becomes

$$a_a^\dagger, a_i \text{ left of } a_i^\dagger, a_a. \tag{4.26}$$

This is called normal ordering with respect to a reference determinant, there also exists normal ordering with respect to vacuum, which we will not use. When a string of operators is normal ordered, we can label it as normal ordered using curly braces as such

$$\{a_i a_a^\dagger a_a\ a_i^\dagger\}. \tag{4.27}$$

The expectation value of a normal ordered string of operators with respect to the reference determinant is zero so long as it includes operators that destroy states

$$\langle\Phi_0|\{a_i a_a^\dagger \dots a_a\ a_i^\dagger\}|\Phi_0\rangle = 0. \tag{4.28}$$

This is because the rightmost operator acting on the reference determinant then always gives zero.

4.4.2 Wick's theorem

Wick's theorem is an incredibly useful theorem that can make any string of creation and annihilation operators into a sum of normal ordered operators and easily evaluated Kronecker-deltas. This will come in handy, as the normal ordered operators will disappear when evaluating expectation values, as seen in the previous section. And recall that the equations we wish to solve, equations 4.21, 4.22, and 4.23, can be written in terms of only creation and annihilation operators. Wick's theorem can therefore reduce these equations into only easily evaluated Kronecker-deltas.

The theorem uses something called contractions. A contraction between two operators is given by

$$\overbracket{AB} \equiv AB - \{AB\}, \tag{4.29}$$

where one must normal order the operators in the curly brackets and change the sign of the term with the curly brackets for each permutation of neighbouring operators necessary to do this. If the two contracted operators are not next to each other, the string of operators must be permuted to bring them together, with the sign changing every permutation.

This means that

$$\begin{aligned}
\wick{\c a^\dagger_i \c a_j} &= a^\dagger_i a_j - \left\{a^\dagger_i a_j\right\} = a^\dagger_i a_i + a_j a^\dagger_i = \delta_{ij}, \\
\wick{\c a_a \c a^\dagger_b} &= a_a a^\dagger_b - \left\{a_a a^\dagger_b\right\} = a_a a^\dagger_b + a^\dagger_b a_a = \delta_{ab}, \\
\wick{\c a^\dagger_a \c a_b} &= \wick{\c a_i \c a^\dagger_i} = 0.
\end{aligned} \tag{4.30}$$

All other contractions involve Kronecker-deltas between hole and particle indices, which are zero.

We are now ready to present Wick's theorem:

$$\begin{aligned}
ABC\cdots XYZ =& \{ABC\cdots XYZ\}_v \\
&+ \sum_{\text{singles}} \{\wick{\c A \c B}\cdots XYZ\} \\
&+ \sum_{\text{doubles}} \{\wick{\c1 A \c2 B \c1 C \cdots XY \c2 Z}\} \\
&+ \cdots
\end{aligned} \tag{4.31}$$

which gives us a way of turning a string of operators into sums of normal ordered string, with some contractions to be evaluated. And the generalized Wick's theorem:

$$\begin{aligned}
\{ABC\cdots\}\{XYZ\cdots\} =& \{ABC\cdots XYZ\} \\
&+ \sum_{\text{singles}} \{\wick{\c A B \cdots \c X YZ}\} \\
&+ \sum_{\text{doubles}} \{\wick{\c1 A \c2 B C \cdots X \c1 Y \c2 Z}\} \\
&+ \cdots .
\end{aligned} \tag{4.32}$$

which describes how to normal order the product of two normal ordered strings. When all the terms in a string are contracted, we call the string fully contracted. In this case a convenient trick applies: If the lines describing the contractions cross an even number of times, the result is positive, if not the result is negative. For instance

$$\{\wick{\c1 a^\dagger_i \c2 a_a \c1 a_j \c2 a^\dagger_b}\} = -\delta_{ij}\delta_{ab}. \tag{4.33}$$

We are now ready to return to the equations at hand.

4.4.3 Getting a workable Hamiltonian

Working with a Hamiltonian written in terms of normal-ordered second-quantized operators will make our life much easier. It can be shown that such a normal ordered Hamiltonian can be defined as

$$\begin{aligned}\hat{H}_N &= \hat{H} - \langle\Phi_0|\,\hat{H}\,|\Phi_0\rangle\,,\\ &= \sum_{pq} f_{pq}\{a_p^\dagger a_q\} + \frac{1}{4}\sum_{pqrs}\langle pq\|rs\rangle\{a_p^\dagger a_q^\dagger a_s a_r\},\\ &= \hat{F}_N + \hat{V}_N.\end{aligned} \tag{4.34}$$

We will use this Hamiltonian instead when solving equations 4.21, 4.22, and4.23. We can now define

$$\bar{H} = e^{-\hat{T}}\hat{H}_N e^{\hat{T}}. \tag{4.35}$$

This expression can be simplified via the so-called Campbell-Baker-Hausdorff formula into

$$\begin{aligned}\bar{H} = e^{-\hat{T}}\hat{H}_N e^{\hat{T}} =& \hat{H}_N + [\hat{H}_N,\hat{T}] + \frac{1}{2!}[[\hat{H}_N,\hat{T}],\hat{T}] + \frac{1}{3!}[[[\hat{H}_N,\hat{T}],\hat{T}],\hat{T}]\\ &+ \frac{1}{4!}[[[[\hat{H}_N,\hat{T}],\hat{T}],\hat{T}],\hat{T}] + \cdots .\end{aligned} \tag{4.36}$$

Many of these terms will disappear when we insert this expression into equations 4.21, 4.22, and 4.23: Since $\hat{H}_N$ is a two-particle operator, whenever the left and right hand side of it differ by more than two orbitals, you will get zero.

Many terms will also cancel, and it can be shown that the only non-zero terms are those in which the Hamiltonian $\hat{H}_N$ has at least one contraction with every cluster operator $\hat{T}_n$ on its right. If we use these results and insert $\hat{T} = \hat{T}_1 + \hat{T}_2$ we get

$$\begin{aligned}\bar{H} = \Big(&\hat{H}_N + \hat{H}_N\hat{T}_1 + \hat{H}_N\hat{T}_2 + \frac{1}{2}\hat{H}_N\hat{T}_1^2 + \frac{1}{2}\hat{H}_N\hat{T}_2^2 + \hat{H}_N\hat{T}_1\hat{T}_2\\ &+ \frac{1}{6}\hat{H}_N\hat{T}_1^3 + \frac{1}{2}\hat{H}_N\hat{T}_1^2\hat{T}_2 + \frac{1}{2}\hat{H}_N\hat{T}_1\hat{T}_2^2 + \frac{1}{6}\hat{H}_N\hat{T}_2^3\\ &+ \frac{1}{24}\hat{H}_N\hat{T}_1^4 + \frac{1}{6}\hat{H}_N\hat{T}_1^3\hat{T}_2 + \frac{1}{4}\hat{H}_N\hat{T}_1^2\hat{T}_2^2 + \frac{1}{6}\hat{H}_N\hat{T}_1\hat{T}_2^3 + \frac{1}{24}\hat{H}_N\hat{T}_2^4\Big)_c\,,\end{aligned} \tag{4.37}$$

where the c tells us to only include contractions where $\hat{H}_N$ has at least one contraction with every cluster operator $\hat{T}_n$ on its right.

4.4.4 Simplifying to matrix-elements

The next step is to use Wick's theorem and the rules of normal ordering to simplify equation 4.21, 4.22, and 4.23, only with $\hat{H}_N$, as discussed in the previous section (So that the energy equation becomes $\langle\Phi_0|\,\bar{H}\,|\Psi_0\rangle = 0$, and the amplitude equations become $\langle\Phi_i^a|\,\bar{H}\,|\Psi_0\rangle = 0$ and $\langle\Phi_0|\,\bar{H}\,\left|\Psi_{ij}^{ab}\right\rangle = 0$).

$\bar{H}$ includes the term $\frac{1}{2}\hat{H}_N\hat{T}_1^2$, which can be written as $\frac{1}{2}(\hat{F}_N+\hat{V}_N)\hat{T}_1^2$. We will now show how to evaluate the term $\langle\Phi_i^a|\,\frac{1}{2}\hat{V}_N\hat{T}_1^2\,|\Phi_0\rangle$ from the $\hat{T}_1$ amplitude equations. We choose this term in particular because when we implemented the Coupled Cluster equations into code, we discovered a discrepancy between the equations provided by Crawford & Schaefer [6], and Shavitt & Bartlett [23]. It is the evaluation of $\langle\Phi_i^a|\,\frac{1}{2}\hat{V}_N\hat{T}_1^2\,|\Phi_0\rangle$ from the $\hat{T}_1$ amplitude equations that differs in the two books.

In equation 152 of Crawford & Schaefer [6], they get the terms

$$-\sum_{klc}\langle kl||ci\rangle t_k^c t_l^a - \sum_{kcd}\langle ka||cd\rangle t_k^c t_i^d. \tag{4.38}$$

While in equation 10.23 of Shavitt & Bartlett [23], they get the terms

$$-\sum_{klc}\langle kl||ic\rangle t_k^a t_l^c + \sum_{kcd}\langle ak||cd\rangle t_i^c t_k^d. \tag{4.39}$$

By swapping the indices k and l in the first sum, and c and d in the second, we get

$$\begin{aligned} &-\sum_{klc}\langle lk||ic\rangle t_l^a t_k^c + \sum_{kcd}\langle ak||dc\rangle t_i^d t_k^c, \\ =&-\sum_{klc}\langle kl||ci\rangle t_k^c t_l^a + \sum_{kcd}\langle ka||cd\rangle t_k^c t_i^d, \end{aligned} \tag{4.40}$$

where we used the fact that we can interchange two indices on one side of the two-body integrals if we also change the sign. We see that the equations from the two books differ in the sign of the second term. We therefore went through the trouble of evaluating these terms ourselves, both to reassure ourselves that the equations from Shavitt & Bartlett were correct, as confirmed by other sources, but also as a good example and exercise.

We start by writing out the expression in terms of annihilation and creation operators

$$\langle\Phi_i^a|\,(\hat{V}_N\hat{T}_1^2)\,|\Phi_0\rangle = \frac{1}{8}\sum_{pqrs}\sum_{ld}\sum_{kc}\langle pq||rs\rangle t_l^d t_k^c\,\langle\Phi_0|\,\{a_i^\dagger a_a\}(\{a_p^\dagger a_q^\dagger a_s a_r\}\{a_d^\dagger a_l\}\{a_c^\dagger a_k\})_c\,|\Phi_0\rangle\,. \tag{4.41}$$

Wick's theorem tells us that we need to evaluate all possible contractions between the strings of operators (and not within the strings), and the c tells us that the two $\hat{T}_1$-s must share at least one contraction with $\hat{V}_N$. Furthermore, since we are computing an expectation value between reference determinants, terms from Wick's theorem that contain un-contracted operators will dissapear. That is, only fully contracted terms must be included. We also only need to consider contractions that give non-zero results. With this in mind, we write out the possible contractions and the matrix elements that follow:

$$
\begin{aligned}
\langle\Phi_i^a|\,(\hat{V}_N\hat{T}_1^2)\,|\Phi_0\rangle &= \frac{1}{8}\sum_{pqrs}\sum_{ld}\sum_{kc}\langle pq|rs\rangle\, t_l^d t_k^c\,\langle\Phi_0|\,\{a_i^\dagger a_a\}(\{a_p^\dagger a_q^\dagger a_s a_r\}\{a_d^\dagger a_l\}\{a_c^\dagger a_k\})_c\,|\Phi_0\rangle\,,\\
&= \sum_{pqrs}\sum_{ld}\sum_{kc}\langle pq|rs\rangle\, t_l^d t_k^c\Big(\\
&\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}\\
&+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}\\
&+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}\\
&+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}\\
&+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}\\
&+\{a_i^\dagger a_a a_p^\dagger a_q^\dagger a_s a_r a_d^\dagger a_l a_c^\dagger a_k\}\Big),\\
&= \frac{1}{8}\sum_{pqrs}\sum_{ld}\sum_{kc}\langle pq|rs\rangle\, t_l^d t_k^c(\delta_{is}\delta_{ad}\delta_{pl}\delta_{qk}\delta_{rc}-\delta_{is}\delta_{ad}\delta_{pk}\delta_{ql}\delta_{rc}-\delta_{is}\delta_{ac}\delta_{pl}\delta_{qk}\delta_{rd}\\
&+\delta_{is}\delta_{ac}\delta_{pk}\delta_{ql}\delta_{rd}-\delta_{ir}\delta_{ad}\delta_{pl}\delta_{qk}\delta_{sc}+\delta_{ir}\delta_{ad}\delta_{pk}\delta_{ql}\delta_{sc}\\
&+\delta_{ir}\delta_{ac}\delta_{pl}\delta_{qk}\delta_{sd}-\delta_{ir}\delta_{ac}\delta_{pk}\delta_{ql}\delta_{sd}-\delta_{il}\delta_{ap}\delta_{qk}\delta_{sd}\delta_{rc}\\
&+\delta_{il}\delta_{ap}\delta_{qk}\delta_{sc}\delta_{rd}+\delta_{il}\delta_{aq}\delta_{pk}\delta_{sd}\delta_{rc}-\delta_{il}\delta_{aq}\delta_{pk}\delta_{sc}\delta_{rd}\\
&+\delta_{ik}\delta_{ap}\delta_{ql}\delta_{sd}\delta_{rc}-\delta_{ik}\delta_{ap}\delta_{ql}\delta_{sc}\delta_{rd}-\delta_{ik}\delta_{aq}\delta_{pl}\delta_{sd}\delta_{rc}\\
&+\delta_{ik}\delta_{aq}\delta_{pl}\delta_{sc}\delta_{rd})\\
&= \frac{1}{8}\Big(\sum_{lkc}\langle lk|ci\rangle\, t_l^a t_k^c-\sum_{lkc}\langle kl|ci\rangle\, t_l^a t_k^c-\sum_{ldk}\langle lk|di\rangle\, t_l^d t_k^a\\
&+\sum_{kld}\langle kl|di\rangle\, t_l^d t_k^a-\sum_{lkc}\langle lk|ic\rangle\, t_l^a t_k^c+\sum_{lkc}\langle kl|ic\rangle\, t_l^a t_k^c\\
&+\sum_{ldk}\langle lk|id\rangle\, t_l^d t_k^a-\sum_{ldk}\langle kl|id\rangle\, t_l^d t_k^a-\sum_{dkc}\langle ak|cd\rangle\, t_i^d t_k^c\\
&+\sum_{dkc}\langle ak|dc\rangle\, t_i^d t_k^c+\sum_{dkc}\langle ka|cd\rangle\, t_i^d t_k^c-\sum_{dkc}\langle ka|dc\rangle\, t_i^d t_k^c\\
&+\sum_{ldc}\langle al|cd\rangle\, t_l^d t_i^c-\sum_{ldc}\langle al|dc\rangle\, t_l^d t_i^c-\sum_{ldc}\langle la|cd\rangle\, t_l^d t_i^c\\
&+\sum_{ldc}\langle la|dc\rangle\, t_l^d t_i^c\Big),\\
&= -\sum_{klc}\langle kl|ci\rangle\, t_k^c t_l^a+\sum_{kcd}\langle ka|cd\rangle\, t_k^c t_i^d.
\end{aligned}
\tag{4.42}
$$

We see that the terms provided from Shavitt & Bartlett were indeed correct. We describe how we used code to help us find the contractions and produce the equations in section 11.1. This is only a very small number of the total number of terms that need to be evaluated for the Coupled Cluster method. Similar calculations need to be performed for every single term in $\bar{H}$ for both the energy, $\hat{T}_1$ and $\hat{T}_2$ equations.

4.5 The simplified equations

By using a normal ordered Hamiltonian and employing Wick's theorem to turn equations 4.21, 4.22, and4.23 into the workable matrix-element form we get the following equations:

4.5.1 The Energy Equations

The difference between the energy of the reference determinant and the energy of the coupled cluster wave function is called the correlation energy, since it is the energy difference arising from accounting for electron correlations. The correlation energy for the CCD wave function is:

$$E_{CCD} - E_0 = \langle \Phi_0 | \bar{H} | \Phi_0 \rangle = \frac{1}{4} \sum_{aibj} \langle ij || ab \rangle t_{ij}^{ab}. \tag{4.43}$$

The correlation energy for the CCSD wave function is:

$$E_{CCSD} - E_0 = \langle \Phi_0 | \bar{H} | \Phi_0 \rangle = \sum_{ia} f_{ia} t_i^a + \frac{1}{4} \sum_{aibj} \langle ij || ab \rangle t_{ij}^{ab} + \frac{1}{2} \langle ij || ab \rangle t_i^a t_j^b. \tag{4.44}$$

4.5.2 The Amplitude Equations

The CCD amplitude equation for the $\hat{T}_2$ amplitudes is:

$$\begin{aligned} 0 =& \langle ab || ij \rangle + \sum_c \left(f_{bc} t_{ij}^{ac} - f_{ac} t_{ij}^{bc} \right) - \sum_k \left(f_{kj} t_{ik}^{ab} - f_{ki} t_{jk}^{ab} \right) \\ &+ \frac{1}{2} \sum_{cd} \langle ab || cd \rangle t_{ij}^{cd} + \frac{1}{2} \sum_{kl} \langle kl || ij \rangle t_{kl}^{ab} + P(ij)P(ab) \sum_{kc} \langle kb || cj \rangle t_{ik}^{ac} \\ &+ \frac{1}{4} \sum_{klcd} \langle kl || cd \rangle t_{ij}^{cd} t_{kl}^{ab} + P(ij) \sum_{klcd} \langle kl || cd \rangle t_{ik}^{ac} t_{jl}^{bd} \\ &- P(ij) \frac{1}{2} \sum_{klcd} \langle kl || cd \rangle t_{ik}^{dc} t_{lj}^{ab} - P(ab) \frac{1}{2} \sum_{klcd} \langle kl || cd \rangle t_{lk}^{ac} t_{ij}^{db}. \end{aligned} \tag{4.45}$$

The CCSD amplitude equation for the $\hat{T}_1$ amplitudes is:

$$
\begin{aligned}
0 =& f_{ai} + \sum_{c} f_{ac} t_i^c - \sum_{k} f_{ki} t_k^a + \sum_{kc} \langle ka \| ci \rangle t_k^c + \sum_{kc} f_{kc} t_{ik}^{ac} + \frac{1}{2} \sum_{kcd} \langle ka \| cd \rangle t_{ki}^{cd} \\
& - \frac{1}{2} \sum_{klc} \langle kl \| ci \rangle t_{kl}^{ca} - \sum_{kc} f_{kc} t_i^c t_k^a - \sum_{klc} \langle kl \| ci \rangle t_k^c t_l^a + \sum_{kcd} \langle ka \| cd \rangle t_k^c t_i^d \\
& - \sum_{klcd} \langle kl \| cd \rangle t_k^c t_i^d t_l^a + \sum_{klcd} \langle kl \| cd \rangle t_k^c t_{li}^{da} - \frac{1}{2} \sum_{klcd} \langle kl \| cd \rangle t_{ki}^{cd} t_l^a - \frac{1}{2} \sum_{klcd} \langle kl \| cd \rangle t_{kl}^{ca} t_i^d,
\end{aligned}
\tag{4.46}
$$

where the reader will recognize the last two terms on the second line as the terms we computed ourselves in the previous section.

The CCSD amplitude equation for the $\hat{T}_2$ amplitudes can be found in appendix .1.

4.5.3 Solving the $\hat{T}_1$ amplitude equation

The $\hat{T}_1$ and $\hat{T}_2$ amplitude equations can be solved with an iterative approach. Given an initial set of amplitudes, we compute a new set of amplitudes which better satisfy the equations. These new amplitudes can then be improved upon once again, and so on, until we (hopefully) have amplitudes which completely satisfy the equations. First, we need to formulate expressions for updating the amplitudes.

We start by manipulating the $\hat{T}_1$ amplitude equation so that we get an expression for t_i^a. We start by separating the diagonal terms from the Fock-Matrix summations.

$$
\begin{aligned}
0 &= \langle \Phi_i^a | \bar{H} | \Phi_0 \rangle , \\
0 &= f_{ai} + f_{aa} t_i^a - f_{ii} t_i^a + \sum_{c} (1 - \delta_{ca}) f_{ac} t_i^c - \sum_{k} (1 - \delta_{ik}) f_{ik} t_k^a + \cdots .
\end{aligned}
\tag{4.47}
$$

Defining

$$
D_i^a = f_{ii} - f_{aa}, \tag{4.48}
$$

we can write

$$
\begin{aligned}
D_i^a t_i^a &= f_{ai} + \sum_{c} (1 - \delta_{ca}) f_{ac} t_i^c - \sum_{k} (1 - \delta_{ik}) f_{ik} t_k^a + \cdots , \\
t_i^a &= \frac{1}{D_i^a} (f_{ai} + \sum_{c} (1 - \delta_{ca}) f_{ac} t_i^c - \sum_{k} (1 - \delta_{ik}) f_{ik} t_k^a + \cdots),
\end{aligned}
\tag{4.49}
$$

giving us an expression for a new t_i^a, which should better satisfy the $\hat{T}_1$ amplitude equation. Using this expression, we can update the $\hat{T}_1$ amplitudes iteratively by setting the amplitudes t_i^a equal to the value we compute for the right hand side.

This approach can have trouble converging however. Section 11.3 describes schemes which improve convergence by cleverly combining the old t_i^a values with the new ones computed in equation 4.49. The use of these schemes motivate finding an expression for difference between these t_i^a values. First, we define

$$\Omega_i^a = \langle \Phi_i^a | \bar{H} | \Phi_0 \rangle . \tag{4.50}$$

Now, the difference between the old t_i^a values and the new ones computed from equation 4.49 can be written as

$$\Delta t_i^a = t_i^a - \frac{1}{D_i^a}(\Omega_i^a - D_i^a t_i^a) = \frac{\Omega_i^a}{D_i^a}. \tag{4.51}$$

To begin the iterative approach, we need to make an initial guess for the amplitudes. We choose to set all amplitudes to 0, and from equation 4.49 then get the initial guess

$$t_i^a = \frac{f_{ai}}{D_i^a}. \tag{4.52}$$

4.5.4 Solving the $\hat{T}_2$ amplitude equation

The steps for finding an expression for t_{ij}^{ab}(for both CCD and CCSD) are the same as for t_i^a. We define

$$\begin{aligned} D_{ij}^{ab} &= f_{ii} + f_{jj} - f_{aa} - f_{bb}, \\ \Omega_{ij}^{ab} &= \left\langle \Phi_{ij}^{ab} \right| \bar{H} | \Phi_0 \rangle . \end{aligned} \tag{4.53}$$

Separating out the diagonal terms from the Fock-Matrix summations we get

$$\begin{aligned} 0 &= \left\langle \Phi_{ij}^{ab} \right| \bar{H} | \Phi_0 \rangle , \\ D_{ij}^{ab} t_{ij}^{ab} &= \langle ab \| ij \rangle + P(ab) \sum_c (1 - \delta_{bc}) f_{bc} t_{ij}^{ac} - P(ij) \sum_k \left(1 - \delta_{kj}\right) f_{kj} t_{ik}^{ab} + \cdots , \\ t_{ij}^{ab} &= \frac{1}{D_{ij}^{ab}}(\langle ab \| ij \rangle + P(ab) \sum_c (1 - \delta_{bc}) f_{bc} t_{ij}^{ac} - P(ij) \sum_k \left(1 - \delta_{kj}\right) f_{kj} t_{ik}^{ab} + \cdots). \end{aligned} \tag{4.54}$$

The difference between the initial values t_{ij}^{ab} and the ones computed from the right hand side of equation 4.54 is

$$\Delta t_{ij}^{ab} = t_{ij}^{ab} - \frac{1}{D_{ij}^{ab}}(\Omega_{ij}^{ab} - D_{ij}^{ab} t_{ij}^{ab}) = \frac{\Omega_{ij}^{ab}}{D_{ij}^{ab}}. \tag{4.55}$$

This difference Δt_{ij}^{ab} will be used along with Δt_i^a to update the $\hat{T}$ amplitudes. Section 11.3 describes schemes that improve upon the crude guesses $t_i^a \to t_i^a + \Delta t_i^a$ and $t_{ij}^{ab} \to t_{ij}^{ab} + \Delta t_{ij}^{ab}$ for the new amplitudes.

The initial guess for the amplitudes is again found by setting them to zero which leads to equation 4.54 giving us the initial guess

$$t_{ij}^{ab} = \frac{\langle ab\|ij\rangle}{t_{ij}^{ab}}. \tag{4.56}$$

4.6 Expectation values other than energy

The Coupled Cluster method is not variational, which means that it can give energies lower than the exact ground state energy. This might seem like a big problem, but the method is typically very accurate, so it isn't [9]. What is a big problem however, is that other expectation values are not guaranteed to have accurate values. Since the Coupled Cluster method is not variational, it does not satisfy the conditions of the Hellman-Feynman theorem, and it therefore doesn't necessarily give accurate values for other expectation values when we get an accurate value for the energy. We should therefore not interpret other expectation values as physically relevant. This does not stop us from computing these expectation values however.

