

BRIDGING QUANTUM MECHANICS
AND MOLECULAR DYNAMICS
WITH ARTIFICIAL NEURAL NETWORKS

by

Svenn-Arne Dragly

THESIS
for the degree of
MASTER OF SCIENCE



Faculty of Mathematics and Natural Sciences
University of Oslo

June 2014



This work is licensed under a
Creative Commons Attribution-ShareAlike 4.0 International License.

Abstract

In this thesis, we explore how a classical potential can be constructed by fitting an artificial neural network to the potential energy surface of an *ab initio* calculation. A Hartree-Fock implementation is explained in detail and used to calculate the potential energy surface. Further, we provide details on how a molecular dynamics code is implemented. This is verified by a simulation of argon crystallization. Results from the Hartree-Fock and molecular dynamics implementations are well aligned with those found in the literature.

To bridge these two implementations in a simulation of hydrogen dissociation, the potential energy surface of hydrogen is fitted with the Fast Artificial Neural Network Library and applied in molecular dynamics. The results are on par with a study using the Kohen-Tully-Stillinger potential. In comparison, the artificial neural network potential is parameterized without empirical data nor initial assumptions about the form of the potential function, but suffers a performance loss by a factor of 10 – 20.

Finally, we discuss different techniques used to visualize molecules, including isosurface and volumetric rendering of electron densities, and billboard rendering of systems with millions of atoms.

Acknowledgements

Two exciting years are coming to an end as I am typing up the final sections of my Master's thesis. In these years, I have come to encounter and work with a great number of awesome people.

I want to thank Milad Hobbi Mobarhan and Henrik Mathias Eiding, with whom I shared an office throughout this endeavor, for the many inspiring discussions we've had, and for the large amount of help and support you have provided. Without you guys, I'm sure I wouldn't have been able to understand as much as I do now.

To Mikael Bull Steen: Thanks for all the inspiring talks about physics and science. You got me truly interested in physics in the first place.

With Anders Hafreager, I have been working on some very exciting side-projects, including visualization tools, virtual reality software, games and optimized C++ code. Some projects even made their way into this text, although they started out just for fun. Thanks for our late hour searches for the most optimized, high-performance visualization code that would still be easily readable and pleasing to the eye. Those have been the greatest moments, and I look forward to many more to come.

Thanks to Filip Sund, for the many talks about everything from setting up RAIDs to landing Kerbals on Mun, and for the great amount of work you have put down in maintaining Smaug with me and Anders. Thanks to Sigve Bøe Skattum for our conversations about everything from graphic design to crowdfunding projects, and to Jørgen Høgberget for many inspiring talks about programming design and compiler magic.

I would like to thank my supervisors, Morten Hjorth-Jensen and Anders Malthe-Sørensen, for the encouragement and motivation you have given me. After consulting you two, I'm always left with a high level of enthusiasm and a sense of endless possibilities. Thanks for believing in us and for making the Computational Physics group an amazing place to be. Speaking of which, I would like to thank everybody at the Computational Physics group for making these years the best.

A huge thanks to my family, who has always cheered for me. And most importantly, to my lovely Ingvild, for providing me with support at all times, and for always boosting my confidence!

Collaboration Details

The Hartree-Fock code used in this thesis has been developed in parallel with two similar projects by H. Mobarhan [1] and Eiding [2]. In detail, for the most part our respective codes have been written single-handedly, but we have collaborated on transforming theory into computational concepts. Some functions and classes have therefore ended up strikingly similar.

The billboard rendering method and virtual reality visualization tools (see Chapter 9) have been developed in collaboration with Hafreager [3].

Svenn-Arne Dragly
Oslo, June 2014

Contents

List of Symbols	1
Atomic Units	2
Source Code	3
1 Introduction	5
1.1 From Quantum Mechanics to Molecular Dynamics	5
1.2 Artificial Neural Network Potentials	6
1.3 Molecular Visualizations	6
1.4 Structure of the Thesis	7
I Introductory Theory	9
2 Introduction to Molecular Dynamics	11
2.1 Potential Energy Surfaces from Quantum Mechanics	12
2.2 Time Integration	13
2.2.1 Euler Methods	13
2.2.2 Velocity Verlet	14
2.3 Common Potentials	15
2.3.1 Lennard-Jones	15
2.3.2 Stillinger-Weber	17
2.3.3 Vashishta-Kalia-Rino-Ebbsjö	18
2.3.4 Kohen-Tully-Stillinger	18
2.3.5 Tersoff	19
2.3.6 ReaxFF	20
2.3.7 Artificial Neural Network Potentials	21
3 Computational Quantum Mechanics	23
3.1 The Many-Body Problem	24
3.1.1 Variational Principle	25
3.1.2 Born-Oppenheimer Approximation	26
3.2 The Slater Determinant: A Guess on the Many-Body Wave Function . .	26
3.3 Quantum Monte Carlo	28
3.3.1 Monte Carlo Integration	29

3.3.2	Metropolis Algorithm	30
3.3.3	Variational Monte Carlo	31
3.3.4	Diffusion Monte Carlo	32
3.4	Hartree-Fock	33
3.4.1	The Energy Expectation Value	34
3.4.2	Defining the J and K Operators	35
3.4.3	Minimizing the Energy Expectation Value	35
3.4.4	The Energy Variation	37
3.4.5	The Fock Operator	38
3.4.6	Obtaining the Hartree-Fock Equations	38
3.4.7	Restricted Hartree-Fock	40
3.4.8	Unrestricted Hartree-Fock	42
3.5	Choice of Single-Particle Orbitals	44
3.5.1	Slater-Type Orbitals	45
3.5.2	Gaussian-Type Orbitals	45
3.6	Integration of Gaussian-Type Orbitals	49
3.6.1	General Properties of Gaussian Functions	49
3.6.2	Overlap Integrals	51
3.6.3	Kinetic Integrals	51
3.6.4	Electron-Nuclei Integrals	52
3.6.5	Electron-Electron Integrals	53
3.7	The Way Forward	53
4	Interpolation with Artificial Neural Networks	55
4.1	An Introduction to Artificial Neural Networks	55
4.2	Network Topology	57
4.2.1	Feedforward Networks	57
4.2.2	Recurrent Networks	57
4.3	Training	58
4.3.1	Backpropagation and the Gradient of the Output	59
4.3.2	Using the Gradient of the Error to Adjust the Weights	61
4.4	Artificial Neural Networks in Molecular Dynamics	61
II	Advanced Theory, Implementation and Results	63
5	A General Hartree-Fock Solver for a Gaussian-Type Basis	65
5.1	Loading data from the Basis Set Exchange	65
5.2	Calculating the Gaussian-Type Orbital Integrals	69
5.3	Solving the Eigenvalue Equations	74
5.3.1	The Hartree-Fock Solver Class	75
5.3.2	Convergence Problems	76
5.4	Symmetries in the Electron-Electron Integrals	77
5.5	Testing the Hartree-Fock Solver	77
5.5.1	Integrator Verification	77
5.5.2	Overlap Integrals and Kinetic Integrals	78

5.5.3	Coulomb Integrals	79
5.6	Results of the Hartree-Fock Solver	80
5.6.1	Ground State Energy of H ₂	80
5.6.2	Pulling the Hydrogen Molecule Apart: Restricted vs. Unrestricted Hartree-Fock	81
5.6.3	Ground State Energies of Molecules	83
5.6.4	Electron Density	83
5.6.5	Electrostatic Potential	86
6	Artificial Neural Networks	91
6.1	Configurations for Two- and Three-Body Potentials	91
6.1.1	Defining the Ranges of Particle Configurations	91
6.1.2	Creating Two-Body Configurations	92
6.1.3	Creating Three-Body Configurations	93
6.1.4	Creating Many-Body Configurations	93
6.2	Training an Artificial Neural Network with FANN	93
6.3	Results and Verification of the Artificial Neural Network	95
6.4	Implementing the Missing Gradient Function in FANN	98
6.5	Using the Artificial Neural Network in Molecular Dynamics	100
7	Advanced Molecular Dynamics	103
7.1	Subdividing the System	103
7.1.1	Verlet Lists	105
7.1.2	Neighbor Cells	105
7.1.3	Neighbor Cells and Verlet Lists Combined	108
7.2	What Goes into the <i>n</i> -Body Terms?	108
7.2.1	One-Body Term	108
7.2.2	Two-Body Term	108
7.2.3	Three-Body Term	109
7.2.4	Many-Body Term	109
7.3	Tail Correction	110
7.3.1	Adding a Shift to the Two-Particle Potentials	110
7.3.2	Damping the Potential Near the Cutoff	110
7.4	Thermostats	111
7.4.1	Measuring Temperature in Molecular Dynamics	112
7.4.2	Andersen Thermostat	113
7.4.3	Berendsen Thermostat	113
7.4.4	Nosé-Hoover Thermostat	114
7.5	Testing the Molecular Dynamics Simulator	114
7.6	Verification: Crystallization of Argon	115
7.7	High-Performance Computing: Parallelization	116
7.7.1	An Argument for Boost MPI	117
7.7.2	Parallel Molecular Dynamics	118

8 Hydrogen Molecules: Results of the Complete Workflow	121
8.1 Simulation Details	121
8.2 Pair Correlation Function	123
8.3 Dissociation	125
8.4 Performance Benchmark	128
III Visualization	131
9 Visualization of Molecules	133
9.1 Densities with Volumetric Rendering	133
9.1.1 Volumetric Vertex Shader	133
9.1.2 Volumetric Fragment Shader	134
9.1.3 Example Application: Denseness	137
9.2 Densities with Isosurfaces	138
9.2.1 Calculating Isosurfaces in Mayavi	138
9.2.2 Final Rendering in Blender	139
9.3 Millions of Atoms with Billboarding	140
9.3.1 Billboarding with a Geometry Shader	141
9.3.2 Billboarding without a Geometry Shader	143
9.3.3 Example Application: Emdee	144
9.3.4 Example Application: Poson	145
IV Conclusions and Future Work	149
10 Conclusion	151
11 Future Implementations and Optimizations	153
11.1 Improved Potential Energy Surface Calculations	153
11.2 A Customized Artificial Neural Network Library	154
11.3 Improved Convergence in Hartree-Fock	154
11.4 Future Visualizations	154
11.5 Applications to New Systems	155
Appendices	157
A Libraries	159
A.1 Plugins	159
A.2 Libraries	159
A.3 Project Structure for Library Based Programs	160
A.3.1 defaults.pri	160
A.3.2 src/	161
A.3.3 app/	161
A.3.4 tests/	162

B Unit Testing	165
B.1 Using a Simple Unit Test Library	165
B.2 Automatic Unit Testing with Jenkins	166
C Visualization Tools and Graphics Programming	169
C.1 Graphics Tools	169
C.1.1 Qt Application Framework	169
C.1.2 OpenGL	169
C.1.3 Qt3D	169
C.2 Shader Programming	171
C.2.1 Vertex Shader	171
C.2.2 Fragment Shader	174
C.2.3 Geometry Shader	175
C.2.4 Tessellation Shader	176
D Visualization Machine	177
D.1 Hardware	177
D.2 Software	177
Glossary	179

List of Symbols

E Energy expectation value 24, 25, 33

G Gaussian function 46

g Activation function of a neuron in a neural network 56, 57, 60, 61, 98

\mathcal{J} Two-body electron-electron interaction operator 34, 42, 77

\mathcal{K} One-body kinetic and electron-nuclei interaction operator 34, 42

H Hamiltonian 24–26, 33, 34

M Number of contracted functions 41, 42, 44, 46, 67, 85, 86, 88

L Number of primitive functions 45, 67

N_e Number of electrons 24–28, 41, 42, 67, 70, 85

N^α Number of electrons with α spin 43, 44, 85, 86

N^β Number of electrons with β spin 43, 44, 85, 86

N_n Number of nuclei 24–26, 88

Ψ Total wave function 23–26, 46, 83, 85

Ψ_T Trial wave function 25–28, 33, 34, 42, 46, 67

Ψ_H Hartree product wave function 26, 28

Ψ_{SD} Slater determinant wave function 23, 28, 46, 85, 88

Ψ_J Jastrow factor for wave function 28

ψ Spin orbital 26, 27, 33–44, 46, 67, 85

ϕ Spatial orbital 23, 40–47, 67, 70, 85

ξ Spin function 40–42, 44, 46, 67

φ Contracted Gaussian function 41, 42, 44–46, 67, 69–71, 77, 85, 86, 88

ϕ Primitive Gaussian function 45–47, 49, 67, 72, 74

Atomic Units

Throughout this thesis, atomic units [4] are implied. If you encounter any number without a unit, it is either in atomic units or unitless. Although this notation is a bit cumbersome, as the unit needs to be deduced from the context, it reduces a lot of clutter. The atomic units used in this thesis are given in the table below, with approximate values in SI units:

Dimension	Defined by/name	Symbol	Approx. value in SI units
mass	electron rest mass	m_e	9.109×10^{-31} kg
charge	elementary charge	e	1.602×10^{-19} C
action	reduced Planck's constant	\hbar	1.054×10^{-31} J s
length	bohr	a_0	5.292×10^{-11} m
energy	hartree	E_h	4.360×10^{-18} J
time		\hbar/E_h	2.419×10^{-17} s
velocity		$a_0 E_h / \hbar$	2.188×10^6 m s ⁻¹
force		E_h / a_0	8.239×10^{-8} N
temperature		E_h / k_B	3.158×10^5 K
pressure		E_h / a_0^3	2.942×10^{13} Pa
electric field		$E_h / (e a_0)$	5.142×10^{11} V m ⁻¹
electric potential		E_h / e	2.721×10^1 V

Source Code

Source code is available online for the applications and libraries developed in this thesis. Permanent links and descriptions are found in the following table:

Name	Description	Link*
Kindfield	Hartree-Fock program and library for atoms and molecules.	kindfield
Denseness	Visualization tool for electron densities.	denseness
FANN-MD	A set of scripts developed for this thesis project.	fann-md
Emdee	Molecular dynamics library, simulator and visualizer.	emdee
Poson	Large-scale molecular dynamics visualization tool with support for virtual reality hardware.	poson

*The links are directed to [http://dragly.org/projects/\[link-name\]](http://dragly.org/projects/[link-name]).

Chapter 1

Introduction

Simulations have become an integrated part of the natural sciences at all scales, paving the way for research in physics, chemistry, medicine, nanotechnology and many other fields. During the past century, our increasing ability to perform quantum mechanical simulations has opened up for a brand new understanding of the matter and chemistry that surround us. We are now able to research properties of materials by modeling large systems of atoms and molecules, and to predict the outcome of experiments by simulating reactions. With current computational tools, researchers are able to discover new materials, such as catalysts [5], and perform detailed studies of reaction mechanisms, including hydrogen combustion [6] and thermal decomposition of nanowires [7].

However, many simulations are still bound by limited computational resources. Most computational methods of quantum mechanics suffer from polynomial growth with system size [8, 9]. Therefore, specialized supercomputers are still needed to perform high-precision dynamics calculations of huge molecules [10]. Not to mention that we often have to make a number of approximations on the way, many of which are possibly devastating to the results we try to obtain.

One of the most important challenges we face in physics and chemistry, is the challenge of scaling from the quantum domain up to the domain of classical physics. We are still in the early stages of predicting the behavior of processes in our cells, such as protein folding [10], with quantum mechanics. We struggle with such microscopic processes because there is no clear-cut answer on how to handle the transition from nanometer scale simulations to the micrometer domain. There are many suggestions on ways to move forward [11], but none are perfect and rarely apply to general problems. Often, we need to build models that are specific for the case we are working with.

1.1 From Quantum Mechanics to Molecular Dynamics

In this thesis, we will explore a procedure that will take us from the interactions of a few atoms to the cumulative effect of millions of atoms. It starts out with solving the Schrödinger equation with the Hartree-Fock method [12]. From this we will be able to calculate a series of physical properties, such as the interatomic forces and the potential energy. The Hartree-Fock method could also be replaced by advanced many-body methods, such as perturbation theory [2, 13], density functional theory (DFT) [14],

Full Configuration Interaction (FCI) [15–17], variational Monte Carlo (VMC) [18, 19], diffusion Monte Carlo (DMC) [19, 20] and coupled cluster theory [13, 21]. However, the main focus will be on Hartree-Fock, due to its simple nature, high efficiency and relatively easy implementation. For this purpose, a Hartree-Fock program, named Kindfield, has been developed from scratch and validated by comparison with results from the literature.

By use of the resulting potentials and forces, we will be able to predict trajectories of atoms in what is known as molecular dynamics (MD) simulations [22]. Nonetheless, Hartree-Fock calculations are expensive when many atoms are involved. Therefore, we will need an efficient way to approximate the potential. The usual approach is to define a functional form of the potential and parameterize this from quantum mechanical calculations. The Lennard-Jones [23] and Stillinger-Weber [24] potentials are examples of such functional forms. The Lennard-Jones potential is usually applied to systems of noble gases, and the Stillinger-Weber potential to systems of silicon. These potentials are very efficient to evaluate, but they cannot be applied to other, more general problems. More advanced potentials, on the other hand, often require many man-hours to tune the functional form and its parameters to new molecules and systems.

1.2 Artificial Neural Network Potentials

Artificial neural networks (ANNs) can be used to fit functional forms to the potential energy surfaces (PESs) calculated with Hartree-Fock. This may result in potentials that are slower to evaluate than traditional alternatives, but the time saved on parameterization will likely make up for it [25]. The goal of this thesis is to describe and apply a procedure that can be used to calculate, fit and apply an ANN potential to a molecular dynamics simulation of arbitrary atoms. To achieve this, the Emdee molecular dynamics simulator has been developed, validated for a simple noble gas system of argon, and prepared for use with ANN potentials.

Further, we have used the above-mentioned Hartree-Fock code to calculate PESs for hydrogen atoms. These have been approximated by an ANN and used in the Emdee program to simulate hydrogen dissociation, $H_2 \rightleftharpoons 2 H$. The results have been compared to a study [26], where the Kohen-Tully-Stillinger potential [27] has been used. While the results are promising, some differences indicate that the Hartree-Fock method is not accurate enough for this purpose.

1.3 Molecular Visualizations

Recently, we have become able to simulate huge molecular systems based on fundamental quantum mechanics. Visualizations will play a core part in our future understanding of the physics of such simulations. Therefore, a large part of this thesis has been devoted to the development and discussion of both traditional and novel techniques for molecular visualization.

At the quantum level, electron densities and electrostatic potentials reveal information about bond strengths, molecular orbitals, and nucleo- and electrophilic regions.

To visualize this, the Denseness application has been developed. It performs live volumetric rendering on the graphics processing unit (GPU) by use of the OpenGL shader language (GLSL). This allows low-latency interaction with the user, who can adjust visualization parameters directly in a Qt-based graphical user interface (GUI). To create artistic renderings, the Mayavi [28] Python package has been used to export isosurface data to the 3D graphics software Blender [29].

In order to make the most out of modern graphics hardware, a large-scale molecular dynamics visualization program has also been developed. This makes use of the *billboarding technique* to draw millions of atoms simultaneously, allowing for real-time exploration of huge datasets. Further, the billboarding technique has been integrated with the Emdee application, which runs a live molecular dynamics simulation on both desktops and mobile devices. In Emdee, the user is given direct control over system properties, like temperature and pressure, and can observe phenomena such as argon crystallization. Additionally, we've enabled support for the Oculus Rift [30] virtual reality headset in the large-scale molecular dynamics visualization program. The possible benefits of using such hardware for scientific purposes have also been reviewed.

1.4 Structure of the Thesis

This thesis consists of four parts: The first part is an introduction to the theories of many-body quantum mechanics, Hartree-Fock, molecular dynamics and ANNs. The second part provides advanced theory, implementation details, and the results of the implemented programs. The third part is about visualization, and the fourth part concludes the text and provides possible future work and applications.

My intention has been to write this thesis as an introductory text for students with little or no knowledge about Hartree-Fock, molecular dynamics, ANNs and molecular visualizations. This has in turn resulted in a text that is a bit lengthy. Readers with prior experience in the above fields may very well skip a few chapters. I do however assume that the reader has a background in basic quantum mechanics, classical mechanics and programming. Additionally, some prior knowledge of chemical terminology (such as covalent bonding) and graphics programming is assumed.

The goal is to enable the reader to use or re-implement the programs discussed in this thesis, and to understand how the different parts can be put together to build a multiscale framework for simulation and visualization of molecules. The style and form of the text is therefore at times more similar to what you will find in a typical tutorial, rather than in a scientific article. This has made it possible to split the text up in smaller parts that may be posted online as guides and tutorials, thereby making it available to a larger community than the rather small group of people that I expect to find their way to a copy of this thesis. In fact, some sections of this text have already been turned into blog posts available on my personal homepage, dragly.org, and on the website of our research group comp-phys.net, in slightly modified versions of what you will find in the following chapters.

Part I

Introductory Theory

Chapter 2

Introduction to Molecular Dynamics

Let's begin by focusing on the motion of atoms. In molecular dynamics, we model atoms as point-like particles. Their interactions are described by classical force fields, and their time-evolution by Newton's equations of motion. This is in other words a simplified picture of the true story. We are applying classical mechanics to a field we know is governed by the laws of quantum. However, this simplification allows us to study much larger systems than we are otherwise able to. Systems of thousands or millions of atoms may be studied with molecular dynamics (MD), and enables us to sample macroscopic properties such as temperature, pressure, diffusion, heat capacity, and much more.

The cost, of course, is that we miss out on some of the complex chemistry that is described by quantum mechanics. The alternative, however, is to solve the time-dependent Schrödinger equation, and this is a very computationally expensive task. With the exception of some very simple systems, the Schrödinger equation cannot be solved on closed form, and numerical methods scales non-linearly with system size. Molecular dynamics simulations do on the other hand scale linearly, if implemented correctly.