A useful quantity when computing one- and two-body expectation values are the density matrices [15]. The one-body density matrix is given by

$$\rho_{qp} = \langle \Psi | a_p^\dagger a_q | \Psi \rangle . \tag{4.57}$$

If we use our CCSD wave function on the right, and $\langle \Psi_0 | e^{-T}$ on the left, we get

$$\begin{aligned}
\rho_{qp} &= \langle \Psi_0 | e^{-T} a_p^\dagger a_q e^{T} | \Psi_0 \rangle , \\
&= \langle \Psi_0 | e^{-T} (\{a_p^\dagger a_q\} + \{\overbracket{a_p^\dagger a_q}\}) e^{T} | \Psi_0 \rangle , \\
&= \langle \Psi_0 | e^{-T} (\{a_p^\dagger a_q\} + \delta_{pq}\delta_{pi}) e^{T} | \Psi_0 \rangle , \\
&= \langle \Psi_0 | e^{-T} \{a_p^\dagger a_q\} e^{T} | \Psi_0 \rangle + \delta_{pq}\delta_{pi}, \\
&= \delta_{pi}\delta_{qa} t_i^a + \delta_{pq}\delta_{pi},
\end{aligned} \tag{4.58}$$

where we used the normal ordering rules discussed earlier to reach the second to last line. To evaluate the final expectation value with the normal ordered $\{a_p^\dagger a_q\}$, we simply used the expectation value computed for $(\hat{F_N})_{pq} = f_{pq}\{a_p^\dagger a_q\}$ earlier. If we used the CCD wavefunction, we would only get the second term.

One of the quantities of interest we can compute is the particle density, the probability density of finding any electron at a given position x.

$$\rho(x_1) = N \left(\prod_{i=2}^{N} \int dx_i \right) |\Psi(x_1, x_2, \ldots, x_N)|^2, \tag{4.59}$$

which we can also express using the one-body density as[9]

$$\rho(x) = \sum_{pq} \phi_p^*(x) \rho_{pq} \phi_q(x). \tag{4.60}$$

Chapter 5

The Variational Monte Carlo method

The variational Monte Carlo method (VMC) is different from the methods we have covered so far in that it is a method where we numerically approximate integrals of our wave function instead of finding closed form simplifications of them that we then minimize using some iterative scheme.

The central principle of VMC is very simple: Approximate the derivative of the energy with respect to your wave function parameters, and use it to minimize the energy. This method relies on the variational principle, hence the name variational. Since we know that the energy of our wave function can get no lower than the ground state energy, we can find the ground state by simply minimizing the energy.

5.1 Expectation Values

We compute the expectation value of the energy of a wave function by

$$E = \langle\Psi| \hat{H} |\Psi\rangle = \int \Psi(\mathbf{X})^* \hat{H} \Psi(\mathbf{X}) d\mathbf{X}. \tag{5.1}$$

We now allow the wave function to not be normalized, and define the probability density

$$P(\mathbf{X}) = \frac{|\Psi(\mathbf{X})|^2}{\int |\Psi(\mathbf{X})| d\mathbf{X}}. \tag{5.2}$$

The probability density can be used to calculate the expectation value of any operator $\hat{O}$

$$\langle\hat{O}\rangle = \int P(\mathbf{X}) \hat{O}(\mathbf{X}) d\mathbf{X}, \tag{5.3}$$

where $\hat{O}(\mathbf{X})$ is defined in an analogous way to how we now define the local energy $E_L(\mathbf{X})$

$$E_L(\mathbf{X}) = \frac{1}{\Psi(\mathbf{X})}\hat{H}\Psi(\mathbf{X}). \tag{5.4}$$

We can now write the expectation value of energy as

$$\langle E_L \rangle = \int P(\mathbf{X}) E_L(\mathbf{X}) d\mathbf{X}. \tag{5.5}$$

Say our wave function depends on parameters θ, using the definition of the gradient of an expectation value[27], we then get

$$\nabla_\theta \langle E_L \rangle = 2\left(\langle E_L \frac{1}{\Psi}\nabla_\theta \Psi \rangle - \langle \frac{1}{\Psi}\nabla_\theta \Psi \rangle \langle E_L \rangle \right). \tag{5.6}$$

We now have expressions for computing both the energy and the derivative of the energy with respect to the wave function parameters. All that is left before we can use the gradient to minimize the energy is a way of actually evaluating these expressions. The solution is Monte Carlo integration. Instead of integrating uniformly over the entire space, we integrate over a selection of M sampled points in space that follow the statistical properties needed for the integral to be valid, while still being efficient. Using Monte Carlo integration, we write the integral of an expectation value as

$$\langle \hat{O} \rangle \approx \frac{1}{M}\sum_{i=1}^{M} \hat{O}(\mathbf{X}_i). \tag{5.7}$$

The way we sample these M points in space is by using the Metropolis algorithm.

5.2 The Metropolis Algorithm

The Metropolis algorithm creates a Markov chain by repeatedly proposing a move of a single random particle and then accepting or rejecting it. The probability function defined earlier is used to decide whether to accept or deny the new move. The algorithm uses the ratio of the new and old probability function (after and before the move), which makes the normalization constant in the probability function disappear. The transition from an old state S_o to a new state S_n has a transition rule $P(S_n, S_o)$, and this rule must satisfy both ergodicity and detailed balance, meaning

$$P(S_n, S_o)P(S_o) = P(S_o, S_n)P(S_n), \tag{5.8}$$

where the functions $P(S)$ are the probability density functions from before. If we now write the transition probabilities as a product of a proposal distribution $T(S_n, S_o)$ and acceptance probability $A(S_n, S_o)$, we get the general expression for the acceptance probability

$$A(S_n, S_n) = min(1, \frac{T(S_o, S_n)P(S_n)}{T(S_n, S_o)P(S_o)}) \tag{5.9}$$

In brute force metropolis sampling, we choose to move a random particle in a random direction a random distance, and require that $T(S_n, S_o) = T(S_o, S_n)$. The probability of accepting the move from the old state S_o to the new state S_n is then

$$A(S_n, S_n) = min(1, \frac{P(S_n)}{P(S_o)}) \tag{5.10}$$

5.3 Importance Sampling

The Metropolis algorithm suffers from sampling completely at random. It can very easily let the particles get stuck in a low-probability areas for long times. Importance sampling solves this issue by modelling the particles as moving dynamically due to a quantum force $F(\mathbf{r})$, given by the Langevin equation[2]

$$\frac{\partial \mathbf{x}(t)}{\partial t} = DF(\mathbf{x}(t)) + \eta(t). \tag{5.11}$$

where D denotes a diffusion constant and $\eta(t)$ accounts for random collisions. Its solution gives us the new rule for suggesting moves

$$\mathbf{x}_{new} = \mathbf{x}_i + DF(\mathbf{x}_i)\Delta t + \xi\sqrt{\Delta t}, \tag{5.12}$$

where Δt denotes a time step we are free to choose, ξ describes a randomly drawn number from a normal distrubution, and

$$F(\mathbf{x}_i) = \frac{2}{\Psi(\mathbf{X})}\nabla_{x_i}\Psi(\mathbf{X}). \tag{5.13}$$

Now that we have a new way of proposing moves, we need a new way of accepting them to compensate. The Fokker - Planck equation

$$\frac{\partial P(\mathbf{x}_i, t)}{\partial t} = D\frac{\partial}{\partial \mathbf{x}_i}\left(\frac{\partial}{\partial \mathbf{x}_i} - F\right)P(\mathbf{x}_i, t), \tag{5.14}$$

which describes the time-dependent evolution of a probability distribution P and is a natural choice when devised alongside the Langevin equation. Its solution is given by the Green's function

$$G(\mathbf{x}_{new}, \mathbf{x}_i, \Delta t) = \frac{1}{(4\pi D\Delta t)^{3N/2}} e^{-(\mathbf{x}_{new} - \mathbf{x}_i - D\Delta t F(\mathbf{x}_i))^2/4D\Delta t}, \tag{5.15}$$

which describes the probability for selecting a transition $\mathbf{x}_{new}$ given that we previously were in a state $\mathbf{x}_i$. The acceptance probability is therefore replaced with

$$A(\mathbf{x}_i, \mathbf{x}_{new}) = \min\left(1, \frac{G(\mathbf{x}_i, \mathbf{x}_{new}, \Delta t)|\Psi(\mathbf{x}_{new})|^2}{G(\mathbf{x}_{new}, \mathbf{x}_i, \Delta t)|\Psi(\mathbf{x}_i)|^2}\right). \tag{5.16}$$

For an in detail derivation of the Green's function ratio as a simplified analytical expression, refer to [10].

5.4 The Procedure

Now that we have a way of efficiently sampling states, we can simply sample the local energy and parameter derivative of the wave function for each state. Letting us compute the derivative of the local energy with respect to the wave function parameters. Section 7.2.1 covers how to use the sampled gradient to optimize the neural network.

After we have finished optimizing the neural network, we can instead focus on only sampling the local energy, to get a good estimate for the energy of our wave function. The next section covers how we properly approximate the error in our estimate.

5.5 The Blocking Method

When we estimate the energy of our wave function using VMC, we take a large number of samples. We use the mean of these samples as our estimate of the expectation value of energy. But how do we calculate the error in our estimate? The most normal estimate of the error of the mean is the standard error of the sample mean, given by

$$\sigma_E = \frac{\sigma}{\sqrt{n}}, \tag{5.17}$$

where σ is the standard deviation of the samples and n is the number of samples. In our case however, this estimate of the error does not apply, as our samples are correlated. Since we only move particles a little at a time, the energy at one iteration is correlated with the energy at the next iteration. To avoid variations in our measurements due to correlations being interpreted as noise, we need a way to eliminate the correlations. The blocking method does just this. In very simple terms, it collapses together neighbouring measurements by taking the average until the measurements are no longer correlated, and then by using how many times the measurements needed to be collapsed, finds the true error in our sample mean. The method was introduced by Flyvbjerg and Petersen in 1989[8], but was made much more accesible by Jonsson in 2018[12]. We use his implementation of the method, which we translated into Julia.

Chapter 6

Evaluating the Slater Determinant

6.1 Factorizing the Restricted Determinant

Given a system of n particles, recall that the restricted Hartree-Fock method produces $n/2$ spatial function $\phi_i(x)$ that we make into n spin-orbitals by adding spin as such

$$\begin{aligned} \phi_i(\mathbf{x}) &= \phi_i(x) \otimes \alpha(m_s) \quad \text{for } i = 1,2,3,...,n/2. \\ \phi_{\frac{n}{2}+i}(\mathbf{x}) &= \phi_i(x) \otimes \beta(m_s) \quad \text{for } i = 1,2,3,...,n/2. \end{aligned} \tag{6.1}$$

The Slater determinant formed by these orbitals can be written as

$$|\Phi_0\rangle = \frac{1}{\sqrt{n!}} \begin{vmatrix} \phi_1(\mathbf{x}_1) & \dots & \phi_{\frac{n}{2}}(\mathbf{x}_1) & \phi_{\frac{n}{2}+1}(\mathbf{x}_1) & \dots & \phi_n(\mathbf{x}_1) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \phi_1(\mathbf{x}_{\frac{n}{2}}) & \dots & \phi_{\frac{n}{2}}(\mathbf{x}_{\frac{n}{2}}) & \phi_{\frac{n}{2}+1}(\mathbf{x}_{\frac{n}{2}}) & \dots & \phi_n(\mathbf{x}_{\frac{n}{2}}) \\ \phi_1(\mathbf{x}_{\frac{n}{2}+1}) & \dots & \phi_{\frac{n}{2}}(\mathbf{x}_{\frac{n}{2}+1}) & \phi_{\frac{n}{2}+1}(\mathbf{x}_{\frac{n}{2}+1}) & \dots & \phi_n(\mathbf{x}_{\frac{n}{2}+1}) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \phi_1(\mathbf{x}_n) & \dots & \phi_{\frac{n}{2}}(\mathbf{x}_n) & \phi_{\frac{n}{2}+1}(\mathbf{x}_n) & \dots & \phi_n(\mathbf{x}_n) \end{vmatrix}. \tag{6.2}$$

Since the Hamiltonian of our system ignores spin, the ground state will consist of $n/2$ electrons with spin-up and $n/2$ electrons with spin-down. We can order these such that the first $n/2$ spatial/spin coordinates $\mathbf{x}_i$ represent electrons with spin-up, and the rest electrons with spin-down. Ordering the electrons like this means that the solution is no longer anti-symmetric under the exchange of electrons with opposite spins, but all expectation values we care about, such as energy, will still be unchanged [16]. With this ordering, the determinant in equation 6.2 will be greatly simplified. The functions ϕ_i that do not represent the spin of their argument $\mathbf{x}_i$ will vanish. And the function that "match" the spin of their argument will have their spin-part evaluated to

1, leaving only the spatial part. With this ordering then, we get

$$|\Phi_0\rangle = \frac{1}{\sqrt{n!}} \begin{vmatrix} \phi_1(x_1) & \dots & \phi_{\frac{n}{2}}(x_1) & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \phi_1(x_{\frac{n}{2}}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}}) & 0 & \dots & 0 \\ 0 & \dots & 0 & \phi_1(x_{\frac{n}{2}+1}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}+1}) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & \phi_1(x_n) & \dots & \phi_{\frac{n}{2}}(x_n) \end{vmatrix}. \quad (6.3)$$

The matrix we are taking the determinant of is now block-diagonal, and the determinant of a block-diagonal matrix is given by the product of the determinants of the blocks. Each block contains only spatial functions, so we can evaluate these determinants as we would with a matrix of numbers. Evaluating the wave function is therefore greatly simplified to:

$$|\Phi_0\rangle = \frac{1}{\sqrt{n!}} \begin{vmatrix} \phi_1(x_1) & \dots & \phi_{\frac{n}{2}}(x_1) \\ \vdots & \vdots & \vdots \\ \phi_1(x_{\frac{n}{2}}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}}) \end{vmatrix} \cdot \begin{vmatrix} \phi_1(x_{\frac{n}{2}+1}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}+1}) \\ \vdots & \vdots & \vdots \\ \phi_1(x_n) & \dots & \phi_{\frac{n}{2}}(x_n) \end{vmatrix}. \quad (6.4)$$

This expression will be evaluated at each step in the Monte-Carlo calculation. But from one evaluation to another, only a single particle with position x_i is moved, which means that only a single row of one of these blocks is changed. Subsection 10.4.3 describes an efficient procedure to evaluate these determinants when only a single row has changed from a previous calculation.

6.2 Derivatives of the Slater determinant

For the variational Monte-Carlo calculation we will need the first and second derivatives of our wave function.

If we were to write out the terms resulting from a determinant of a $n \times n$ matrix, we would get $n!$ terms, each being a product of n elements from the matrix, with none of the elements coming from the same row or column. One of these terms would be the product of the diagonal elements, but all possible combinations of elements that don't share rows or columns will be included. For our Slater determinant, this means that all the $n!$ terms will include a given positional argument x_i exactly once. The term including only the diagonal elements for instance, is written out as

$$\phi_1(x_1)\phi_2(x_2)\dots\phi_{n-1}(x_{n-1})\phi_n(x_n). \quad (6.5)$$

The derivative of such terms wrt. a position x_i is quite simple, only the function that takes x_i as an argument is affected.

$$\frac{\partial}{\partial x_1}\phi_1(x_1)\phi_2(x_2)\dots\phi_{n-1}(x_{n-1})\phi_n(x_n) = \frac{\partial\phi_1(x_1)}{\partial x_1}\cdot\phi_2(x_2)\dots\phi_{n-1}(x_{n-1})\phi_n(x_n). \tag{6.6}$$

The derivative of the entire determinant wrt. x_i can therefore be written quite compactly, with only the row containing functions of x_i being affected.

$$\frac{\partial}{\partial x_1}|\boldsymbol{\Phi}_0\rangle = \frac{1}{\sqrt{n!}}\begin{vmatrix} \frac{\partial\phi_1(x_1)}{\partial x_1} & \dots & \frac{\partial\phi_{\frac{n}{2}}(x_1)}{\partial x_1} \\ \vdots & \vdots & \vdots \\ \phi_1(x_{\frac{n}{2}}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}}) \end{vmatrix} \cdot \begin{vmatrix} \phi_1(x_{\frac{n}{2}+1}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}+1}) \\ \vdots & \vdots & \vdots \\ \phi_1(x_n) & \dots & \phi_{\frac{n}{2}}(x_n) \end{vmatrix}. \tag{6.7}$$

And the second derivative can similarly be written as

$$\frac{\partial^2}{\partial x_1^2}|\boldsymbol{\Phi}_0\rangle = \frac{1}{\sqrt{n!}}\begin{vmatrix} \frac{\partial^2\phi_1(x_1)}{\partial x_1^2} & \dots & \frac{\partial^2\phi_{\frac{n}{2}}(x_1)}{\partial x_1^2} \\ \vdots & \vdots & \vdots \\ \phi_1(x_{\frac{n}{2}}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}}) \end{vmatrix} \cdot \begin{vmatrix} \phi_1(x_{\frac{n}{2}+1}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}+1}) \\ \vdots & \vdots & \vdots \\ \phi_1(x_n) & \dots & \phi_{\frac{n}{2}}(x_n) \end{vmatrix}. \tag{6.8}$$

We wrote out the derivatives wrt. x_1 for compactness, but for any position x_i, the result is same, with the derivatives being taken of the corresponding row in the corresponding block.

The problem of evaluating all of these determinants efficiently is covered in section 10.4.3.

Chapter 7

Feedforward Neural Networks

A feedforward neural network (FNN) is a model central to the field of machine learning. Even though it is a relatively simple model, it still sees use among more complicated models due to its flexibility and performance.

FNNs are a type of artificial neural network which can be used for approximating any arbitrary continuous function. The universal approximation theorem for neural networks states that a neural network with only one layer can approximate any continuous function from one finite dimensional space to another [11]. The ground state wave function we are trying to approximate falls within this category of functions, so it should be possible to approximate it with a FNN. The universal approximation theorem requires an infinite number of neurons in a single layer though, which is not possible in practice. Despite this, FNNs often achieve good results with a limited number of neurons over a limited number of layers.

It is therefore worth investigating how well a feedforward neural network can solve the Schrödinger Equation, as has been done by others[13] [1].

This chapter will cover the inner workings of a FNN and how it is trained to better approximate target data from the target function. We will finish the chapter by delving into automatic differentiation, which will be used to calculate some derivatives of the network. From here on we will write "neural network", or "network" instead of "feedforward neural network", for brevity.

7.1 Network Structure

A neural network takes an input vector $\mathbf{a}^{(0)}$ and after many operations on this input returns an output vector $\mathbf{a}^{(L)}$. These operations are done sequentially, each one belonging to what we call a layer.

A layer in a neural network takes an input vector $\mathbf{a}^{(l-1)}$ and returns the output vector $\mathbf{a}^{(l)}$ which comes from the operation

$$\mathbf{a}^{(l)} = \sigma_l(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}), \tag{7.1}$$

where $\mathbf{W}^{(l)}$ is the weight matrix, $\mathbf{b}^{(l)}$ is the bias, and σ_l is the activation function for layer l. The activation function σ_l is a simple $\mathbb{R} \to \mathbb{R}$ function which acts element-wise on the elements of its input vector.

This means that each element of $\mathbf{a}^{(l)}$ is a weighted sum of the elements of $\mathbf{a}_{i-1}$, plus an element of the bias $\mathbf{b}^{(l)}$, all sent through the activation function. This is a very powerful operation. If $\mathbf{a}^{(l)}$ and $\mathbf{a}^{(l-1)}$ both contain 50 numbers, the 50 numbers of $\mathbf{a}^{(l)}$ are combined in 50 different ways, giving 50 potential useful interpretations of the data. The activation function can then dampen, amplify or transform the results in a non-linear way, introducing even more flexibility in what can be modelled.

The entire neural network of L layers can be described by the set of L operations

$$\begin{aligned}
\mathbf{a}^{(1)} &= \sigma_1(\mathbf{W}^{(1)}\mathbf{a}^{(0)} + \mathbf{b}^{(1)}). \\
\mathbf{a}^{(2)} &= \sigma_2(\mathbf{W}_2\mathbf{a}^{(1)} + \mathbf{b}^{(2)}). \\
&\dots \\
\mathbf{a}^{(L)} &= \sigma_L(\mathbf{W}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}).
\end{aligned} \tag{7.2}$$

The flexibility offered by a single layer is only increased when chaining them together like this. The intermediate operations make $\mathbf{a}_i$ a more useful representation than $\mathbf{a}_{i-1}$, so that finally, the final operation can leverage the information in $\mathbf{a}_{L-1}$ to produce the desired result.

7.1.1 Explainability

Often, the only numbers that make any intuitive sense in this process is the input and the output. Consider a network which takes a picture of an animal as input(a vector of pixel values), and then returns a "cat"-score. Naturally, we want the network to give pictures of cats a high cat score and pictures of other animals a low cat score. The input of the network has a simple interpretation, it is the pixel values of a picture of an animal. The output of the network also has a simple interpretation: A high, medium or low score means the network has high, middling or low confidence the picture is of a cat. But what about the other numbers in the operation? What do the weights $\mathbf{W}^{(l)}$ and biases $\mathbf{b}^{(l)}$ say about what the network knows about cats? What can an intermediate value $\mathbf{a}^{(l)}$ tell us about anything?

These questions belong to the field of model explainability, and sometimes we can get answers [7], maybe we can for instance single out a part of the network which is sensitive to the appearance of whiskers. But sometimes, the inner workings of a neural network are a mystery. If it gives the right answer we don't know how, and if it gives the wrong answer we don't know why. The use of heuristics and trial and error are therefore common when choosing the number of layers and the shape of their weight matrices.

7.1.2 Activation Functions

The activation functions in a FNN serve multiple purposes. They can introduce non-linearity to the model as is the case of the ReLU activation function:

$$\mathrm{ReLU}(x) = \mathrm{max}(0, x). \tag{7.3}$$

They can also make model results less volatile by dampening large values when they appear as is the case of the Sigmoid and tanh activation functions (which are also non-linear)

$$\begin{aligned} \mathrm{Sigmoid}(x) &= \frac{1}{1+e^{-x}}. \\ \mathrm{tanh}(x) &= \frac{e^{2x}-1}{e^{2x}+1}. \end{aligned} \tag{7.4}$$

These are commonly used for these features, as well as for their simple derivatives. ReLU especially is very fast to compute, and is therefore often used in very large networks to gain non-linearity at a low cost. The plots of these three functions is shown in figure 7.1.

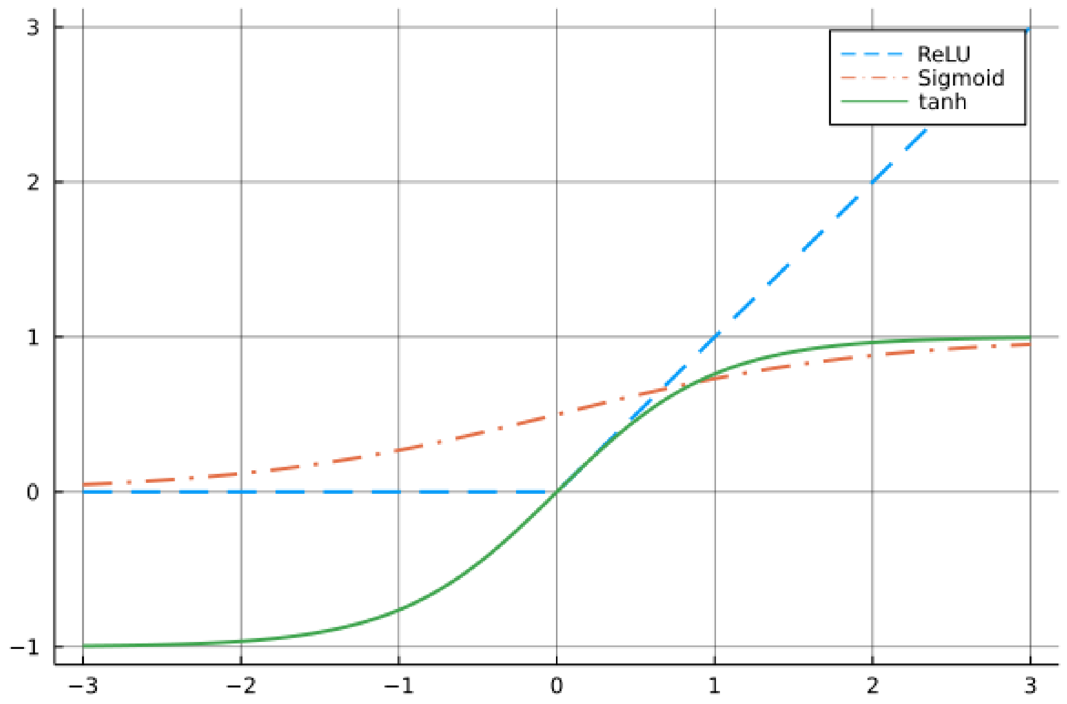


Figure 7.1: The ReLU, Sigmoid and $\tanh$ activation functions. Notice the ReLUs sharp angle at 0, where negative values are all set to zero. The Sigmoid function has an upper bound at 1 and lower bound at 0, which it approaches asymptotically. $\tanh$ is very similar, but has a lower bound at -1, and is much steeper around 0.

For our final layer, which produces the wavefunction output, we employ the exponential function as activation, inspired by [20]. This function will hopefully amplify the network output and make the output more sensitive to subtle differences in input.

7.2 Optimization

Now that we have shown how the input to a network is turned into the output, let's look at how we change the network so that it gives the output we want.

7.2.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a method of minimizing a function by calculating its gradient using a small random subset of data (even though the actual gradient requires the full data set), and updating the function parameters by the negative gradient. The following section uses a simple example model to show how and why SGD is used.

Consider the linear function

$$y(x, \alpha, \beta) = \alpha x + \beta. \tag{7.5}$$

We can use it to model something we think has a linear nature, for instance the height (model input x) and weight (model output y) of an adult.

To use our model, we need to specify our parameters. We set them to some initial guess, $\alpha = 0.8$ and $\beta = -80$. We demonstrate how the model can be used with some example data: From a data set of 1 million people, we randomly pick a person with a height of 170cm and a weight of 60kg. The model predicts the weight to be $\alpha x + \beta = 0.8 * 170 - 80 = 56\text{kg}$.

To judge how well our linear function models the target data, we choose a cost function: the squared distance

$$C(y, t) = \frac{1}{2}(y - t)^2, \tag{7.6}$$

where y is the model prediction, and t is the true value. A low cost is good, and a high cost is bad.

The cost in the example is then $\frac{1}{2}(y - t)^2 = \frac{1}{2}(56 - 60)^2 = 8$.

If we want to minimize the cost, we need to change y, and to change y we need to change the parameters of our model, in this case α and β. With the input data set, we write the cost as a function of the model parameters: $C(\alpha, \beta)$. This function has two inputs, and a single number, the cost, as an output, which means we can visualize it as a 3D height map like the one shown in figure 7.2.

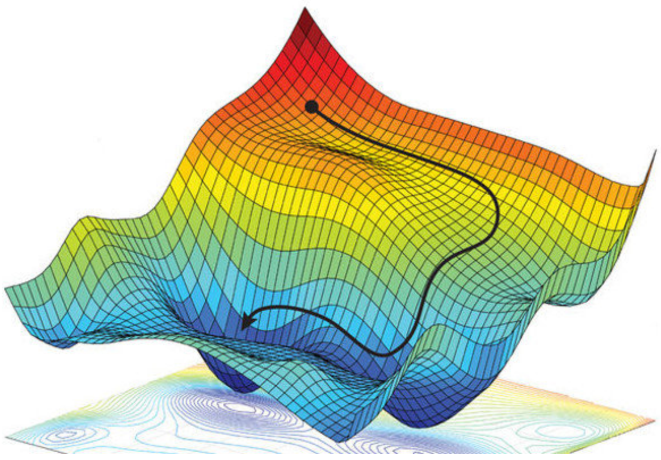


Figure 7.2: An illustration of the path that can be taken on the when using SGD to minimize a cost function, illustrated by the height map.

The lowest point on this graph is where the cost is the lowest. The parameters that correspond to this lowest point can be found by evaluating the gradient of the cost function.

First, some intuition: Say you were standing on the surface shown in figure 7.2 blindfolded, at some random point. If you wanted to get to the lowest point, a good strategy would be to move in the downhill direction until you reached some bottom point. Simply feeling your way downwards. Although you could get stuck in a pit which is not at the very lowest point, it's still much better than just moving around randomly. The mathematical way of finding the lowest point of a function is the same. You take the gradient to find which direction the value is decreasing most, then you shift your parameters in that direction. By repeating this operation, you are bound to reach some bottom point (if there is one). The very bottom is called the global minimum, while other "pits in the terrain" are called local minima.

While this procedure will reduce the cost for this single prediction, it might make the model worse for future predictions. We do not want to optimize the model for this single person, we want to optimize it for the entire data set. An alternative is to compute the gradient of our model for every single person in our data set, and use the average gradient to minimize the cost. This is called gradient descent. However, this is computationally expensive, and we can easily get stuck at a local minimum.

Stochastic gradient descent attempts to remedy these problems. While gradient descent takes the gradient of $C(\alpha, \beta)$ with complete data and uses it to minimize the cost, SGD uses the gradient of $C(\alpha, \beta)$ with small set of randomly chosen data. This has two advantages: First, we do not need to compute the gradient with the entire data set, which is computationally expensive. Second, we get a gradient which moves our parameters in mostly the right direction, leading us slowly towards the bottom with randomness that hopefully lets us escape local minima.

Knowing this, we optimize our model with a single step of SGD using our randomly chosen data point. The gradient of our model using this data is found by

$$\begin{aligned} \frac{\partial C}{\partial y} &= y - t, \\ \frac{\partial C}{\partial \alpha} &= \frac{\partial y}{\partial \alpha}\frac{\partial C}{\partial y} = \alpha * (y - t) = 0.8 * (-4) = -3.2, \\ \frac{\partial C}{\partial \beta} &= \frac{\partial y}{\partial \beta}\frac{\partial C}{\partial y} = 1 * (y - t) = -4. \end{aligned} \tag{7.7}$$

This suggests that both α and β should be higher (negative gradient gives the direction of decrease in cost). Using SGD, we change our parameters by some small amount proportional to the negative gradient. This proportionality is called the learning rate, and its optimal value depends on the problem and model at hand. After updating our parameters, we choose some new data, compute the gradient, and update our parameters once more. When the gradient gets very small or we have reached our desired number of steps, the SGD process is complete, and we have our final model.

Instead of usign a simple learning rate, we can also choose to use a more sophisticated optimization algorithm, which instead of using just a learning rate, also uses a combination of the previously computed gradients to improve convergence. We implement on of these optimizers, ADAM[14], which greatly improves convergence.

7.2.2 Back-propagation

As the previous section has shown, to optimize our model, we need its gradient. In the previous section this was easy, we only had to chain together a couple derivatives, and we had the gradient. But in an FNN there are many more parameters, and many more operations, which means we need a more systematic way to compute the gradient. This section will show such a way, an algorithm called back-propagation.

First we define a new term $\mathbf{z}^{(l)}$, an intermediate term in the evaluation of a layer

$$\begin{aligned} \mathbf{a}^{(l)} &= \sigma_i(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) = \sigma_i(\mathbf{z}^{(l)}), \\ &\Rightarrow \mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}. \end{aligned} \tag{7.8}$$

Then we define the cost function we want to take the derivative with respect to, we call it C. We now take the derivatives of this cost function with respect to the different layer parameters. We use the chain rule and our new term $\mathbf{z}^{(l)}$, as this will prove useful.

The following expressions include derivatives with vectors, so we need to take extra care with the shapes of all terms. NOTE: We choose here to use the unnatural denominator layout, meaning that the first dimension of derivatives has the length of the denominator and the second dimension has the length of the numerator (the transpose of the natural way). Therefore, the terms coming from the chain rule are in reverse order (since they have all been transposed and $(AB)^T = B^T A^T$). We choose this layout because it leads to (slightly) faster and cleaner Julia code. Julia arrays are stored in column-major order, which means that operating along columns is faster, and the standard vector is a column vector. The denominator layout lets us avoid using row-vectors in the implementation. It does however require us to pay extra attention for terms like

$$\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{a}_i^{(l-1)}} = (\mathbf{W}^{(l)})^T, \tag{7.9}$$

where the transpose comes from the choice of layout.

With this in mind we start working towards the main results we are after, the parameter derivatives:

$$\begin{aligned} \frac{\partial C}{\partial \mathbf{W}_{ij}^{(l)}} &= \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}_{ij}^{(l)}} \frac{\partial C}{\partial \mathbf{z}^{(l)}}, \\ \frac{\partial C}{\partial \mathbf{b}^{(l)}} &= \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \frac{\partial C}{\partial \mathbf{z}^{(l)}}. \end{aligned} \tag{7.10}$$

We start with simplifying the leftmost factors in the expressions. Following the definition of $\mathbf{z}^{(l)}$.

$$\begin{aligned} \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}_i^{(l)}} &= [0, ..., 0, \overset{i}{1}, 0, ..., 0], \\ \Rightarrow \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} &= I, \end{aligned} \tag{7.11}$$

where the floating i indicates the i'th index in the row vector. This gives

$$\frac{\partial C}{\partial \mathbf{b}^{(l)}} = \frac{\partial C}{\partial \mathbf{z}^{(l)}}. \tag{7.12}$$

The derivative of $\mathbf{z}^{(l)}$ wrt. $\mathbf{W}^{(l)}$ is a 3-dimentional tensor, so we index into it to make working with it more manageable. Its elements are

$$\begin{aligned}\frac{\partial \mathbf{z}_k^{(l)}}{\partial \mathbf{W}_{ij}^{(l)}} &= \delta_{ki}\mathbf{a}_j^{(l-1)},\\ \Rightarrow \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}_{ij}^{(l)}} &= [0, ..., 0, \overset{i}{\mathbf{a}_j^{(l-1)}}, 0, ..., 0],\end{aligned} \tag{7.13}$$

where δ_{ki} is the kronecker delta function which equals 1 for $k = i$ and 0 for $k \neq i$.

Recall from equation 7.10 that we take the vector product

$$\frac{\partial C}{\partial \mathbf{W}_{ij}^{(l)}} = \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}_{ij}^{(l)}}\frac{\partial C}{\partial \mathbf{z}^{(l)}}. \tag{7.14}$$

Notice that the first vector is a row vector, while the second is a column vector. This means that it is an inner product, and since the first vector only has one non-zero term, as we just showed in equation 7.13, we can make some large simplifications. We now know that $\frac{\partial C}{\partial \mathbf{W}^{(l)}}$ is a matrix where an element in row i and column j is the product of the i'th element of $\frac{\partial C}{\partial \mathbf{z}^{(l)}}$ and $\mathbf{a}_j^{(l-1)}$. This can be written compactly as the outer product

$$\frac{\partial C}{\partial \mathbf{W}^{(l)}} = \frac{\partial C}{\partial \mathbf{z}^{(l)}}(\mathbf{a}^{(l-1)})^T. \tag{7.15}$$

The rightmost factors in equation 7.10 are the same for both derivatives, but require some more work.

$$\begin{aligned}\frac{\partial C}{\partial \mathbf{z}^{(l)}} &= \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}}\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}}\frac{\partial C}{\partial \mathbf{z}^{(l+1)}},\\ &= \frac{\partial \sigma_i(\mathbf{z}^{(l)})}{\partial \mathbf{z}^{(l)}}(\mathbf{W}^{(l+1)})^T\frac{\partial C}{\partial \mathbf{z}^{(l+1)}}.\end{aligned} \tag{7.16}$$

We again used the definition of $\mathbf{z}_i$ to simplify a partial derivative, and we used the definition of $\mathbf{a}_i$ to replace its derivative with the derivative of $\sigma_i(z_i)$, which can be calculated directly.

The only term left which is not directly known is $\frac{\partial C}{\partial z_{i+1}}$. The keen reader will notice however, that equation 7.16 does in face give us a direct expression for this term, only for all $i \neq L$. For $L = i$ we get

$$\frac{\partial C}{\partial z_L} = \frac{\partial C}{\partial a_L}\frac{\partial a_L}{\partial z_L} = \frac{\partial C}{\partial a_L}\frac{\partial \sigma_L(z_L)}{\partial z_L}, \tag{7.17}$$

where both terms in the final expression can be calculated directly, as a_L is the direct input to C and $\mathbf{z}_L$ is the direct input to σ_i.

Using these equations we can calculate all weight and bias derivatives using the recursive algorithm called back-propagation:

1. Calculate all a_i and z_i in the network (Forward pass)
2. Now, at layer L: Calculate $\frac{\partial C}{\partial z_L}$, use it to calculate $\frac{\partial C}{\partial W_L}$ and $\frac{\partial C}{\partial b_L}$
3. For each layer i from layer $L-1$ to the first layer:
    (a) Use $\frac{\partial C}{\partial z_{i+1}}$ to calculate $\frac{\partial C}{\partial z_i}$
    (b) Use $\frac{\partial C}{\partial z_i}$ to calculate $\frac{\partial C}{\partial W_i}$ and $\frac{\partial C}{\partial b_i}$

7.3 Neural Network Wave Function

We are going to use a neural network as part of a wave function used in a variational Monte-Carlo calculation. Let the positions of the electrons be given by $\mathbf{x}$. We choose to make the input to the network the positions of the electrons, sorted by value. That is

$$\mathbf{a}^{(0)} = \chi(\mathbf{x}), \tag{7.18}$$

where χ is a function that sorts the positions $x_1, x_2, \ldots, x_n$ by value. The output of the network will simply be the output of the final layer, which has the exponential function as activation, and returns a single number. We denote the network by the function

$$NN(\chi(\mathbf{x})). \tag{7.19}$$

Since we will be using it as part of a wave function for VMC, we will need to be able to evaluate its output, its first and second derivatives with respect its inputs, and its derivative with respect to its parameters.

We start with the derivative with respect to its parameters, the weights and biases of the linear layers. We want derivatives of the output of the network, so we let the cost function equal the output of the network

$$C = NN(\chi(\mathbf{x})), \tag{7.20}$$

the derivatives of the network with respect to the parameters are then found using the back-propagation procedure described in section 7.2.2.

7.3.1 Quantum Force

When using the Neural Network as part of a wave function we will need to compute the quantum force. We will therefore need the derivative

$$\frac{\partial NN(\chi(\mathbf{x}))}{\partial x_i}. \tag{7.21}$$

Remember that $\mathrm{NN}(\chi(\mathbf{x})) = C$ and that $x_i = \mathbf{a}_j^{(0)}$, where index j is where element i of $\mathbf{x}$ is moved when we sort $\mathbf{x}$. We can now find an expression for the partial derivatives of all $\mathbf{a}_j^{(0)}$ at the same time

$$\frac{\partial C}{\partial \mathbf{a}^{(0)}} = \frac{\partial C}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{a}^{(0)}} = \frac{\partial C}{\partial \mathbf{z}^{(1)}} (\mathbf{W}^{(1)})^T, \tag{7.22}$$

which can then be mapped back to the derivatives of x_i by swapping the elements back. The factor $\frac{\partial C}{\partial \mathbf{z}^{(1)}}$ is found during the back-propagation algorithm, so we can simply reuse that result here, making computing the first derivative a very fast operation.

The second derivatives are not as simple to compute however. We could set up the analytical expressions and get a recursive result similar to the back-propagation algorithm, but we will instead use automatic differentiation to compute them, saving on development time, and showcasing an efficient, flexible technique for computing derivatives.

7.3.2 Automatic Differentiation

In the previous section we used to chain rule and reuse of gradients to very efficiently calculate derivatives. This approach is not limited to taking gradients of neural network parameters however, it can also be used to efficiently take the derivative of many functions implemented in code.

All functions written in code are built up by simple operations such as addition, multiplication, exponentiation, sin, etc. . By breaking functions into these component parts, a program can use the chain rule to compute the derivative of the entire function. This process is called automatic differentiation (AD), or autodiff for short. There are two popular types of AD: Forward accumulation AD (forwarddiff, or forward mode AD), and reverse accumulation AD (reversediff, or reverse mode AD).

We make use of both these types when computing the double derivative of the neural network wrt. its inputs. Section 12.3 covers how and why. Here, we give an introduction to the theory of the two types.

Forward Accumulation

Consider the following computations:

$$\begin{aligned} x_2 &= a(x_1). \\ x_3 &= b(x_2). \\ y_1 = c(x_3). \quad y_2 &= d(x_3). \end{aligned} \tag{7.23}$$

We use the chain rule to write out the derivatives of the outputs wrt. the input

$$\begin{aligned}\frac{\partial y_1}{\partial x_1} &= \frac{\partial y_1}{\partial x_3}\cdot\frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial x_1} = \frac{\partial c(x_3)}{\partial x_3}\cdot\frac{\partial b(x_2)}{\partial x_2}\frac{\partial a(x_1)}{\partial x_1}.\\ \frac{\partial y_2}{\partial x_1} &= \frac{\partial y_2}{\partial x_3}\cdot\frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial x_1} = \frac{\partial d(x_3)}{\partial x_3}\cdot\frac{\partial b(x_2)}{\partial x_2}\frac{\partial a(x_1)}{\partial x_1}.\end{aligned} \tag{7.24}$$

If we want to evaluate these two derivatives and avoid unnecessary calculations, should we:

1. Compute the two derivatives separately
2. Accumulate derivative products from the left and share them as they become useful
3. Accumulate derivative products from the right and share them as they become useful

Clearly, option 3 is best suited here. This sequence of calculations allows for reuse of derivatives:

$$\begin{aligned}1.t_2 &= \frac{\partial a(x_1)}{\partial x_1},\\ 2.t_3 &= \frac{\partial b(x_2)}{\partial x_2}t_2,\\ 3.\frac{\partial y_1}{\partial x_1} &= \frac{\partial c(x_3)}{\partial x_3}t_3,\\ 4.\frac{\partial y_2}{\partial x_1} &= \frac{\partial d(x_3)}{\partial x_3}t_3.\end{aligned} \tag{7.25}$$

Admittedly, not many calculations are saved here, but the principle is clear: Accumulating products of derivatives in the right order can save calculations. This ordering, starting at the input and accumulating derivatives forward in the same order as the original calculation (in this case equation 7.23), is called forward accumulation automatic differentiation. Note that this entire sequence of steps only gives you derivatives with respect to a single input.

Implementing forward accumulation of derivatives is very straightforward. Simply perform the normal evaluation of the original calculation, but every time you compute a new value, compute and save the accumulated derivative together with that value.

For a demonstration, look back at equation 7.23 and 7.25. We are derivating with respect to x_1. We start by evaluating x_2, and save it together with its derivative t_2. When we then use x_2 to compute something new, like x_3, we use t_2 to quickly compute x_3s derivative t_3. If we later used x_2 or x_3 to compute some new value, we could use t_2 or t_3 to compute its derivative wrt.

the input x_1. This is how we efficiently compute the derivative of everything wrt. the chosen input, with no repeated work.

If we want to compute a derivative wrt. a different input, we need to do the entire forward accumulation process again. Forward accumulation is therefore best suited when there are few inputs. The advantage is that it can quickly compute the derivatives regardless of the number of outputs.

Forward accumulation autodiff is offered by several programming libraries and packages. Examples include Autodiff which offers a python implementation, and ForwardDiff which offers a Julia implementation.

Reverse Accumulation

Now consider this slightly different set of calculations:

$$\begin{aligned} x_3 &= a(x_1) + b(x_2), \\ x_4 &= c(x_3), \\ y_1 &= d(x_4). \end{aligned} \tag{7.26}$$

This time there are two inputs and one output, instead of one input and two inputs. We use the chain rule to write out the derivatives of the output wrt. the inputs

$$\begin{aligned} \frac{\partial y_1}{\partial x_1} &= \frac{\partial y_1}{\partial x_4}\frac{\partial x_4}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_1} = \frac{\partial d(x_4)}{\partial x_4}\frac{\partial c(x_3)}{\partial x_3} \cdot \frac{\partial a(x_1)}{\partial x_1}, \\ \frac{\partial y_1}{\partial x_2} &= \frac{\partial y_1}{\partial x_4}\frac{\partial x_4}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} = \frac{\partial d(x_4)}{\partial x_4}\frac{\partial c(x_3)}{\partial x_3} \cdot \frac{\partial b(x_2)}{\partial x_2}. \end{aligned} \tag{7.27}$$

Which of the three options from before should we choose to optimize these calculations? This time, option 2 wins out, because the leftmost derivative factors are the same in both expressions. Accumulating derivative products from the left we get the calculations

$$\begin{aligned} &1. t_1 = \frac{\partial d(x_4)}{\partial x_4}, \\ &2. t_2 = t_1 \frac{\partial c(x_3)}{\partial x_3}, \\ &3. \frac{\partial y_1}{\partial x_1} = t_2 \frac{\partial a(x_1)}{\partial x_1}, \\ &4. \frac{\partial y_1}{\partial x_2} = t_2 \frac{\partial b(x_2)}{\partial x_2}. \end{aligned} \tag{7.28}$$

This time, we are computing derivatives of a chosen *output* wrt. the terms used to calculate it. This means that we only need to complete this process once for each output we care about. In forward accumulation, we computed derivatives of all outputs wrt. a chosen *input*, and needed to repeat the process for each input. This makes reverse accumulation much more efficient when the calculation we are derivating has many more inputs than outputs.

- #inputs $\gg$ #outputs $\Rightarrow$ Use reverse accumulation
- #inputs $\ll$ #outputs $\Rightarrow$ Use forward accumulation
- #inputs $\approx$ #outputs $\Rightarrow$ It depends

Reverse accumulation is more complex than the simple forward accumulation procedure described earlier. First, the entire function we are derivating is evaluated, and intermediate values are stored. Then, starting at the final calculation, we use the intermediate values to compute carefully chosen partial derivatives (computing $\frac{\partial c(x_3)}{\partial x_3}$ for instance, required x_3). These partial derivatives are accumulated and propagated towards the start of the calculation.

To avoid unnecessary work, we do not take all possible derivatives. We specify which ones we are interested in, and then compute only the strictly required ones. Finding out which partial derivatives are required and setting up the sequence of calculations to compute them is neither trivial nor fast. However, we only need to do it once for a given function. After this setup, we can use reverse accumulation to quickly compute the derivatives with any input. Note that the back-propagation algorithm described in section 7.2.2 is a special case of reverse accumulation automatic differentiation, only we took care of the setup of operations by hand instead of a program doing it "automatically".

Setting up the sequence of calculations used for reverse accumulation and then performing them is handled by many different programming libraries and packages. Examples include the Python libraries Autograd, TensorFlow and PyTorch, and the Julia packaged ReverseDiff and Zygote.

7.3.3 Kinetic energy - Forward over Reversediff

For the kinetic energy of our wave function we will need the double derivatives

$$\frac{\partial^2 \mathrm{NN}(\chi(\mathbf{x}))}{\partial x_i} = \frac{\partial^2 \mathrm{NN}(\mathbf{a}^{(0)})}{\partial \mathbf{a}_j^{(0)}}, \tag{7.29}$$

where j is the index that element i of $\mathbf{x}$ is moved when $\mathbf{x}$ is sorted.

To compute these derivatives we will use automatic differentiation. The first derivative is of an n to 1 function, so we will use reverse accumulation to compute it.

We can then take the derivative of the first derivative to get the second derivative. The first derivative is an n to n function, so we tried both reverse and forward accumulation to take the second derivative, but we found forward to be much faster. This process actually produces all second order partial derivatives, that is, the entire hessian matrix for the neural network. Still, it is a very fast operation, and by extracting the diagonal of the hessian, we get the second derivatives. We go into further details in section 12.3 of the implementation.

Chapter 8

The Slater-Neural Network Wave Function

As stated previously, our main goal is to combine the decent trial wave function from Hartree-Fock with a neural network that we train to model correlations between electrons. The variational Monte Carlo method lets us compute expectation values and variationally optimize our wave function, given that we can evaluate our wave function and its derivatives.

In this chapter, we will show how to evaluate a Slater determinant and its derivatives, and how we will combine it with a neural network.

8.1 Combining Wave Functions

With the use of a neural network as a wave function covered earlier, and with the Slater determinant being made workable in the previous section, we are ready to consider combining the two to try and find a good approximation to the ground state of quantum mechanical systems. We define our combined wave function as

$$\Omega(\mathbf{x}) = \Phi(\mathbf{x})\mathrm{NN}(\gamma(\mathbf{x})), \tag{8.1}$$

where $\gamma(\mathbf{x})$ sorts the positions x_i from lowest to highest. This ensures that the interchange of two electrons will not change the sign of the $\mathrm{NN}(x)$ part of the wave function. Since the Slater determinant is anti-symmetric, the total wave function become anti-symmetric, as it must be for fermions. It is possible to allow the Neural Network instead take the raw input and have it learn to be quite symmetric, but we opted for a more straightforward approach which also ensures perfect anti-symmetry.

8.1.1 Derivatives

The derivatives of our total wave function will require the derivatives of its components, but no new terms we haven't seen before.

For the quantum force we will need the first derivative. The product rule gives us

$$\frac{\partial \Omega(\mathbf{x})}{\partial x_i} = \frac{\partial \Phi(\mathbf{x})}{\partial x_i} NN(\gamma(\mathbf{x})) + \Phi(\mathbf{x}) \frac{\partial NN(\gamma(\mathbf{x}))}{\partial x_i}. \tag{8.2}$$

The quantum force is then

$$\frac{2}{\Omega(\mathbf{x})} \frac{\partial \Omega(\mathbf{x})}{\partial x_i} = \frac{2}{\Phi(\mathbf{x})} \frac{\partial \Phi(\mathbf{x})}{\partial x_i} + \frac{2}{NN(\gamma(\mathbf{x}))} \frac{\partial NN(\gamma(\mathbf{x}))}{\partial x_i}. \tag{8.3}$$

The first term can be computed using the formulas found in the previous section. Section 10.4.3 covers a very efficient way of computing such terms.