In this thesis, the focus is on what we will call “classical molecular dynamics”, where the motion of the nuclei will be determined by Newton's equations, and the interacting forces are defined from the gradient of a potential energy surface (PES). The PES is represented by a mathematical function, often split into a sum over two- and three-body interactions. Other approaches we could have chosen, are Born-Oppenheimer molecular dynamics and the Car-Parinello method, both belonging to a group of methods known as Ehrenfest molecular dynamics, or quantum-classical molecular dynamics (QCMD) [31]. These approaches constantly provide updates to the PES and the forces by performing quantum mechanics calculations on all the involved atoms for each time step. They have become very popular in computational chemistry, due to their predictive power, but suffer from bad scalability once the number of interacting particles are growing. The performance of classical molecular dynamics is superior in cases where the atomic interactions may be well described by PES of only a few interacting atoms [32]. For more information about these methods, we refer to the books by

Marx and Hutter [32], Griebel, Knapek, and Zumbusch [31], and the master thesis of H. Mobarhan [1].

In this chapter, we will start out with a discussion of the foundations of classical molecular dynamics, followed by details about time-integration and commonly used functions to define a PES.

2.1 Potential Energy Surfaces from Quantum Mechanics

The potential energy surface (PES) describes the potential energy $V(\mathbf{r})$ of a system, for any configuration of the atomic positions $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N)$. The expression for $V(\mathbf{r})$ may chosen from empirical knowledge about the system, or from computational quantum mechanics. Either way, to make use of this, we assume that the equations of motions of the atoms are determined only by the motion of the nuclei. From the classical point of view, this is the same as baking the velocity and positions of the electrons into the potential energy of the system. From a quantum mechanics perspective, the degrees of freedom for the nuclei are removed from the Schrödinger equation. This is known as the Born-Oppenheimer approximation.

To calculate the ground state energy of the system in quantum mechanics, we fix the positions of the nuclei, and solve the many-body Schrödinger equation for the electrons only. The Coulomb repulsion of the nuclei is only added to the energy after the quantum calculation. If the PES is calculated exactly with an accurate computational quantum mechanics method, then the minimum of the PES should correspond to the ground state of the system [32].

In a molecular dynamics simulation, we need the PES and its respective gradients to calculate the forces for each time step. If we base the PES on quantum calculations, we will either have to perform a new calculation for each time step, as mentioned in the introduction to this chapter, or approximate the PES by a predefined functional form. We will do the latter, and assume that it can be expanded in a sum of n -body terms,

$$V(\mathbf{r}) \approx \sum_{k=1}^N V_1(\mathbf{r}_k) + \sum_{k < l}^N V_2(\mathbf{r}_k, \mathbf{r}_l) + \sum_{k < l < m}^N V_3(\mathbf{r}_k, \mathbf{r}_l, \mathbf{r}_m) + \dots . \quad (2.1)$$

Each term will then be determined by performing a quantum calculation for a number of configurations, and fitted by a suitable functional form.

This is, however, a crude approximation that is not easily justified. Determining the level of truncation can be really hard, and may even have to be found by simple experimentation, where simulations are run and the properties of the results are checked against our expectations. For instance, noble gases such as argon may be well described by truncating all the way down to the level of two-body potentials, V_2 , while molecules with strong angular dependencies on their bonds, such as water, certainly will need terms of the third order, V_3 , and higher. There is also no obvious way to find the necessary atomic configurations to use during the fitting process. For two- and three-body terms, it is usually possible to span a large set of available configurations, but once we include four or more atoms, the task of selecting probable configurations becomes large. Running *ab initio* simulations for each configuration is also quite expensive.

Additionally, it might be that a chosen functional form of V_n obtains different parameter values if fitted to results from *ab initio* calculations, than if fitted to empirical data. Even so, reasonable results have been produced for many systems with truncations down to two- and three-body terms.

In Chapter 3 we will introduce Hartree-Fock as a quantum mechanics method to calculate the above potential terms. Further, in Chapter 4 we will discuss how to construct functional forms that fits these terms, with ANNs. Finally, in the implementation part, we will work with putting it all together to study the time-evolution of the atoms.

2.2 Time Integration

After deciding to deal with time evolution by the use of classical mechanics, we need to choose a proper numerical time integration method. The time-integration will take the system from its initial configuration of positions and velocities ($\mathbf{r}(0), \mathbf{v}(0)$), to a configuration at a later time t : ($\mathbf{r}(t), \mathbf{v}(t)$). Without influencing the system further, this will allow us to sample the phase space of the microcanonical ensemble (NVE: constant number of particles, volume and energy), and collect statistics of macroscopic properties such as temperature and pressure [22].

2.2.1 Euler Methods

In classical mechanics, the time evolution of the velocities and positions of a particle, are described by integrating the acceleration and velocity, respectively:

$$\mathbf{v}(t) = \mathbf{v}(0) + \int_0^t \mathbf{a}(t) dt \quad (2.2a)$$

$$\mathbf{r}(t) = \mathbf{r}(0) + \int_0^t \mathbf{v}(t) dt \quad (2.2b)$$

On a computer, we need to discretize this procedure in time. To do this, we will assume that the acceleration or velocity is approximately constant for a short time step Δt , and to move ahead step by step. A common way to define such a procedure is to start out with a Taylor expansion of the velocity and position.

We may now write out the Taylor expansion of the velocity and position one time step after t :

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \mathbf{a}^n \Delta t + \mathcal{O}(\Delta t^2) \quad (2.3a)$$

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \mathbf{v}^n \Delta t + \mathcal{O}(\Delta t^2) \quad (2.3b)$$

Where we have used that $\mathbf{r}'(t) = \mathbf{v}(t)$ and $\mathbf{v}'(t) = \mathbf{a}(t)$, and discrete approximations to the values of the positions and velocities at given time steps, where $\mathbf{v}^n \approx \mathbf{v}(t)$ and $\mathbf{v}^{n+1} \approx \mathbf{v}(t + \Delta t)$. This is known as the forward Euler method. By truncating the series at Δt , we see that we end up with a method where the local error is of the order $\mathcal{O}(\Delta t^2)$.

It is worth noting that this approach also makes sense from an intuitive viewpoint: If you want to know the position of the particle at a time Δt from now, you may assume

that the velocity is approximately constant for that short period of time, and thus that the new position will simply be your current position plus the velocity times Δt .

A variant of the forward Euler method is derived by using the velocities at the new time, \mathbf{v}^{n+1} , when calculating the positions for the next time step, \mathbf{r}^{n+1} :

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \mathbf{a}^n \Delta t + \mathcal{O}(\Delta t^2) \quad (2.4a)$$

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \mathbf{v}^{n+1} \Delta t + \mathcal{O}(\Delta t^2) \quad (2.4b)$$

This method has better energy conservation properties than the forward Euler method, but it still has a local error of second order.

In fact, there is a whole family of Euler methods, and a multitude of other methods available for time integration. For the interested reader, I would recommend the course notes on finite difference methods by Langtangen [33].

2.2.2 Velocity Verlet

The key differences between the integration methods reside in the error introduced in the truncation in the time step (the term with the lowest order of those left out in the Taylor expansion), the stability of the solution, and in properties that are important in molecular dynamics, such as time reversibility, long and short term energy conservation, and more. The Euler-family of methods are much used in simple problems, but they do not work well with molecular dynamics simulations due to bad energy conservation properties [31]. Therefore, we will make use of a method that has good long term energy conservation, namely the velocity Verlet method.

The velocity Verlet algorithm is similar to Euler-Cromer, but has a local error that is in the fourth order for the position. This is because it is based on two Taylor expansions taken to the third order to approximate the second order derivative:

$$\mathbf{r}(t + \Delta t) \approx \mathbf{r}(t) + \mathbf{r}'(t)\Delta t + \frac{1}{2}\mathbf{r}''(t)\Delta t^2 + \frac{1}{6}\mathbf{r}'''(t)\Delta t^3 + \mathcal{O}(\Delta t^4) \quad (2.5a)$$

$$\mathbf{r}(t - \Delta t) \approx \mathbf{r}(t) - \mathbf{r}'(t)\Delta t + \frac{1}{2}\mathbf{r}''(t)\Delta t^2 - \frac{1}{6}\mathbf{r}'''(t)\Delta t^3 + \mathcal{O}(\Delta t^4) \quad (2.5b)$$

Adding these two together and reordering the terms, we find

$$\mathbf{r}(t + \Delta t) \approx 2\mathbf{r}(t) - \mathbf{r}(t - \Delta t) + \mathbf{r}''(t)\Delta t^2 + \mathcal{O}(\Delta t^4). \quad (2.6)$$

It may be tempting to use this expression directly in a discrete manner, known as the Störmer-Verlet method:

$$\mathbf{r}^{n+1} = 2\mathbf{r}^n - \mathbf{r}^{n-1} + \mathbf{a}^n \Delta t^2 \quad (2.7)$$

This is however a dangerous approach, because it involves adding the very small Δt^2 term, to much larger terms like \mathbf{r}^n and \mathbf{r}^{n+1} . In numerical programming, one should always watch out for such cases, because they may lead to large round-off errors.

The velocity Verlet method does on the other hand include intermediate calculations of the velocity. These are also useful if we want to sample the kinetic energy. First we compute the velocities by stepping forward $1/2\Delta t$

$$\mathbf{v}^{n+1/2} = \mathbf{v}^n + \frac{1}{2}\mathbf{a}^n \Delta t \quad (2.8)$$

Then we update the positions and calculate the new acceleration based on the new positions:

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \mathbf{v}^{t+1/2} \Delta t \quad (2.9)$$

$$\mathbf{a}^{n+1} = \frac{\mathbf{F}(\mathbf{r}^{n+1})}{m} \quad (2.10)$$

Finally, the velocity is again calculated by stepping forward $1/2\Delta t$. This aligns the time of the velocities and the positions:

$$\mathbf{v}^{n+1} = \mathbf{v}^{n+1/2} + \frac{1}{2}\mathbf{a}^{n+1} \Delta t \quad (2.11)$$

This ensures that we may sample potential and kinetic energies at the same point in time, as well as other observables. The procedure is arithmetically the same as the Störmer-Verlet method, but does not suffer from the possible round-off errors [31].

The velocity Verlet time integration is implemented in the `VelocityVerletIntegrator` class of the Emdee program (see page 3 for information on how to obtain the source code).

2.3 Common Potentials

It is illustrative to speak of some examples that may fit into the above term. Later, we will fit these terms using neural networks, but the common way to do so, is to come up with a function form of the potential and fit its parameters. Here we will briefly mention a few of the most common potentials.

2.3.1 Lennard-Jones

The Lennard-Jones potential [34] is likely one of the best of the simplest potentials out there. It is known to reproduce the behavior of noble gases well [23], and yet it is both simple in its form and inexpensive to use in computer calculations. It is a two-body potential, involving only configurations of pairs of atoms, not taking into account for instance the angles nor the bond-order of molecules. In other words, this is a potential that only works well for systems where there are simple interactions depending only on the distance between pairs of atoms - such as interactions in noble gases.

The general form of the Lennard-Jones 12-6 potential is

$$V_{\text{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right] = \epsilon \left[\left(\frac{r_m}{r} \right)^{12} - 2 \left(\frac{r_m}{r} \right)^6 \right], \quad (2.12)$$

where r is the distance between the two atoms and ϵ is the depth of the potential well. The parameter σ controls the zero-point of the potential, such that $V(\sigma) = 0$, and r_m is the distance for which the potential is at its minimum. The first term describes the repulsion between the atoms, while the second term is the attraction at long distances - the so-called van der Waals force. The potential is illustrated in Figure 2.1.

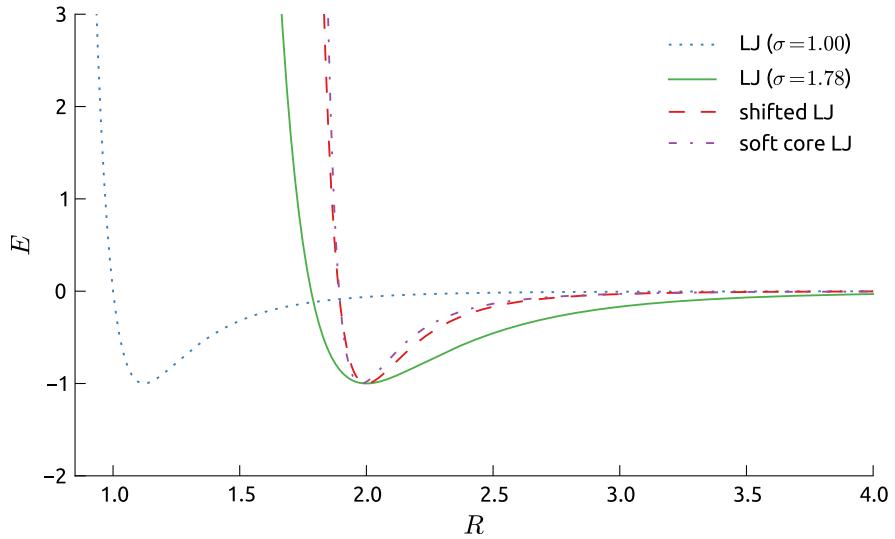


Figure 2.1: Comparison of the regular Lennard-Jones potential (LJ) with its soft-core and shifted counterparts. We see that both the shifted version and the soft-core version are capable of moving the potential curve to a different minimum point, without making the potential shape change in this region. All values are in atomic units.

Soft-Core Lennard-Jones

The Lennard-Jones potential is useful for many systems, but it tends to be hard to fit to the form of the potentials we will work with later in this thesis. Therefore, it is useful to introduce the soft-core Lennard-Jones potential [35]. This has a new parameter, b , which can be adjusted to shift the distance to the potential bottom, allowing for a smoother ascent towards zero - a softer core:

$$V_{\text{softLJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{b\sigma^2 + r^2} \right)^6 - \left(\frac{\sigma}{b\sigma^2 + r^2} \right)^3 \right], \quad (2.13)$$

The potential is illustrated in Figure 2.1. The downside is that the soft-core version is a bit hard to work with if we want the Lennard-Jones potential for a given σ just shifted to a different distance. It also turns out that it is a bit hard for a general least squares implementation, such as the `scipy.optimize.curve_fit` function in Python's SciPy package, to make a good fit with this functional form.

Shifted-Core Lennard-Jones

To deal with the above-mentioned issues of the soft-core Lennard-Jones potential, we may simply introduce a small offset, a , instead - in what is an equivalent functional form of the soft-core Lennard-Jones potential:

$$V_{\text{shiftedLJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{a+r} \right)^{12} - \left(\frac{\sigma}{a+r} \right)^6 \right], \quad (2.14)$$

This dead simple change to the potential allows more flexibility than regular Lennard-Jones and is easier to fit to the results of *ab initio* calculations we will work with in later chapters. The potential is illustrated in Figure 2.1.

However, behind the potential wall, i.e. for distances lower than a , the potential will have a finite value. Some care should be taken that the molecular dynamics simulation is not initialized with distances below a . Having a distance lower than a should in any case not happen, though, because this means that the atoms are basically on top of each other, which in any case would result in all atoms flying out of control immediately.

2.3.2 Stillinger-Weber

In 1985, Frank H. Stillinger and Thomas A. Weber proposed a potential-energy function of two- and three-body interactions between Si atoms [24]. The potential function and its parameters were tuned to obtain the wanted macroscopic properties of Si, such as ensuring that the diamond crystal structure became the most energetically favorable at low pressure.

The two-body part of the Weber-Stillinger potential consists of a factor similar to the Lennard-Jones potential, multiplied by an exponential decay that ensures a zeroing out of the potential at the cutoff, a :

$$V_2(r) = \begin{cases} \varepsilon A \left(\frac{B}{r^p} - \frac{1}{r^q} \right) \exp\left(\frac{1}{r-a}\right), & r < a \\ 0, & r \geq a, \end{cases} \quad (2.15)$$

The three-body term consists of three equal terms, one for each combination of three atoms, i , j and k :

$$V_3(r_{ij}, r_{ik}, \theta_{jik}) = h(r_{ij}, r_{ik}, \theta_{jik}) + h(r_{ji}, r_{jk}, \theta_{ijk}) + h(r_{ki}, r_{kj}, \theta_{ikj}), \quad (2.16)$$

Here θ_{jik} is the angle between \mathbf{r}_j and \mathbf{r}_k at the vertex i . The function h is dependent on the angle between the Si atoms and contains an exponential decay factor that ensures that the potential goes to zero once both r_{ij} and r_{ik} approach the cutoff:

$$h(r_{ij}, r_{ik}, \theta_{jik}) = \lambda \exp\left(\frac{\gamma}{r_{ij}-a} + \frac{\gamma}{r_{ik}-a}\right) \left(\cos \theta_{jik} + \frac{1}{3}\right)^2 \quad (2.17)$$

This function is non-zero for distances below the cutoff where the angle θ_{jik} is about 109° , such that

$$\cos \theta_{jik} = -\frac{1}{3}. \quad (2.18)$$

With this functional form, the three-body potential favors a tetragonal structure of the Si atoms, in accordance with the wanted diamond cubic crystal structure.

In their study, Stillinger and Weber [24] found the optimal parameters to be

$$\begin{aligned} A &= 7.049\,556\,277, \quad B = 0.602\,224\,558\,4, \\ p &= 4, \quad q = 0, \quad a = 1.80, \quad \lambda = 21.0, \quad \gamma = 12.0 \end{aligned} \quad (2.19)$$

This potential function has turned out to be very useful for simulating systems of solid and liquid silicon with molecular dynamics, in turn making their original paper one of the most cited on the topic of molecular dynamics potentials.

2.3.3 Vashishta-Kalia-Rino-Ebbsjö

The potential developed by Vashishta et al. [36] has a similar three-body term to that of the Stillinger-Weber potential, but a more complicated two-body term. This enables modeling of interactions between silicon and oxygen, so that it can be used with systems of silicates (SiO_4). The two-body term has the form

$$V_2(r) = \frac{H_{ij}}{r^{\eta_{ij}}} + \frac{Z_i Z_j}{r} - \frac{\frac{1}{2} (\alpha_i Z_j^2 + \alpha_j Z_i^2)}{r^4} e^{-r/r_{4s}}, \quad (2.20)$$

while the three-body term has the following form:

$$V_3(r_{ij}, r_{ik}, \theta_{jik}) = B_{ijk} f(r_{ij}, r_{ik}) p(\theta_{jik}, \bar{\theta}_{jik}) \quad (2.21)$$

with

$$f(r_{ij}, r_{ik}) = \begin{cases} \exp\left(\frac{l}{r_{ij}-r_0} + \frac{l}{r_{ik}-r_0}\right), & r_{ij}, r_{ik} < r_0 \\ 0, & \text{otherwise,} \end{cases} \quad (2.22)$$

and

$$p(\theta_{jik}, \bar{\theta}_{jik}) = (\cos \theta_{jik} - \cos \bar{\theta}_{jik})^2. \quad (2.23)$$

For details on the different terms and parameter values in this potential, see [36].

In this thesis, this potential will not be discussed further, but it has been implemented as part of the Emdee program and is available for the interested reader (see page 3 for information on how to obtain the source code).

2.3.4 Kohen-Tully-Stillinger

In 1998, Kohen, Tully and Stillinger made an expansion to the Stillinger-Weber potential by adding terms for the interaction of hydrogen atoms [27]. In their study, they investigated the interaction of hydrogen with silicon surfaces, where they found that only short-range two- and three-body potential function terms were needed to model the system.

The form of the two-body term is the same as for the Stillinger-Weber potential, while the three-body term for the Si-H and H-H interactions are defined with a new h -function, with the following form:

$$h(r_{ji}, r_{ik}, \theta_{ijk}) = \begin{cases} \lambda_{ijk} a \exp\left(\frac{\gamma_{ij(k)}^3}{r_{ji}-r_0} + \frac{\gamma_{(i)jk}}{r_{jk}-r_0}\right), & r_{ij}, r_{ik} < r_0 \\ 0 & \text{otherwise.} \end{cases} \quad (2.24)$$

where

$$a = \left(1 + \mu_{ijk} \cos \theta_{ijk} + \nu_{ijk} (\cos \theta_{ijk})^2\right) \quad (2.25)$$

For details on the different terms and parameter values in this potential, see [27].

Kohen, Tully, and Stillinger [27] notes a few important approximations made by choosing this potential form: First, the potential is limited to two- and three-body terms. This is the minimum required to include angle-dependencies and bond-orders, because it allows favoring certain angles for which a third atom must be placed in the proximity of two other atoms. The second approximation is limiting the potential to a short range, excluding long-range forces. The third is that the potential is separated into an angular and a radial component, making the distance and angle uncorrelated. This potential can in other words not favor two different angles for separate distances.

The Kohen-Tully-Stillinger potential was used in 2013 by Skorpa et al. [26] to model the dissociation of hydrogen. In Chapter 8, we will compare the results of their study to molecular dynamics simulations using an artificial neural network (ANN) potential function.

2.3.5 Tersoff

A different family of potentials, developed by Tersoff [37], has the atomic bonds in focus, rather than the atoms. Tersoff [37] argued that the N -body form of the potential was not the right assumption to make, because a three-body interaction would not be sufficient to describe the cohesive energy of silicon for a range of geometries. Four- and five-body terms, would on the other hand be intractable to parameterize, due to the high number of free parameters. The cure, he argued, was to include more information about the nearby environment of every atom, in the form of bonds, directly into the potential. The form of the potential was thus as follows:

$$V_2(r_{ij}) = f_C(r_{ij}) [a_{ij}f_R(r_{ij}) + b_{ij}f_A(r_{ij})] \quad (2.26)$$

Although this looks like a two-body potential, it requires much more information than what is given just from the value r_{ij} . The terms f_R and f_A represent an repulsive and an attractive two-body term, typically given as exponentially decaying functions:

$$f_R(r) = A \exp(-\lambda_1 r) \quad (2.27)$$

$$f_A(r) = -B \exp(-\lambda_2 r) \quad (2.28)$$

The function f_C is a smooth cutoff function, removing the need for an extra tail-correction of the potential (see Section 7.3 for information on tail-corrections). For the smoothing region, it is defined as

$$f_C(r) = \frac{1}{2} - \frac{1}{2} \left(\frac{\pi}{2} \frac{(r - R)}{D} \right), \quad R - D < r < R + D, \quad (2.29)$$

while it is $f_C(r) = 1$ for $r < R - D$ and $f_C(r) = 0$ for $r > R + D$.