The double derivative will be needed for computing the kinetic energy. The product rule now gives us

$$\frac{\partial^2 \Omega(\mathbf{x})}{\partial x_i^2} = \frac{\partial^2 \Phi(\mathbf{x})}{\partial x_i^2} NN(\gamma(\mathbf{x})) + \Phi(\mathbf{x}) \frac{\partial^2 NN(\gamma(\mathbf{x}))}{\partial x_i^2} + 2 \frac{\partial \Phi(\mathbf{x})}{\partial x_i} \frac{\partial NN(\gamma(\mathbf{x}))}{\partial x_i}. \tag{8.4}$$

The kinetic part of the local energy then becomes

$$\begin{aligned} -\frac{1}{2} \frac{1}{\Omega(\mathbf{x})} \frac{\partial^2 \Omega(\mathbf{x})}{\partial x_i^2} = -\frac{1}{2} \Bigg(& \frac{1}{\Phi(\mathbf{x})} \frac{\partial^2 \Phi(\mathbf{x})}{\partial x_i^2} + \frac{1}{NN(\gamma(\mathbf{x}))} \frac{\partial^2 NN(\gamma(\mathbf{x}))}{\partial x_i^2} \\ & + 2 \frac{1}{\Phi(\mathbf{x})} \frac{\partial \Phi(\mathbf{x})}{\partial x_i} \frac{1}{NN(\gamma(\mathbf{x}))} \frac{\partial NN(\gamma(\mathbf{x}))}{\partial x_i} \Bigg). \end{aligned} \tag{8.5}$$

We have already covered how to compute the terms and factors included in this expression.

The final derivative we will need is the derivative of the wave function with respect to its parameters, all divided by the wave function. This will be simplified to only including the neural network, since the Slater determinant has no parameters.

$$\frac{1}{\Omega(\mathbf{x})} \frac{\partial \Omega(\mathbf{x})}{\partial \theta} = \frac{1}{NN(\gamma(\mathbf{x}))} \frac{\partial NN(\gamma(\mathbf{x}))}{\partial \theta}. \tag{8.6}$$

Back-propagation of the neural network gives us all of the parameter derivatives, which means we now have all the expression we need to put this wave function into use.

8.2 Limitations of our method

The Slater-NN wave function has some inherent limitations.

It only supports one dimensional systems, as we haven't computed one- and two-body integrals in higher dimensions, which we need to get a Hartree-Fock determinant that works for higher dimensions. The higher dimensional two-body integrals are very expensive to compute. Though one could bite the bullet and run the expensive calculations, using a lower resolution might be worth it. A small decrease in the accuracy of the determinant might not matter too much, as the Neural Network might not need it. A smaller basis set or a worse resolution over the grid can greatly speed up the calculations. Another way around the expensive two-body integrals would be to use Gaussian type orbitals. Although that would make the fast evaluation of the harmonic oscillator functions, and their suitability to the quantum dot problem no longer an advantage. A final approach is to use a Slater determinant that has not been optimized by Hartree-Fock.

While the 1D limitation of our Hartree-Fock determinant is very much possible possible to get around, a neural network supporting multiple dimensions would require an entirely different approach. We sort the input of the network to make it symmetric, but this would not work as well in higher dimensions. In 1D, sorting is a very smooth operation, a small change in the positions will only ever make a small change in the sorted positions. Sorting in 2D or 3D would introduce an instability, as if you sort by distance from the center, two electrons at opposite sides of the center might suddenly swap positions in the ordering after one was moved slightly further out. A different approach to symmetry is therefore needed in higher dimensions.

Another large limitation of the Slater-NN wave function is that it does not have the distance between electrons as input. The Jastrow factor, which is a function often used for modelling correlations, uses the distance between electrons explicitly, letting it directly make it less likely for electrons to get close. On the one hand, this makes the Slater-NN wave function require less physical insight about the system, on the other hand, it might give a much worse result. This could of course be remedied by having the distance between electrons as explicit inputs to the network, though this would take some work to implement. Another approach could be to include the Jastrow factor, together with the Slater determinant and the Neural Network.

FermiNet[17] is a neural network introduced in 2020 that solves many of these problems. It has an entirely different approach to anti-symmetry of the wave functions that works in higher dimensions, and has both the distance and difference between positions as inputs to the network. It is also trained using variational Monte Carlo, and has proven to be an excellent solver, being competitive and sometimes outperforming other state of the art methods.

While we see the newly introduced FermiNet as superior to our method, the exciting developments in the field of machine learning used for solving quantum systems motivates us to trying novel approaches, to see what they have to offer.

Part II

Implementation

Chapter 9

The Julia Programming Language

We have chosen to use the Julia programming language to implement the methods discussed in this thesis. We chose Julia because it is fast, easy to write, easy to optimize, and because it has great packages for scientific computing. Julia also has a powerful type system which allows for code that is both readable and efficient.

Specifically, we found python to be ill suited for variational Monte Carlo (VMC) simulations, as such simulations often require a good amount of performance critical native code to be written, which of course is very slow to run using the python virtual machine. There are ways to get around this in python, such as using numba or cython which compiles your python code into fast machine code, but these put big limitations on which language features can be used in your code, such as struggling with compiling functions that use many other functions and classes. These compilers also still have to contend with the fact that Python was not designed to be type safe, making it hard for the compiler to properly optimize complex code.

We also wanted to avoid writing in C++, as it lacks the ease of use of python. Being able to run code as you're writing it, and testing and profiling features with minimal setup leads to much more productive coding. The plotting and linear algebra libraries of C++ also pale in comparison to the ease of use of matplotlib and numpy in python. Lastly, while code in C++ is much more explicit in what it does, and offers more control, it is not strictly necessary for the compiler to produce highly performant code. Whether you start a for loop with "`for (int i = 1; i <= n; i++)`" in C++ or "`for i in 1:n`" in Julia, the compiler will be able to produce exactly as fast machine code.

The solution to our problems with python and C++ was the language Julia. Julia is both fast and readable. It can be run interactively, while still compiling to highly optimized machine code. It has great libraries for linear algebra and plotting, and has great profiling tools for finding bottlenecks in your code. This chapter will cover how Julia achieves this using just-in-time compilation and multiple dispatch, but also some of the big drawbacks Julia has because of this design. Finally, we will present some of the things to keep in mind when trying to make Julia code fast.

9.1 Just-in-time compilation

Julia's greatest strength, and its greatest weakness, is its compiler. When you define a function in Julia, it is not compiled right away. It is only when you first call the function that it is compiled. The compiler looks at the types of the arguments you called the function with, and compiles specialized machine code for exactly these input types. The function is then run, and the compiled code is saved for the next time you might call the function with these types. This is called just-in-time (JIT) compilation, as the function is compiled just as you need to call it.

If you now call the same function again, but with arguments with different types, the function is compiled again for these types, saved for later use, and then executed. Of course, not all functions can take all types as inputs, so we can limit the accepted types by using type annotations, or just hope that the code crashes and gives a useful error message if the wrong thing is sent into a function. We called this Julia's greatest strength, because this design allows us to write a single function definition that can work with a large number of types, such as this one:

```julia
function square_sum(a, b)
    return a^2 + 2 * a * b + b^2
end
```

If you call this function with two Int64 numbers, Julia will compile a specialized function for that. Or if you call it with a Float64 and a complex Float64, or any combination of such numerical types, Julia will compile a specialized function for that. You can even use this function element wise on two arrays of any type of numbers using Julia's dot syntax `square_sum.(A, B)`. This lets you worry more about the mathematics than the intricacies of type promotion. The compiler does it for you, and it does it in an efficient way.

We also called Julia's compiler its greatest weakness. This is because JIT compilation has some massive drawbacks. Every time you load a package or some of your own code for the first time in a Julia session and want to run a function, you need to compile it first, which can take some time. Some larger functions and libraries can take the order of tens of seconds to compile, though most take much less time. The plotting library Plots.jl and its `plot()` are the most notorious examples, taking a combined 25 seconds to load and compile, meaning it takes at least 25 seconds from starting up a Julia session to getting a plot on the screen. Most often however, it only takes a few seconds to have you session and libraries up and running. And once you have paid this upfront cost, everything runs very fast. This upfront cost is also present if you want to export your code however. Calling Julia from python for instance, requires loading the Julia compiler. And creating an executable from your code will in most cases require adding the entire compiler into the executable. And running Julia on embedded devices might incur potentially fatal unexpected delays when a new function is called and execution needs to stop in order to compile it. All of these shortcomings are actively being worked on by the developers, and massive improvements are made regularly. While there are many things Julia cannot do well, it is still very good at the things it was designed to do.

9.2 Multiple Dispatch

The previous section showed how Julia's compiler can compile multiple functions from the same function definition. This section will cover another one of Julia's central design elements, multiple dispatch. We can define many functions with the same signature, and when calling using this function signature, Julia can choose which function to call at runtime, by finding the one which best matches the types of the inputs. With these three function defines, for instance

```
function say_hello(a::String, b::String)
    println("Hello from \$(a) to \$(b)!")
end
function say_hello(a::Number, b::Number)
    println("Hello number \$(a+b)!")
end
function say_hello(a::String, b::Number)
    println("Hello \$(a), you are \$(b)!")
end
```

The following function calls would give the following inputs

```
say_hello("John", "Adam")
say_hello(2, 3)
say_hello("Adam", 4)
say_hello(3, "Adam")

Hello from John to Adam!
Hello number 5!
Hello Adam, you are 4!
MethodError: no method matching say_hello(::Int64, ::String)
Closest candidates are:
  say_hello(::Number, ::Number) at In[1]:5
  say_hello(::String, ::String) at In[1]:1
  ...
```

This might not seem too special, but it allows us to model complex systems with great flexibility without having to use classes and inheritance. We can define a method `kinetic(positions, wf)` for instance, which gives the kinetic energy given a set of positions and a wave function, for each different type of wave function we implement. This would be implemented as the method `wf.kinetic(positions)` of a wave function object in an object oriented language. Multiple dispatch goes further than allowing us to recreate that simple object oriented pattern however, as the function called depends on the type of all the arguments. In an object oriented language, the method called depends only on the type of the object used to call the method.

A very useful consequence of multiple dispatch is operator overloading. In Julia, the operators +, ., *, / and so on are all functions. These functions all have definitions when acting on number types, but there is nothing stopping us from defining what these function do when acting on other types, like our own custom types. This in turn allows us to evaluate more complex functions built up by these basic operators, on our custom types. If you create your own type for describing colors, and define addition and multiplication between instances of this type, and multiplication between this type and an Int64, you can call the `square_sum(a, b)` function with these color instances, and it will work.

An example of a custom type from our code is the type

```
struct Dense_Grad
    layer_idx::Int64
    W_g::Matrix{Float64}
    b_g::Vector{Float64}
end
```

which holds gradient information for a linear layer in the neural network. If we want to multiply the gradient by a number, we would need to manually access the fields of the gradient and multiply them element wise. We would instead like to just write `gradient = gradient * f`, so we implement the multiplication function for these types as follows:

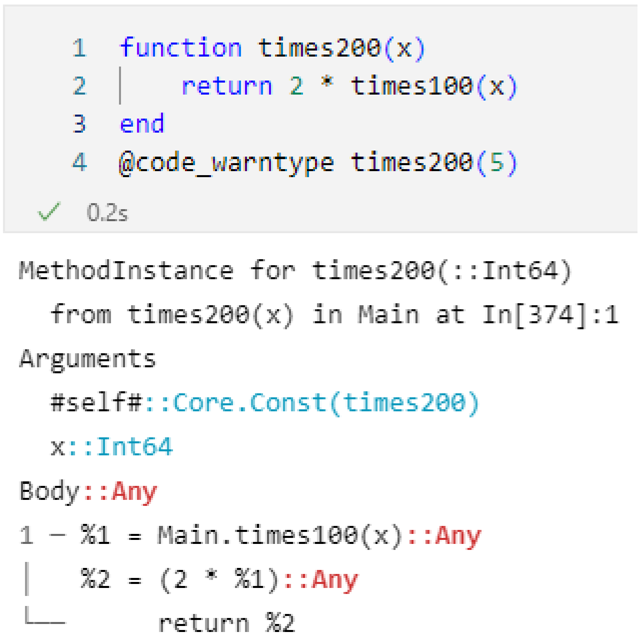
```
function *(d::Dense_Grad, f::Number)
    d.W_g .*= f
    d.b_g .*= f
    return d
end
```

9.3 Writing fast Julia

So far we have shown how the Julia compiler and type system are very flexible and powerful. But this comes at a risk. The compiler is so flexible that it will compile even when it doesn't know all the types used in the function, though this comes with a huge hit to performance. This section covers how to make sure that doesn't happen, and other important performance tips. For more performance tips, see https://docs.julialang.org/en/v1/manual/performance-tips/.

9.3.1 Ensuring type safety

Sometimes when compiling a function, the compiler has no way of knowing what type an operation will return. This can happen when a changeable global variable is used, or when the return type of some operation is dependant on the value of its inputs. The compiler has no choice then but to write machine code that can support any type, which of course will run very slowly. The solution to this is to avoid using global variables, and to write code that is type safe, code that always returns the same types when given the same types as input. It can sometimes be hard to spot that a function is not type safe. If you are unsure of whether or where a function is not type safe, you can use the `@code_warntype` macro. Here we have a function `times200(x)` that is slowing down our code. We see if it is type safe by using the `@code_warntype` macro like this:



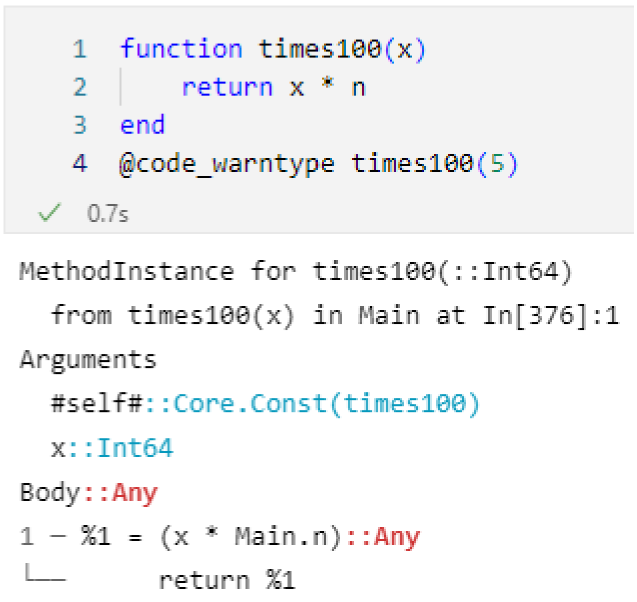
```julia
function times200(x)
    return 2 * times100(x)
end
@code_warntype times200(5)
```

✓ 0.2s

```
MethodInstance for times200(::Int64)
  from times200(x) in Main at In[374]:1
Arguments
  #self#::Core.Const(times200)
  x::Int64
Body::Any
1 ─ %1 = Main.times100(x)::Any
│   %2 = (2 * %1)::Any
└──      return %2
```

Figure 9.1

We see that the type of x is known at compile time, but that the return type of the function `times100(x)` is unknown. This is bad, which is why it is conveniently marked in red. We know know to look at `times100(x)`, since the problem resides there. Finding the implementation of `times100(x)` and using the macro again lets us further pin down our search of where the type un-safety comes from:



```julia
function times100(x)
    return x * n
end
@code_warntype times100(5)
```

```
MethodInstance for times100(::Int64)
  from times100(x) in Main at In[376]:1
Arguments
  #self#::Core.Const(times100)
  x::Int64
Body::Any
1 ─ %1 = (x * Main.n)::Any
└──      return %1
```

Figure 9.2

Here the macro is unneeded of course, as we can plainly see that some global variable n is used. While n might have equaled 100 when the function was compiled, it might be changed to anything at a later point, invalidating the compiled code. Julia therefore need to check what type and value n is in the global scope each time the function is called, leading to slow code. Why does Julia allow the use of global variables in functions then? Because it is sometimes convenient, and can otherwise be avoided.

9.3.2 Reducing Allocations

In many compiled languages, if a function does not allocate and deallocate the memory it uses for its calculations, the resulting memory leak might greatly decrease performance or cause the program to crash. Julia instead has a garbage collector, like many other languages. After too much memory has been allocated, Julia stops running the code, finds all the allocated memory that is no longer needed, frees it, and then continues running. This can quickly become a huge problem for the performance of your programs as both allocating memory and garbage collection are costly operations. The two main ways of solving this problem is to write code that allocates less memory, or to pre-allocate memory and then reuse that memory over and over in a performance critical part of your code.

The benefit of pre-allocation in our VMC code can be showed by changing a small part of our code to instead allocate memory and comparing the speed of the VMC simulation. If we simulate a system of 2 electrons, 20 harmonic oscillator functions, and use a single hidden layer with 10 neurons in our network, running 10 million VMC iterations while computing the gradient takes 3.6 seconds, and allocates 7.595 MiB of memory over 151.65 k allocations. If we change a single line in our code, so that the matrix-vector product in the linear layers allocates some memory:

```
#la.mul!(output, W, input) # The original code puts the result straight in out
output .= W * input # This allocates a vector to hold the result of W*input
```

The 10 million iterations instead take 4.9 seconds and allocates 1.949 GiB of memory over 20.26 M allocations.

A more subtle example of allocations is when we loop over the vector of layers in the network in reverse order to do back-propagation. If we loop over `reverse(layers)`, we allocate the reversed vector of pointers to the layers. If we instead loop over `Iterators.reverse(layers)`, we allocate nothing, as we instead get an iterator which gives us the layers in reverse. Using the same example simulation as above, using `reverse(layers)` makes the simulation take 4.3 instead of 3.6 seconds.

Throughout our code, we have tracked down allocations and come up with many such optimizations to remove allocations. The evaluation of the Slater determinant, the back-propagation through the network, the sorting of the input, the evaluation of the Hermite polynomials, the derivatives and double derivatives, they have all been implemented to allocate no memory. The results of these optimizations is that our code allocates absolutely no memory in performance critical loops. Running 10 or 10 million VMC iterations allocates the same amount of memory. These optimizations really add up, as memory is a very limited resource in such large calculations.

9.3.3 Profiling

The optimizations described in the previous section would have been very difficult to find if not for Julia's excellent profiling tools. You can time and track the number of allocations of any function call with the `@time` macro, or do a more thorough benchmark with the BenchmarkTools.jl package. As mentioned earlier, the `@code_warntype` macro lets you track down variables that are not type safe in your code.

Two of the most powerful profiling tools however, are `@profview` and `@profview_allocs` from ProfileCanvas. jl. These macros give you an interactive figure showing how much time the different parts of a function takes to execute, and how much memory each part allocated, respectively. You can then explore recursively through the function calls to see how the time taken or the allocations made is distributed. In figure 9.3 below, you can see the interactive figure provided when using `@profview` on the entire VMC calculation for 2 electrons described above. The figure below shows how the time is spent during the VMC calculation using our code.

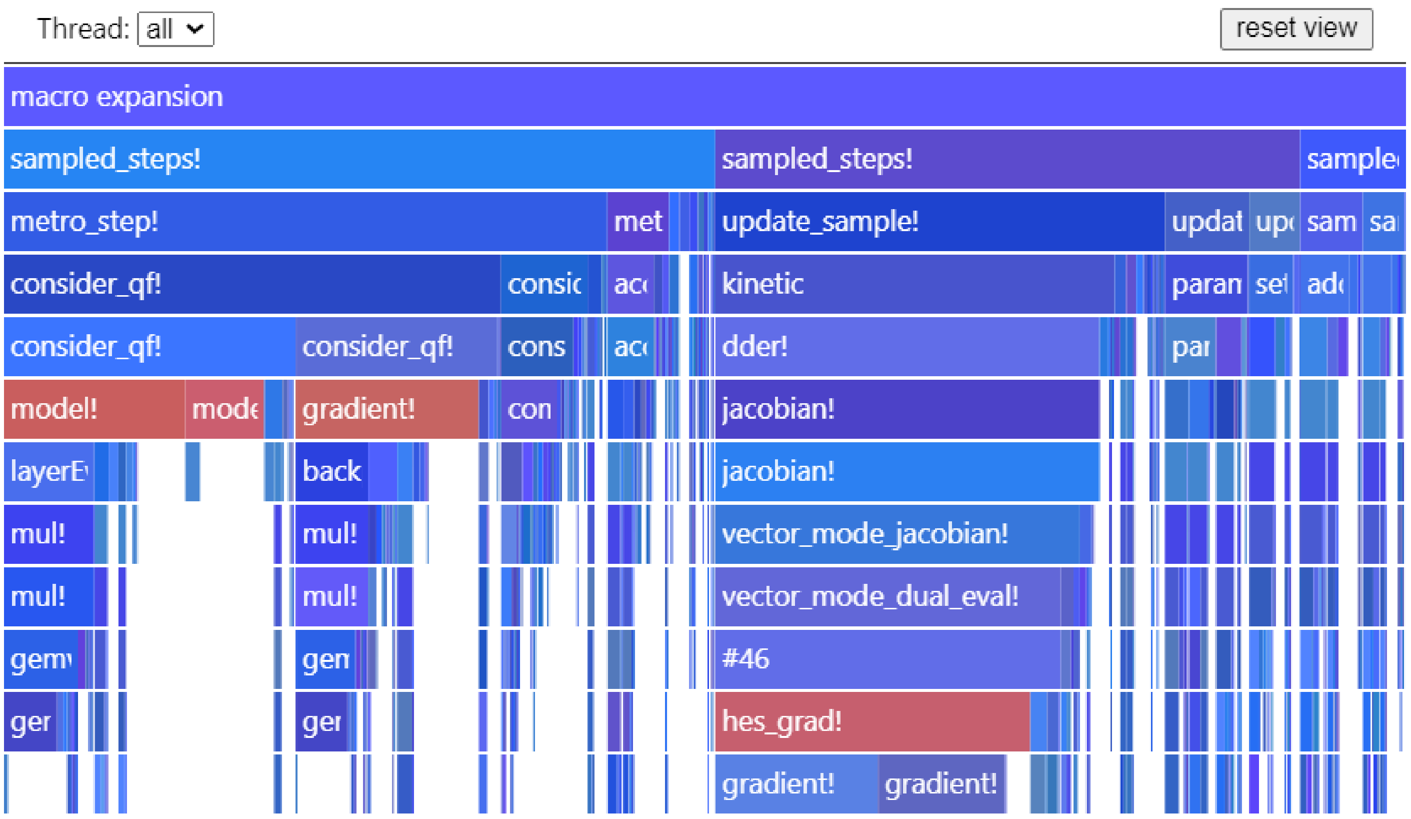


Figure 9.3

Each box is a function that is called, and below each box are the functions that function calls. The width of each box is the time spend executing that function. The boxes that are too small to read can be viewed by selecting one of the boxes above it to limit the view to the functions called by the selected function. We see that the neural network is what is taking the most time, and not the Slater determinant, as the `model!`, `gradient!` and `jacobian!` functions are taking up most of the horizontal space. This tells us that we should focus on optimizing the code for the neural network first. `@profview_allocs` gives us a similar figure, only that it tracks allocations made by the functions. These tools are invaluable when tracking down which parts of your code are slowing it down, and where to spend your efforts optimizing it. And they can all be used from both a terminal and a Jupyter notebook.

Chapter 10

Hartree-Fock Implementation

The use of Hartree-Fock in our work has four main parts:

- Setting up the basis vector integrals for the harmonic oscillator basis
- Finding the Hartree-Fock orbitals using the computed integrals
- Setting up the basis vector integrals for the Hartree-Fock orbitals for use in coupled cluster
- Evaluating the Hartree-Fock determinant very efficiently in the VMC calculation

This chapter covers how we chose to solve these four problems.

10.1 Basis Vector Integrals

The one-body integrals $\hat{h}_{ai} = \langle\psi_a|\,\hat{h}\,|\psi_i\rangle$ are given analytically, since we are working with the harmonic oscillator basis in a harmonic potential. The off-diagonal elements are 0 since the basis is orthogonal, and the diagonal elements are the one-body energies for the system. These energies are known analytically as $E_n = (n + \frac{1}{2})\omega$, in atomic units. The matrix form of $\hat{h}$ is then found with the code

```
function onebody(ho::HOBasis, grid)
    (; l, ω) = ho
    return la.Diagonal([(n + 0.5) * ω for n in 0:l-1])
end
```

Computing the two-body integrals $\langle ab||ij\rangle$ however, is often the bottleneck when using Hartree-Fock or similar methods. If you only work with a basis of L spatial functions there L^4 of them, and if you spin-double the basis, as we do, there are $(2L)^4$ of them. These quickly becomes very expensive to both compute and store, as if you want high accuracy or to study a large system, you will need many basis functions. Furthermore, when working in 2 or 3 dimensions, the integrals become so expensive to compute that you must either use a very small basis, or use basis functions that offer analytical shortcuts when evaluating the integrals, such as Gaussian orbitals or DVR functions.

We choose to stick to 1D, and compute the integrals with no analytical shortcuts. The final form of the integrals include functions with spin, and anti-symmetrization, but we begin by only computing the integrals between the spatial functions.

$$\int\int dx_1 dx_2 \psi_a^*(x_1)\psi_b^*(x_2)\frac{1}{r_{12}}\psi_i(x_1)\psi_j(x_2). \tag{10.1}$$

This integral has no analytical solution, so we will need to compute it numerically over a 1D grid. The most straightforward approach would be a double loop over some limited range of x_1 and x_2, and to simply evaluate the functions $\psi_n(x)$ and $\frac{1}{r_{12}}$ at those points. With the function evaluations, we could use the trapezoidal method to approximate the integral. This would however lead to many unneeded repeated evaluations, as we are evaluating this integral L^4 times for different choices of basis functions.

The efficient way to compute these integrals is to first evaluate all the functions $\psi_n(x)$ over a 1D grid, and storing the values. Evaluating the harmonic oscillator functions is a relatively cheap part of this entire operation, as we only need to evaluate them once over a small finite 1D grid. We still have an optimized implementation for doing this however, which is covered in section 10.4.1. This is mostly for the VMC calculation, where we need to evaluate the functions many more times.

Next, we compute and store the value of

$$\text{inner}(b, j, x_1) = \int dx_2 \psi_b^*(x_2)\frac{1}{r_{12}}\psi_j(x_2), \tag{10.2}$$

for all values of b, j and x_1. Looping over x_1 first allows reuse of the values of $\frac{1}{r_{12}}$ when looping over b and j. We call these the inner integrals, and they can be reused greatly when looping over all the values of a and i. Notice also how swapping b and j in the inner integrals is the same as taking the complex conjugate, we use this to reduce the number of inner integrals we need to compute. We also make use of pre-allocation and multi-threading to speed up the calculation. The code below computes the inner integrals given the spatial functions in the vector of vectors `spfs`

```
function inner_ints(spfs, grid, V::Interaction)
    l = length(spfs)
    inner_int = zeros(typeof(spfs[1][1]), l, l, length(spfs[1]))
    interactions = [similar(grid) for i in 1:Threads.nthreads()]
    fs = [similar(spfs[1]) for i in 1:Threads.nthreads()]

    @inbounds Threads.@threads for xi in eachindex(grid)
        x1 = grid[xi]
        f_vals = fs[Threads.threadid()] # Pre-allocated vector for this thread
        interaction = interactions[Threads.threadid()] # ^^^

        interaction = interaction_over_grid!(interaction, x1, grid, V)
        for b in 1:l
            for j in b:l
                f_vals .= conj.(spfs[b]) .* interaction .* spfs[j]
                res = trapz(f_vals, grid)
                inner_int[b, j, xi] = res
                inner_int[j, b, xi] = res'
            end
        end
    end
    return inner_int
end
```

The complete two-body integral can then be computed as

$$\int dx_1 \psi_a^*(x_1)\psi_i(x_1)\text{inner}(b, j, x_1), \tag{10.3}$$

which allows for a massive speedup, as each inner integral can be reused L^2 times. Here we also use pre-allocation and multi-threading:

```
function outer_int(spfs, grid, inner_ints)
    l = length(spfs)
    outer_int = zeros(typeof(spfs[1][1]), l, l, l, l)

    fs = [similar(spfs[1]) for i in 1:Threads.nthreads()]
    is = [similar(spfs[1]) for i in 1:Threads.nthreads()]

    @inbounds Threads.@threads for b in 1:l
        f_vals = fs[Threads.threadid()]
        inner = is[Threads.threadid()]

        for j in 1:l
            @views inner .= inner_ints[b, j, :]
            for a in 1:l
                for i in 1:l
                    f_vals .= conj.(spfs[a]) .* inner .* spfs[i]
                    outer_int[a, b, i, j] = trapz(f_vals, grid)
                end
            end
        end
    end
    return outer_int
end
```

After these integrals between the spatial functions have been computed, we get the integrals between the orbitals with spin by noting that the integrals are non-zero only when the spins of the orbitals being integrated line up. When the integrals are non-zero (1/4 of the time), they are the same as the integrals we just computed between the spatial function. We have implemented code which figures out which elements are non-zero, and that inserts the corresponding two-body integrals between spatial function.

This approach to computing the two-body integrals was adapted to Julia from the python library quantum-systems developed by Øyvind Sigmundson Schøyen. The optimizations of pre-allocation, threading, and looping over x_i first when computing the inner integrals were my own. The last of these optimizations was later added to the quantum-systems library by me, giving a significant speedup when setting up the anti-symmetrized two-body integrals of a spin-doubled basis. The calculation went from taking 2.5 to 1.4 seconds with $L = 10$, and 3.9 to 2.6 seconds with $L = 15$ basis functions, making the library more user friendly when working interactively. Though our Julia implementation takes only 0.02 seconds and 0.05 seconds on the same calculations, as is shown in section 14.1.

10.2 Convergence Acceleration

When starting the self-consistent field procedure, we have an initial guess for $\mathbf{C}_0$, which gives an initial guess $\mathbf{F}_0$. The eigenvectors of $\mathbf{F}_0$ forms the columns of the new $\mathbf{C}_1$. And the new $\mathbf{C}_1$ can then form a new $\mathbf{F}_1$ using equation 3.15, or equation 3.20 for a restricted Hartree-Fock calculation. This can be repeated until $\mathbf{C}$ and $\mathbf{F}$ don't change, at which point we have found the optimal orbitals.

As mentioned in section 3.3, this procedure might not converge for some systems. In these cases, the use of a convergence acceleration method might allow for convergence. And in the cases where the SCF procedure does converge, a convergence acceleration method often makes it converge faster.

We implement two convergence acceleration methods, the alpha filter, and the direct inversion of the iterative subspace (DIIS). The alpha filter is quite simple: Let $\mathbf{F}$ be the Fock matrix at a given iteration. We call the next Fock matrix given by the method described above (from the Fock-matrix equation) $\mathbf{F}_{new}$. The difference between these, we call $\Delta\mathbf{F} = \mathbf{F}_{new} - \mathbf{F}$. The alpha filter method has a parameter $\alpha \in [0, 1]$ which tells us how much of the new solution to use, and how much of the old solution to use. With a given α, we update the Fock-matrix for the next iteration as

$$\mathbf{F} \to \mathbf{F} + \alpha\Delta\mathbf{F}. \tag{10.4}$$

DIIS is explained in section 11.3.2. When using it to help the convergence of Hartree-Fock, $\mathbf{F}$ takes the place of t, and $\Delta\mathbf{F}$ takes the place of Δt. The error vector used for DIIS is given by $\mathbf{FD} - \mathbf{DF}$ [22].

10.3 Transforming integrals to a new basis

After the Hartree-Fock method has given us the optimal orbitals $\phi_i(\mathbf{x})$, we need to transform our one- and two-body integrals to be expressed in terms of these orbitals. These transformed orbitals are used in the coupled cluster calculations.

The one-body integrals in the new basis can be found using the old one-body integrals $\hat{h}$ and the coefficients $\mathbf{C}$ that define the new basis. The transformed one-body integrals can be written as

$$\begin{aligned} \tilde{h}_{ab} &= \langle\phi_a|\,\hat{h}\,|\phi_b\rangle = \sum_{ij} C^*_{ia}C_{jb}\,\langle\psi_i|\,\hat{h}\,|\psi_j\rangle\,. \\ \Rightarrow \tilde{h} &= C^\dagger\hat{h}C. \end{aligned} \tag{10.5}$$

The transformed two-body integrals u2[a, b, c, d] = $\langle ab\|cd\rangle$ between the Hartree-Fock orbitals $\phi_a, \phi_b, \ldots$ can similarly be written using $\mathbf{C}$ and the old two-body integrals u[i, j, k, m] = $\langle ij \| km \rangle$ between the original orbitals. Expressed in code, the transformation looks like

```
function transform_twobody_slow(u, C)
    u2 = zero(u)
    l = length(C[:,1])

    for a in 1:l, b in 1:l, c in 1:l, d in 1:l
        for i in 1:l, j in 1:l, k in 1:l, m in 1:l
            @inbounds u2[a, b, c, d] += C[i, a] * C[j, b]
                                       * C[k, c] * C[m, d] * u[i, j, k, m]
        end
    end
    return u2
end
```

This octuple loop is too slow to run for any serious number of orbitals. We can avoid it by transforming one orbital at a time. This process can be written as a tensor product, for which there are specialized packages, like the TensorOperations package in Julia. Using this package, the transformation is given by the code

```
function transform_twobody(u, C)
    @tensor begin
        u2[a, b, c, i] := u[a, b, c, d] * C[d, i]
        u3[a, b, i, d] := u2[a, b, c, d] * C[c, i]
        u2[a, i, c, d] = u3[a, b, c, d] * C[b, i]
        u3[i, b, c, d] = u2[a, b, c, d] * C[a, i]
    end
    return u3
end
```

The octuple loop implementation takes 68 seconds with $L = 20$ functions. The TensorOperations implementation takes only 0.003 seconds. The TensorOperations implementation also scales very well, taking 8 only seconds with $L = 100$ functions.

10.4 Efficient Slater Determinant Evaluation

10.4.1 Recursive HO-Basis Evaluation

Recall that we need to evaluate products of determinants that look like this:

$$|\Phi_0\rangle = \frac{1}{\sqrt{n!}} \begin{vmatrix} \phi_1(x_1) & \dots & \phi_{\frac{n}{2}}(x_1) \\ \vdots & \vdots & \vdots \\ \phi_1(x_{\frac{n}{2}}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}}) \end{vmatrix} \cdot \begin{vmatrix} \phi_1(x_{\frac{n}{2}+1}) & \dots & \phi_{\frac{n}{2}}(x_{\frac{n}{2}+1}) \\ \vdots & \vdots & \vdots \\ \phi_1(x_n) & \dots & \phi_{\frac{n}{2}}(x_n) \end{vmatrix}. \tag{10.6}$$

We only move one particle between every VMC step, so all the values in the two matrices above are saved. When we a particle with position x_i is moved, we then only need to update the corresponding row in the corresponding matrix, and recompute the determinant. This section we will discuss how we go about computing the values that go into this changed row.

When we move the electron with position x_i we need to reevaluate the functions $\phi_1(x_i), \phi_2(x_i), \dots, \phi_{\frac{n}{2}}(x_i)$. These are given by linear combinations of Harmonic Oscillator basis functions

$$\phi_j(x_i) = \sum_n^L C_{nj}\psi_n(x_i), \tag{10.7}$$

where the Harmonic Oscillator basis functions are given by

$$\psi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{\omega}{\pi}\right)^{\frac{1}{4}} H_n(\xi) e^{-\xi^2/2}, \tag{10.8}$$

where $\xi = \sqrt{\omega}x$ and atomic units are used. $H_n(\xi)$ are the physicists Hermite polynomials, which are computed recursively to improve speed and accuracy. The polynomials quickly get coefficients which are too large and unpractical to work with. The polynomials are computed using the relations

$$\begin{aligned} H_0(x) &= 1. \\ H_1(x) &= 2x. \\ H_{n+1}(x) &= 2xH_n(x) - 2nH_{n-1}(x). \end{aligned} \tag{10.9}$$

The above definitions suggest an efficient strategy for evaluating the functions $\phi_j(x_i)$: First, evaluate all the HO basis functions using the recursive nature of the Hermite polynomials, then reuse all thsese for the linear combinations building up the functions $\phi_j(x_i)$.

This recursive approach gives a significant speedup over evaluating the functions separately, as will be shown in the results section. But it also means that we don't need to implement functions for evaluating all the different Hermite polynomials and derivatives of Hermite polynomials we will need. This leads to shorter and more elegant code.

10.4.2 Recursive HO-Derivatives

If you look back at equations 6.7 and 6.8, you see that we also need to compute the first and second derivatives of the functions $\phi_j(x_i)$. We will use the same strategy as before, evaluating the first and second derivatives of the entire HO-basis before combining them in different ways to form the derivatives we then insert into the given row of our slater determinant. All we need are the expression for the HO-basis derivatives. Using the rule that $H_n'(x) = 2nH_{n-1}(x)$ and the product rule, we can reach the following expressions

$$\psi_n'(x) = \frac{1}{\sqrt{2^n n!}}\left(\frac{\omega}{\pi}\right)^{\frac{1}{4}}(\sqrt{\omega}2nH_{n-1}(\xi) - \omega x H_n(\xi))e^{-\xi^2/2}.$$
$$\psi_n''(x) = \frac{1}{\sqrt{2^n n!}}\left(\frac{\omega}{\pi}\right)^{\frac{1}{4}}\omega((\omega x^2 - 1)H_n(\xi) - 4nx\sqrt{\omega}H_{n-1}(\xi) + 4n(n-1)H_{n-2}(\xi))e^{-\xi^2/2}. \tag{10.10}$$

These expressions, as well as the expressions for ψ_n, share many factors and they all use the Hermite polynomials which are computed recursively as they are needed. We can therefore make our calculation much more efficient by computing the basis functions and the first and second derivatives at the same time, saving many repeat calculations. The function we wrote to achieve this is shown below

```
function fast_ho_all!(x, ho::HOBasis)
    (; ω, hermites, hos, ho_der, ho_dder) = ho

    ξ = √ω * x

    #hermites[1] = 1.0
    ho_fac = (ω / π)^0.25 * exp(-ξ^2 / 2)

    hos[1] =     ho_fac * hermites[1]
    ho_der[1] = -ω * x * hos[1]
    ho_dder[1] = ω * (ω * x^2 - 1) * hos[1]

    hermites[2] = 2ξ
    ho_fac *= 1 / √2

    hos[2] =     ho_fac * hermites[2]
    ho_der[2] =  ho_fac * (√ω * 2 * hermites[1] - ω * x * hermites[2])
    ho_dder[2] = ho_fac * ω * ((ω * x^2 - 1)
            * hermites[2] - √ω * x * 4 * hermites[1])

    @inbounds for n in 2:length(hos)-1
        hermites[n+1] = 2ξ * hermites[n] - 2(n-1) * hermites[n-1]
        ho_fac *= 1 / sqrt( 2n )

        hos[n+1] = ho_fac * hermites[n+1]
        ho_der[n+1] =  ho_fac * (√ω * 2 * n * hermites[n] - ω * x * hermites[n+1])
        ho_dder[n+1] = ho_fac * ω * ((ω * x^2 - 1) * hermites[n+1]
                - √ω * x * 4n * hermites[n] + 4 * (n-1)*n * hermites[n - 1])
    end

    return hos, ho_der, ho_dder
end
```

10.4.3 Fast evaluation of determinants after row-change

The second derivative wrt. all n positions x_i are used when computing the kinetic energy. We also need first derivatives for the old and new quantum force used for importance sampling. Finally, we need to evaluate the new amplitude when moving a particle. In total, we need to evaluate $n+3$ determinants for every step in the variational Monte-Carlo calculation, and even though we have split each determinant into two smaller determinants, this is still quite an expensive operation.

Crucially however, all of these determinants only differ in a single row from one we will have already evaluated. When we move a single particle, we only change the row of the moved particle. When we take a derivative wrt. a particle position, we only change the corresponding row. We can therefore use some results from linear algebra to speed up the computations significantly.

The determinant of a matrix A can be written as

$$|A| = \sum_i a_{ij}c_{ij} = \sum_j a_{ij}c_{ij}, \tag{10.11}$$

where a_{ij} are the elements of A, and c_{ij} are the elements of C, the comatrix of A. The index that is not summed over can be chosen arbitrarily. The comatrix can be expressed using A-s inverse, A^{-1}. Using its elements a_{ij}^{-1}, we write

$$c_{ij} = a_{ji}^{-1}|A|. \tag{10.12}$$

Note that $\sum_j a_{ij}a_{ji}^{-1} = 1$, as a matrix times its inverse gives the identity, and this sum returns one of the diagonal elements of that matrix product (at index i, i).

We now introduce a matrix B which differs from A in only row i. Row i in the comatrix of A is independent of row i of A. So if we sum over columns j of A and B, with row i locked, A and B share the same comatrix values. We can therefore write the ratio of their determinants as

$$R = \frac{|B|}{|A|} = \frac{\sum_j b_{ij}c_{ij}}{\sum_j a_{ij}c_{ij}} = \frac{\sum_j b_{ij}a_{ji}^{-1}|A|}{\sum_j a_{ij}a_{ji}^{-1}|A|} = \sum_j b_{ij}a_{ji}^{-1}. \tag{10.13}$$

This expression lets us quickly compute the ratio between our old determinant and ones where a single row has been changed, exactly what we needed.

This is all great, but if we want to update our old determinant we will need to compute a new inverse, yet another expensive operation we will need to circumvent.

Column i in B^{-1} can be found using row i of the comatrix of A, giving

$$b_{ki}^{-1} = c_{ik}\frac{1}{|B|} = a_{ki}^{-1}\frac{1}{R}. \tag{10.14}$$

For all other columns k, the new inverse elements are given by

$$b_{kj}^{-1} = a_{kj}^{-1} - \frac{S_j}{R}a_{ki}^{-1}, \tag{10.15}$$

where S_j is given by

$$S_j = \sum_{l=1}^{N} b_{il}a_{lj}^{-1}. \tag{10.16}$$

With these final expression we are able to quickly compute the ratio of new/old determinants when we change a single row, and we can quickly update the inverse of the old matrix when we want to go forward with the changed row.

To start the procedure, the inverse and determinant of the starting matrix needs to be computed, but this only happens once. If we want the new determinant, we can simply use the old determinant and the ratio to find it, though we will always just use the ratio directly. The struct holding the information about the current matrix (called D in the code) and its default constructor is shown below

```
mutable struct Fast_Det
    const D::Matrix{Float64}
    const D_inv::Matrix{Float64}
    const n::Int64
    det::Float64


    function Fast_Det(D)
        D_inv = la.inv(D)
        det = la.det(D)
        n = size(D)[1]
        return new(copy(D), D_inv, n, det)
    end
end
```

With the inverse matrix saved, we can easily compute the ratio of the new/old determinant using the inner product computed in the following function

```
function ratio_new_old_det(fd::Fast_Det, new_row::Vector{Float64}, i)
    (; D_inv, n) = fd
    R = 0.0
    for j in 1:n
        R += new_row[j] * D_inv[j, i]
    end
    return R
end
```

This is much faster than evaluating a whole new determinant. The spin-up determinant and the spin-down determinant have separate Fast_Det structs. We only need to use one at a time, as we only compute ratios, and when the relevant particle position doesn’t appear in one of the matrices, that matrix doesn’t change the ratio of new/old.

Chapter 11

Coupled Cluster

11.1 Wick's Theorem Implementation

In equation 4.42 we showed all the full contractions from the expression

$$\langle \Phi_i^a | (\hat{V}_N \hat{T}_1^2) | \Phi_0 \rangle = \frac{1}{8} \sum_{pqrs} \sum_{ld} \sum_{kc} \langle pq|rs \rangle t_l^d t_k^c \langle \Phi_0 | \{a_i^\dagger a_a\} (\{a_p^\dagger a_q^\dagger a_s a_r\} \{a_d^\dagger a_l\} \{a_c^\dagger a_k\})_c | \Phi_0 \rangle, \tag{11.1}$$

coming from the CCSD $\hat{T}_1$ amplitude equations. We quickly realized finding them by hand would be very difficult. There are many of them, and seemingly no straightforward way to find them all. We therefore opted for a programmatic approach which we will outline in this section.

We know from the discussion of normal ordering, that the only single contractions that give non-zero results are $a_i^\dagger a_j = \delta_{ij}$ where i and j are hole states, and $a_a a_b^\dagger = \delta_{ab}$ where a and b are particle states. Wick's theorem also tells us that contractions only need to be evaluated between normal ordered strings, and not within them. These two results lets us greatly limit the total number of "valid" single contractions.

We represent each operator as being particle or hole and dagger or no dagger, and giving it the index of the string of operators it belongs to. For instance $a_d^\dagger$ is a hole operator, with a dagger and index 3. Strings and dictionaries are employed to represent this in code. By then looping over all possible pairs of operators we can test whether they obey the rules stated above with a couple of if-statements. Of 45 possible, we are left with 16 valid single contractions.

Next we loop over all the combinations of 5 valid single contractions, see which ones have no repeat operators, and that have contractions between $\hat{V}$ and both $\hat{T}_1$-s. Sets are used to check for repeat operators, and a regular expression is used to check for the required single contractions. Of 4368 combinations, 16 are valid full contractions.

Finally we turn these full contractions, along with their corresponding Kronecker delta products into Latex code using string interpolation, writing the resulting Latex code to file. The simple and elegant syntax of the simplerwick Latex package was of great help. The signs of the Kronecker delta products were corrected by hand by simply counting the number of crossings in the fully contracted terms. The remaining operations of turning Kronecker delta products into sums of matrix elements, and then simplifying these sums was done by hand since it took little time.

11.2 Amplitude Equation Evaluation

There are many ways to evaluate the right hand sides of the amplitude equations, but we have chosen a simple one, to save time developing the code.

The right hand side of the $\hat{T}_1$ amplitude equation (equation 4.46) must be evaluated for every amplitude t_i^a, that is, $n(l-n)$ times. The equation includes a quadruple loop with $n^2(l-n)^2$ evaluations. The computational complexity of evaluating the $\hat{T}_1$ amplitude equation is thus $O(n^3 l^3)$.

The right hand side of the $\hat{T}_2$ amplitude equation (equation 1) must be evaluated for every amplitude t_{ij}^{ab}, where $a < b$ and $i < j$. The amplitudes where $b < a$ and $j < i$ can be found by interchanging a with b and i with j, remembering that each interchange flips the sign. Where $a = b$ and $i = j$, the amplitudes are always zero. This means that there are $n(n-1)/2 \cdot (l-n)(l-n-1)/2$ evaluations (A restricted double loop has first $n-1$ evaluations of the inner loop, then $n-2$, and so on. $(n(n-1))/2$ in total). The equation includes a quadruple loop with $n^2(l-n)^2$ evaluations. The computational complexity of evaluating the $\hat{T}_2$ amplitude equation is thus $O(n^4 l^4)$.

To speed up evaluation, parallelization is utilized. Since we are only looking at systems with $n = 2$, we parallelize over the $(l-n)$ a values, for both equations, to make use of more CPU threads.

The computational complexity of our CCSD implementation is $O(n^4 l^4)$. However, factoring the large sums in the $\hat{T}_2$ amplitude equation into smaller terms, and storing intermediate values can lead to a complexity of only $O(l^6)$ [6], which is often much faster. This does come at the cost of writing more values to memory, and more time writing the actual code. We opted to implement the equations as written, since more speed was not needed for the small systems we are interested in.

11.3 Convergence Acceleration

The iterative scheme to solve the amplitude equations shown in section 4.5.3 is not guaranteed to converge quickly, if at all. In this section we present two schemes that aim to speed up or even allow convergence of the iterative scheme.

From equation 4.52 and 4.56 we have initial amplitudes we together now call t, and from equations 4.51 and 4.55 we have the crude updates to t which we together now call Δt. (Whenever we "combine" tensors of numbers like this in this section, we first unravel them into vectors, and then concatenate them in the corresponding code. After we have computed the new t, we reshape everything back how it was.) The simplest iterative scheme which we will improve upon is this:

$$t \to t + \Delta t. \tag{11.2}$$

11.3.1 Alpha filter

If the iterative scheme does not converge when updating t by Δt, a natural alternative is to update it by a scaled down Δt, by "taking smaller steps". This is often an improvement upon the crude approach. Introducing a scaling parameter $\alpha \in [0, 1]$, we define a new iterative scheme using an Alpha filter

$$t \to t + \alpha \Delta t. \tag{11.3}$$

11.3.2 Direct inversion of the iterative subspace

A more sophisticated approach is to look at the m previous t-s and Δt-s, and see which combination of them best fulfills the amplitude equations. Recall that the amplitude equations are

$$\begin{aligned} 0 &= \langle \Phi_i^a | \bar{H} | \Phi_0 \rangle . \\ 0 &= \left\langle \Phi_{ij}^{ab} \right| \bar{H} | \Phi_0 \rangle . \end{aligned} \tag{11.4}$$

and that we defined

$$\begin{aligned} \Omega_i^a &= \langle \Phi_i^a | \bar{H} | \Phi_0 \rangle . \\ \Omega_{ij}^{ab} &= \left\langle \Phi_{ij}^{ab} \right| \bar{H} | \Phi_0 \rangle . \end{aligned} \tag{11.5}$$

Ω_i^a and Ω_{ij}^{ab} are clearly good measures of how well the t-s satisfy the amplitude equations, when they are all zero, the equations are satisfied. We will therefore use them as a direct measure of the error. We will from now on call the combination of these errors Ω.