The terms a_{ij} and b_{ij} include the bond-ordering of the potential. They hold information about the atoms in the vicinity of the involved pair ij :

$$a_{ij} = (1 + \alpha^n \eta_{ij}^n)^{-1/2n} \quad (2.30)$$

$$\eta_{ij} = \sum_{k \neq i,j} f_C(r_{ik}) \exp \left(\lambda_3^3 (r_{ij} - r_{ik})^3 \right) \quad (2.31)$$

and

$$b_{ij} = (1 + \beta^n \zeta_{ij}^n)^{-1/2n} \quad (2.32)$$

$$\zeta_{ij} = \sum_{k \neq i,j} f_C(r_{ik}) g(\theta_{ijk}) \exp\left(\lambda_3^3(r_{ij} - r_{ik})^3\right) \quad (2.33)$$

$$g(\theta) = 1 + \frac{c^2}{d^2} - \frac{c^2}{d^2 + (h - \cos(\theta))^2}. \quad (2.34)$$

Tersoff [37] found this new form of the potential to provide good results with systems of silicon.

2.3.6 ReaxFF

Recently, reactive force-field (ReaxFF) potentials have gained much attention in the field of molecular dynamics simulations. Initially developed by Van Duin et al. [38], these potentials include many more terms that are targeting higher bond-orders, bonded and non-bonded interactions, over- and under-coordination corrections, and more [39]. This results in quite a large number of parameters to fit to the system at hand, and according to some sources, 10-50 times more expensive calculations than simpler, non-reactive potentials [6].

In total, the potential energy expression may be written out in the following terms [7]:

$$V = V_{\text{bond}} + V_{\text{over}} + V_{\text{under}} + V_{\text{penalty}} + V_{\text{valence}} + \\ V_{\text{torsion}} + V_{\text{conj}} + V_{\text{van der Waals}} + V_{\text{Coulomb}}, \quad (2.35)$$

where the terms represent energy contributions from bonds, over-coordination, under-coordination, valence angles, torsions, conjugation, van der Waals forces and Coulomb interactions, respectively. See Refs. [6, 7, 38, 39] for details on the different terms and how to implement ReaxFF potentials in molecular dynamics.

The development of complex functional forms such as ReaxFF, may indicate an increased demand for simulations of more advanced systems with accurate results. This is definitely the case in computational chemistry, where the demand for more large-scale simulations comes at the cost of expensive *ab initio* methods. Even though the cost of using ReaxFF is higher than simpler potentials, the cost scales much better ($\mathcal{O}(N \log N)$) than *ab initio* calculations ($\mathcal{O}(N^3) \sim \mathcal{O}(N^4)$ for Hartree-Fock) [6, 8].

However, there might also be room for molecular dynamics potentials that are constructed automatically. After all, creating a potential like ReaxFF depends heavily on investing a large amount of person-hours to come up with the correct functional forms and tweaking the parameters for adaption to new systems. This often requires much theoretical and empirical knowledge of the systems we want to model. In some cases, the lack of such knowledge may be the reason why computer simulations are required.

This sets the stage for other options, such as completely general potentials based on *ab initio* calculations and interpolated by machine learning techniques like ANNs.

2.3.7 Artificial Neural Network Potentials

While potentials based on ANNs are the topic of the coming chapters, let me just briefly compare them to the above mentioned potentials.

The neural network potentials are similar in the way that they are also functions of variables such as distances and angles, but they are much more like black boxes. The mathematical expression for the complete potential function is tedious to write out, because it involves many more recursive functions, sums and a plentiful of parameters. One disadvantage with this approach is that we cannot obtain a theoretical understanding of the potential by looking at its terms. The concepts of neural networks are, on the other hand, quite simple.

So I'll spare you any functional expression of the neural network potentials for now. We will instead delve into the details in Chapter 4. For the time being, just think of them as black boxes,

$$V_{\text{NN}}(\mathbf{r}) = \blacksquare. \quad (2.36)$$

Chapter 3

Computational Quantum Mechanics

To construct the needed potential energy surface for molecular dynamics (MD) simulations, we may either try to fit a functional form to experimental data, or dig deep into the theory of quantum mechanics. The benefit of building upon experimental data is that the potential typically results in a good fit in to the properties of the chosen systems. The downside is that the predictive value of the potential is limited, because it has been fitted to a given system, and the potential may not be generalizable to other systems. By using quantum mechanics as our starting point, we will be able to derive predictive potentials, but are no longer guaranteed that approximations we make along the way won't compromise the results for the same systems. In many cases, the latter approach does turn out to give good predictions, as is shown by the many applications of *ab initio* molecular dynamics, where quantum mechanical calculations are used to calculate the forces on all particles live. Fitting a classical potential to results from *ab initio* calculations should in other words be a good option.

To find many-body potentials to use in molecular dynamics calculations, we first need to familiarize ourselves with available computational methods of many-body quantum mechanics. In addition to discussing Hartree-Fock in detail, which is the main method applied in this thesis we will also make a few notes on Quantum Monte Carlo (QMC) in this chapter. Other methods, such as density functional theory (DFT), coupled-cluster theory, many-body perturbation theory and configuration interaction, will not be discussed, but feel free to check out the literature referred to in Chapter 1 for more information. In the case of many-body perturbation theory, a Hartree-Fock implementation similar to the one described in the coming chapters has been implemented and extended with Rayleigh-Schrödinger perturbation theory, in the Master's thesis of Eiding [2].

In this chapter, we will first explore the challenges of the many-body quantum mechanics problem and how it may be solved by QMC. Next, details on how to approximate the many-body electronic wave function Ψ by a Slater determinant Ψ_{SD} will be discussed, followed by the Hartree-Fock method and a discussion of available choices of spatial orbitals ϕ . The choice falls on Gaussian-type orbitals, and details on how to perform the necessary Hartree-Fock integrals are provided.

3.1 The Many-Body Problem

The many-body wave-function Ψ is the central piece of this chapter. For any system, we postulate that there exists such a wave function, and that operators may act upon it to return observables of the system. An operator O acting on Ψ could return a scalar value Q of the system property, which is to say that the wave function is an eigenfunction of the operator:

$$O\Psi = Q\Psi. \quad (3.1)$$

Our main property of interest is the energy of the system. The Hamiltonian operator, which represents the total energy of the system, acts on the wave equation to return the energy eigenvalue:

$$H\Psi = E\Psi. \quad (3.2)$$

Because we are working with systems of many atoms, the Hamiltonian operator is the sum of the kinetic energies of the nuclei and electrons, as well as the electron-electron repulsions, the nucleus-nucleus repulsions and electron-nucleus attractions. The complete Hamiltonian (in atomic units), may be written out as:

$$\begin{aligned} H = & - \sum_i^{N_e} \frac{1}{2} \nabla_{r_i}^2 - \sum_n^{N_n} \frac{m_e}{2m_p} \nabla_{R_i}^2 + \sum_{n < m}^{N_n} \frac{Z_n Z_m}{|\mathbf{R}_n - \mathbf{R}_m|} \\ & - \frac{1}{2} \sum_i^{N_e} \sum_n^{N_n} \frac{Z_n}{|\mathbf{r}_i - \mathbf{R}_n|} + \sum_{i < j}^{N_e} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}. \end{aligned} \quad (3.3)$$

The separate terms represent the following:

- the kinetic energy of the electrons,

$$- \sum_i^{N_e} \frac{1}{2} \nabla_{r_i}^2, \quad (3.4)$$

- the kinetic energy of the nuclei,

$$- \sum_n^{N_n} \frac{m_e}{2m_p} \nabla_{R_i}^2, \quad (3.5)$$

- the Coulomb repulsion of the nuclei,

$$\sum_{n < m}^{N_n} \frac{Z_n Z_m}{|\mathbf{R}_n - \mathbf{R}_m|}, \quad (3.6)$$

- the Coulomb attraction of the nuclei and electrons,

$$- \sum_i^{N_e} \sum_n^{N_n} \frac{Z_j}{|\mathbf{r}_i - \mathbf{R}_n|}, \quad (3.7)$$

- and the Coulomb repulsion of the electrons,

$$\sum_{i < j}^{N_e} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}. \quad (3.8)$$

In the case of more complicated systems, more terms could be added for contributions such as external electric potentials and magnetic fields, but for the scope of this thesis, we will focus solely on purely atomic systems.

The Hamiltonian is defined by the positions of the N_e electrons and N_n nuclei,

$$H = H(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_{N_e}, \mathbf{R}_1, \mathbf{R}_2, \dots, \mathbf{R}_{N_n}), \quad (3.9)$$

while the wave function is defined in terms of the combined spin-coordinate $\mathbf{x}_i = (\mathbf{r}_i, \chi_i)$ of the same particles:

$$\Psi = \Psi(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{N_e}, \mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_{N_n}). \quad (3.10)$$

The particles in quantum mechanics are all described by the same wave function Ψ , resulting in correlations that are more complex than the interaction between classical particles. A change in any part of the wave function requires the entire function to adapt, because the wave function must fulfill the Schrödinger equation for every point in space at all times. This is part of what makes the Schrödinger equation (and wave equations in general) hard to work with in the first place.

There are multiple eigenfunctions Ψ_i that fulfill the Schrödinger equation. Each such eigenfunction has an eigenvalue E_i . These are assumed to be orthonormal. The integral over all coordinates of the product of two wave functions therefore results in the Kroenecker delta. In simplified terms, where $\int d\mathbf{r}$, is an integral over all $3N_e + 3N_n$ coordinates, we have

$$\int \Psi_i^* \Psi_j d\mathbf{x}_1 \dots d\mathbf{x}_{N_e} d\mathbf{X}_1 \dots d\mathbf{X}_{N_n} = \delta_{ij}, \quad (3.11)$$

where $\delta_{ij} = 1$ if $i = j$, and $\delta_{ij} = 0$ otherwise.

3.1.1 Variational Principle

Computational methods of quantum mechanics are often classified as either “variational” or “non-variational”. If a method is variational, it obeys the *variational principle*:

$$E = \langle H \rangle = \langle \Psi_T | H | \Psi_T \rangle \geq \langle \Psi | H | \Psi \rangle = E_0 \quad (3.12)$$

This means that the calculated expectation value of the energy will be an upper bound to the ground state energy. It allows us to determine the quality of a computational method: the lower the energy expectation value, the closer the trial wave function is to the ground state (except if it is close to a local minimum). Further, if the trial wave function is dependent on a set of parameters, mathematical minimization methods can be used to find the optimal parameters.

However, the trial wave function should meet certain requirements. It should be a uniquely defined, single-valued function for a given set of coordinates. Additionally, taking the integral of the modulus squared of the trial wave function must equal to unity:

$$\int |\Psi_T|^2 \, d\mathbf{x}_1 \dots d\mathbf{x}_{N_e} = 1 \quad (3.13)$$

And finally, it should be an everywhere continuous function, and continuously differentiable.

3.1.2 Born-Oppenheimer Approximation

In the Born-Oppenheimer approximation, the degrees of freedom for the nuclei are frozen out, and the Hamiltonian of the system only includes the contributions of the electrons:

$$H = - \sum_i^{N_e} \frac{1}{2} \nabla_{r_i}^2 - \frac{1}{2} \sum_i^{N_e} \sum_n^{N_n} \frac{Z_n}{|\mathbf{r}_i - \mathbf{R}_n|} + \sum_{i < j}^{N_e} \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}, \quad (3.14)$$

This is justified by the high ratio between the mass of the nuclei and electrons. The nuclei are assumed to adapt slowly to changes in the electron distribution, and we may assume that they appear fixed in the reference frame of the electrons. The wave function is explicitly dependent of the electron coordinates, while it is implicitly dependent on the nuclear coordinates:

$$\Psi = \Psi(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{N_e}). \quad (3.15)$$

The kinetic energy and the internuclear repulsion contributions are no longer part of the quantum Hamiltonian. Instead, they are included in the molecular dynamics calculations. The potential energy in molecular dynamics is the total ground state energy expectation value of the electron-only quantum problem, plus the Coulomb repulsion of the nuclei. If it was not for the Born-Oppenheimer approximation, we would not be able to define a potential energy surface (PES) at all. The position of the nuclei would be expressed in terms of a probability distribution, rather than that of point-like particles.

3.2 The Slater Determinant: A Guess on the Many-Body Wave Function

Thanks to the variational principle and the Born-Oppenheimer approximation, we are now able to choose any configuration of nuclei and start looking for trial wave functions. The target is to find the lowest expectation value of the energy, and in general, the wave function could have any form (under the above-mentioned constraints). However, we need to integrate the function to find the expectation values of observables. Therefore, it is a good idea to be somewhat cautious in our search for a trial wave function.

One common starting point is to use a product of functions of one spin-coordinate, so called single-particle states (or spin orbitals):

$$\Psi_H = \psi_1(\mathbf{x}_1) \psi_2(\mathbf{x}_2) \dots \psi_{N_e}(\mathbf{x}_{N_e}). \quad (3.16)$$

This is known as a Hartree-product. It does however not obey the Pauli exclusion principle, which states that a fermionic wave function must be antisymmetric. Fortunately, there is a straightforward way to turn a Hartree-product into an antisymmetric wave function: by using a Slater determinant. It is a sum of Hartree-products, but with alternating signs and index permutations (which makes it antisymmetric):

$$\Psi_T = \frac{1}{\sqrt{N_e!}} \begin{vmatrix} \psi_1(\mathbf{x}_1) & \psi_1(\mathbf{x}_2) & \cdots & \psi_1(\mathbf{x}_{N_e}) \\ \psi_2(\mathbf{x}_1) & \psi_2(\mathbf{x}_2) & \cdots & \psi_2(\mathbf{x}_{N_e}) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_{N_e}(\mathbf{x}_1) & \psi_{N_e}(\mathbf{x}_2) & \cdots & \psi_{N_e}(\mathbf{x}_{N_e}) \end{vmatrix}. \quad (3.17)$$

The Slater determinant may also be written on the following form:

$$\Psi = \frac{1}{\sqrt{N_e!}} \sum_p^{N_e!} (-1)^p \mathbf{P}(p) \psi_1(\mathbf{x}_1) \psi_2(\mathbf{x}_2) \cdots \psi_{N_e}(\mathbf{x}_{N_e}), \quad (3.18)$$

where $\mathbf{P}(p)$ is a permutation operator, permuting the labels of the spin orbitals. The index p denotes which permutation it will use. Note that it only permutes the spin-orbital labels and not their coordinates. Otherwise, the permutation operator would have no effect.

We can check that the Slater determinant is normalized by considering the inner product between two terms appearing in the Slater determinant:

$$\begin{aligned} \langle \psi_i \psi_j \cdots \psi_k | \psi_l \psi_m \cdots \psi_n \rangle &= \int \psi_i(\mathbf{x}_1) \psi_l(\mathbf{x}_1) d\mathbf{x}_1 \int \psi_j(\mathbf{x}_2) \psi_m(\mathbf{x}_2) d\mathbf{x}_2 \\ &\quad \cdots \int \psi_k(\mathbf{x}_{N_e}) \psi_n(\mathbf{x}_{N_e}) d\mathbf{x}_{N_e} \\ &= \delta_{il} \delta_{jm} \cdots \delta_{kn}. \end{aligned} \quad (3.19)$$

The final step is done by assuming that the spin orbitals $\psi_i(\mathbf{x})$ are orthonormal. In the complete Slater determinant inner product, $\langle \Psi | \Psi \rangle$, there will be $N!$ cases where all the Dirac delta functions equal to 1 (we can make $N!$ permutations of the spin-orbital coordinates), and the sum of all those terms is thereby $N!$ itself. This is why we include the $1/\sqrt{N!}$ normalization factor in front of the Slater determinant, resulting in:

$$\begin{aligned} \langle \Psi | \Psi \rangle &= \left(\frac{1}{\sqrt{N_e!}} \sum_p^{N_e!} (-1)^p \mathbf{P}(p) \psi_1(\mathbf{x}_1) \psi_2(\mathbf{x}_2) \cdots \psi_{N_e}(\mathbf{x}_{N_e}) \right) \\ &\quad \cdot \left(\frac{1}{\sqrt{N_e!}} \sum_p^{N_e!} (-1)^p \mathbf{P}(p) \psi_1(\mathbf{x}_1) \psi_2(\mathbf{x}_2) \cdots \psi_{N_e}(\mathbf{x}_{N_e}) \right) \\ &= \frac{1}{\sqrt{N_e!} \sqrt{N_e!}} N_e! \\ &= 1. \end{aligned} \quad (3.20)$$

By approximating the wave function by a single Slater determinant, we are introducing an approximation: that the electrons are in separable single-particle states. In other words, apart from the antisymmetry, the electrons are assumed to be uncorrelated. If had used a plain Hartree product Ψ_H , we would have assumed completely uncorrelated electrons. Even so, when speaking of correlations in quantum mechanics, we are usually speaking of correlations other than the antisymmetry. However, if the Hamiltonian did not include the electron interaction term, there would be no correlation. In that case, the exact solution of the time-independent Schrödinger equation could have been expressed as a Slater determinant [40].

To speak of single-particle states does not make much sense without a Hartree product or Slater determinant. It implies the assumption that the electrons are uncorrelated. This assumption is also what leads to orbital theory in chemistry. When we will later speak of atomic and molecular orbitals, we will actually be talking about the single-particle wave functions we assume here. The real wave function of an atom or a molecule is, on the other hand, possibly not separable into single-particle states. It could be a function with strong correlations between the electrons.

3.3 Quantum Monte Carlo

The most brute force way to solve the Schrödinger equation is probably by use of the Quantum Monte Carlo (QMC) methods. The idea is to choose a trial wave function, calculate the energy expectation value by Monte Carlo integration and apply the variational principle to search for the ground state. The Monte Carlo way of integration is to pick a number of random points to sample - in our case, configurations of the electrons - and sum the sampled integrand for all those points. Once the energy has been obtained, the wave function is varied, and the energy recalculated. By application of the variational principle, we hope to find the ground state wave function by searching for an energy minimum.

This is one of the fastest ways to calculate quantum mechanical integrals once the choice of wave function is too complicated for the integral to be solved analytically. The reason is that the integrals are many-dimensional, and therefore quickly become slow to compute with other numerical methods.

Due to the numerical nature of QMC methods, we can build upon our Slater determinant wave function with more complicated components. Given the Hamiltonian described in equation (3.14), a trial wave function should include the effects of the external potential as well as the potential energy from the interactions between the particles. The former, non-interactive part, is the above mentioned Slater determinant, $|\Psi_{SD}\rangle$. The latter can be included by a Jastrow factor, multiplied with this above Slater determinant:

$$\Psi_T = \Psi_{SD} \Psi_J. \quad (3.21)$$

The Jastrow factor is often chosen on the form

$$\Psi_J = \sum_{i < j}^{N_e} \exp \left(\frac{a_{ij} r_{ij}}{(1 + \beta r_{ij})} \right), \quad (3.22)$$

where i and j are the particle indices. The parameter a_{ij} is equal to 1 when the electrons have anti-parallel spins and $a = 1/3$ when the spins are parallel. β is a variational parameter. This is just one of many popular extensions used to improve the trial wave function in QMC.

3.3.1 Monte Carlo Integration

Monte Carlo methods are among the most efficient at sampling multi-dimensional integrals, such as the ones we are challenged with in quantum mechanics. The idea behind Monte Carlo integration is to sample values for an integral by a random selection of variables. Further, the integration may be improved by importance sampling, which prioritizes regions of the integral with high probability density.

As mentioned above, we are interested in finding the energy expectation value of a many-body system. The expectation value of a function $f(x)$ can be written as

$$\langle f \rangle = \int w(x) f(x) dx, \quad (3.23)$$

where $w(x)$ is the probability distribution. Monte Carlo integration is, however, performed discretely. The above integral is therefore approximated by a sum,

$$\langle f \rangle \approx \frac{1}{N} \sum_{i=1}^N w(x_i) f(x_i). \quad (3.24)$$

Here the x_i 's are randomly chosen points to sample.

The central limit theorem guarantees that the statistical error is reduced as

$$\sigma_N = \frac{\sigma}{\sqrt{N}}. \quad (3.25)$$

This is true regardless of the dimensionality of the integral. Therefore, Monte Carlo integration is often more efficient than other methods for such integrals. The variance can be calculated according to

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (w(x_i) f(x_i))^2 - \left(\frac{1}{N} \sum_{i=1}^N w(x_i) f(x_i) \right)^2, \quad (3.26)$$

while the error for uncorrelated samples is

$$\text{err} = \sqrt{\frac{\sigma^2}{N}}. \quad (3.27)$$

Random number generators do not generally produce uncorrelated values. Therefore, we need to make a better estimate of the variance and error through a method called *blocking*. We will, however, not go into those details here. See the lecture notes of Hjorth-Jensen [41] for more information.

3.3.2 Metropolis Algorithm

Extending upon the theory of Monte Carlo integration, the Metropolis algorithm incorporates the use of random walkers to perform importance sampling in the multidimensional space. A random walker has a defined probability to move around in space while sampling the energy at each move, making the basis of a Markov chain.

Markov chains resemble a microscopic Brownian motion, and as with Brownian motion, the Markov chain will reach the most likely state of the system after running for a long time. In a Markov process a random walker has a selected probability for making a move. The new move is independent of the previous history of the system and depends only on the current state.

For a random walk to be characterized as a Markov chain, it must have the following two important properties:

- **Ergodicity:** From any random starting point, the Markov chain should be able to reach every possible state of the system. Even if the probability of the system being in the state is very small.
- **Detailed balance:** At equilibrium each move in the Markov chain should be of equal probability as its reverse move

$$W_{i \rightarrow j} w_i = W_{j \rightarrow i} w_j \quad (3.28)$$

Here $W_{i \rightarrow j}$ is the probability of making a move from i to j , while w_i is the probability of being in the state i .

From detailed balance we get the relation

$$\frac{w_i}{w_j} = \frac{W_{j \rightarrow i}}{W_{i \rightarrow j}}. \quad (3.29)$$

Because we may model the transition probability $W_{i \rightarrow j}$ in any manner, we choose the form

$$W_{i \rightarrow j} = g_{i \rightarrow j} A_{i \rightarrow j} \quad (3.30)$$

where $g_{i \rightarrow j}$ is the probability of suggesting a move from i to j , while $A_{i \rightarrow j}$ is the probability for this move to be accepted.