We want to find the coefficients c_i such that

$$t_{new} = \sum_i^m c_i (t_i + \Delta t_i), \tag{11.6}$$

is the combination of t-s and Δt-s that best satisfies the amplitude equations. Combining the m previous errors Ω_i in the same way gives us the error vector.

$$\Omega_{new} = \sum_i^m c_i \Omega_i. \tag{11.7}$$

We want to minimize the norm of the error vector under the constraint of the c-s adding to 1, so that we get a useful solution, and not the trivial $c_i = 0$ solution. We introduce a Lagrange multiplier λ to satisfy this demand. We now minimize

$$\begin{aligned} L &= ||\Omega_{new}||^2 - 2\lambda\left(\sum_i c_i - 1\right), \\ &= \sum_{ij} c_j B_{ij} c_i - 2\lambda\left(\sum_i c_i - 1\right), \end{aligned} \tag{11.8}$$

where $B_{ij} = \langle \Omega_j, \Omega_i \rangle$.

Taking the derivative wrt. c_k and equating to 0 gives

$$\begin{aligned} \frac{\partial L}{\partial c_k} &= 0, \\ \sum_j B_{kj} c_j + \sum_i B_{ik} c_i - 2\lambda &= 0, \\ 2\sum_j B_{kj} c_j - 2\lambda &= 0, \\ \sum_j B_{kj} c_j - \lambda &= 0. \end{aligned} \tag{11.9}$$

The derivatives of the c_k-s and λ form a system of $(m+1)$ linear equations which we can solve to find the c_k-s.

$$\begin{bmatrix} B_{11} & B_{12} & B_{13} & \dots & B_{1m} & -1 \\ B_{21} & B_{22} & B_{23} & \dots & B_{2m} & -1 \\ B_{31} & B_{32} & B_{33} & \dots & B_{3m} & -1 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & B_{m3} & \dots & B_{mm} & -1 \\ 1 & 1 & 1 & \dots & 1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_m \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}. \tag{11.10}$$

Each iteration of solving the Coupled-Cluster amplitude equations using direct inversion of the iterative subspace(DIIS) can be summarized as

1. Compute Ω and Δt
2. Store the current t, Ω and Δt

3. Collect the last m t-s, Ω-s and Δt-s
4. Set up the matrix of equations
5. Solve the equations to get the coefficients c_k
6. Compute your new t using equation 11.6

Chapter 12

Feedforward Neural Networks

Our neural network wave function code needs to support four main functions:

- Evaluation of the network
- The derivative of the network wrt. its parameters (backpropogation)
- The derivative of the network wrt. its input (quantum force)
- The double derivative of the network wrt. its input (kinetic energy)

To write well optimized code for all of these functions, we have chosen to implement our own neural network code from the ground up. The following sections describe the design of this code, and the unique approaches used to solve each of the four problems. The final section is a note on why other machine learning libraries were not used instead.

12.1 Fast evaluation

The evaluation of a neural network is very straightforward. As shown in equation 7.2, it is a series of matrix-vector multiplication, vector-vector addition, and element-wise evaluation of activation functions.

All of these operations have fast, in-place implementations in Julia. To make use of this, we preallocate all the memory required when the network is created. Every layer in the network has a vector where it will put its output, and every layer has a reference to the previous layer's output. This means that for the millions of times the network is evaluated, no memory needs to be allocated and garbage-collected, resulting in a speed-up.

The function for evaluating the output of a linear layer is as follows:

```
function in_place_eval!(layer::Dense)
    (; W, b, input, output) = layer

    mul!(output, W, input)
    output .+= b
end
```

The first line in this function extracts the weight matrix and the bias-, input- and output-vectors of the layer for later use.

The function `mul!()` computes the matrix-vector product of **W** and **input** and places the result in **output** during the calculation. The special Julia syntax `.+=` adds each element of **b** to the corresponding element of **output**, again without any intermediary memory allocation.

Compare this with the similar function `simple_eval()`.

```
function simple_eval(input, layer::Dense)
    (; W, b) = layer

    return W * input + b
end
```

This function takes the input to the layer as an input, and performs the matrix-vector multiplication and vector-vector addition with no preallocated output vector used. A benchmark of these two functions is shown in table 12.1

	Time	Allocations
in-place 10x10	117.737 ns	0 allocations: 0 bytes
simple 10x10	194.929 ns	2 allocations: 288 bytes
in-place 100x100	58.800 µs	0 allocations: 0 bytes
simple 100x100	59.500 µs	2 allocations: 1.75 KiB

Table 12.1: In-place and simple layer evaluations benchmarked. The size of the weight matrix is shown for each benchmark. We see that the in-place operation allocates less memory, and is faster, especially for small matrices.

We see that in-place evaluation is faster and allocates no memory. This effect will be amplified as we make heavy use of many small weight matrices. Since we will be evaluating the network very many times during the VMC process, these small allocations very quickly add up to a lot of memory, requiring the program to stop so that the allocated, but now unused memory can be garbage collected.

The activation functions are treated as their own layers and perform similar in-place operations using their input and output vectors. To illustrate, here is the in-place evaluation of the tanh activation function

```
function in_place_eval!(layer::Tanh)
    (; input, output) = layer
    output .= tanh.(input)
end
```

We also have functions for evaluating activation functions without the use of in-place operations. We will come back to the use of these in section 12.3.

12.2 Back-propagation

Section 7.2.2 described the back-propagation algorithm. It consists of moving from the output of the network, all the way to the input, recursively computing partial derivatives of the network output along the way. An important detail is that it requires the outputs of all the layers in the network, so we need to evaluate it before we use back-propagation.

The central operation performed at each layer is given by

$$\frac{\partial C}{\partial \mathbf{z}^{(l)}} = \frac{\partial \sigma_i(\mathbf{z}^{(l)})}{\partial \mathbf{z}^{(l)}} (\mathbf{W}^{(l+1)})^T \frac{\partial C}{\partial \mathbf{z}^{(l+1)}}. \tag{12.1}$$

Since we have implemented activation functions as being their own layers, we split this process into two parts:

$$\begin{aligned} \frac{\partial C}{\partial \mathbf{a}^{(l)}} &= (\mathbf{W}^{(l+1)})^T \frac{\partial C}{\partial \mathbf{z}^{(l+1)}}, \\ \frac{\partial C}{\partial \mathbf{z}^{(l)}} &= \frac{\partial \sigma_i(\mathbf{z}^{(l)})}{\partial \mathbf{z}^{(l)}} \frac{\partial C}{\partial \mathbf{a}^{(l)}}. \end{aligned} \tag{12.2}$$

The first equation describes how to update the recursive gradient for linear layers, and the second for activation layers. As an example, a tanh activation layer will propagate gradients using the function

```
function backprop!(layer::Tanh, delta)
    (; input) = layer
    delta .*= sech.(input).^2
end
```

Note that the Tanh layer saved the input it got during the evaluation of the network in order to use it for the gradient. The linear layers use equations 7.12 and 7.15 for updating their own parameter gradients. A linear layer propagates gradients, and updates their parameter gradient, with the function

```
function backprop!(layer::Dense, delta_in)
    (; W, W_g, b_g, input, delta) = layer

    #la.mul!(W_g, delta_in, transpose(input))
    vec_outer!(W_g, delta_in, input)
    b_g .= delta_in # Bias gradient

    la.mul!(delta, transpose(W), delta_in) # Propagating gradient
    return delta
end
```

Julia uses BLAS to perform many linear algebra operations, such as outer products of vectors. In our case however, we found that implementing our own outer product in pure Julia specialized for vectors made the computation much faster. We believe that this is because we mostly deal with relatively small vectors, and because all threads are busy, making the overhead of BLAS not worth it. Our outer product implementation is simply

```
function vec_outer!(M::Matrix{Float64}, u, vT)
    @inbounds for (i, ui) in enumerate(u)
        for (j, vi) in enumerate(vT)
            M[i, j] = ui * vi
        end
    end
    return M
end
```

This optimization can lead to a speedup of almost 2 times for small networks, and a significant speedup for even large networks. The activation layers propagate the gradient in similar ways as the dense layers. The function `gradient!()` can update the parameter gradients of the whole network as follows:

```
function gradient!(nn::NeuralNetwork)
    # Computes the derivative of the network output wrt. the parameters
    delta = nn.delta_start # The delta vector of the last layer
    delta .= 1 # Derivative of network wrt. last layer output is 1

    for layer in reverse(nn.layers)
        delta = backprop!(layer, delta)
    end

    return nn
end
```

This implementation only uses pre-allocated memory. Note that we compute the derivative of our network parameters using analytical formulas, in contrast to most popular machine learning libraries, which use automatic differentiation.

Python packages such as Pytorch and TensorFlow, or Julia packages such as Flux, use automatic differentiation, also known as autodiff, to compute gradients. The use of autodiff is a necessity when working with complicated models such as the ones offered by these libraries. Autodiff gives reliable gradients with very little work required, at a low cost to speed.

When working with only a feedforward neutral network however, the derivatives are simple enough to quickly be implemented by hand. Not having to rely on autodiff makes the computation slightly faster, and it gives much more flexibility in how to store the parameters. This was the main reason we went this route. The autodiff packages ForwardDiff and ReverseDiff, for instance, requires the parameters to be stored in a single vector. While the autodiff library Zygote is flexible, it is often slow.

12.3 Kinetic energy - Forward over Reversediff

In section 7.3.3, we wrote shortly on how we plan to use automatic differentiation to take the double derivative of the neural network with respect to its inputs

$$\frac{\partial^2 \mathrm{NN}(\chi(\mathbf{x}))}{\partial x_i} = \frac{\partial^2 \mathrm{NN}(\mathbf{a}^{(0)})}{\partial \mathbf{a}_j^{(0)}}, \tag{12.3}$$

where j is the index that element i of $\mathbf{x}$ is moved when $\mathbf{x}$ is sorted.

To achieve this, we will use the forward mode automatic differantiation package ForwardDiff.jl, and the reverse mode automatic differentiation package ReverseDiff.jl.

The main function we use for evaluating the neural network is `model!(nn, x)`, but this function uses pre-allocated vectors to make the evaluation fast and cost no additional memory for each VMC iteration. This function will therefore not work with automatic differentiation, as the pre-allocated vectors don't support the special numerical types used to perform automatic differentiation.

The solution is to implement evaluation of the network without using pre-allocated outputs, so that any numerical types can be used. Examples of functions written for this purpose are shown below

```
function layerEval(x, layer::Dense)
    (; W, b) = layer
    return W * x + b
end

function layerEval(x, layer::Exp)
    return exp.(x)
end
```

While the original pre-allocating functions `layerEval!(x, layer)` (notice the ! due to them mutating the pre-allocated vectors) required x to be a vector of floats, these functions accept vectors of any numerical type. This is one of the strengths of Julia, if you call these functions with a vector of ForwardDiff's DualNumber type for instance, a version of the function specialized for this type is compiled, used and then saved for later reuse. This way Julia can be a compiled language, but functions can still support a large number of types without needing to code a function for every possible input type. Evaluating the whole network using these flexible layer evaluation functions can be done with the function.

```
function diffable_model(layers, x)
    for layer in layers
        x = layerEval(x, layer)
    end
    return x
end
```

This function is much slower than the in-place `model!(nn, x)`, but this is fine, as it will only be used for setting up the forward and reverse automatic differentiation.

At this point we could get the double derivatives using any of the approaches below

```
hessian = ForwardDiff.hessian(x -> diffable_model(layers, x)[1], x)

hessian = ReverseDiff.hessian(x -> diffable_model(layers, x)[1], x)

hessian = ReverseDiff.jacobian(
x -> ForwardDiff.gradient(x -> diffable_model(layers, x)[1], x), x
)

hessian = ForwardDiff.jacobian(
x -> ReverseDiff.gradient(x -> diffable_model(layers, x)[1], x), x
)
```

We will choose to use the fourth one, forward over reverse. But written this way the calculation would be painfully slow. These automatic differentiation libraries work best when the calculation is prepared.

Setting up these libraries is normally quite simple, but when combining them like this it takes more care. The reverse mode gradient needs to compile gradient tapes for both Float64 and ForwardDiffs DualNumber type, so a Dictionary holding the tapes needs to be used and setup. In addition to this, all results and intermediary results have pre-allocated memory prepared. All of this setup leads to a more complicated setup, but also much faster speeds. The code used for the forward over reverse automatic differentiation, and turning it into the kinetic energy is shown below

```
# In the constructor for the neural network
...
# Reverse mode setup
hes_grad_result = zeros(n)
hes_grad_tapes = Dict{DataType, Any}()
# Forward mode setup
hes_jac_result = zeros(n, n)
hes_jac_config = fd.JacobianConfig(nothing, hes_grad_result, hes_grad_result)
...

function hes_grad!(y, x::Array{T}, nn::NeuralNetwork) where {T<:Real}
    if !haskey(nn.hes_grad_tapes, T)
        # Creating a tape if it hasn't been created yet
        config = rd.GradientConfig(x)
        tape = rd.compile(
        rd.GradientTape(x -> diffable_model(nn.layers, x)[1], x, config)
        )
        nn.hes_grad_tapes[T] = tape
    end
    # Getting the correct gradient tape for the input given from jacobian!
    tape = nn.hes_grad_tapes[T]
    return rd.gradient!(y, tape, x) # This is where the gradient is taken
end

function kinetic(x::Vector{Float64}, nn::NeuralNetwork)
    # Taking the jacobian of the gradient gives the hessian
    fd.jacobian!(nn.hes_jac_result, (y, x) -> hes_grad!(y, x, nn),
    nn.hes_grad_result, x, nn.hes_jac_config
    )
    return -0.5 * la.tr(nn.hes_jac_result) / nn.output[1]
end
```

12.4 Smart Sorting and Un-sorting

We will here describe a method to efficiently sort the positions of the particles in order to make the neural network a symmetric function. This method also covers how to efficiently permute the derivatives of the neural network using the reverse operation of the original sorting to make the index of the derivatives correctly match up to the index of the particles.

When computing a derivative or double derivative of the neural network with respect to its inputs $\chi(\mathbf{x})$, we first sort $\mathbf{x}$ into $\mathbf{a}^{(0)}$, and then we get out the partial derivatives with respect to the elements of $\mathbf{a}^{(0)}$. To get the derivatives with respect to the elements of $\mathbf{x}$, we therefore need to do the reverse of the original sorting so that derivative number i corresponds to position number i. As an example, lets say $\mathbf{a}^{(0)} = (x_2, x_3, x_1)$, then

$$\frac{\partial^2 NN(\chi(\mathbf{x}))}{\partial^2 x_1} = \frac{\partial^2 NN(\mathbf{a}^{(0)})}{\partial^2 \mathbf{a}_3^{(0)}}. \tag{12.4}$$

We will need a mapping to go from the original index i to the sorted index j. This is achieved with the function `p = sortperm(x)` (called argsort in some other languages). In the above example, p[1] would give the index 3. We will also need a mapping to go in the reverse direction (unsorting in a way), this is achieved with the operation `p_rev = sortperm(p)`. In the above example p_rev[3] would give the index 1.

These sorting functions are good, and even have versions that only allocate 16 bytes of memory, regardless of input size, but we can write our own faster, non-allocating sorting algorithm for our specific needs. The following is a specialized implementation of insertion sort, tailored to our specific needs.

Each time we move a particle, we only change a single element of $\mathbf{x}$, and therefore we only need to change a single element of the sorted $\mathbf{a}^{(0)}$ from the previous VMC step. When we have changed an element in the previously sorted $\mathbf{a}^{(0)}$, we can check if the new value is larger or smaller than the old value, and then move it to the right or left until $\mathbf{a}^{(0)}$ is sorted again, since all other values are still sorted with respect to each other. Each time we need to move the new value, we update `p` and `p_rev` accordingly. To do this, we have implemented a struct MiniSort which holds the sorted positions, the original index to sorted index mapping, and the sorted index to original index mapping. When we move a particle with index new_idx to the positions new_pos, we update the sorted positions, as well as these mapping with the following function

```
function update_sort!(minisort, new_idx, new_pos)
    (; n, x_sort, p, p_rev) = minisort

    @inbounds sort_idx = p_rev[new_idx]
    @inbounds old_pos = x_sort[sort_idx]
    @inbounds x_sort[sort_idx] = new_pos

    if new_pos > old_pos
        if sort_idx != n
            r = sort_idx + 1
            @inbounds while new_pos > x_sort[r]
                x_sort[r], x_sort[r - 1] = x_sort[r - 1], x_sort[r]
                p[r], p[r - 1] = p[r - 1], p[r]
                p_rev[p[r]], p_rev[p[r - 1]] = p_rev[p[r - 1]] , p_rev[p[r]]

                r += 1
                if r == n + 1
                    break
                end
            end
        end
    elseif new_pos < old_pos
        if sort_idx != 1
            l = sort_idx - 1
            @inbounds while new_pos < x_sort[l]
                x_sort[l], x_sort[l + 1] = x_sort[l + 1], x_sort[l]
                p[l], p[l + 1] = p[l + 1], p[l]
                p_rev[p[l]], p_rev[p[l + 1]] = p_rev[p[l + 1]] , p_rev[p[l]]

                l -= 1
                if l == 0
                    break
                end
            end
        end
    end
    return minisort
end
```

Part III

Results

Chapter 13

Ground state energies

The three methods we have implemented, the Hartree-Fock method, the coupled cluster method, and the variational Monte-Carlo method, are all first and foremost methods for finding the ground state of a system. When judging the accuracy of these methods, the ground state energy is therefore the most important measure. In this thesis, we have limited ourselves to studying the 1D harmonic quantum dot, as we found few other studies of it, and because it made some aspects of the implementation easier, or more theoretically robust. While this system is not physically possible, unlike the 3D and (in some sense) 2D harmonic quantum dots which can be studied in experiments, it still acts as an accessible way of testing how well our methods solve the Schrödinger equation.

This chapter mostly covers results on a system of 2 electrons, as this is the system where we had access to exact results for both energies and one-body densities to compare with. We also give less focus to systems of more than 2 electrons because we found VMC calculations on such systems in 1D to be quite unstable using our approach.

13.1 2 Electrons

For the 1D harmonic quantum dot with two electrons, we have exact results from Zanghellini[28] for a harmonic potential with strength $\omega = 0.25$ and a Coulomb force with shielding $a = 0.25$. We will use these same parameters for all of our results. We will use the exact energy E_∞ from Zanghellini[28] as the goal, and include the relative error of the different methods, given by $\frac{E-E_\infty}{E_\infty}$, in our tables.

13.1.1 Comparing RHF with GHF

This system is strongly correlated, so we expect to see a large difference between the Hartree-Fock energy and the exact ground state energy. For the Hartree-Fock method, we used a basis of L harmonic oscillator functions. The one- and two-body integrals between the L basis functions were computed over a grid of 2001 points from $x = -10$ to $x = 10$. We then added spin to the basis functions so that we get L spin up and L spin down basis functions. We then used the restricted Hartree-Fock(RHF) method and the general Hartree-Fock method (GHF) (with the Alpha-mixer with $\alpha = 0.5$ to speed up convergence) to find the optimal basis. This basis is then used to construct the optimal single Slater determinant, and find the Hartree-Fock energy. The energies computed with these methods and $L = 10$ basis functions is shown in table 13.1, together with the energy of just using a determinant of harmonic oscillator functions.

Method	Energy	Error
HO Functions	1.3837	0.5590
RHF	1.1796	0.3549
GHF	0.8450	0.0203
Exact	0.8247	0.0000

Table 13.1: The ground state energies computed for the 1D harmonic quantum dots system with restricted and general Hartree-Fock. The exact value is for an infinite basis.

We see that only using the 2 lowest Harmonic Oscillator basis functions as the ground state solution gives a very high energy compared to the exact solution. This is to be expected, as the harmonic quantum dots system is a highly correlated system, far different to the non-interactive case. RHF also gives a poor estimate, which is also expected, as electron correlations are the thing we need Coupled Cluster and the neural network to properly account for.

GHF gives a surprisingly good energy. We know that the exact ground state has a single doubly occupied orbital, so one could expect that RHF would be able to find the best single-determinant wave fucntion. But under the constraint of only using a single determinant, GHF found that the lowest energy is instead reached using a determinant with two completely different occupied orbitals that have neither the same spatial part nor opposite spins, a determinant RHF could not find. The stark difference between the doubly occupied orbitals of the RHF method and the unphysical orbitals found by GHF, is shown in figures 13.1 and 13.2.

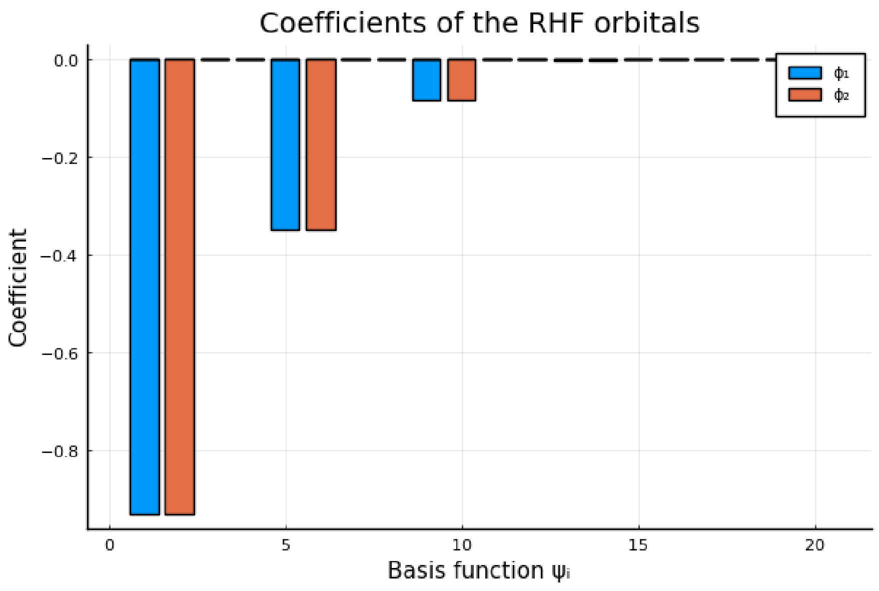


Figure 13.1: The coefficients the RHF method gives. We see that the orbitals are doubly occupied.

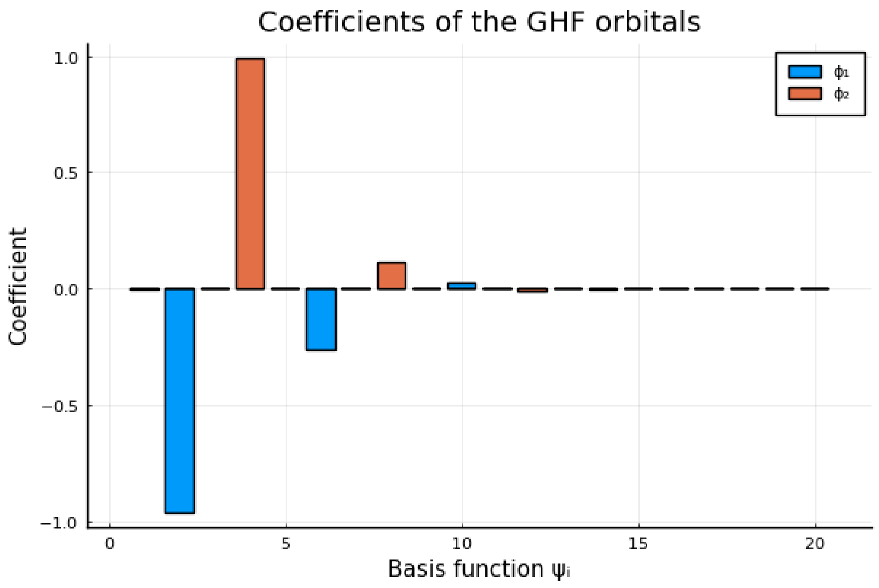


Figure 13.2: The coefficients the GHF method gives. We see that the orbitals are not doubly occupied.

These figure show that the GHF method clearly does not give a single doubly occupied orbital. And since virtually only the evenly numbered basis functions are included, both orbitals are mostly spin down, which should not be the case. The physical interpretation of the GHF solution is shaky at best. Since we are interested in approximating the true ground state of the system, we will not consider the GHF solution valid.

13.1.2 Comparing CCD with CCSD

We used the basis from both RHF and GHF in both coupled cluster doubles (CCD) and coupled cluster singles doubles (CCSD), using DIIS to speed up convergence. This let us find linear combinations of determinants that minimize the energy, within the limits of the methods. For two electrons, CCSD is exact within the space spanned by its basis, so it should come very close the exact energy E_∞.

Method	Energy	Error
HO Functions	1.3837	0.5590
RHF	1.1796	0.3549
GHF	0.8450	0.0203
CCD	1.0517	0.2270
RHF+CCD	0.8384	0.0137
GHF+CCD	0.8377	0.0130
CCSD	0.8253	0.0006
RHF+CCSD	0.8253	0.0006
GHF+CCSD	0.8374	0.0127
Exact	0.8247	0.0000

Table 13.2: The ground state energies computed for the 1D harmonic quantum dots system with several different methods. The exact value is for an infinite basis.

CCD on its own struggles. Double excitations are expected to be the most important when optimizing the energy due to Brillouin's theorem [24], but that is only when using a Hartree-Fock basis. With the help of the GHF and RHF basis however, we see that the Couple-Cluster method comes quite close to the exact ground state.

When only looking at two electrons, CCSD gives the exact solution within the space spanned by the basis function. It therefore does not require a basis optimized by Hartree-Fock in this case. The difference between the energy reached and the exact energy of the system is only due to the limited finite basis. Interestingly, it actually gives a higher energy when using the general Hartree-Fock basis. We have no other explanation for this than that the unphysical general Hartree-Fock basis adds some unexpected effects when using it for further calculations.

13.1.3 Changing the number of basis functions

One of the main limiting factors of the methods discussed so far is that they are limited by the size and applicability of the basis chosen, as the wave functions they find are limited to the space spanned by the basis. The only option for increasing the potential accuracy is improving the basis. The most straightforward way is to simply increase the size. This comes at increased computational cost, and severely diminishing returns, as the lowest energy basis functions are typically the most important ones in the ground state.

Table 13.3 shows that RHF and GHF practically don't make use of more than 10 basis functions in their optimized Slater determinant. We also saw this in figures 13.1 and 13.2. This is expected, as using more of one function means using less of another, so the best single particle functions win out. GHF converged much slower with increased L, as the matrices that are operated on are twice as large as those for RHF, and the SCF procedure seems to converge very slowly for large matrices.

CCD and CCSD show a small improvement with a larger basis however, as they are more limited by the space spanned by the functions, and less by having to only use the best single-particle functions. We see that we will quickly reach a limit where increasing the size of the basis is no longer a viable strategy for improving the energy.

Method	L	E[a.u.]	Rel. Error	Time[s]
RHF	10	1.1796	0.4303	0.01
RHF	20	1.1796	0.4303	0.01
RHF	40	1.1796	0.4303	0.06
GHF	10	0.845	0.0247	0.53
GHF	20	0.845	0.0247	10.95
GHF	40	0.845	0.0247	660.74
CCD	10	0.8384	0.0166	0.02
CCD	20	0.8376	0.0157	0.34
CCD	40	0.8374	0.0154	12.61
CCSD	10	0.82532	0.0008	0.14
CCSD	20	0.82498	0.0003	4.9
CCSD	40	0.82488	0.0002	142.96

Table 13.3: The ground state energies computed for the 1D harmonic quantum dots system with 2 electrons for different methods and different numbers L of harmonic oscillator basis functions. CCD and CCSD both use the basis given by RHF. The relative error is compared to the exact result from Zanghellinin [28].

13.1.4 Comparing with the Slater-NN solution

We have so far computed and discussed the ground state energies computed for some traditional many-body methods. We now turn our attention to the more novel approach of the Slater-NN wave function.

We tried many different network architectures, and found a single hidden layers of 32 nodes and $\tanh$ activation to give the lowest ground state energy. We also chose to use $L = 10$ basis functions for the RHF calculation that fed into the Slater-NN wave function. We trained it over 500 iterations of ADAM optimization. The gradient was computed each time with 10000 equilibration steps and 10000 sampled steps. Importance sampling with a step of 0.01 was used. The final energy was computed with $2^{27} \approx 1.34 \cdot 10^8$ samples. The error in the estimate of the energy was computed using the blocking method. Table 13.4 shows the ground state energy computed using the methods we have discussed so far. GHF is excluded we do not consider its solution valid, as discussed earlier.

Method	n	L	E[a.u.]	Rel. Error	Time[s]
RHF	2	20	1.1796	0.4303	0.24
CCD	2	40	0.83743	0.0154	7.41
CCSD	2	40	0.82488	0.0002	119.40
Exact	2	∞	0.8247	0.0000	-
Slater-NN	2	-	0.82514(8)	0.0004	24.64 + 86.03

Table 13.4: The ground state energies computed for the 1D harmonic quantum dots system with 2 electrons for the different methods described so far. The number in parentheses is the uncertainty in the final digit. The time of the Slater-NN calculation is split into the time for the gradient descent and the time for the sampling and blocking estimation of the energy.

We see that the Slater-NN wave function is able to reach an energy very close to the exact ground state energy, much closer than even CCD. Reaching a lower energy than CCSD is not expected, as CCSD is exact within its basis for 2 electrons, which is about as good as one can get most of the time. We unfortunately don't have 1D benchmarks for machine learning methods similar to the Slater-NN wave function, so further work is required to see how our approach compares other machine learning methods.

For the $n = 2$ electron system, we got by far the best performance from the Slater-NN wave function when using only a single hidden layer. Increasing the number of neurons in the hidden layers improved performance, up until 32 neurons, which gave the best performance in our search. The performance of some other selected network architectures is shown in table 13.5.

Network Structure	E[a.u.]	Rel. Error	GD Time[s]	Blocking Time [s]
$2 \to 8 \to \tanh \to 1 \to \exp$	0.8290(3)	0.0052	19.48	5.714
$2 \to 16 \to \tanh \to 1 \to \exp$	0.8267(2)	0.0024	19.06	6.556
$2 \to 32 \to \tanh \to 1 \to \exp$	0.82514(8)	0.0004	24.64	86.03*
$2 \to 48 \to \tanh \to 1 \to \exp$	0.8279(1)	0.0039	28.66	10.37
$2, 32, \tanh, 8, \tanh, 1, \exp$	0.8627(3)	0.0461	33.42	12.27

Table 13.5: The ground state energies computed for the 1D harmonic quantum dots system with 2 electrons for the different Slater-NN network architectures. The number in parentheses is the uncertainty in the final digit. The parameters for the calculation are the same as those described above, except the energy estimation is done with $2^2 3 \approx 8.3 * 10^6$ steps for all networks except the one marked with *.

Even for a single layer with 8 hidden neurons the Slater-NN wave function is able to outperform CCD. We also see that the computational time scales very well with the number of hidden neurons, which is promising for applications to larger systems. The reason the network with 32 hidden neurons can outperform the network with 48 hidden neurons is that more parameters makes training harder, and can lead to overfitting, where the network learns to replicate useless specific features instead of useful general features of whatever it is training to model. This is the same reason the network with the extra layer performed worse, despite having more representative power.

13.2 4 Electrons

We use the same systems and parameters as for the 2 electrons system, except we use 4 electrons, of course. Now that there are 4 electrons, CCSD is no longer exact within the given basis, as triple and quadruple excitations now play a part in the optimal linear combination of excited determinants.

An exact solution is still feasible however, with a moderate basis. As mentioned in section 4.3.1, the configuration interaction (CI) method finds the optimal linear combination of determinants, including all excited determinants. While this method is too slow and requires too much memory for larger systems, at 4 electrons, it can barely give good results on a standard desktop computer. Øyvind Schøyen has provided a CI code which we could run with at most 12 harmonic oscillator basis functions. While this is a slightly modest basis, it still gives a lower energy than CCSD with 30 basis functions. There is also the CISD method, which finds the optimal combination of determinants, allowing only single and double excitations. Code for this calculation was also provided by Øyvind Schøyen. The ground state energies computed with these methods are shown in table 13.6.

Method	n	L	E[a.u.]	Time[s]
RHF	4	20	4.4667	0.03
CCD	4	30	3.8103	21.76
CCSD	4	30	3.7931	289.85
CISD	4	30	3.9164	170.88
CI	4	12	3.7871	222.03

Table 13.6: The ground state energies computed for the 1D harmonic quantum dots system with 4 electrons.

We see that CI has the lowest energy, despite its small basis. This again goes to show that the first few basis functions are the most important when approximating the ground state. We expect CCSD to outperform CISD, as CCSD can indirectly include triply and quadruply excited determinants, which it does. We also see that CCD outperforms CISD. This can be interpreted as the indirect quadruple excitations of CCD being more important than the direct single excitations of CISD. This makes sense, as Brillouin's theorem [24] tells us that when using a Hartree-Fock basis, the first order correction to the energy comes from the doubly excited determinants, which makes it less of a stretch that quadruple excitations are more important than single excitations.

The Slater-NN wave function was unable to improve upon the Slater determinant for 4 electrons in 1D. The simulation was far too unstable. We tested the code written by Hovden for his thesis on 4 electrons in 1D and found the same behavior, even though simulations of more electrons in higher dimensions were stable. We therefore theorize that this unstable behavior is due to the Coulomb force in 1D being highly unstable, even with shielding. Distributing 4 electrons in 1D space, with a potential pulling them together, leads to erratic behavior. This is only a theory however, so we would welcome more insight on 1D VMC calculations of many electrons.

Chapter 14

Performance

The work on this thesis has been greatly focused on writing fast Julia code. Much time could have been saved by writing slower code, but the skills developed along the way proved worth it. This chapter contains performance benchmarks for the most performance critical functions in our code. Where similar code written by others was available, a comparison is made, but always on the same hardware.

All performance benchmarks were run on a desktop computer with an i7-8700K CPU @ 3.70GHz and 16GB of RAM. The python benchmarks were run with Python 3.7.4. The Julia benchmarks were run with Julia 1.8.1 with 12 threads.

14.1 Two-body integrals

L	Julia time [s]	Python time [s]
10	0.02	1.43
15	0.05	2.58
20	0.1	5.0
25	0.18	8.73
30	0.29	14.17
40	0.98	33.65
50	1.64	69.73
60	3.1	132.99

Table 14.1: The time taken to set up the anti-symmetrized two-body integrals between L harmonic oscillator basis functions that are then spin-doubled. The python implementation is from the quantum-systems library. The Julia implementation is from the OrbitalNeuralMethods.jl package written for this thesis.

14.2 VMC Performance

A main focus of our work has been to write very performant code, specifically fast code for the VMC calculation. Solving the many-electron Schrödinger equation is generally a very computationally expensive problem, and VMC calculations are no different. Writing more efficient code saves on computational resources and allows results to be produced faster. VMC calculations also benefit from being fast by allowing larger simulations which can give better statistical properties. With this in mind we present some benchmarks of our code.

Figure 14.1 shows the execution time of computing 100000 samples of energy with the Slater-NN wave function. A basis of $L = 50$ harmonic oscillator functions was used, and a neural network with two hidden layers with sigmoid activation. The number of neurons in these hidden layers is given by the labels in the figure. In case the colors are hard to differentiate: Using more hidden units took more time, so the lowest line is for 25 hidden neurons, the one above for 50 and so on.

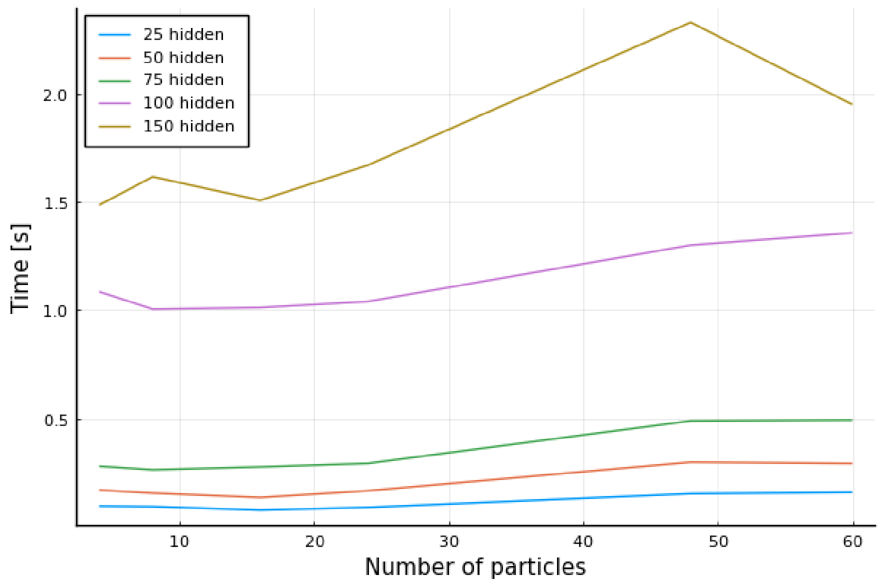


Figure 14.1: The time taken to compute 100000 samples of energy with a given number of particles and nodes in two hidden layers with sigmoid activation.

We see that the number of particles make little difference to the execution time, while the size of the neural makes a large difference. This is to be expected, as the matrix operations in the neural network scale very quickly with an increasing number of neurons, while the operations performed on the positions mostly scale linearly.

14.3 Automatic Differentiation

One of the unique aspects of our implementation of neural networks for variational Monte-Carlo is the use of automatic differentiation for the second derivatives. Others, like Hovden and Samseth, chose to use analytical formulas instead. This is the most sensible choice for simple feed forward networks, because as Samseth noted in his thesis, computing the second derivative with respect all positions using automatic differentiation requires computing the entire Hessian matrix, even though we are only interested in the diagonal. Because there are analytical formulas for just the terms we need, implementations of the analytical formulas will scale better.

Despite this, we chose to still use automatic differentiation because we discovered that by combining forward and reverse mode automatic differentiation, we could get a massive speedup over the most typical way of using reverse mode twice. Table 14.2 shows how quickly (in microseconds) different implementations of automatic differentiation can compute the kinetic energy for the same network and different numbers of inputs.

n	Reverse^2 [μs]	Forward^2 [μs]	Reverse-Forward [μs]	Forward-Reverse [μs]
10	3008	86.4	5013	6.4
50	19163	3819	232337	52.8
100	71267	27282	1231000	171.3

Table 14.2: How quickly different implementations of automatic differentiation can compute the kinetic energy with n inputs. The neural network used had two hidden layers with 32 nodes and the sigmoid activation function, and a single output node with no activation. The time is given in microseconds, meaning the slowest time was slightly over a second.

We see that forward over reverse mode automatic differentiation is faster than all other double derivatives using automatic differentiation, by at least two orders of magnitude for 100 inputs, and by at least one order of magnitude for 10 inputs.

To compare these speeds to the speed of computing the double derivatives using the analytical formulas, we used the implementation written by Hovden, which was also written in Julia. The speed of his implementation and ours using the same network structure, weighs and number of inputs is shown in table 14.3. The evaluation of the entire analytical expression is not strictly needed when also computing the derivatives of parameters, as the first derivative with respect to the input can be reused. We therefore also timed computing the kinetic energy when the first derivatives are already given.

n	Forward-Reverse [μs]	Analytical [μs]	Analytical w/o gradient [μs]
10	6.4	11.3	7.8
25	22.4	17.9	12.2
50	52.8	27.9	19.6
100	171.3	47.0	32.8
500	2675.0	368.7	251.6
1000	10453.0	565.0	379.9

Table 14.3: How quickly automatic differentiation and an implementation of the analytical expressions can compute the kinetic energy with n inputs. The neural network used had two hidden layers with 32 nodes and the sigmoid activation function, and a single output node with no activation. The time is given in microseconds.

We see in table 14.3 that the analytical implementation scales better than automatic differentiation, as expected. Interestingly however, automatic differentiation was faster for 10 inputs. This is not necessarily unexpected, as for this few inputs, the optimization of the implementation is much more important than the scaling. The analytical implementation scales slightly better than linearly for small n, which is expected, since for small vectors, combinations of vectorized operations scale better than n due to the constant cost of some overhead operations. For larger n however, the scaling becomes worse than linear, which might be due to the added costs of memory manipulations.

The automatic differentiation implementation shows scaling slightly worse than linear for up to around 50 inputs, before the quadratic scaling kicks in and the analytical implementation becomes greatly superior. Even at 500 inputs however, which is far more than we will ever need, the analytical implementation is only about 10 times faster. This shows that automatic differentiation can be efficiently used for taking double derivatives of neural networks with a moderate number of inputs. This opens the door for other functions for which we don't have analytical expression for double derivatives easily available.

These results must be taken with a grain of salt however, as the true difference in performance might be much larger for different types of networks, or even for this network. We have not verified if the implementation by Hovden is properly optimized, and the optimization of the derivative given by automatic differentiation will of course differ for different networks.

We also do not know how applicable this approach is in different languages. Popular machine learning frameworks like PyTorch and TensorFlow mostly use reverse mode accumulation[18][25], as it is the best choice of taking the derivative with respect to the many parameters in a machine learning model. While they both support forward mode[19][26], we do not know if they support combining the two modes. We were able to combine the two modes quite simply in Julia, as its powerful type system allows functions from one package to seamlessly be used on types from another package, so that for instance the Dual Number type from ForwardDiff.jl could be operated on by ReverseDiff.jl functions.

14.4 Recursive harmonic oscillator functions

We have chosen to evaluate the harmonic oscillator basis functions recursively. If say we are using a basis of 20 functions, and we want to evaluate the 20th basis function, we first need to evaluate the first 19. This is slower than just evaluating the 20th function by itself, but faster if we need all 20, which is always the case.

As described in section 10.4.1, every harmonic oscillator function is evaluated every time a particle is moved. All the harmonic oscillator functions and their first and second derivatives can be defined recursively, and central to all three of these recursive definition are the Hermite polynomials, which can also be defined recursively. Evaluating the functions at the same time as their first and second derivatives therefore saves a lot of time, as shown in table 14.4.

Functions	Evaluation Time [ns]
Hermite Direct	102.67
Hermite Recursive	26.08
HO Recursive	75.92
HO & Derivatives Recursive	83.26

Table 14.4: The time to evaluate 10 Hermite polynomials directly, and the time to evaluate 10 Hermite polynomials and harmonic oscillator basis functions, and their derivatives recursively.

Table 14.4 shows that evaluating the Hermite polynomials recursively is almost 4 times faster than evaluating them directly. Evaluating all the harmonic oscillator functions and both their derivatives is also faster than directly evaluating the Hermite polynomials. We theorize that evaluating the harmonic oscillator functions and both their derivatives directly would take more than three times as long as calculating the Hermite polynomials directly, so the speedup from this optimization is significant.

14.5 Smart Sorting and Un-sorting

In section 12.4, we described a method to efficiently sort the positions of the particles, and to efficiently permute a different vector using the reverse operation of the position sorting. Here, we compare our method (MiniSort) with a more crude method of simply sorting the positions from scratch each time, and setting up the reverse sorting each time. Table 14.5 shows how the two methods compare.

n	MiniSort [s]	Sorting [s]
2	0.0377	0.4146
10	0.0307	0.4548
25	0.0407	0.8812
50	0.0587	1.8565
100	0.0946	4.2778

Table 14.5: The time taken to update a sorted copy of an array of n elements after a single element has been changed since the last iteration, 10^6 times. The index mappings of the unsorted array to the sorted array and vice versa are also updated. Our specialized method "MiniSort" presented in section 12.4 is compared with the built-in sorting functions of Julia.

We see that our method is about an order of magnitude faster for a small number of particles, but that it scales much better, and therefore approaches being 50 times faster for 100 particles. This comes as no surprise, as our method should scale linearly, while sorting in general scales as $n \log_2(n)$. Figures 14.2 and 14.3 confirm that the methods scale as n and approximately $n \log_2(n)$ as expected.

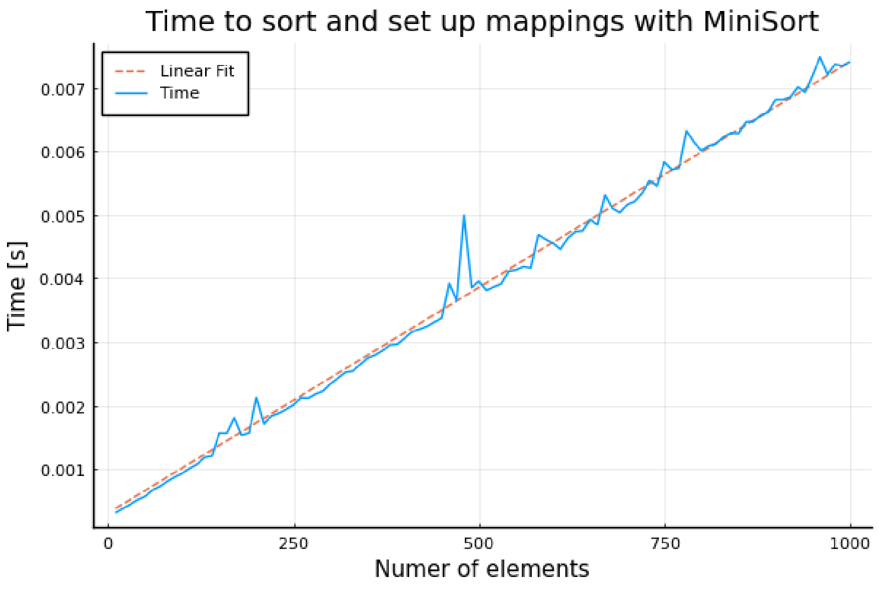


Figure 14.2: The time taken for our specialized method "MiniSort" to update a sorted copy of an array of n elements after a single element has been changed since the last iteration, 10^4 times. The index mappings of the unsorted array to the sorted array and vice versa are also updated. A linear fit of the endpoints is shown for comparison.

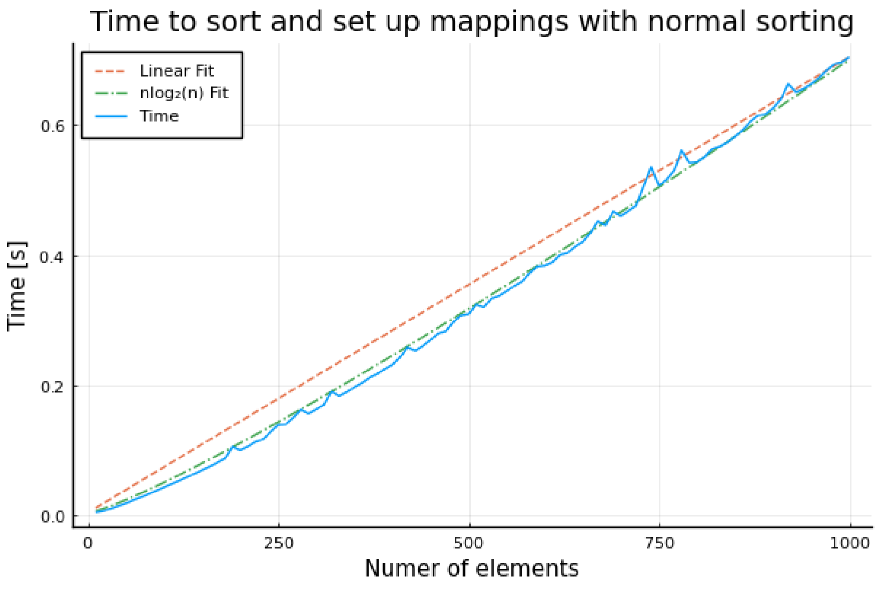


Figure 14.3: The time taken for built-in sorting methods to update a sorted copy of an array of n elements after a single element has been changed since the last iteration, 10^4 times. The index mappings of the unsorted array to the sorted array and vice versa are also updated. A linear fit and an $n\log_2(n)$ fit of the endpoints are shown for comparison.

Chapter 15

Training the Neural Network

The training part of the VMC calculation is very important to get right. Small changes to the optimization algorithm can mean the difference between quickly getting a good result and never getting one. To produce our results, we used the ADAM optimization algorithm with a learning rate of $\gamma = 0.01$, and 500 iterations of training. At each iteration, the gradient was computed with only 10000 samples, and 10000 calibration steps. We found more samples to be unnecessary, only slowing down the computations, and likely doing a worse job of staying out of local minima, since more samples means less noise. The energy at each iteration when optimizing a Slater-NN with 32 hidden neurons, as discussed in section 13.1.4 is shown in figure 15.1.

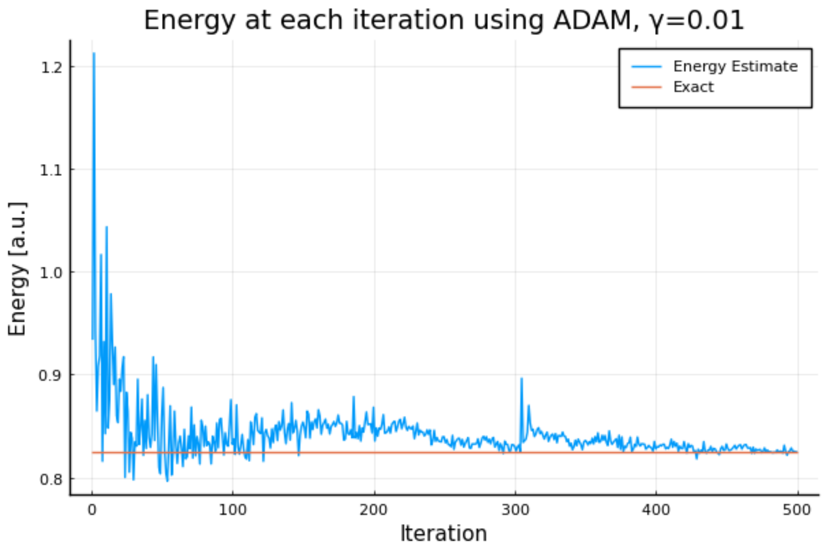


Figure 15.1: The energy per iteration of the gradient descent method. The parameters are given in the text.