The master equation in the Metropolis algorithm is

$$\frac{dw_i}{dt} = \sum_j [W_{j \rightarrow i} w_j - W_{i \rightarrow j} w_i] \quad (3.31)$$

and should at equilibrium be equal to zero. This means that the system should go from a state j to the final state i at the same rate as it goes from i to j . This is fulfilled by detailed balance.

The acceptance ratio to make a move from i to j is defined as

$$R = \frac{g_{j \rightarrow i} w_i}{g_{i \rightarrow j} w_j}. \quad (3.32)$$

For uniform transition probability this is simplified by setting $g_{i \rightarrow j} = 1$, and is the simple or brute-force Metropolis algorithm. Metropolis-Hastings include non-uniform transition probabilities, and the acceptance probability is then given by

$$A_{i \rightarrow j} = \min\{R, 1\}, \quad (3.33)$$

also known as importance sampled Metropolis. In discrete manners on a computer we usually implement this by comparing R with a number between 0 and 1.

A usual implementation of the Metropolis algorithm is illustrated as follows:

- Generate an initial position \mathbf{r}_i for a random walker.
- Suggest a new move $\mathbf{r}_j = \mathbf{r}_i + \delta\mathbf{r}$ where $\delta\mathbf{r}$ is a randomly chosen vector of a predefined length. This length must be set to a reasonable size and is often chosen to be such that the moves are accepted 50% of the time.
- The move is accepted if

$$\frac{w_i}{w_j} \geq \eta, \quad \eta \in [0, 1], \quad (3.34)$$

otherwise the move is rejected and the walker stays in the same place.

- If the move is accepted, the walkers position is set to $\mathbf{r}_i = \mathbf{r}_j$.
- All observables of interest are sampled regardless of whether the move was accepted or rejected. One such observable is the local energy, which will be described in the following section.
- The above is repeated until enough samples have been made.

3.3.3 Variational Monte Carlo

By combining the above mentioned methods of performing Monte Carlo integration and the variational principle, we get variational Monte Carlo (VMC). First, we need to introduce a quantity known as the local energy,

$$E_L = \frac{1}{\Psi_T(\mathbf{R})} \hat{H} \Psi_T(\mathbf{R}). \quad (3.35)$$

which is a sample of the energy “locally” in a given configuration \mathbf{R} of the particles’ positions. It is a function of α and β because it depends on the trial wave function Ψ_T . This allows us to rewrite the expectation value of the energy as

$$E = \langle H \rangle = \int \rho(\mathbf{R}) E_L d\mathbf{R} \quad (3.36)$$

with ρ as the probability density distribution, which is defined as

$$\rho = \frac{|\Psi_T(\mathbf{R})|^2}{\int |\Psi_T(\mathbf{R})|^2 d\mathbf{R}}. \quad (3.37)$$

The trial wave function may now be varied to search for the minimum energy - i.e. the ground state.

Finally the integral has taken on a Monte Carlo form, and we are ready to transform it into a discrete form,

$$\langle H \rangle \approx \frac{1}{N} \sum_i^N E_{L_i}. \quad (3.38)$$

This may now be implemented in terms of sampling with the Metropolis algorithm.

3.3.4 Diffusion Monte Carlo

While the variational Monte Carlo methods are limited by the initial guess at the form of the wave function, diffusion Monte Carlo is capable of finding the energy minimum without a perfect guess. This in turn makes it a strong method for finding the ground state energy of any system even without much a priori knowledge of the system.

Diffusion Monte Carlo should in principle converge to the correct form of the wave function, but a good primary guess at the wave function will help the method converge faster and require fewer walkers. One may think of the final wave function as a superposition of the initial guess, and the closer the guess is to the solution, the easier it is to construct the final superposition with fewer walkers.

The idea behind diffusion Monte Carlo is the use of a special evolution operator that makes all other components but the ground state vanish from our wave function. The only catch is that we need to know the ground state energy for this operator to pick out the ground state component, which is what we wanted to find in the first place. We can, however, use the special evolution operator on a wave function that is close to the ground state, hopefully below the first excited eigenstate, and still be able to pick out the ground state. To explain how, we need to take a closer look at the evolution operator.

The special evolution operator resembles the time-evolution operator, and is defined as

$$\exp(-\hat{H}\tau) \quad (3.39)$$

where $\tau = -it$ is the imaginary time. Applying this to an arbitrary wave function gives

$$\exp(-\hat{H}\tau) \Psi(x) = \sum_i c_i \exp(-\varepsilon_i \tau) \phi_i(x). \quad (3.40)$$

Letting the imaginary time run towards infinity, $\tau \rightarrow \infty$, we see that all negative energies blow up, while positive energies vanish. Adding an energy shift to the equation makes us able to control this and set the threshold. All energies below the threshold will blow up, while energies above vanish. Energies that equal the threshold would be left intact. The effect of the special evolution operator now becomes

$$\Psi(x, \tau) = e^{-(\hat{H}-E_T)\tau} \Psi(x) = \sum_i c_i e^{-(\varepsilon_i - E_T)\tau} \phi_i(x). \quad (3.41)$$

If we are so lucky that E_T equals the ground state energy ε_0 , taking the limit $\tau \rightarrow \infty$ will leave us only with the ground state wave function because all $\varepsilon_i > \varepsilon_0$ will vanish,

and there are of course no energies below the ground state. This leaves us with

$$\lim_{\tau \rightarrow \infty} \Psi(x, \tau) = c_0 \phi_0(x). \quad (3.42)$$

The clue here is to note that if the trial energy E_T is below the first excited state, all excited states will vanish, except for the ground state, which blows up. Even though it blows up, it is still the only one left, which makes it possible to pick it out by adjusting this method a bit.

For more details on variational and diffusion Monte Carlo, see the thesis of Høgberget [19] and the lecture notes of Hjorth-Jensen [41].

3.4 Hartree-Fock

Before delving into the details, we will begin by summarizing the steps involved in deriving the Hartree-Fock method. This will serve as a reference in the following, where details for each step will be provided.

1. Assume a Slater determinant as trial wave function

The trial wave function is a Slater determinant of single-particle wave functions. Together with the Hamiltonian, it gives a proposed solution of the Schrödinger equation: $H\Psi_T = E\Psi_T$.

2. Find the energy expectation value

Multiply the above Schrödinger equation by the trial wave function from the left and integrate over all of space and spin. Because the trial wave function should be normalized, the energy E is just multiplied by 1, and we find $E = \langle \Psi_T | H | \Psi_T \rangle$.

3. Define the J and K -operators

The expression for the eigenvalue of the energy may be simplified by introducing two new operators, J and K . These will be defined in Section 3.4.2.

4. Minimize the energy by applying the variational principle

Because the ground state is upper-bound by the trial wave function energy, we can differentiate the energy E with regards to the wave function to define a minimization scheme. In combination with Lagrange multipliers, this gives the single-particle equations to solve.

5. Define the Fock-operator and obtain the Hartree-Fock equations

Define $\mathcal{F} = h + J - K$ as the Fock operator (J and K will be defined in Section 3.4.2) and set up the eigenvalue equations $\mathcal{F}\psi_k = \epsilon_k \psi_k$. These are the Hartree-Fock equations.

6. Get rid of the spin-dependency and solve

Our Hamiltonian does not depend on spin. The spin part of our trial wave function may hence be integrated out of the equations. However, what spin each spin orbital should be associated with must be chosen. The choice will either lead to the Roothaan equations of restricted Hartree-Fock (RHF), or the Pople-Nesbet equations of unrestricted Hartree-Fock (UHF).

We set up the Schrödinger equation and applied the Born-Oppenheimer approximation in Section 3.1. Let's therefore begin with the energy expectation value. In the rest of this section, we will follow closely the derivation of J.M. Thijssen [12].

3.4.1 The Energy Expectation Value

We start out by rewriting our Hamiltonian in terms of the operators \hbar_i and \mathcal{J}_{ij} :

$$H = \sum_i \hbar_i + \sum_{i < j} \mathcal{J}_{ij}, \quad (3.43)$$

where

$$\hbar_i = -\frac{1}{2} \nabla_i^2 - \sum_n \frac{Z_n}{|\mathbf{r}_i - \mathbf{R}_n|}. \quad (3.44)$$

is the kinetic energy of each electron, plus the potential energy of the electron-nuclei interactions. The term \mathcal{J} is defined as

$$\mathcal{J}_{ij} = \frac{1}{|\mathbf{r}_i - \mathbf{r}_j|}, \quad (3.45)$$

and represents the energy contribution from the electron-electron interaction.

We multiply the term \hbar by the trial wave function Ψ_T from the left and the right, and integrate, to get

$$\begin{aligned} \langle \Psi_T | \sum_i \hbar_i | \Psi_T \rangle &= \int \psi_1^*(\mathbf{q}_1) h_1 \psi_1(\mathbf{q}_1) d\mathbf{q}_1 + \int \psi_2^*(\mathbf{q}_2) h_2 \psi_2(\mathbf{q}_2) d\mathbf{q}_2 + \dots \\ &= \sum_k \langle \psi_k | \hbar | \psi_k \rangle. \end{aligned} \quad (3.46)$$

Here we drop the subscript for \hbar in the final expression because the name of the integration variable is arbitrary. This means that \hbar , without any subscript, is defined as

$$\hbar = -\frac{1}{2} \nabla^2 - \sum_n \frac{Z_n}{|\mathbf{r} - \mathbf{R}_n|} \quad (3.47)$$

We get the result in (3.46) because the \hbar_i operator only acts on particle i . The other spin orbitals in the Slater determinant are not affected by the operator, and are integrated away due to orthonormality.

The electron-electron interaction is slightly more complicated because the \mathcal{J} operator includes the two indices i and j . This results in two terms:

$$\left\langle \Psi \left| \sum_{i < j} \mathcal{J}_{ij} \right| \Psi \right\rangle = 2 \left(\sum_{kl} \langle \psi_k \psi_l | \mathcal{J} | \psi_k \psi_l \rangle - \sum_{kl} \langle \psi_k \psi_l | \mathcal{J} | \psi_l \psi_k \rangle \right). \quad (3.48)$$

Finally, the energy expectation value is given by

$$E = \langle \Psi | H | \Psi \rangle = \sum_k \langle \psi_k | \hbar | \psi_k \rangle + \frac{1}{2} \sum_{kl} \left(\langle \psi_k \psi_l | g | \psi_k \psi_l \rangle - \langle \psi_k \psi_l | g | \psi_l \psi_k \rangle \right). \quad (3.49)$$

The energy calculation is usually the final step in a Hartree-Fock implementation. Before we can make use of (3.49), we need to find the set of spin orbitals $\{\psi_i\}$ that minimize the energy.

3.4.2 Defining the J and K Operators

The expression for the energy expectation value in (3.49) can be simplified by introducing a shorthand notation,

$$E = \sum_k \left\langle \psi_k \left| h + \frac{1}{2}(J - K) \right| \psi_k \right\rangle. \quad (3.50)$$

Here, the operators J and K are given by

$$J = \sum_l J_l \quad (3.51)$$

$$K = \sum_l K_l, \quad (3.52)$$

where the single-particle terms J_l and K_l are defined as

$$\begin{aligned} J_l(\mathbf{q}_i)\psi_k(\mathbf{q}_i) &= \int \psi_l^*(\mathbf{q}_j)g_{ij}\psi_l(\mathbf{q}_j)\psi_k(\mathbf{q}_i) d\mathbf{q}_j \\ &= \langle \psi_l | g | \psi_l \rangle \psi_k(\mathbf{q}_i), \end{aligned} \quad (3.53)$$

and

$$\begin{aligned} K_l(\mathbf{q}_i)\psi_k(\mathbf{q}_i) &= \int \psi_l^*(\mathbf{q}_j)g_{ij}\psi_l(\mathbf{q}_i)\psi_k(\mathbf{q}_j) d\mathbf{q}_j \\ &= \langle \psi_l | g | \psi_k \rangle \psi_l(\mathbf{q}_i). \end{aligned} \quad (3.54)$$

Note that k and l are interchanged in the two integrals. The K term therefore represents the “exchange” contribution of our Slater determinant, while J is the “direct” Coulomb contribution.

3.4.3 Minimizing the Energy Expectation Value

We seek an energy minimum by variation of the spin orbitals ψ_k . From mathematical optimization, we know that a minimum is found when any displacement of the spin orbitals, $\psi_k \rightarrow \psi_k + \delta\psi_k$, results in zero displacement of the energy:

$$\delta E = 0. \quad (3.55)$$

We also require that the spin orbitals ψ_k are orthonormal,

$$\langle \psi_k | \psi_l \rangle = \delta_{kl}, \quad (3.56)$$

where δ_{kl} is the Dirac delta function. This constraint may be defined as an equation,

$$T [\psi_k, \psi_l] = \langle \psi_k | \psi_l \rangle - \delta_{kl} = 0, \quad (3.57)$$

where T is a functional of two spin orbitals ψ_k and ψ_l . In combination with the minimization criterion $\delta E = 0$, we find the displacement of T by varying ψ_k and ψ_l :

$$\delta T [\psi_k, \psi_l] = T [\psi_k + \delta\psi_k, \psi_l + \delta\psi_l] - T [\psi_k, \psi_l]. \quad (3.58)$$

The first term is written out as follows:

$$\begin{aligned}
 T[\psi_k + \delta\psi_k, \psi_l + \delta\psi_l] &= \int (\psi_k^*(\mathbf{q}) + \delta\psi_k^*(\mathbf{q})) (\psi_l(\mathbf{q}) + \delta\psi_l(\mathbf{q})) d\mathbf{q} \\
 &= \int \left(\psi_k^*(\mathbf{q})\psi_l(\mathbf{q}) + \psi_l(\mathbf{q})\delta\psi_k^*(\mathbf{q}) \right. \\
 &\quad \left. + \psi_k^*(\mathbf{q})\delta\psi_l(\mathbf{q}) + \delta\psi_k^*(\mathbf{q})\delta\psi_l(\mathbf{q}) \right) d\mathbf{q} \\
 &= \langle \psi_k | \psi_l \rangle + \langle \delta\psi_k | \psi_l \rangle \\
 &\quad + \langle \psi_k | \delta\psi_l \rangle + \langle \delta\psi_k | \delta\psi_l \rangle - \delta_{kl}.
 \end{aligned} \tag{3.59}$$

Further, the term $\langle \delta\psi_k | \delta\psi_l \rangle$ may be omitted because it is likely to be very small:

$$T[\psi_k + \delta\psi_k, \psi_l + \delta\psi_l] \approx \langle \psi_k | \psi_l \rangle + \langle \delta\psi_k | \psi_l \rangle + \langle \psi_k | \delta\psi_l \rangle - \delta_{kl}. \tag{3.60}$$

This yields the final expression for the displacement in T :

$$\delta T[\psi_k, \psi_l] = \langle \delta\psi_k | \psi_l \rangle + \langle \psi_k | \delta\psi_l \rangle. \tag{3.61}$$

This must be equal to zero because we are only concerned with spin orbitals ψ_k that satisfy $T[\psi_k, \psi_l] = 0$, and thus the displacement cannot change T :

$$\delta T[\psi_k, \psi_l] = 0. \tag{3.62}$$

Following the typical procedure for solving constraint equations, we introduce an arbitrary function Λ_{kl} and multiply (3.62) by this:

$$\Lambda_{kl}\delta T[\psi_k, \psi_l] = \Lambda_{kl} \left(\langle \delta\psi_k | \psi_l \rangle + \langle \psi_k | \delta\psi_l \rangle \right) = 0. \tag{3.63}$$

This equation needs to be fulfilled for any variation of the spin orbitals. There is one such constraint equation for each pair of indices k and l , and all these equations are coupled with (3.55), which determines the variation of the energy.

To solve an equation system with constraints, we start out by subtracting the constraint equations (3.63) from the energy variation (3.55). This results in what is known as the method of Lagrange multipliers¹:

$$\delta E - \sum_{kl} \Lambda_{kl} \left[\langle \delta\psi_k | \psi_l \rangle + \langle \psi_k | \delta\psi_l \rangle \right] = 0. \tag{3.64}$$

The next step is to solve this equation for the Lagrange multipliers Λ_{kl} . However, in the following, it will be handy to use $\Lambda_{kl} = \Lambda_{lk}^*$. Let's prove this before moving on.

¹This is a very specialized example of the method of Lagrange multipliers, but the derivation of the method is very similar to one outlined here. For a more general derivation, see [42].

Symmetry in the Lagrange Multipliers

By using that $\langle \psi_k | \psi_l \rangle = \langle \psi_l | \psi_k \rangle^*$, we find the complex-conjugated counterpart to equation (3.64):

$$\begin{aligned} 0 &= \delta E - \sum_{kl} \Lambda_{kl} \left[\langle \delta \psi_k | \psi_l \rangle + \langle \psi_k | \delta \psi_l \rangle \right] \\ &= \delta E - \sum_{kl} \Lambda_{kl}^* \left[\langle \delta \psi_k | \psi_l \rangle^* + \langle \psi_k | \delta \psi_l \rangle^* \right] \\ &= \delta E - \sum_{kl} \Lambda_{kl}^* \left[\langle \psi_l | \delta \psi_k \rangle + \langle \delta \psi_l | \psi_k \rangle \right]. \end{aligned} \quad (3.65)$$

The indices in the sum are dummy indices. We are therefore free to swap them:

$$\begin{aligned} 0 &= \delta E - \sum_{lk} \Lambda_{lk}^* \left[\langle \psi_k | \delta \psi_l \rangle + \langle \delta \psi_k | \psi_l \rangle \right] \\ &= \delta E - \sum_{kl} \Lambda_{lk}^* \left[\langle \delta \psi_k | \psi_l \rangle + \langle \psi_k | \delta \psi_l \rangle \right]. \end{aligned}$$

This is the same as equation (3.64), except for Λ_{kl} being replaced by Λ_{lk}^* . Solving the equation for either Λ_{kl} or Λ_{lk}^* will give the same result. The two must therefore be equal:

$$\Lambda_{kl} = \Lambda_{lk}^*. \quad (3.66)$$

We now move on to solve equation (3.64) for Λ_{kl} .

3.4.4 The Energy Variation

The explicit expression for δE is as follows:

$$\begin{aligned} \delta E &= E [\psi_1 + \delta \psi_1, \psi_2 + \delta \psi_2, \dots, \psi_N + \delta \psi_N] - E [\psi_1, \psi_2, \dots, \psi_N] \\ &\approx \sum_k \langle \delta \psi_k | h | \psi_k \rangle + \text{complex conjugate} \\ &\quad + \frac{1}{2} \sum_{kl} (\langle \delta \psi_k \psi_l | g | \psi_k \psi_l \rangle + \langle \psi_l \delta \psi_k | g | \psi_l \psi_k \rangle \\ &\quad - \langle \delta \psi_k \psi_l | g | \psi_l \psi_k \rangle - \langle \psi_l \delta \psi_k | g | \psi_k \psi_l \rangle) + \text{complex conjugate} \\ &= \sum_k \langle \delta \psi_k | h + (J - K) | \psi_k \rangle + \text{complex conjugate} \end{aligned} \quad (3.67)$$

In the last step we have used the symmetry of the two-electron elements:

$$\langle \delta \psi_k \psi_l | g | \psi_k \psi_l \rangle = \langle \psi_l \delta \psi_k | g | \psi_l \psi_k \rangle \quad (3.68)$$

and equivalently for the complex conjugate elements.

3.4.5 The Fock Operator

At this point, it will be useful to define the Fock operator, \mathcal{F} , to simplify the upcoming equations:

$$\mathcal{F} = h + (J - K). \quad (3.69)$$

The variation of the energy may thus be written as:

$$\delta E = \sum_k \left[\langle \delta\psi_k | \mathcal{F} | \psi_k \rangle + \langle \psi_k | \mathcal{F} | \delta\psi_k \rangle \right]. \quad (3.70)$$

3.4.6 Obtaining the Hartree-Fock Equations

We now have three coupled sets of equations that needs to be solved simultaneously, namely

$$\left. \begin{array}{l} \delta E = 0 \\ T_{kl} = 0 \\ \delta T_{kl} = 0 \end{array} \right\} \text{for all } k, l = \{1, 2, \dots, N\}. \quad (3.71)$$

As stated in equation (3.64), these can be solved by the Lagrange multiplier method,

$$\delta E - \sum_{kl} \Lambda_{kl} \left[\langle \delta\psi_k | \psi_l \rangle + \langle \psi_k | \delta\psi_l \rangle \right] = 0, \quad (3.72)$$

which can be written out as

$$\begin{aligned} & \sum_k \left(\langle \delta\psi_k | \mathcal{F} | \psi_k \rangle + \langle \psi_k | \mathcal{F} | \delta\psi_k \rangle \right) \\ & - \sum_{kl} \Lambda_{kl} \langle \delta\psi_k | \psi_l \rangle - \sum_{kl} \Lambda_{kl} \langle \psi_k | \delta\psi_l \rangle = 0. \end{aligned} \quad (3.73)$$

Because k and l are only dummy-indices, we swap them to find

$$\begin{aligned} & \sum_k \left(\langle \delta\psi_k | \mathcal{F} | \psi_k \rangle + \langle \psi_k | \mathcal{F} | \delta\psi_k \rangle \right) \\ & - \sum_{kl} \Lambda_{kl} \langle \delta\psi_k | \psi_l \rangle - \sum_{lk} \Lambda_{lk} \langle \psi_l | \delta\psi_k \rangle = 0. \end{aligned} \quad (3.74)$$

Further, we extract the sum over k ,

$$\begin{aligned} & \sum_k \left[\langle \delta\psi_k | \left(\mathcal{F} | \psi_k \rangle - \sum_l \Lambda_{kl} | \psi_l \rangle \right) + \right. \\ & \left. \left(\langle \psi_k | \mathcal{F} - \sum_l \Lambda_{lk} \langle \psi_l | \right) | \delta\psi_k \rangle \right] = 0, \end{aligned} \quad (3.75)$$

and rewrite the second term in the square brackets by its complex conjugate:

$$\sum_k \left[\langle \delta\psi_k | \left(\mathcal{F} |\psi_k\rangle - \sum_l \Lambda_{kl} |\psi_l\rangle \right) + \langle \delta\psi_k | \left(\mathcal{F} |\psi_k\rangle - \sum_l \Lambda_{lk}^* |\psi_l\rangle \right) \right] = 0. \quad (3.76)$$

Earlier, we found that $\Lambda_{kl} = \Lambda_{lk}^*$. The two terms above are therefore equal. Adding them together and dividing by 2 results in the following, simplified equation:

$$\sum_k \langle \delta\psi_k | \left(\mathcal{F} |\psi_k\rangle - \sum_l \Lambda_{kl} |\psi_l\rangle \right) = 0. \quad (3.77)$$

Here comes the closing argument in the method of Lagrange multipliers: Because the variations $\delta\psi_k$ are arbitrary, the only solution is for all the integrands to be zero. We thus obtain

$$\mathcal{F}\psi_k = \sum_l \Lambda_{kl} \psi_l. \quad (3.78)$$

Unfortunately, there is no trivial way to solve this for the spin orbitals ψ_k . However, we may use our constraints to guide our solutions. We know that $T_{kl} = 0$ for all k and l . Therefore, the spin orbitals ψ_k must be orthonormal. To fulfill this requirement, we demand that the spin orbitals ψ_k are eigenvectors of the Fock operator, with eigenvalues ϵ_k , such that

$$\Lambda_{kl} = \epsilon_k \delta_{kl}. \quad (3.79)$$

This results in one eigenvalue equation for each k , known as the Hartree-Fock equations:²

$$\mathcal{F}\psi_k = \epsilon_k \psi_k. \quad (3.81)$$

These have to be solved iteratively because the Fock operator \mathcal{F} is a function of the solutions ψ_k . For each iteration, the new solutions ψ_k will, presumably, come closer to the accurate solution of the Hartree-Fock equations. Once the method has converged, the energy expectation value may be calculated.

The convergence criterion is usually defined by a threshold difference between the eigenvalues obtained by two preceding iterations. We are, however, not guaranteed that this method will converge if the initial guess is far off. A few techniques that aid convergence are discussed in Section 5.3.

The Hartree-Fock equations have (infinitely) many solutions with different eigenvalues and eigenvectors. Those corresponding to the lowest eigenvalues, ϵ_k , we will result

²To get the solutions of equation (3.78), we could transform the set of eigenstates $\{\psi_k\}$ by a unitary transformation,

$$\psi'_k = \sum_l U_{kl} \psi_l, \quad (3.80)$$

and use the new states ψ'_k in equation (3.78). However, there is no good reason to do this because we have already found a useful set of spin orbitals ψ_k .

in the best ground state approximation. Because all equations (3.81) are essentially equal for all values of k , we may be tempted to select the same solution - the one with the lowest eigenvalue - for all the equations. However, this is not an option because we still need to fulfill our constraint equations: $T_{kl} = 0$ for all k and l . To fulfill the constraints, we must therefore select the first solution when solving the equation for ψ_1 , the second solution for ψ_2 , etc.

Note that the eigenvalues ϵ_k do not correspond to the energy expectation value of the system. They are defined from the Lagrange multipliers in equation (3.79). However, we may define the energy expectation value in terms of the eigenvalues ϵ_k [12]:

$$E = \frac{1}{2} \sum_k \left(\epsilon_k + \langle \psi_k | h | \psi_k \rangle \right). \quad (3.82)$$

This is equivalent to the energy eigenvalue definition in equation (3.49).

3.4.7 Restricted Hartree-Fock

By assuming a closed-shell system, we may work with the spatial part of the spin orbitals only, and not have to worry about the spin at all. This is done by grouping the n orbitals in pairs with the same spatial part of the wave function, but opposite spin.

The spin orbital is assumed to be separable into a product of the spatial function $\phi(\mathbf{r})$ and the spin function $\xi(s)$,

$$\psi_k(\mathbf{x}) = \phi_k(\mathbf{r})\xi_k(s), \quad (3.83)$$

where ξ could either be $\alpha(s)$ or $\beta(s)$. These functions are in turn defined such that

$$\begin{aligned} \alpha(\uparrow) &= 1 \\ \alpha(\downarrow) &= 0 \\ \beta(\uparrow) &= 0 \\ \beta(\downarrow) &= 1, \end{aligned} \quad (3.84)$$

where the arrows represent positive or negative spin $1/2$ values,

$$\uparrow = +\frac{1}{2}, \quad \downarrow = -\frac{1}{2}. \quad (3.85)$$

The spatial function can be expressed in either *real space* or *reciprocal space*. The latter is useful in periodic systems and commonly applied in the literature. We will, however, stick to real space, and refer to the text of Sholl and Steckel [43] for more information on computational quantum mechanics in reciprocal space.

In the restricted Hartree-Fock (RHF) method, we assume that spin orbitals come in pairs, sharing the same spatial function ϕ , but with opposite spin functions:

$$\{\psi_{2k-1}(\mathbf{q}), \psi_{2k}(\mathbf{q})\} = \{\phi_k(\mathbf{r})\alpha(s), \phi_k(\mathbf{r})\beta(s)\}, \quad k = 1, \dots, n/2 \quad (3.86)$$

With this assumption in place, we may now integrate out the spin from the J and K operators:

$$\begin{aligned} J(\mathbf{r})\phi(\mathbf{r})\xi(s) &= \sum_l^{N_e} \iint \phi_l^*(\mathbf{r}')\xi_l(s')\phi_l(\mathbf{r}')\xi_l(s') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi(\mathbf{r})\xi(s) d\mathbf{r}' ds' \\ &= \sum_l^{N_e} \int \phi_l^*(\mathbf{r}')\phi_l(\mathbf{r}') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi(\mathbf{r})\xi(s) d\mathbf{r}' \end{aligned} \quad (3.87)$$

$$\begin{aligned} K(\mathbf{r})\phi(\mathbf{r})\xi(s) &= \sum_l^{N_e} \iint \phi_l^*(\mathbf{r}')\xi_l(s')\phi_l(\mathbf{r}')\xi_l(s') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi_l(\mathbf{r})\xi_l(s) d\mathbf{r}' ds' \\ &= \frac{1}{2} \sum_l^{N_e} \int \phi_l^*(\mathbf{r}')\phi_l(\mathbf{r}') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi_l(\mathbf{r})\xi_l(s) d\mathbf{r}' \end{aligned} \quad (3.88)$$

Note that K has a factor of $1/2$ because ξ_l is different from ξ in half the terms. When they differ, at least one of them has to be zero when they are both evaluated with the same spin s' .

Since two and two spin orbitals will share the same spatial function (with opposite spin), we introduce the new spatial operators \tilde{J} and \tilde{K} :

$$\begin{aligned} \tilde{J}(\mathbf{r})\phi(\mathbf{r}) &= \sum_l^{N_e/2} \int \phi_l^*(\mathbf{r}')\phi_l(\mathbf{r}') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi(\mathbf{r}) d\mathbf{r}' \\ \tilde{K}(\mathbf{r})\phi(\mathbf{r}) &= \sum_l^{N_e/2} \int \phi_l^*(\mathbf{r}')\phi_l(\mathbf{r}') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi_l(\mathbf{r}) d\mathbf{r}'. \end{aligned} \quad (3.89)$$

Because the sums now include only half the particles, we need to compensate for this in what we will call *the restricted Fock operator*:

$$\tilde{\mathcal{F}}(\mathbf{r}) = h(\mathbf{r}) + 2\tilde{J}(\mathbf{r}) - \tilde{K}(\mathbf{r}). \quad (3.90)$$

Note that the \tilde{K} operator is equivalent to half the K operator.

Matrix Form and the Roothaan Equation

We now introduce a set of basis states φ_p that will be joined in a linear combination to expand the spin orbitals ψ_k ,

$$\phi_k(\mathbf{r}) = \sum_p^M C_{pk}\varphi_p(\mathbf{r}). \quad (3.91)$$

This turns the Hartree-Fock equations into the following:

$$\sum_p^M C_{pk}\mathcal{F}(\mathbf{r}_1)\varphi_p(\mathbf{r}_1) = \epsilon_k \sum_p^M C_{pk}\varphi_p(\mathbf{r}_1) \quad (3.92)$$

By multiplying with $\varphi_q^*(\mathbf{r}_1)$ and integrating, we find

$$\sum_p^M C_{pk} \int \varphi_q^*(\mathbf{r}_1) \mathcal{F}(\mathbf{r}_1) \varphi_p(\mathbf{r}_1) d\mathbf{r}_1 = \epsilon_k \sum_p^M C_{pk} \int \varphi_q^*(\mathbf{r}_1) \varphi_p(\mathbf{r}_1) d\mathbf{r}_1, \quad (3.93)$$

which can be written as a matrix equation:

$$\mathbf{FC}_k = \epsilon_k \mathbf{SC}_k. \quad (3.94)$$

Here we have defined the matrices \mathbf{F} and \mathbf{S} , with elements

$$\begin{aligned} F_{pq} &= \int \varphi_q^*(\mathbf{r}_1) \mathcal{F}(\mathbf{r}_1) \varphi_p(\mathbf{r}_1) d\mathbf{r}_1 \\ &= h_{pq} + \sum_{rs}^M \sum_a^{N_e/2} C_{ra} C_{sa}^* \left(2Q_{prqs} - Q_{prsq} \right). \end{aligned} \quad (3.95)$$

$$S_{pq} = \int \varphi_q^*(\mathbf{r}_1) \varphi_p(\mathbf{r}_1) d\mathbf{r}_1, \quad (3.96)$$

while \mathbf{C}_k is a vector of the $K \times K$ matrix \mathbf{C} of the expansion coefficients C_{pk} . Here, the one-particle matrix elements h_{pq} are defined as

$$h_{pq} = \langle p | \hbar | q \rangle = \int \varphi_p^*(\mathbf{r}) \hbar \varphi_q(\mathbf{r}) d\mathbf{r}, \quad (3.97)$$

while two-particle matrix elements Q_{prqs} are defined as

$$Q_{prqs} = \langle pr | \mathcal{G} | qs \rangle = \iint \varphi_p^*(\mathbf{r}_1) \varphi_r^*(\mathbf{r}_2) \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \varphi_q(\mathbf{r}_1) \varphi_s(\mathbf{r}_2) d\mathbf{r}_1 d\mathbf{r}_2. \quad (3.98)$$

Equation (3.94) is a generalized eigenvalue equation. If we turn this into a regular eigenvalue form, it may then be solved by an ordinary eigenvalue solver. Such a procedure is explained in Section 5.3. Remember also that solving (3.94) has to be done iteratively because the solutions \mathbf{C}_k are used to build the \mathbf{F} matrix.

3.4.8 Unrestricted Hartree-Fock

In restricted Hartree-Fock (RHF), we assumed that the spin-orbitals ψ_k came in pairs where they shared the spatial function $\phi(\mathbf{r})$ and held opposite spin functions $\xi(s)$. This does, however, cause some problems for systems with odd numbers of electrons, or for molecules in dissociated states. For instance, in the case where two hydrogen atoms of H_2 are widely separated, the two spin-orbitals should be centered at each core, and not share a common spatial orbital. In these cases, it might be beneficial to vary the spins independently.

Unrestricted Hartree-Fock (UHF) allows for such independent variations, but comes at the cost higher complexity. Further, in UHF, the resulting trial wave function Ψ_T , will no longer be an eigenstate of the spin operator \mathcal{S}^2 , which results in *spin contamination* [44]. In spite of these drawbacks, UHF generally provides better results than

RHF for dissociated states. For more information on spin-contamination, Sonnenberg, Schlegel, and Hratchian [44] provides a general overview, and Eiding [2] gives details in a context similar to the one found here.

In this section, we will derive the Pople-Nesbet equations, which are similar to the Roothan equations, but are defined for UHF. The derivation in this section follows closely that of Szabo and Ostlund [45].

We first need to define some new spin orbitals that allow electrons to not only appear in pairs with the same spatial orbital and separate spins, but also with different spatial orbitals as well. We define two new sets of spin orbitals, one with spin $\alpha(s)$ and one with spin $\beta(s)$:

$$\psi_j^\alpha = \phi_j^\alpha \alpha(s), \quad (3.99a)$$

$$\psi_j^\beta = \phi_j^\beta \beta(s). \quad (3.99b)$$

Picking a spin orbital with index k from the complete set now means that we pick an orbital from either of the above sets:

$$\psi_k = \begin{cases} \psi_j^\alpha & \text{for } k \in [0, N^\alpha - 1], \\ \psi_j^\beta & \text{for } k \in [N^\alpha, N^\alpha + N^\beta - 1], \end{cases} \quad (3.100)$$

where N^α and N^β is the number of particles we choose to have the α and β spin functions, respectively.

We now need to derive the spatial equations to find $\{\phi_j^\alpha\}$ and $\{\phi_j^\beta\}$, which gives This is done by inserting (3.100) into the general Hartree-Fock equation.

$$\mathcal{F}\phi_j^\alpha(\mathbf{r})\alpha(s) = \epsilon_j^\alpha \phi_j^\alpha \alpha(s). \quad (3.101)$$

Because this leads to a set of equations that are the equivalent for both α and β , we only write out the equations for α explicitly. Next, we multiply by $\alpha^*(s)$ and integrate over the spin to find

$$\tilde{\mathcal{F}}^\alpha \phi_j^\alpha = \epsilon_j^\alpha \phi_j^\alpha(\mathbf{r}_1), \quad (3.102)$$

where we now have introduced *the unrestricted Fock operator*,

$$\tilde{\mathcal{F}}^\alpha(\mathbf{r}_1) = \int \alpha^*(s_1) \mathcal{F}(\mathbf{x}_1) \alpha(s_1) ds_1. \quad (3.103)$$

This can also be written as

$$\tilde{\mathcal{F}}^\alpha = h + [\tilde{J}^\alpha - \tilde{K}^\alpha] + \tilde{J}^\beta. \quad (3.104)$$

Notice the similarity to the restricted Fock-operator $\tilde{\mathcal{F}}$ in (3.90). The α spin functions have both direct and exchange interactions with all other α spin functions, but only direct interactions with β spin. The sums in \tilde{J}^α , \tilde{J}^β , \tilde{K}^α and \tilde{K}^β run to the number of particles with the given spin, rather than half the total number of particles:

$$\tilde{J}^\alpha \phi(\mathbf{r}) = \sum_l^{N^\alpha} \int \phi_l^{\alpha*}(\mathbf{r}') \phi_l^\alpha(\mathbf{r}') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi(\mathbf{r}) d\mathbf{r}', \quad (3.105a)$$

$$\tilde{K}^\alpha \phi(\mathbf{r}) = \sum_l^{N^\alpha} \int \phi_l^{\alpha*}(\mathbf{r}') \phi(\mathbf{r}') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi_l^\alpha(\mathbf{r}) d\mathbf{r}'. \quad (3.105b)$$

This leads to the two matrix equations:

$$\begin{aligned}\mathbf{F}^\alpha \mathbf{C}^\alpha &= \epsilon^\alpha \mathbf{S} \mathbf{C}^\alpha \\ \mathbf{F}^\beta \mathbf{C}^\beta &= \epsilon^\beta \mathbf{S} \mathbf{C}^\beta.\end{aligned}\quad (3.106)$$

These will have to be solved simultaneously because the α -equation depends on the β -orbitals, and vice versa. Further, the Fock matrix elements are now

$$F_{pq}^\alpha = h_{pq} + \sum_{rs}^M \sum_a^{N^\alpha} C_{ra}^\alpha \left(C_{sa}^\alpha \right)^* \left(Q_{prqs} - Q_{prsq} \right) + \sum_{rs}^M \sum_a^{N^\beta} C_{ra}^\beta \left(C_{sa}^\beta \right)^* Q_{prqs}, \quad (3.107)$$

$$F_{pq}^\beta = h_{pq} + \sum_{rs}^M \sum_a^{N^\beta} C_{ra}^\beta \left(C_{sa}^\beta \right)^* \left(Q_{prqs} - Q_{prsq} \right) + \sum_{rs}^M \sum_a^{N^\alpha} C_{ra}^\alpha \left(C_{sa}^\alpha \right)^* Q_{prqs}, \quad (3.108)$$

where the h_{pq} and Q_{prqs} elements are defined as in the previous section. Other than that, the procedure of solving these equations is pretty much the same as for solving the Roothan equation from the restricted case.

3.5 Choice of Single-Particle Orbitals

In the Hartree-Fock method, we have chosen the trial wave function to be a Slater-determinant. This is built up of spin-orbitals, also known as single-particle wave functions or molecular orbitals. These are further separated into spatial and spin functions, as defined in equation (3.83), and restated here:

$$\psi_k(\mathbf{x}) = \phi_k(\mathbf{r}) \xi_k(s). \quad (3.109)$$

However, we have not yet decided on the form of our spatial orbitals ϕ_k . Let's have a look at our available options.

A common approach is to expand the spatial orbital functions in a set of basis functions, which we will call “contracted basis functions”:

$$\phi_k(\mathbf{r}) = \sum_p^M C_{pk} \varphi_p(\mathbf{r}). \quad (3.110)$$

To name these “contracted” does not make much sense yet, but will soon be justified. For now, just assume that the spatial orbitals are expanded in a basis of contracted functions.

The number of possibilities for the form of these functions are endless. Quantum chemists have come up with many suggestions for the single-particle orbitals over the years. Nevertheless, there are only a few types of basis functions that are commonly used. These are all established by expanding the spatial orbitals in a basis of atomic orbitals, centered on the nuclei.

In the following sections, \mathbf{r} denotes the electron position, while \mathbf{A} is the position of nucleus A . The vector \mathbf{r}_A is the difference between the two, $\mathbf{r}_A = \mathbf{r} - \mathbf{A}$, and its components are $\mathbf{r}_A = [x_A, y_A, z_A]$.

3.5.1 Slater-Type Orbitals

An attractive type of basis functions are the Slater-type orbitals (STOs). They closely resemble the familiar hydrogenic atomic orbitals, 1s, 2s, 2p, 3s, 3p, 3d, etc. The 1s Slater-type orbital is defined as:

$$\phi_{1s}^{\text{SF}}(\gamma, \mathbf{r}_A) = \sqrt{\frac{\gamma^3}{\pi}} \exp(-\gamma r_A). \quad (3.111)$$

The Slater-type orbitals decay exponentially, and have a finite slope at $\mathbf{r}_A = 0$. At large distances, it has been shown that both molecular and atomic orbitals decay exponentially, making the Slater-type orbitals a tempting choice. However, they are computationally expensive in integral evaluations of molecules, and are therefore not our optimal choice.

We will not go further into details about the Slater-type orbitals, but refer the reader to literature of Szabo and Ostlund [45] and Cramer [8].

3.5.2 Gaussian-Type Orbitals

A more affordable alternative to Slater-type orbitals, are Gaussian-type orbitals (GTOs). In the following, we will refer to GTOs as *primitive Gaussians*. In its normalized form, the 1s Gaussian-type orbital is defined as

$$\phi_{1s}^{\text{GF}}(\alpha, \mathbf{r}_A) = \left(\frac{2\alpha}{\pi}\right)^{3/4} \exp(-\alpha r_A^2). \quad (3.112)$$

The normalization factor, $(2\alpha/\pi)^{3/4}$ ensures that the integral over the squared norm of this function is unity,

$$\int |\phi_{1s}^{\text{GF}}(\alpha, \mathbf{r}_A)|^2 d\mathbf{r} = 1. \quad (3.113)$$

As we will soon learn, the Gaussian-type orbitals result in very efficient integral evaluations. However, they do not have the same qualitative properties as STOs, such as exponential decay. This may, on the other hand, be alleviated by constructing a linear combination of Gaussian-type orbitals to approximate a Slater-type orbital. In the following, we are going to work only with primitive Gaussians. The GF superscript will hence be dropped from here on.

The reason why Gaussian-type orbitals are efficient in integral evaluation, is because the product of two Gaussian functions is another Gaussian centered at a point between the two. Any two-centered integral becomes one-centered, and four-centered integrals become two-centered.

We will use the primitive Gaussians to construct the above-mentioned *contracted Gaussians*:

$$\varphi_p(\mathbf{r}_A) = \sum_{a_p=1}^{L_p} d_{a_p} \phi_{a_p}(\alpha_{a_p}, \mathbf{r}_A). \quad (3.114)$$

Note that they all are centered on the same nucleus, and that L_p is the number of primitives in this contracted function. Remember also that the spatial orbitals ϕ_i are

a linear combination of contracted Gaussian functions, resulting in

$$\phi_k(\mathbf{r}) = \sum_p^M C_{kp} \varphi_p(\mathbf{r}_{A_p}) = \sum_p^M C_{kp} \sum_{a_p=1}^{L_p} d_{a_p} \phi_{a_p}(\alpha_{a_p}, \mathbf{r}_{A_p}). \quad (3.115)$$

Note that the different contracted functions in this expansion can be centered on separate nuclei.

At this point, we have included many levels of abstraction and expansion. It may be useful to review them all. In the below table, I have listed the different functions and their symbols:

Symbol	Name
Ψ	Total wave function
Ψ_T	Trial wave function
Ψ_{SD}	Slater determinant wave function
ψ_k	Single-particle spin-orbital
ϕ_k	Spatial part of single-particle orbital
ξ_k	Spin part of single-particle orbital
φ_p	Contracted orbitals
ϕ_a	Primitive orbitals
G_{ijk}	Gaussian function

Gaussian functions are just primitive Gaussian orbitals without the normalization factor. These will be discussed in the next section, along with integration details.

Further, we should note that a general Gaussian primitive is not of the 1s type, as shown in equation (3.112). In a Cartesian coordinate system, we can multiply the Gaussian primitive by polynomial factors, x^i , y^j and z^k , to better approximate p -, d - and f -type orbitals. The definition of a general Gaussian primitive is then as follows:

$$\phi_a^{ijk}(\alpha, \mathbf{r}_A) = D_{ijk}^\alpha x_A^i y_A^j z_A^k \exp(-\alpha r_A^2), \quad (3.116)$$

where D_{ijk}^α is the normalization constant

$$D_{ijk}^\alpha = \left(\frac{2\alpha}{\pi} \right)^{3/4} \left(\frac{(8\alpha)^{i+j+k} i! j! k!}{(2i)!(2j)!(2k)!} \right)^{1/2}. \quad (3.117)$$

With the notation used here, $\phi_{a_1}^{000}$ is a function that mimics an s-type atomic orbital and has the label a_1 . Further, $\phi_{a_2}^{100}$ mimics a p_x-type atomic orbital and has the label a_2 . The label is used to distinguish different primitive Gaussians used to approximate the same type of orbital.