This is the best case scenario, as the energy changes greatly at the start, quickly arriving near the target, and then slowly inches closer to a good result. With a higher learning rate, the energy might never converge, as seen in figure 15.2

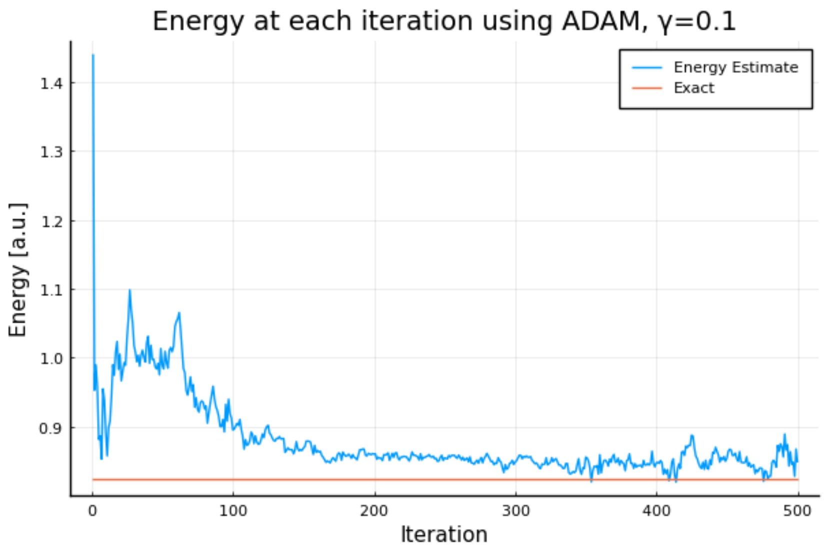


Figure 15.2: The energy per iteration of the gradient descent method. The parameters are given in the text.

While with a lower learning rate, the method might get stuck in local minima, or simply reach the target energy too slowly, as shown in figure 15.3.

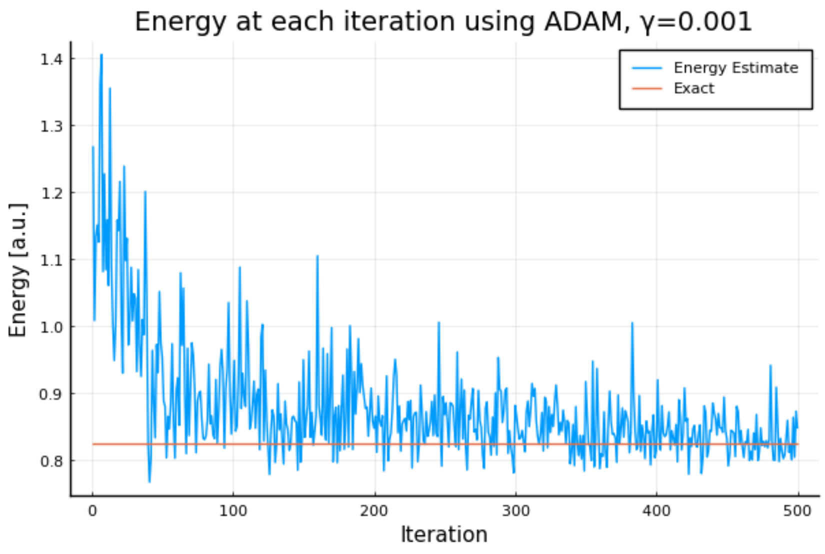


Figure 15.3: The energy per iteration of the gradient descent method. The parameters are given in the text.

We avoided using normal gradient descent altogether, as the energy did not converge properly.

Chapter 16

One-body density

Another measure of the accuracy of our methods is the one-body density. For the Hartree-Fock method, we compute it directly using the coefficients and the spatial orbitals. For the variational Monte Carlo method however, we get the one-body density from sampling the particle positions to build a histogram of where the particles tend to be observed. For the system of two quantum dots in 1D, using the same parameters as in section **??**, we have the one-body density of the exact ground state from Zanghellini[28], shown in figure 16.1. The solid line is the exact ground state, while the dashed line is the one-body density found with RHF in their article.

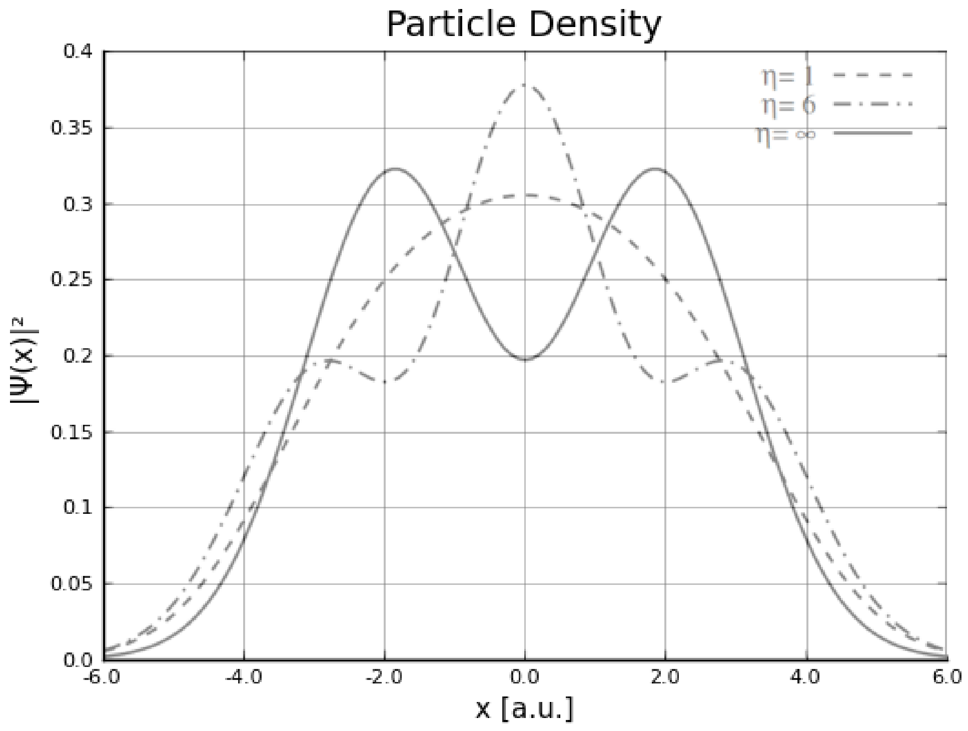


Figure 16.1: The one-body density of two quantum dots. The solid line is exact, the dashed line is from RHF. Results from [28].

We see that the electrons are most likely to be observed at two peaks at -2 and 2, but are still quite likely to be observed in the center of the harmonic potential at $x = 0$. The RHF solution has completely the wrong shape, with the particles being most likely to be observed in the middle.

We now overlay the one-body densities we found using RHF and GHF, and the one-body density computed from our optimal Slater-NN wave function, and get figure 16.2. We used 10^8 samples of the positions to compute the one-body density.

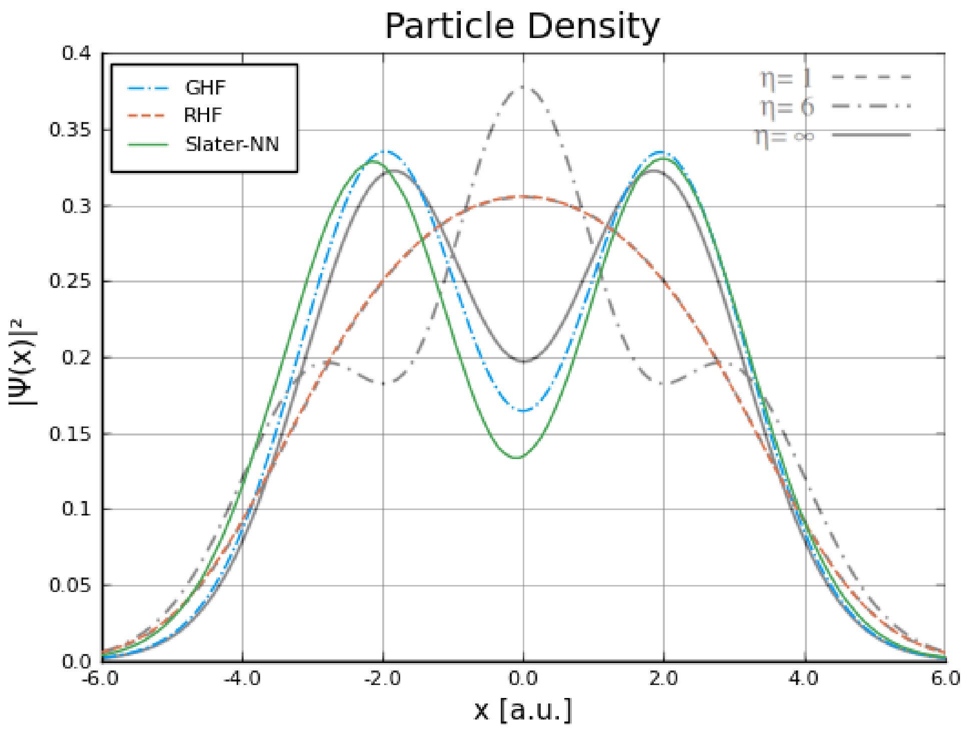


Figure 16.2: The one-body density of two quantum dots in 1D, found with different methods.

We see that our RHF solution perfectly overlaps the RHF solution of Zanghellini. More interestingly however, we see that the GHF one-body density comes quite close to the exact one-body density. The freedom to make the occupied orbitals have different spatial components let GHF find this invalid state which has a one-body density quite close to the exact ground state.

The one-body density of the Slater-NN wave function is slightly left leaning, matching up with the GHF solution quite well on the right, but not on the left. It has the general shape of the exact ground state, but seems to favor the electrons being further apart than the exact ground state. It is surprising that the wave function can match up with the energy of the ground state so well, but differ so much in the one-body density.

Part IV

Conclusion

Chapter 17

Summary Remarks

One of the goals of this thesis was to use the Hartree-Fock method to give an initial guess for the ground state of a many-electron system, and then model correlations using only a neural network. Using this approach, we have approximated the ground state energy of two quantum dots in 1D with a relative error of only 0.0004, coming close the the performance of the coupled cluster singles doubles(CCSD) method which has a relative error of 0.0002. Unlike some other similar methods, our approximation is anti-symmetric. We have also demonstrated new significant optimizations to the variational Monte Carlo calculation of such wave functions, including specialized ways of evaluating neural networks, evaluating harmonic oscillator basis functions, and sorting. Despite our implementation having limited applicability, being limited to 1 dimension, these optimizations and proofs of concept can be utilized in further studies of similar, more general methods. Perhaps our most significant finding was the great efficiency of computing the second derivative of a neural network using automatic differentiation. The use of forward mode automatic differentiation in conjunction with reverse mode allows for 400 times faster computations than the typical approach of using reverse mode twice. The speedup gained from this optimization allows automatic differentiation to stay competitive with analytical formulas for small to moderate systems. This might allow for the efficient use of new types of wave functions that don't have easily expressed derivatives, or allow for much faster development of similar variational approaches.

In addition to this implementation of VMC, we have presented and implemented restricted Hartree-Fock, general Hartree-Fock, coupled cluster doubles, and coupled cluster singles doubles. We have also developed codes for computing and transforming the one- and two-body integrals used in these methods, in addition to a limited implementation of Wick's theorem. All of these codes have been implemented in Julia and combined into the package OrbitalNeuralMethods which provides a common interface for these methods that also allow for seamless reuse between the methods, such as using the same Hamiltonian and basis for both VMC and CCSD. With the exception of the coupled cluster methods, all of these implementations have been heavily optimized, sometimes being orders of magnitude faster than the implementations of our peers.

17.1 Future Studies

The Slater-NN wave function we implemented has great potential for further studies. Following the lead of the FermiNet[17] authors, finding a way of explicitly inputting the distances between electrons into the network could allow for even better modelling of correlations. Alternatively, studying combinations of Hartree-Fock determinants, neural networks and other functions which can model correlations, like Jastrow-factors that satisfy the cusp condition of Coulomb interactions, could be fruitful. And of course, extending the method to 2 and 3 dimensions is a logical next step.

We found that the matrix operations in the network scale very poorly compared to the rest of the method, so the use of GPUs for performing matrix operations is a possible area to look into.

The forward over reverse implementation of automatic differentiation is perhaps the most exciting future prospect following this work. The ability to potentially use any anti-symmetric function as a wave function can lead to some very exciting developments. In addition to this flexibility, the automatic differentiation implementation in fact computes the entire hessian matrix of the neural network, which of course is the source of the poor scaling for large systems, but future authors might find a use for it.

Appendices

.1 The CCSD T2 Equation

$$
\begin{aligned}
0 =& \langle ab\|ij\rangle + \sum_{c}\left(f_{bc}t_{ij}^{ac} - f_{ac}t_{ij}^{bc}\right) - \sum_{k}\left(f_{kj}t_{ik}^{ab} - f_{ki}t_{jk}^{ab}\right) + \frac{1}{2}\sum_{kl}\langle kl\|ij\rangle t_{kl}^{ab} \\
&+ \frac{1}{2}\sum_{cd}\langle ab\|cd\rangle t_{ij}^{cd} + P(ij)P(ab)\sum_{kc}\langle kb\|cj\rangle t_{ik}^{ac} + P(ij)\sum_{c}\langle ab\|cj\rangle t_{i}^{c} \\
&- P(ab)\sum_{k}\langle kb\|ij\rangle t_{k}^{a} + \frac{1}{2}P(ij)P(ab)\sum_{klcd}\langle kl\|cd\rangle t_{ik}^{ac}t_{lj}^{db} + \frac{1}{4}\sum_{klcd}\langle kl\|cd\rangle t_{ij}^{cd}t_{kl}^{ab} \\
&- P(ab)\frac{1}{2}\sum_{klcd}\langle kl\|cd\rangle t_{ij}^{ac}t_{kl}^{bd} - P(ij)\frac{1}{2}\sum_{klcd}\langle kl\|cd\rangle t_{ik}^{ab}t_{jl}^{cd} + P(ab)\frac{1}{2}\sum_{kl}\langle kl\|ij\rangle t_{k}^{a}t_{l}^{b} \\
&+ P(ij)\frac{1}{2}\sum_{cd}\langle ab\|cd\rangle t_{i}^{c}t_{j}^{d} - P(ij)P(ab)\sum_{kc}\langle kb\|ic\rangle t_{k}^{a}t_{j}^{c} + P(ab)\sum_{kc} f_{kc}t_{k}^{a}t_{ij}^{bc} \\
&+ P(ij)\sum_{kc} f_{kc}t_{i}^{c}t_{jk}^{ab} - P(ij)\sum_{klc}\langle kl\|ci\rangle t_{k}^{c}t_{lj}^{ab} + P(ab)\sum_{kcd}\langle ka\|cd\rangle t_{k}^{c}t_{ij}^{db} \\
&+ P(ij)P(ab)\sum_{kcd}\langle ak\|dc\rangle t_{i}^{d}t_{jk}^{bc} + P(ij)P(ab)\sum_{klc}\langle kl\|ic\rangle t_{l}^{a}t_{jk}^{bc} \\
&+ P(ij)\frac{1}{2}\sum_{klc}\langle kl\|cj\rangle t_{i}^{c}t_{kl}^{ab} - P(ab)\frac{1}{2}\sum_{kcd}\langle kb\|cd\rangle t_{k}^{a}t_{ij}^{cd} \\
&- P(ij)P(ab)\frac{1}{2}\sum_{kcd}\langle kb\|cd\rangle t_{i}^{c}t_{k}^{a}t_{j}^{d} + P(ij)P(ab)\frac{1}{2}\sum_{klc}\langle kl\|cj\rangle t_{i}^{c}t_{k}^{a}t_{l}^{b} \\
&- P(ij)\sum_{klcd}\langle kl\|cd\rangle t_{k}^{c}t_{i}^{d}t_{lj}^{ab} - P(ab)\sum_{klcd}\langle kl\|cd\rangle t_{k}^{c}t_{l}^{a}t_{ij}^{db} + P(ij)\frac{1}{4}\sum_{klcd}\langle kl\|cd\rangle t_{i}^{c}t_{j}^{d}t_{kl}^{ab} \\
&+ P(ab)\frac{1}{4}\sum_{klcd}\langle kl\|cd\rangle t_{k}^{a}t_{l}^{b}t_{ij}^{cd} + P(ij)P(ab)\sum_{klcd}\langle kl\|cd\rangle t_{i}^{c}t_{l}^{b}t_{kj}^{ad} \\
&+ P(ij)P(ab)\frac{1}{4}\sum_{klcd}\langle kl\|cd\rangle t_{i}^{c}t_{k}^{a}t_{j}^{d}t_{l}^{b}
\end{aligned}
\tag{1}
$$

.2 Additional Codes

.3 Adding spin

When we make a spin-up and spin-down duplicate of each basis function, we get many new integrals to compute between these new basis functions. However, most of these integrals will be zero due to opposite spins between the basis functions, or they will be the same as the old integrals, if the spins align.

We loop over the latter two indices in our two-body integral, ν and λ. The matrix of elements at the indeces u_new[:, :, ν, λ] then correspond to integrals where the first two basis functions in the integral have spins alternating up and down (Note that we here assume a spin doubling where every other function has spin up, and two and two functions have the same spatial part).

This 2x2 pattern of spins (that we have marked with ″ in the comment below) repeats, and all elements in the 2x2 pattern has the same set of spatial functions. But only one element will be non-zero, the one that has the same spins as the basis functions numbered ν and λ. The code below finds which one of these elements in the 2x2 pattern will be non-zero, and then uses this to turn the two-body-integrals without spin into the two-body-integrals with spin.

```
"""
'up-up'   'up-down'   up-up   ...
'down-up' 'down-down' down-up ...
 up-up     up-down    up-up   ...
 ...       ...        ...     ...
"""
function add_spin_u(u_old)
    l = size(u_old)[1] * 2
    u_new = zeros((l, l, l, l))
    for ν in 1:l
        for λ in 1:l
            if (ν % 2 == 1)      # --- UP ---
                if (λ % 2 == 1) # UP - UP
                    pattern = [1 0; 0 0]
                else             # UP - DOWN
                    pattern = [0 1; 0 0]
                end
            else                 # --- DOWN ---
                if (λ % 2 == 1) # DOWN - UP
                    pattern = [0 0; 1 0]
                else             # DOWN - DOWN
                    pattern = [0 0; 0 1]
                end
            end

            ν_old = Int( ceil(ν / 2) )
            λ_old = Int( ceil(λ / 2) )
            @views kron!(u_new[:, :, ν, λ], u_old[:, :, ν_old, λ_old], pattern)
        end
    end
    return u_new
end
```

Bibliography

[1] Corey Adams et al. “Variational Monte Carlo Calculations of $A \leqslant 4$ Nuclei with an Artificial Neural-Network Correlator Ansatz”. In: *Physical Review Letters* 127 (2021). DOI: 10.1103/physrevlett.127.022502.

[2] Brage Brevig and Karl Henrik Fredly. *A Numerical Study of Trapped Bosons Devising a Variational Monte Carlo Solver*. 2021. URL: https://github.com/KarlHenrik/FYS4411-Gruppe/blob/main/Project1/Report/Project_1_FYS4411.pdf (visited on 03/10/2022).

[3] J. iek and J. Paldus. “Correlation problems in atomic and molecular systems III. Rederivation of the coupled-pair many-electron theory using the traditional quantum chemical methods”. In: *International Journal of Quantum Chemistry* 5.4 (), p. 359. DOI: https://doi.org/10.1002/qua.560050402.

[4] Jií íek. “On the Correlation Problem in Atomic and Molecular Systems. Calculation of Wavefunction Components in UrsellType Expansion Using QuantumField Theoretical Methods”. In: *The Journal of Chemical Physics* 45.11 (1966), p. 4256. DOI: 10.1063/1.1727484.

[5] Jií íek. “On the Use of the Cluster Expansion and the Technique of Diagrams in Calculations of Correlation Effects in Atoms and Molecules”. In: *Advances in Chemical Physics*. John Wiley & Sons, Ltd, 1969, p. 35. DOI: https://doi.org/10.1002/9780470143599.ch2.

[6] T. Daniel Crawford and Henry F. Schaefer III. “An Introduction to Coupled Cluster Theory for Computational Chemists”. In: *Reviews in Computational Chemistry*. John Wiley & Sons, Ltd, 2000. DOI: https://doi.org/10.1002/9780470125915.ch2.

[7] Raphaël Feraud and Fabrice Clerot. “A methodology to explain neural network classification”. In: *Neural networks : the official journal of the International Neural Network Society* 15 (Apr. 2002), pp. 237–46. DOI: 10.1016/S0893-6080(01)00127-7.

[8] H. Flyvbjerg and H. G. Petersen. “Error estimates on averages of correlated data”. In: *The Journal of Chemical Physics* 91.1 (1989), pp. 461–466. DOI: 10.1063/1.457480.

[9] Trygve Helgaker, Poul Jørgensen, and Jeppe Olsen. "Spin in Second Quantization". In: *Molecular ElectronicStructure Theory*. John Wiley & Sons, Ltd, 2000. Chap. 2, 13. DOI: https://doi.org/10.1002/9781119019572.ch2.

[10] Jørgen Høgberget. "Quantum Monte-Carlo Studies of Generalized Many-body Systems". In: *Master thesis* (June 2013). URL: http://urn.nb.no/URN:NBN:no-38645.

[11] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), p. 359. DOI: https://doi.org/10.1016/0893-6080(89)90020-8.

[12] Marius Jonsson. "Standard error estimation by an automated blocking method". In: *Phys. Rev. E* 98 (4 2018), p. 043304. DOI: 10.1103/PhysRevE.98.043304.

[13] J.W.T. Keeble and A. Rios. "Machine learning the deuteron". In: *Physics Letters B* 809 (Aug. 2020), p. 135743. DOI: 10.1016/j.physletb.2020.135743.

[14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980.

[15] Per-Olov Löwdin. "Quantum Theory of Many-Particle Systems. I. Physical Interpretations by Means of Density Matrices, Natural Spin-Orbitals, and Convergence Problems in the Method of Configurational Interaction". In: *Phys. Rev.* 97 (1955), p. 1474. DOI: 10.1103/PhysRev.97.1474.

[16] Dan Nissenbaum. "The stochastic gradient approximation: an application to li nanoclusters". In: *Physics Dissertations* (Jan. 2008).

[17] David Pfau et al. "Ab initio solution of the many-electron Schrödinger equation with deep neural networks". In: *Phys. Rev. Research* 2 (3 Sept. 2020), p. 033429. DOI: 10.1103/PhysRevResearch.2.033429.

[18] PyTorch. *Automatic Differentiation with torch.autograd*. 2022. URL: https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html (visited on 02/10/2022).

[19] PyTorch. *Forward-mode Automatic Differentiation (Beta)*. 2022. URL: https://pytorch.org/tutorials/intermediate/forward_ad_usage.html (visited on 02/10/2022).

[20] Hiroki Saito. "Method to Solve Quantum Few-Body Problems with Artificial Neural Networks". In: *Journal of the Physical Society of Japan* 87 (2018), p. 074002. DOI: 10.7566/JPSJ.87.074002.

[21] Øyvind Schøyen. *Quantum systems*. 2022. URL: https://github.com/Schoyen/quantum-systems (visited on 03/10/2022).

[22] Øyvind Sigmundson Schøyen. “Real-time quantum many-body dynamics”. MA thesis. University of Oslo, 2019, p. 140. URL: http://urn.nb.no/URN:NBN:no-76008.

[23] Isaiah Shavitt and Rodney J. Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge Molecular Science. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511596834.

[24] Attila Szabo and Neil S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. First. Mineola: Dover Publications, Inc., 1996.

[25] TensorFlow. *Introduction to gradients and automatic differentiation*. 2022. URL: https://www.tensorflow.org/guide/autodiff (visited on 02/10/2022).

[26] TensorFlow. *tf.autodiff.ForwardAccumulator*. 2022. URL: https://www.tensorflow.org/api_docs/python/tf/autodiff/ForwardAccumulator (visited on 02/10/2022).

[27] C. J. Umrigar and Claudia Filippi. “Energy and Variance Optimization of Many-Body Wave Functions”. In: *Phys. Rev. Lett.* 94 (15 Apr. 2005), p. 150201. DOI: 10.1103/PhysRevLett.94.150201.

[28] Jürgen Zanghellini et al. “Testing the multi-configuration time-dependent HartreeFock method”. In: *Journal of Physics B: Atomic, Molecular and Optical Physics* 37 (2004), p. 763. DOI: 10.1088/0953-4075/37/4/004.