Choosing the proper parameters α and d of the primitive Gaussian functions is performed beforehand, either by fitting the linear combination to a Slater-type orbital, or by a variational calculation [45].

A Minimal Basis: STO-3G

The simplest approach is to have one contracted function for each atomic orbital. This results in the so-called single-zeta basis sets. They have one contracted Gaussian for each of the 1s, 2s, $2p_x$, $2p_y$, $2p_z$, etc. atomic orbitals of the atom. This is how the STO-LG basis sets are constructed. They are fitted to Slater-type orbitals by using L primitive Gaussian functions for each contracted basis function, and the coefficients α and d are tuned to give the best fit.

In the case of the carbon atom, which has electrons in the 1s, 2s and 2p orbitals in its ground state, an STO-3G basis consists of three primitive functions for 1s, three for 2s and three for each of the 2p-orbitals. By convention, the α_a^{2s} exponents of the 2s orbitals are fitted under the constraint that they will be equal to the α_a^{2p} exponents. This is a detail that we won't have to worry about, but it's worth noting that the exponents for this reason often are listed together under the same 2sp heading.

If there are equal atoms in the system, each atom will usually contribute the same number of contracted and primitive functions to the solver. This means that in the STO-3G basis, two carbon atoms, C^A and C^B , will have a contracted function fitted to 1s with three primitives, all centered on the first atom, \mathbf{R}_A , and the same contracted function centered on the second atom, \mathbf{R}_B . Further, there will be one contracted function fitted to 2s, one to $2p_x$, one to $2p_y$, etc.

Let's illustrate this by writing out all the terms in the expansion of one of the spin-orbitals:

$$\begin{aligned} \phi_k = & C_{k1} \left(d_1 \phi_1^{000}(\alpha_1, \mathbf{r}_A) + d_2 \phi_2^{000}(\alpha_2, \mathbf{r}_A) + d_3 \phi_3^{000}(\alpha_3, \mathbf{r}_A) \right) \\ & + C_{k2} \left(d_4 \phi_4^{000}(\alpha_4, \mathbf{r}_A) + d_5 \phi_5^{000}(\alpha_5, \mathbf{r}_A) + d_6 \phi_6^{000}(\alpha_6, \mathbf{r}_A) \right) \\ & + C_{k3} \left(d_7 \phi_7^{100}(\alpha_7, \mathbf{r}_A) + d_8 \phi_8^{100}(\alpha_8, \mathbf{r}_A) + d_9 \phi_9^{100}(\alpha_9, \mathbf{r}_A) \right) \\ & + C_{k4} \left(d_7 \phi_7^{010}(\alpha_7, \mathbf{r}_A) + d_8 \phi_8^{010}(\alpha_8, \mathbf{r}_A) + d_9 \phi_9^{010}(\alpha_9, \mathbf{r}_A) \right) \\ & + C_{k5} \left(d_7 \phi_7^{001}(\alpha_7, \mathbf{r}_A) + d_8 \phi_8^{001}(\alpha_8, \mathbf{r}_A) + d_9 \phi_9^{001}(\alpha_9, \mathbf{r}_A) \right) \\ & + \text{the same terms centered at } \mathbf{R}_B \text{ with } C_{k6}, C_{k7}, \dots \end{aligned} \quad (3.118)$$

Here, ϕ_1^{000} denotes a primitive with exponents tuned to fit the 1s orbital, ϕ_4^{000} fits the 2s orbital, and similarly for the other primitives and orbitals.

Note that this expansion is just for one of the spatial orbitals ϕ_k . However, all the other spatial orbitals are equal, except for the coefficients C_{kp} . The primitives are also equal for both atoms, because the atoms are equal. For each directional 2p-orbital, the d - and α -coefficients are shared. As mentioned, the same α coefficients are shared with the 2s-orbital as well, but we'll write them out explicitly for the sake of whatever remainder there may be of readability.

Multiple-zeta Bases: 3-21G, 6-31G** and 6-311++G**

The STO-LG basis sets are useful as a starting point, but because they are fitted to Slater-type orbitals, which again have been fitted to the atomic orbitals, they won't

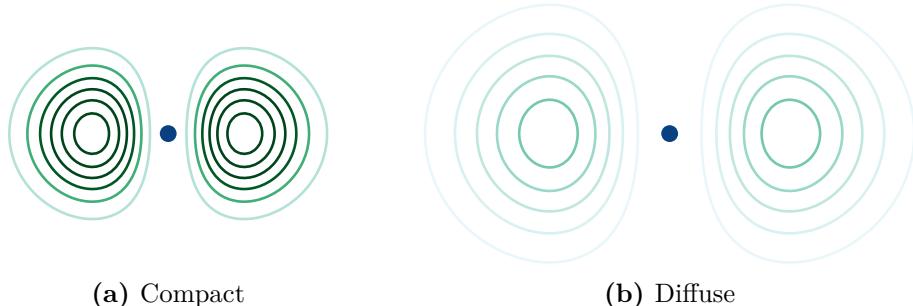


Figure 3.1: The difference between a compact and a diffuse primitive basis function. In this figure, we show the absolute value squared of two basis functions of p-type ($i = 1$ and $j = k = 0$). The exponent α of the compact function (a) has a greater value than that of the diffuse function (b). The blue dot indicates the position of the nucleus.

do well with molecular bonds. The next step is therefore to add another contracted function for each atomic orbital, resulting in what is known as double-zeta basis sets.

With two contracted Gaussians per atomic orbital, the common approach is to use primitive Gaussians with exponents slightly above or below those used in an STO-LG basis [45]. This allows expanding or contracting the spin-orbitals by varying the coefficients C_{kp} , rather than varying the coefficients d or the exponents α . By doing so, the Hartree-Fock solver may favor a more compact or diffuse orbital (see Figure 3.1). The consequence may then well be that a spatial volume gets a higher electron density, for instance forming what we know as a chemical bond.

In the following, we will divide the atomic orbitals into two categories: *Core orbitals* are those of the inner, closed shells of an atom. For an atom in a periodic table, this is the same as all the orbitals of the last atom in the preceding row - the noble gases. This is why carbon is said to have the electron configuration $[He] 2s^2 2p^2$, which means that the core orbitals of carbon are all the orbitals of helium. *Valence orbitals* are all the other orbitals. In the case of carbon, this would be $2s^2$ and $2p^2$. The superscript in p^2 means that there are 2 electrons occupying the p -orbitals. Note that hydrogen and helium only have 1s as their valence orbital and no core orbitals.

The naming convention in the Pople family [46] of Gaussian-type orbitals (3-21G, 4-31G, 6-31G, etc.) is such that the number of digits preceding the dash (-) indicate the number of contracted Gaussians used for each of the core orbitals. The digits succeeding the dash indicate the number of contracted Gaussians used for each of the valence orbitals. Further, the digit itself denotes the number of primitive Gaussians used in each contracted Gaussian.

This means that 3-21G is a basis set with one contracted function with three primitives for each of the core orbitals, one contracted function with 2 primitives for each of the valence orbitals, and one contracted function with one primitive for each of the valence orbitals. Basis sets such as 3-21G are thus not true double-zeta basis sets, because we only split the valence orbitals and not the core orbitals.

Even better representations of molecular bonds are found by including contracted functions with primitive Gaussians of even higher order. Doing this results in what is

known as polarized basis sets. For the hydrogen and helium atoms, this would amount to adding contracted functions for the p-orbitals, while for the atoms Li to F, d-orbitals. When this is done for Li and above, we add one asterisk to the basis name. Further, two asterisks are added if p-orbitals are also used for the H and He atoms.

Taking this even further, we can add diffuse functions on top of, or instead of, the polarized functions. These are typically functions with lower exponents, allowing for electron densities further away from the nuclei. The diffuse functions are necessary in situations where electrons are weakly bound to the atoms, like in anions. In the naming convention of the Pople family of Gaussian basis functions, the G is prepended by one or two pluses, to indicate that the atoms Li and above have gained another set of functions for s-orbitals and another set for p-orbitals. The H and He atoms will gain another set of s-functions. More details on basis functions are found in the literature. See, for instance, Cramer [8], Szabo and Ostlund [45] or H. Mobarhan [1].

At this point, it may be useful to note that the d-orbitals in the Pople family of basis sets, uses six Cartesian d-functions, as opposed to the regular five spherical d-functions [8].

3.6 Integration of Gaussian-Type Orbitals

In this section, we will discuss how to perform the necessary integrals to solve the Hartree-Fock equations. These are the overlap, kinetic, electron-nuclei and electron-electron integrals. We will follow closely the text of Helgaker, Jorgensen, and Olsen [47], in addition to the notation in Helgaker's slides [48, 49].

3.6.1 General Properties of Gaussian Functions

Before we get started, some general properties of Gaussian functions should be mentioned. First of all, we will ease up on our notation by introducing a new, simplified Gaussian function:

$$G_a = G_{ijk}(a, \mathbf{r}_A) = x_A^i y_A^j z_A^k \exp(-ar_A^2). \quad (3.119)$$

where $\mathbf{r}_A = \mathbf{r} - \mathbf{A}$ is the vector pointing from the nucleus center \mathbf{R}_A to the electron position \mathbf{r} . As above, the i, j and k powers control the polarization of the Gaussian-type orbital. This is related to a general primitive Gaussian function by

$$\phi^{ijk}(a, \mathbf{r}_A) = D_{ijk}^a G_{ijk}(a, \mathbf{r}_A) = D_{ijk}^a G_a. \quad (3.120)$$

Note that we used a as summing index in the previous section, while α was used as the exponent. Further, we drop the summation index, and a takes on the role of α . The change in this section is done to stay in sync with the notation of Helgaker [48].

Factorization

The three-dimensional Gaussian function may be factorized into a product of each Cartesian direction:

$$G_a = G_{ijk}(a, \mathbf{r}_A) = G_i(a, x_A) G_j(a, y_A) G_k(a, z_A) \quad (3.121)$$

where

$$G_i = x_A^i \exp(-ax_A^2) \quad (3.122)$$

and similarly for the y - and z -directions. Note that this is only a property of Cartesian Gaussian functions. It does not hold true for solid-harmonic Gaussian functions or Slater-type orbitals.

Differentiation

Differentiation of Gaussian functions is pretty straightforward, but it is also useful to note that the differentiation leads to a recurrence relation, which we will make use of later:

$$\frac{\partial G_i(a, x_A)}{\partial A_x} = -\frac{\partial G_i(a, x_A)}{\partial x} = 2aG_{i+1}(a, x_A) - iG_{i-1}(a, x_A) \quad (3.123)$$

This way we end up with two Gaussian functions that need no further differentiation. Note that all the i 's in this expression do not represent the imaginary number, but the index from G_i .

Multiplication

Multiplying two one-dimensional Gaussian functions of the zeroth order, $G_0(a, x_A)$ and $G_0(b, x_B)$, results in a new Gaussian centered between the two:

$$G_0(a, x_A)G_0(b, x_B) = \exp(-ax_A^2)\exp(-bx_B^2) = \exp(-\mu X_{AB}^2)\exp(-px_P^2). \quad (3.124)$$

Here, we have introduced the following new parameters:

$$P_x = \frac{aA_x + bB_x}{p} \quad (3.125)$$

$$p = a + b \quad (3.126)$$

$$X_{AB} = A_x - B_x \quad (3.127)$$

$$\mu = \frac{ab}{a + b}. \quad (3.128)$$

Even though this may look like a product of another two Gaussian functions, it should be noted that the first factor on the right hand side,

$$K_{AB} = \exp(-\mu X_{AB}^2), \quad (3.129)$$

is just an exponential prefactor. It is not dependent on the electron position x , and thus may be factored out of the coming integrals over x , y and z .

If we multiply two general one-dimensional Gaussian functions of i -th and j -th order, $G_i(a, x_A)$ and $G_j(a, x_B)$, we get what is known as the overlap distribution:

$$\Omega_{ij}(x) = G_i(a, x_A)G_j(b, x_B) \quad (3.130)$$

Because we may factor out K_{AB} from the above product, this reduces a two-centered integral to a one-centered integral:

$$\int \Omega_{ij} dx = K_{AB} \int x_A^i x_B^j \exp(-px_P^2) dx \quad (3.131)$$

In the following sections, we will look into the details of solving these integrals.

3.6.2 Overlap Integrals

The Hermite Gaussian functions simplify the integration of a Gaussian functions greatly. These may be written generally as

$$\Lambda_t = \frac{\partial^t}{\partial P_x^t} \exp(-px_P^2). \quad (3.132)$$

They are related to the product of two Gaussians as

$$G_i(a, x_A) G_j(b, x_B) = \sum_{t=0}^{i+j} E_t^{ij} \Lambda_t(x_P), \quad (3.133)$$

where the factor E is given by a recurrence relation, where the first term is K_{AB} :

$$E_0^{00} = K_{AB}, \quad (3.134)$$

$$E_t^{i+1,j} = \frac{1}{2p} E_{t-1}^{ij} + X_{PA} E_t^{ij} + (t+1) E_{t+1}^{ij}, \quad (3.135)$$

$$E_t^{i,j+1} = \frac{1}{2p} E_{t-1}^{ij} + X_{PB} E_t^{ij} + (t+1) E_{t+1}^{ij}. \quad (3.136)$$

Here $E_t^{ij} = 0$ if $t < 0$ or $t > i + j$. For more details, we refer the reader to the paper of McMurchie and Davidson [50], who first introduced these recurrence relations.

The benefit of rewriting these Gaussian products as Hermite expansions is that the integrals become extremely simple to solve. The integral of a Hermite Gaussian is

$$\int_{-\infty}^{\infty} \Lambda_t(x) dx = \delta_{0t} \sqrt{\frac{\pi}{p}}, \quad (3.137)$$

with δ_{0t} being the Dirac delta function. The complete overlap integral may thus be expressed as

$$S_{ab} = \langle G_a | G_b \rangle = \int G_a(\mathbf{r}) G_b(\mathbf{r}) d\mathbf{r} = S_{ij} S_{kl} S_{mn} = E_0^{ij} E_0^{kl} E_0^{mn} \left(\frac{\pi}{p} \right)^{3/2}, \quad (3.138)$$

with $G_a = G_{ikm}(\mathbf{r}_A, a)$ and $G_b = G_{jln}(\mathbf{r}_B, b)$.

3.6.3 Kinetic Integrals

The overlap integrals make it easy to calculate the kinetic integrals too. The kinetic integrals are the result of the Laplace operator, leading to terms on the form,

$$T_{ab} = -\frac{1}{2} \left\langle G_a \left| \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right| G_b \right\rangle \quad (3.139)$$

$$= T_{ij} S_{kl} S_{mn} + S_{ij} T_{kl} S_{mn} + S_{ij} S_{kl} T_{mn}, \quad (3.140)$$

where

$$T_{ij} = -\frac{1}{2} \left\langle G_i(x_A) \left| \frac{\partial^2}{\partial x^2} \right| G_j(x_B) \right\rangle. \quad (3.141)$$

Because Cartesian Gaussians are subject to the following recurrence relations upon differentiation,

$$\frac{\partial}{\partial x} G_j(x_B) = -2bG_{j+1} + jG_{j-1} \quad (3.142)$$

$$\frac{\partial^2}{\partial x^2} = 4b^2G_{j+2} - 2b(2j+1)G_j + j(j-1)G_{j-2}, \quad (3.143)$$

each of the T_{ij} -terms are expressed in terms of the one-dimensional overlap integrals:

$$T_{ij} = 4b^2S_{i,j+2} - 2b(2j+1)S_{i,j} + j(j-1)S_{i,j-2}. \quad (3.144)$$

This is all we need to calculate the kinetic integral.

3.6.4 Electron-Nuclei Integrals

The electron-nuclei interactions are defined by an integral of two Gaussians over the nucleus position. Note that the position of the nucleus C does not have to be the same as the Gaussians, A and B . The electrons-nuclei integrals are taken over all nuclei, for all combinations of primitives.

The Coulomb integral is given by

$$V_{ab} = \int \frac{G_a(\mathbf{r})G_b(\mathbf{r})(\mathbf{r})}{r_C} d\mathbf{r} = \int \frac{\Omega_{ab}(\mathbf{r})}{r_C} d\mathbf{r}. \quad (3.145)$$

Again, we can rewrite this into one-center Hermite integrals:

$$V_{ab} = \sum_{tuv} E_{tuv}^{ab} \int \frac{\Lambda_{tuv}(p, \mathbf{r}_P)}{r_C} d\mathbf{r}, \quad (3.146)$$

where $\mathbf{r}_P = \mathbf{r} - \mathbf{P}$, and $\mathbf{r}_C = \mathbf{r} - \mathbf{R}_C$. The “mass center” vector, \mathbf{P} is defined as in (3.125).

The Hermite Gaussian is defined as

$$\Lambda_{tuv}(p, \mathbf{r}_P) = \frac{\partial^{t+u+v} \exp(-pr_P^2)}{\partial P_x^t \partial P_y^u \partial P_z^v}, \quad (3.147)$$

so the Coulomb integral can be written out as

$$V_{ab} = \sum_{tuv} E_{tuv}^{ab} \frac{\partial^{t+u+v}}{\partial P_x^t \partial P_y^u \partial P_z^v} \int \frac{\exp(-pr_P^2)}{r_C} d\mathbf{r}. \quad (3.148)$$

Now, the r_C^{-1} factor is a bit cumbersome to work with. Fortunately, it is possible to rewrite this into a spherical integral. The details are provided by Helgaker [48], and we will simply state the final result here:

$$V_{ab} = \frac{2\pi}{p} \sum_{tuv} E_{tuv}^{ab} R_{tuv}^0(p, \mathbf{R}_{PC}), \quad (3.149)$$

where $\mathbf{R}_{PC} = \mathbf{P} - \mathbf{C}$. The function R_{tuv} is defined as

$$R_{tuv}^n(a, \mathbf{A}) = (-2a)^n \frac{\partial^{t+u+v} F_n(aA^2)}{\partial A_x^t \partial A_y^u \partial A_z^v}, \quad (3.150)$$

where F_n is the Boys-function, defined as

$$F_n(x) = \int_0^1 \exp(-xt^2) t^{2n} dt. \quad (3.151)$$

The Boys function is readily available for fast calculations, and may be tabulated for most of the values we need. Additionally, there is a recurrence relation between the Hermite integrals in R , that we may make use of:

$$R_{t+1,u,v}^n = t R_{t-1,u,v}^{n+1} + A_x R_{tuv}^{n+1}, \quad (3.152)$$

$$R_{t,u+1,v}^n = u R_{t,u-1,v}^{n+1} + A_y R_{tuv}^{n+1}, \quad (3.153)$$

$$R_{t,u,v+1}^n = v R_{t,u,v-1}^{n+1} + A_z R_{tuv}^{n+1}, \quad (3.154)$$

where we only need to define the starting values for all n :

$$R_{000}^n(a, \mathbf{A}) = (-2p)^n F_n(aA^2). \quad (3.155)$$

For more details, we refer the reader to the paper of McMurchie and Davidson [50], who first introduced these recurrence relations.

3.6.5 Electron-Electron Integrals

The electron-electron integrals result in a similar expression, which may be constructed by the above recurrence relations. The derivation of this expression is much the same as for the electron-nuclei integrals. For the sake of brevity, we will only state the result here:

$$V_{abcd} = \left\langle G_a(1)G_b(1) \left| \frac{1}{r_{12}} \right| G_c(2)G_d(2) \right\rangle \quad (3.156)$$

$$= \frac{2\pi^{5/2}}{pq\sqrt{p+q}} \sum_{tuv} E_{tuv}^{ab} \sum_{\tau\nu\phi} E_{\tau\nu\phi}^{cd} (-1)^{\tau+\nu+\phi} R_{t+\tau,u+\nu,v+\phi}(\alpha, \mathbf{R}_{PQ}), \quad (3.157)$$

where $\mathbf{R}_{PQ} = \mathbf{P} - \mathbf{Q}$, with \mathbf{P} being the center of mass of G_a and G_b , as defined in equation (3.125), and \mathbf{Q} equivalently for G_c and G_d .

3.7 The Way Forward

Integral evaluation was the final theoretical piece needed to solve the Hartree-Fock equations. With this machinery at hand, we do not even have to evaluate the primitive Gaussian functions directly, unless we want to visualize spatial properties, such as electron density. The Gaussian-type orbitals make integral evaluations in methods like Hartree-Fock, density functional theory (DFT), coupled-cluster and perturbation

theory convenient. In Quantum Monte Carlo (QMC) methods, however, they don't make much of a difference. There, we always have to evaluate the functions directly for each configuration of coordinates. However, using the results of a Hartree-Fock calculation can be a good starting point for QMC. The resulting Slater-determinant of a Hartree-Fock calculation may be used as input to a QMC calculation, and further enhanced by adding correlation components, such as Jastrow-factors.

Other observables may also be calculated with a similar integral evaluation machinery as above. For example, we could have found similar expressions for the dipole moment [48] and the interatomic forces (by application of the Hellmann-Feynman theorem [12]). In this thesis, however, we will mainly be concerned with the potential energy surface.

Considering that our goal is to do molecular dynamics simulations, it may seem strange to leave out the force calculations. However, as we will show in the following chapter, once the potential energy surface has been fitted by a artificial neural network (ANN), the derivatives are also readily available. Should you, on the other hand, need the forces immediately, please refer to the thesis project of H. Mobarhan [1].

Chapter 4

Interpolation with Artificial Neural Networks

While a qualified guess for the functional form of a potential can lead to good results and fast code, it requires both chemical intuition and empirical or *ab initio* data. As discussed back in Section 2.3, potentials like ReaxFF include many terms that must be fitted either to experimental data or *ab initio* calculations.

What if there are important features of a potential energy surface (PES) that are left unnoticed or ignored in the construction of such a potential? Perhaps some terms, such as bond order, are only consequences of more fundamental properties? Defining well-known terms helps us understand the underlying physics and chemistry, but a restricted focus on prior knowledge could leave out new and unpredicted physics. By allowing a more generalized functional form of the potential, we may be able to include such features. This is the strength of artificial neural networks (ANNs).

An ANN is like a black box with a given number of inputs and outputs. It is trained to return an expected output for a certain set of inputs. This is done by modifying the network connections until the output of the network matches the expected output. In our case, the input will be the positions of a set of particles, and the output will be the PES. The use of ANNs to fit PESs has been applied for over 20 years. Raff et al. [51] provide a very detailed overview of the literature on ANN fitting to molecular dynamics (MD) potentials and electronic structures.

In this chapter, we will go into details of basic ANN theory. We will discuss how they are structured, trained and tested. We follow closely the text of Steffen Nissen [52], the author of the Fast Artificial Neural Network Library (FANN). In Chapter 6, we will continue with details on how we make use of the FANN library to fit PESs, while we in Chapter 8 will apply them in molecular dynamics simulations.

4.1 An Introduction to Artificial Neural Networks

An ANN is composed of connected neurons. It is a computational model that mimics the neural network of a biological nervous system. Each neuron can fire pulses that are received by other neurons. The received pulses are used as input to an *activation*

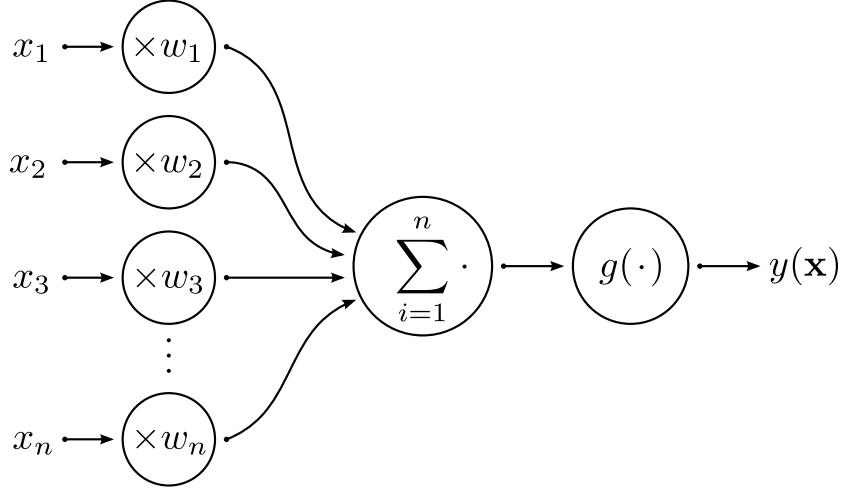


Figure 4.1: The neuron receives a number of weighted inputs. These are summed and used as the argument of the activation function, resulting in the final output of the neuron.

function, which is used to fire new pulses (Figure 4.1). The activation function mimics how a biological neuron must be excited above a threshold before returning an output.

The neuron may be defined mathematically as

$$y(\mathbf{x}) = g \left(\sum_{i=1}^n w_i x_i \right). \quad (4.1)$$

Here \mathbf{x} is the input received by the neuron from its n dendrites (x_1, \dots, x_n), w_i are a set of weights for each input and $y(\mathbf{x})$ is the output. The function g is the activation function. This should be differentiable, for reasons that will become evident when we discuss network training in Section 4.3. Sigmoid and Gaussian functions are typically used [52, 53]. Sigmoid functions are S-shaped, while Gaussian functions are bell-shaped (Figure 4.2). Two examples of sigmoid functions (modified with an offset t and a scaling parameter s) are the logistic function,

$$g(x) = \frac{1}{1 + e^{-2s(x+t)}}, \quad (4.2)$$

and the hyperbolic tangent,

$$g(x) = \tanh(s(x+t)). \quad (4.3)$$

The Gaussian function is defined as

$$g(x) = e^{-s^2 x^2}. \quad (4.4)$$

To train the neural network, we will vary the weights w_i together with the parameters s and t .

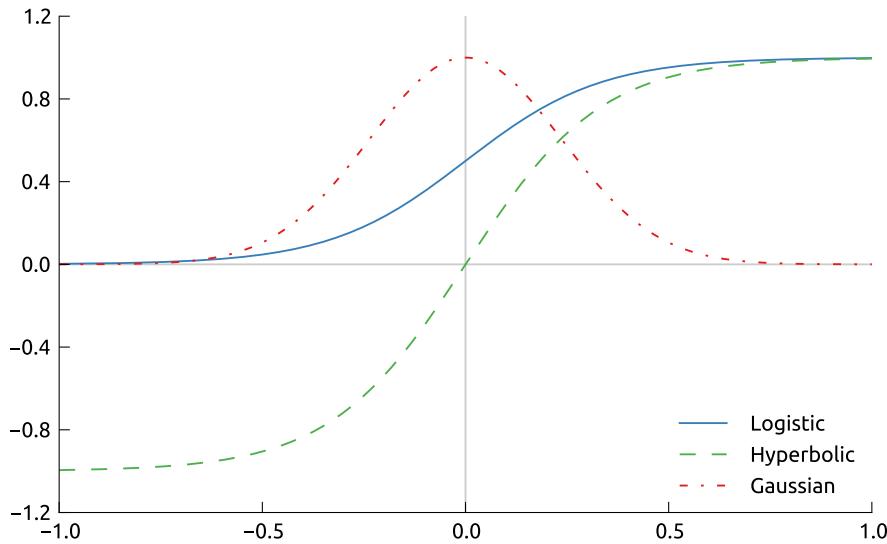


Figure 4.2: Commonly used activation functions are the logistic function, $g(x) = 1/(1 + \exp(-2s(x + t)))$, the hyperbolic tangent, $g(x) = \tanh(s(x + t))$, and the Gaussian function $g(x) = e^{-s^2x^2}$. The functions are modified from their traditional forms by the inclusion of the offset and scaling parameters t and s , respectively.

4.2 Network Topology

A neural network can be structured in many ways, as long as we have the same number of inputs and outputs in the network as in the function we want to approximate. It is common to structure the network in layers [52], where the output of all neurons in one layer are connected to the inputs of the neurons in the next layer. It can be showed [54] that a network with only three layers - one input, one output and one hidden layer in between - can approximate any continuous function.

4.2.1 Feedforward Networks

When the output of each layer is sent only to succeeding layers, we get a *feedforward neural network* (Figure 4.3). This is divided into an input layer, an output layer and a number of hidden layers. The standard feedforward network passes the output of one layer to the next layer. However, it is possible to create shortcut networks, where the output of one layer may be directed to any succeeding layer. The output of a feedforward network is swiftly calculated because information only needs to be propagated through each layer once.

4.2.2 Recurrent Networks

Networks with feedback loops are known as *recurrent neural networks*. Calculations of these networks need multiple iterations. The main benefit of feedback networks is that they possess internal memory that can be used in time-dependent computations. The

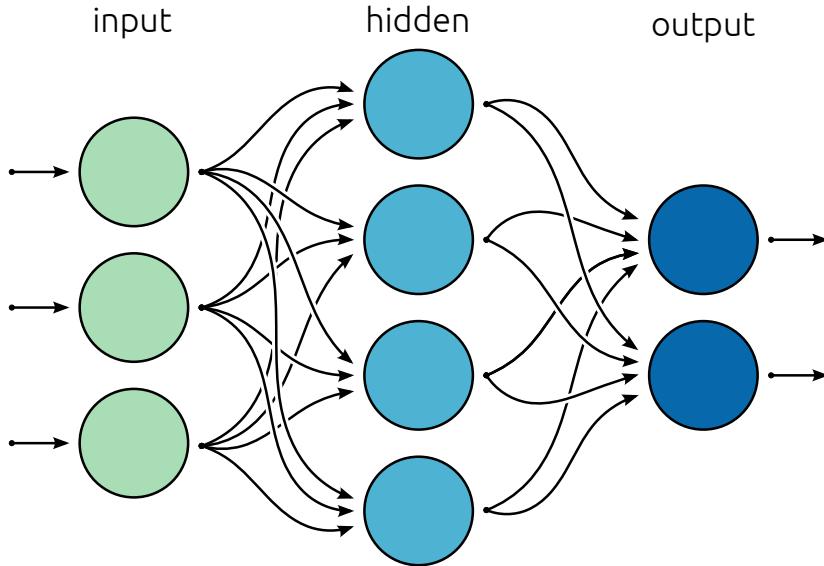


Figure 4.3: An example of a feedforward neural network with three layers. In this example, there are three input neurons, four hidden neurons and two output neurons. A general neural network can have multiple hidden layers and an arbitrary number of neurons in each layer.

network is not only affected by the current input, but also previous inputs. This is also useful in handwriting recognition [55]. However, such networks need to be handled with great care. Feedback loops tend to either decay the propagated signal, or make it grow exponentially. Several techniques need to be applied to structure the network to fix the shortcomings of recurrent neural networks. A recent overview of useful techniques has been given by Graves et al. [55].

4.3 Training

To train a neural network, we modify the connection weights until the output of the network matches the expected output. In our training, we first define a list of expected output values for given input values. This list is divided into three subsets: One is used for training, one for validation and one for final testing [56].

A common problem is that the network can become specialized to the training data, and not be able to reproduce data outside this set. This is known as *overfitting* and is resolved by testing the network on the validation data at certain intervals in the training process. If the network is getting better at predicting both the validation data and the training data, we let the training continue. If the network starts to lose its generality (such that the error in the validation set grows), we stop the training and store the current version of the network.

Unfortunately, the training could stop before we have reached the desired error threshold. Therefore, multiple networks are trained at the same time, and the best is

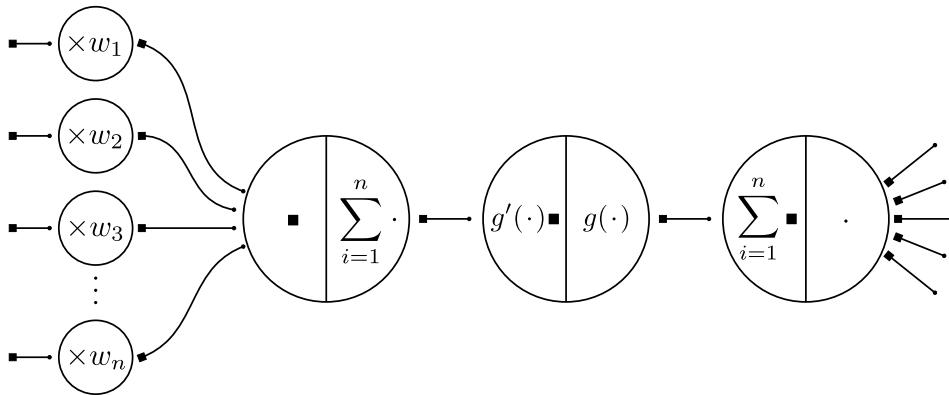


Figure 4.4: Each node in the neuron can be split in two. The right side of a node is used under function composition, and the left side of a node is used under backpropagation. The square symbol ■ represents the value propagated back from nodes to the right. The small dot · represents the value sent from nodes to the left. Note that the derivative of the activation function is taken with the left value · as argument and multiplied by the right value ■. The left value · must therefore be stored on the node during function composition. The weight nodes are equal in both backpropagation and function composition.

picked once the training is complete. This is done by testing the networks on the final data set.

4.3.1 Backpropagation and the Gradient of the Output

To train the network, we need a method to modify the connection weights. A common method used in neural network training is backpropagation [52, 53]. In backpropagation, the output of the network is calculated and compared to the desired output. The error is then propagated backwards to adjust the weights – hence the name. Backpropagation is a type of gradient descent, and inherits some common limitations of this group of algorithms, such as being slow close to the minimum. Other methods are therefore often used to build upon basic backpropagation, such as resilient backpropagation (RPROP) and Quickprop [52].

Let's first look at how backpropagation is used to calculate the gradients of the inputs of the network. In the next section, we will return to how the error is backpropagated to adjust the weights. Following a modified version of the graphical approach of Rojas [53], we will remodel our neuron by using a *backpropagation diagram*. This is illustrated in Figure 4.4. Each node in the neuron now has a right and left side. The right side is the same as in our original neuron model, and is used in function composition. The left side is used for backpropagation. Under function composition, there will be flow from left to right in the diagram, while during backpropagation, there will be flow from right to left.

The method of backpropagation is an implementation of the chain rule, and will allow us to calculate the partial derivatives (and thereby the gradient) of the network.

First, function composition is performed, where each neuron sums all its inputs multiplied with their weights, and then uses this sum as the argument of the activation function. Then the backpropagation begins at the right hand side of the network, and as we progress leftwards, the derivative of each neuron is evaluated.

The derivative of a neuron is the derivative of the activation function with respect to its argument (the sum of all weighted inputs). The argument was obtained in the function composition stage. We need to store both the argument and the output of the neuron during function composition. As we progress leftwards in backpropagation, the derivatives of the preceding neurons are multiplied with the current neuron's derivative, just like in the chain rule. The output neurons have no neurons to their right. Therefore, their derivatives are simply multiplied by 1.

The derivative of an output neuron k with respect to its input is

$$\delta_k = g'_k(x_k) = g'_k \left(\sum_j w_{jk} y_j \right). \quad (4.5)$$

where the output of a preceding neuron j is y_j and the input to a neuron k has been defined as

$$x_k = \sum_j w_{jk} y_j. \quad (4.6)$$

We propagate backwards by multiplying this value with the derivatives of the neurons to its left and the connections weights. Each of these neurons stores the resulting value:

$$\delta_j = g'_j(x_j) w_{jk} \delta_k = g'_j(x_j) w_{jk} g'_k(x_k). \quad (4.7)$$

This is then propagated further backwards. When there are multiple values propagated back to the same neuron, the values are summed (Figure 4.4). So in the preceding layer, a neuron will obtain the following value:

$$\delta_i = g'_i(x_i) \left(\sum_j w_{ij} \delta_j \right) w_{jk} \delta_k = g'_i(x_i) \left(\sum_j w_{ij} g'_j(x_j) \right) w_{jk} g'_k(x_k). \quad (4.8)$$

This is continued until we reach the input neuron we wish to derive with respect to. At that point, we will have found the derivative of the output neuron with respect to the input neuron.

One amazing feature of this method is that we will retrieve all the terms in the gradient for one output neuron, all at once. The chain rule works in the way that there is only one factor dependent on the variable we take the derivative with respect to. When the derivatives of the whole network is calculated, the derivative of the output with respect to a given input is actually stored on that input neuron. Note that this is not the case in the other end of the chain. We must only propagate back the derivative of one output neuron at the time - all others are ignored - otherwise we would end up with a peculiar sum of the derivatives of the outputs at each input neuron.

The backpropagation procedure is implemented in FANN, but only as an internal function used in the training of the network. Because the function is internal, FANN

does not provide an interface to calculate the gradients of the network outputs. Fortunately, with FANN being an open-source library, we are able to extend the library with this functionality by doing a few modifications to the backpropagation function. This will be discussed in Section 6.4.

4.3.2 Using the Gradient of the Error to Adjust the Weights

The first step in training with backpropagation is to calculate the output of the network by propagating the input through the network. The mean square error (MSE) is then calculated from the output and propagated back through the network while adjusting the weights to reduce the error. The error in the value of output neuron k can be written as

$$e_k = d_k - y_k, \quad (4.9)$$

where d_k is the desired value and y_k is the current output. This allows us to find the change we need in the input of the output neurons. We now multiply the derivative of all the output neurons with their respective errors to find

$$\delta_k = e_k g'_k(x_k), \quad (4.10)$$

where $g'_k(x_k)$ is the derivative of the activation function of output neuron k with respect to its argument. As above, the argument is the input $x_k = \sum_j w_{jk}y_j$ from the preceding layer. This is propagated back to find the δ_j values of the previous layer:

$$\delta_j = g'_k(x_j) \sum_k w_{jk} \delta_k = g'_j(x_j) \sum_k w_{jk} e_k g'_k(x_k). \quad (4.11)$$

Note that the sum in the above equation includes the derivatives of all the output neurons. When we targeted the derivative of the output with respect to the inputs, we only included one output neuron. When training, however, we propagate back all the errors to the entire network at once.

This is repeated until we have found δ values for all the neurons. Once the δ values are found, each weight can be adjusted by

$$\Delta w_{jk} = \delta_j y_k, \quad (4.12)$$

to obtain a better result. The whole method is then repeated until the MSE of the network reaches a given threshold, or the training stops due to overfitting.

4.4 Artificial Neural Networks in Molecular Dynamics

The target of this thesis is to use ANNs to approximate potentials from *ab initio* calculations on molecules. We have this far discussed the fundamental theory of molecular dynamics, Hartree-Fock and ANNs. Soon we'll discuss the implementation of a machinery that does molecular dynamics calculations on systems of arbitrary atoms and geometries. This can be done with little or no intervention from the user.

After everything has been implemented, our final work flow begins with choosing the configurations to calculate for the two- and three-body potentials. This is done

by calculating the potential energy surface for two- and three-body systems with the Hartree-Fock code for a large range of distances and angles.

Then we set up many configurations within these bounds, calculate the energies using the Kindfield code and train ANNs to fit the potential. The best one is picked and verified by visual and numeric comparison of the function fit. Finally, we set up a molecular dynamics system, load the ANN and run the simulation. The results of the final simulation may then be analyzed and compared with data from other studies or experiment.

Part II

Advanced Theory, Implementation and Results

Chapter 5

A General Hartree-Fock Solver for a Gaussian-Type Basis

Before moving on to molecular dynamics (MD) simulations with artificial neural networks (ANNs), we need a method to calculate potential energy surfaces (PESs). We will implement a Hartree-Fock code for this purpose. In this chapter, we will deal with the implementation details of the theory outlined in Sections 3.4 to 3.6. This has been used to develop the Kindfield program. The main flow of the program is shown in Figure 5.1 for reference.

The program starts by reading a YAML [57] configuration file. In this file, the atom types and their positions are listed together with the basis set name. The `ElectronSystem` class is responsible for keeping track of this information. The `GaussianCore` and `TurboMoleParser` classes (not depicted in the figure) are used to load basis set files containing the coefficients of the primitive Gaussian basis functions. From this information, the contracted Gaussian basis functions are created, and passed on to the `ElectronSystem` class.

The basis set files are downloaded from a website named Basis Set Exchange [58]. Here, numerous basis sets are kindly provided in various formats. We will focus on the format used by Turbomole [59] because this has the most intuitive structure.

The next step is to calculate the integral matrices. For this purpose, we make use of the recurrence relations defined in Section 3.6. Further, we set up and solve the eigenvalue equations of restricted Hartree-Fock (RHF) and unrestricted Hartree-Fock (UHF). This is performed iteratively, and once convergence is reached, we harvest our results. In the following sections, we will discuss the implementation details.

The source code of the Kindfield program is available online (see page 3 for information on how to obtain the source code).

5.1 Loading data from the Basis Set Exchange

The website Basis Set Exchange [58] provides the weights and coefficients of Gaussian type orbitals for a range of atoms. These are gathered from the literature and are available in many formats. However, before we make use of this data, lets recap our

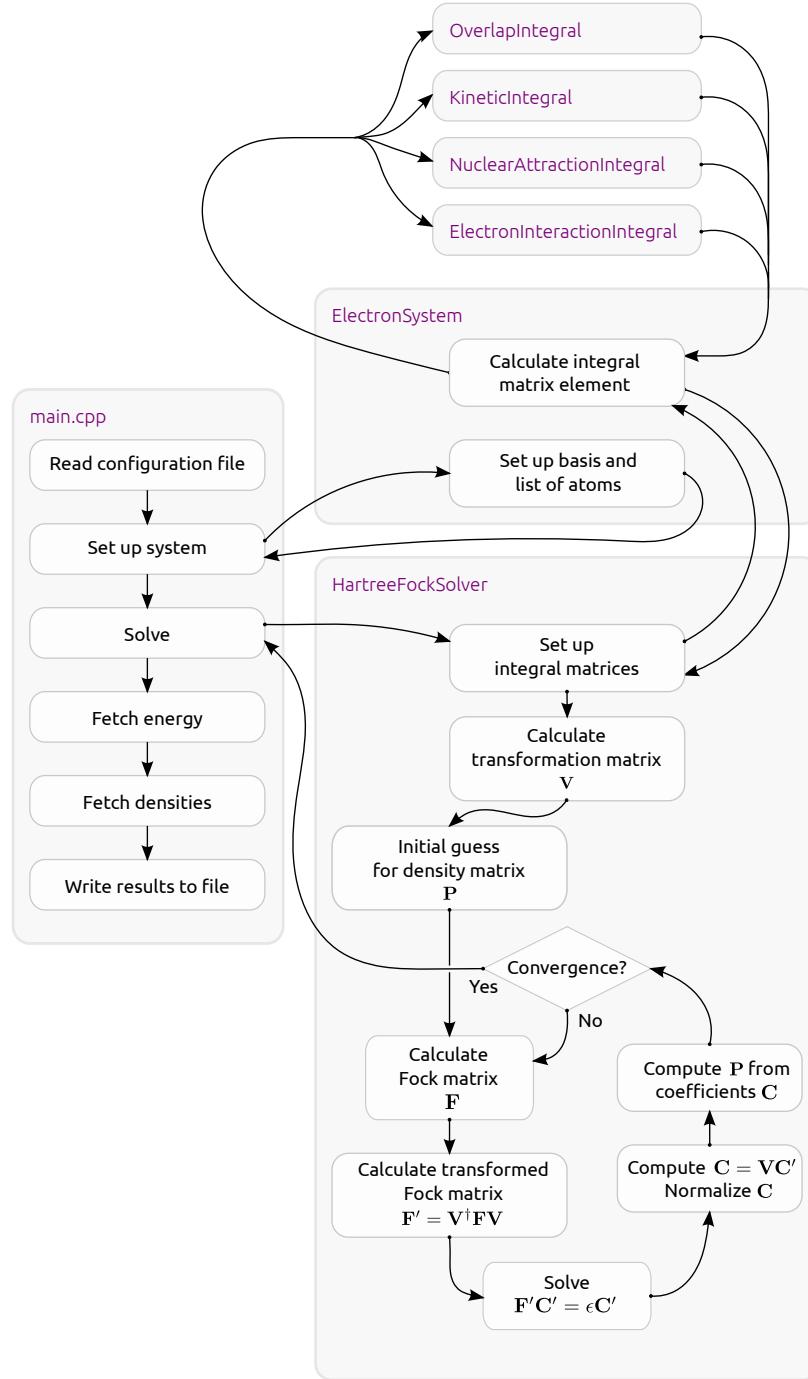


Figure 5.1: Flow chart of the Kindfield program. The program reads a YAML [57] configuration file containing the atom types, positions, and basis set name. The `HartreeFockSolver` class sets up the necessary integral matrices by calling the methods of `ElectronSystem` to calculate the matrix elements. The Fock matrix is constructed from the integral matrices and an initial guess on the density matrix. Finally, the Hartree-Fock equations are solved iteratively until the method has converged.

definitions from previous chapters.

The trial wave function we use is the Slater determinant defined in Section 3.2:

$$\Psi_T = \frac{1}{\sqrt{N_e!}} \begin{vmatrix} \psi_1(\mathbf{x}_1) & \psi_1(\mathbf{x}_2) & \cdots & \psi_1(\mathbf{x}_{N_e}) \\ \psi_2(\mathbf{x}_1) & \psi_2(\mathbf{x}_2) & \cdots & \psi_2(\mathbf{x}_{N_e}) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_{N_e}(\mathbf{x}_1) & \psi_{N_e}(\mathbf{x}_2) & \cdots & \psi_{N_e}(\mathbf{x}_{N_e}) \end{vmatrix}. \quad (5.1)$$

The spin-orbitals ψ_k are separated into a spatial orbital function and a spin function:

$$\psi_k(\mathbf{x}) = \phi_k(\mathbf{r})\xi_k(s). \quad (5.2)$$

These may in turn be written as a linear combination of *contracted* Gaussian functions

$$\phi_k(\mathbf{r}) = \sum_p^M C_{pk} \varphi_p(\mathbf{r}), \quad (5.3)$$

which are a linear combination of *primitive* Gaussian functions

$$\varphi_p(\mathbf{r}_A) = \sum_{a_p=1}^{L_p} d_{a_p} \phi_{a_p}(\alpha_{a_p}, \mathbf{r}_A), \quad (5.4)$$

where

$$\phi_a^{ijk}(\alpha, \mathbf{r}_A) = D_{ijk}^\alpha x_A^i y_A^j z_A^k \exp(-\alpha r_A^2). \quad (5.5)$$

The Basis Set Exchange website provides the d_a coefficients (non-normalized, see below) and the α_a exponents of the primitive Gaussian functions. A number of formats are available, but we'll stick with Turbomole [59] because it is quite comprehensible at first sight.

Below, the Turbomole basis file for the 3-21G basis of oxygen is shown:

```
$basis
*
o 3-21G
*
3 s                      # <-- results in 1 contracted (1s)
322.0370000   0.0592394
48.4308000   0.3515000
10.4206000   0.7076580
2 s                      # <-- results in 1 contracted (2s)
7.4029400    -0.4044530
1.5762000    1.2215600
1 s                      # <-- results in 1 contracted (2s)
0.3736840    1.0000000
2 p                      # <-- results in 3 contracted (2px,2py,2pz)
7.4029400    0.2445860
1.5762000    0.8539550
1 p                      # <-- results in 3 contracted (2px,2py,2pz)
0.3736840    1.0000000
*
$end
```

The first four lines denotes the header information of the file, telling us that this is the 3-21G basis for the oxygen atom. Following these are the definitions of primitive basis functions that will be contracted. A line with a number and a letter, such as “3 s” tells us that the following lines define a contracted function of the *s*-orbital type with 3 primitives (not to be confused with the 3s atomic orbital!). This next line corresponds to the number 3 in the 3-21G basis name, which should provide 1 contracted function with 3 primitives per core orbital of the atom. Oxygen has only one core orbital, namely 1s, and thus only 1 contracted function with 3 primitives in total. The next three lines are structured such that the first column is the exponent α_a , and the second column is the *non-normalized* weight coefficient d_a .

By non-normalized, we mean that we need to multiply d_a by the normalization factor we defined in Section 3.5:

$$D_{ijk}^{\alpha} = \left(\frac{2\alpha}{\pi} \right)^{3/4} \left(\frac{(8\alpha)^{i+j+k} i! j! k!}{(2i)!(2j)!(2k)!} \right)^{1/2}. \quad (5.6)$$

Although d and D are treated separately in theory, we will only store the result of d_a multiplied with $D_{ijk}^{\alpha_a}$ in the code. Therefore, the `m_weight` member of the `GaussianPrimitiveOrbital` class, refers to the following product:

$$w_a = D_{ijk}^{\alpha_a} d_a. \quad (5.7)$$

The next lines, starting with “2 s”, define the contracted functions for the valence orbitals of oxygen. Oxygen has 4 valence orbitals: The 2s orbital and the three *p*-orbitals, $2p_x$, $2p_y$ and $2p_z$. The lines starting with “2 s” and “1 s” define one contracted each for the 2s orbital, with 2 and 1 primitive functions, respectively. The line starting with “2 p” defines three contracted functions, one for each of the $2p_x$, $2p_y$ and $2p_z$ orbitals. They all have two primitives each. The line starting with “1 p” defines another three contracted functions, one for each of the $2p_x$, $2p_y$ and $2p_z$ orbitals. They all have one primitive each.

This elaborate discussion should hopefully provide you with enough insight to understand the Turbomole file format. The `TurboMoleParser` class has been implemented to parse this format using Boost.Regex [60]. The `TurboMoleParser::load` function is shown below in a stripped-down version (this won’t compile, see the source code for a complete version):

```
bool TurboMoleParser::load(string fileName)
{
    setlocale(LC_ALL, "C"); // Ensure that numbers are read locale-independently
    ifstream dataFile(fileName);
    string line;
    while (getline(dataFile, line))
    {
        bool ignoredLine = false;
        ignoredLine |= regex_match(line, regex("#.*"));
        ignoredLine |= regex_match(line, regex("$basis.*"));
        ignoredLine |= regex_match(line, regex("$end.*"));
        ignoredLine |= regex_match(line, regex("\\".*"));
    }
}
```

```

    if(ignoredLine) {
        continue;
    }
    smatch what;
    regex basisRegex("^\\s*([a-zA-Z]+)");
    while(regex_search(line, what, basisRegex)) {
        atomTypeAbbreviation = string(what[1]);
        m_atomType = HF::abbreviationToAtomType(atomTypeAbbreviation);
        break;
    }
    regex orbitalRegex("\\s*([0-9])\\s*([spdf])\\s*");
    if(regex_search(line, what, orbitalRegex)) {
        mergePrimitivesIntoContracted();
        if(what[2] == "s") {
            m_currentOrbitalType = HF::sOrbitalType;
        } else if(what[2] == "p") {
            ...
        }
    }
    regex exponentWeightRegex("\\s*(-?[0-9]+\\.\\.?-[0-9]+)\\s*(-?[0-9]+\\.\\.?-[0-9]+)\\s*");
    // matches lines like "2.1 4.9"
    if(regex_search(line, what, exponentWeightRegex)) {
        double exponent = stod(string(what[1]));
        double weight = stod(string(what[2]));
        GaussianPrimitiveOrbital primitive;
        primitive.setWeight(weight);
        primitive.setExponent(exponent);
        m_collectedPrimitiveBasisFunctions.push_back(primitive);
    }
}
}
}

```

This function will read each line of the file and construct `GaussianPrimitiveOrbital` objects that are pushed to a C++ `vector`. Whenever a line indicating a new contracted function (such as `3 s`) is read, the collected primitives are merged into a `GaussianContractedOrbital` object and stored in a new `vector`. This `vector` is made available on the `TurboMoleParser` object once the `load` function has completed.

5.2 Calculating the Gaussian-Type Orbital Integrals

For brevity, we will only discuss the RHF implementation in the following. UHF should be a straightforward extension to this.

To solve the eigenvalue equation of RHF,

$$\mathbf{FC}_k = \epsilon_k \mathbf{SC}_k, \quad (5.8)$$

we first need to set up the Fock matrix,

$$F_{pq} = \int \varphi_q^*(\mathbf{r}_1) \mathcal{F}(\mathbf{r}_1) \varphi_p(\mathbf{r}_1) d\mathbf{r}_1, \quad (5.9)$$

where

$$\tilde{\mathcal{F}}(\mathbf{r}) = h(\mathbf{r}) + 2\tilde{J}(\mathbf{r}) - \tilde{K}(\mathbf{r}), \quad (5.10)$$

and

$$\begin{aligned}\tilde{J}(\mathbf{r})\phi(\mathbf{r}) &= \sum_l^{N_e/2} \int \phi_l^*(\mathbf{r}') \phi_l(\mathbf{r}') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi(\mathbf{r}) d\mathbf{r}', \\ \tilde{K}(\mathbf{r})\phi(\mathbf{r}) &= \sum_l^{N_e/2} \int \phi_l^*(\mathbf{r}') \phi(\mathbf{r}') \frac{1}{|\mathbf{r}' - \mathbf{r}|} \phi_l(\mathbf{r}) d\mathbf{r}'.\end{aligned}\quad (5.11)$$

Further, we need to set up the overlap matrix

$$S_{pq} = \int \varphi_q^*(\mathbf{r}_1) \varphi_p(\mathbf{r}_1) d\mathbf{r}_1. \quad (5.12)$$

All these expressions were derived and defined in Section 3.4.7.

The integrals above have Gaussian contracted functions φ in their integrands. These will be expanded by Gaussian primitives, resulting in a series of primitive Gaussian integrals. The Fock matrix is set up in the `RestrictedHartreeFockSolver` class. It has a pointer to the `ElectronSystem` class, and probes this for calculations of the elements of the integral matrices. Initializing the `RestrictedHartreeFockSolver` class thus involves calling functions that set up all these integrals:

```
void HartreeFockSolver::setup() {
    setupOverlapMatrix();
    setupUncoupledMatrix();
    setupCoupledMatrix();
}
...
void RestrictedHartreeFockSolver::setup() {
    HartreeFockSolver::setup(); // common for RHF and UHF
    resetCoefficientMatrix();
    resetFockMatrix();
    setupDensityMatrix();
}
```

In the case of the overlap matrix, the `ElectronSystem` is requested for each element in the matrix:

```
void HartreeFockSolver::setupOverlapMatrix() {
    ElectronSystem* f = m_electronSystem;
    uint nk = f->nBasisFunctions();
    m_overlapMatrix = zeros(nk,nk);
    for(uint p = 0; p < nk; p++) {
        for(uint q = 0; q < nk; q++) {
            m_overlapMatrix(p,q) = f->overlapIntegral(p, q);
        }
    }
}
```

For implementation details of the other `setup*` functions, please refer to the source code of the Kindfield program (see page 3 for information on how to obtain the source code).

`GaussianSystem` inherits from `ElectronSystem`, and makes use of the `Gaussian*Integral` classes to calculate the overlap, kinetic, electron-nucleus and electron-electron integrals of primitives. We found efficient ways to calculate the primitive integrals in Section 3.6. The overlap integral is given by

$$S_{ab} = S_{ij} S_{kl} S_{mn} = E_0^{ij} E_0^{kl} E_0^{mn} \left(\frac{\pi}{p} \right)^{3/2}, \quad (5.13)$$

where we need to calculate the Hermite coefficients E_0^{ij} for all three Cartesian directions.

The Hermite expansion coefficients E_t^{ij} were defined as

$$E_0^{00} = K_{AB}, \quad (5.14)$$

$$E_t^{i+1,j} = \frac{1}{2p} E_{t-1}^{ij} + X_{PA} E_t^{ij} + (t+1) E_{t+1}^{ij}, \quad (5.15)$$

$$E_t^{i,j+1} = \frac{1}{2p} E_{t-1}^{ij} + X_{PB} E_t^{ij} + (t+1) E_{t+1}^{ij}. \quad (5.16)$$

where

$$K_{AB} = \exp(-\mu X_{AB}^2), \quad (5.17)$$

and

$$P_x = \frac{aA_x + bB_x}{p} \quad (5.18)$$

$$p = a + b \quad (5.19)$$

$$X_{AB} = A_x - B_x \quad (5.20)$$

$$\mu = \frac{ab}{a+b}. \quad (5.21)$$

Here, the nucleus positions A_x and B_x also define the centers of the Gaussian primitive functions.

Figure 5.2 (with its caption) illustrates how the recurrence relations may be applied in practice. It shows how to build the entire E_t^t cube, even though we don't need all elements for the overlap integrals. However, all will be needed for the upcoming Coulomb integrals, and reusing the same implementation here saves us some work later.

The full implementation of the recurrence relations is found in the `HermiteExpansionCoefficients` class. This class holds three Armadillo [61] `cube` objects, with one E_t^{ij} for each Cartesian direction. One object of the `HermiteExpansionCoefficients` type is stored on each `GaussianOverlapIntegral`, and is responsible for setting up the E_t^{ij} cubes by use of the recurrence relations.

When the `GaussianSystem` class gets a request to calculate the overlap integral of two contracted functions φ_p and φ_q , it will loop over all combinations of the primitive functions for each contracted:

```
double GaussianSystem::overlapIntegral(int p, int q)
{
```

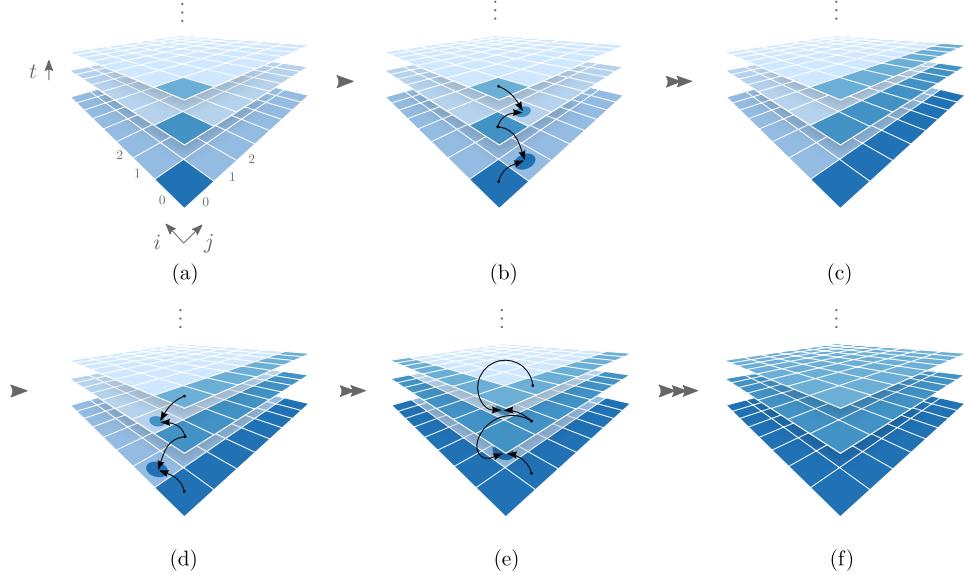


Figure 5.2: To build the Hermite expansion coefficient cube E_{ij}^t , first (a) calculate $E_{00}^0 = K_{AB}$ and set $E_{00}^t = 0$ for all $t > 0$. Then (b) iterate over all j for $i = 0$ and use the recurrence relations to (c) complete this “wall”. (d) Continue with the wall for which $j = 0$, then (e) $j = 1, \dots$, until (f) all elements are computed. The vertical dots above each figure indicate that this procedure is equivalent for any number of layers, i.e., for the highest t needed.

```

double result = 0;
const GaussianContractedOrbital& pBF = m_basisFunctions.at(p);
const GaussianContractedOrbital& qBF = m_basisFunctions.at(q);
for(const GaussianPrimitiveOrbital& pP : pBF.primitiveBasisFunctions()) {
    for(const GaussianPrimitiveOrbital& qP : qBF.primitiveBasisFunctions()) {
        m_overlapIntegral.set(pBF.corePosition(), qBF.corePosition(), pP,
                             qP);
        result += pP.weight() * qP.weight() *
                  m_overlapIntegral.overlapIntegral(pP, qP);
    }
}
return result;
}

```

The call to `GaussianOverlapIntegral::overlapIntegral(a, b)` will return the actual integral of the two primitives ϕ_a and ϕ_b :

```

double GaussianOverlapIntegral::overlapIntegral(int dim, int iA, int iB)
{
    double p = m_exponentSum;
    const cube &E_dim = (*m_hermiteExpansionCoefficient)[dim];
    return E_dim(iA,iB,0) * sqrt(M_PI / p);
}
double GaussianOverlapIntegral::overlapIntegral(
    const GaussianPrimitiveOrbital& primitiveA,

```

```

        const GaussianPrimitiveOrbital& primitiveB) {
    int iA = primitiveA.xExponent();
    ...
    return overlapIntegral(0, iA, iB) * overlapIntegral(1, jA, jB) *
           overlapIntegral(2, kA, kB);
}

```

The kinetic integral is calculated by a function similar to the one above, namely `GaussianSystem::kineticIntegral`. This will in turn call the functions of a `GaussianKineticIntegral` object to return the actual functions. The expression of the kinetic integral includes a few evaluations of the overlap integral,

$$T_{ab} = T_{ij} S_{kl} S_{mn} + S_{ij} T_{kl} S_{mn} + S_{ij} S_{kl} T_{mn} \quad (5.22)$$

with

$$T_{ij} = 4b^2 S_{i,j+2} - 2b(2j+1)S_{i,j} + j(j-1)S_{i,j-2}, \quad (5.23)$$

which is why the `GaussianKineticIntegral` object holds a `GaussianOverlapIntegral` to perform these evaluations. A possible optimization that could be done here is to store the directional overlap matrix elements S_{ij} , when setting up S_{ab} . This has not been done, due to the added code complexity, and because calculating the overlap and kinetic integrals amounts to very little computation time in comparison to the electron-electron integrals.

The first Coulomb integral, for the electron-nucleus interactions, is set up similarly to the above integral matrices, and is defined for primitive evaluations in `GaussianColoumbAttractionIntegral`. The expression for this integral was found in Section 3.6 to be

$$V_{ab} = \frac{2\pi}{p} \sum_{tuv} E_{tuv}^{ab} R_{tuv}^0(p, \mathbf{R}_{PC}), \quad (5.24)$$

where R_{tuv}^0 are the Hermite integrals, defined by the recurrence relations

$$R_{t+1,u,v}^n = t R_{t-1,u,v}^{n+1} + A_x R_{tuv}^{n+1}, \quad (5.25)$$

$$R_{t,u+1,v}^n = u R_{t,u-1,v}^{n+1} + A_y R_{tuv}^{n+1}, \quad (5.26)$$

$$R_{t,u,v+1}^n = v R_{t,u,v-1}^{n+1} + A_z R_{tuv}^{n+1}, \quad (5.27)$$

where $R_{000}^n(a, \mathbf{A}) = (-2p)^n F_n(aA^2)$ and $F_n(x)$ is the Boys function.

The Hermite integrals are implemented in the `HermiteIntegral` class, which sets up and uses the above recurrence relation to store the necessary values for R_{tuv}^0 , needed for our integrals. See Figure 5.3 for a graphical explanation of how this is set up. Note that the number of elements shown calculated in this figure is higher than necessary for cases where $u \neq v$. See the source code for an implementation that takes this into account to reduce the computational cost.

The electron-electron integrals are calculated by the `GaussianElectronInteractionIntegral` class. It sets up two E 's and one R to evaluate the expression,

$$V_{abcd} = \frac{2\pi^{5/2}}{pq\sqrt{p+q}} \sum_{tuv} E_{tuv}^{ab} \sum_{\tau\nu\phi} E_{\tau\nu\phi}^{cd} (-1)^{\tau+\nu+\phi} R_{t+\tau, u+\nu, v+\phi}(\alpha, \mathbf{R}_{PQ}). \quad (5.28)$$

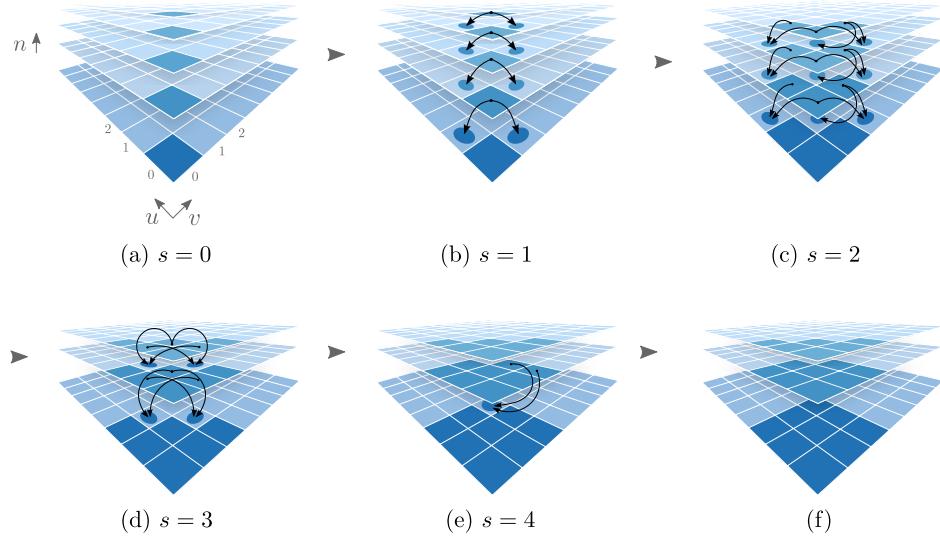


Figure 5.3: To compute the Hermite integral R_{tuv}^0 , (a) we begin by calculating R_{000}^n for all $n \leq s = t + u + v$. In this example, we'll calculate R_{022}^0 , which gives $n_{\max} = 4$. Because $t = 0$, only u and v are shown, but the method is equivalent if $t > 0$. (b) We calculate all values of R_{tuv}^n for which $s = 1$. Note that only values for which $n \leq n_{\max} - s$ are needed, hence only four layers are shown. This leads to a staircase-like structure of calculated R_{tuv}^n elements. (c) We do the same for $s = 2$, (d) for $s = 3$, and (e) for $s = 4$. (f) Finally, we have found R_{022}^0 .

Note that the order of the indices in V_{abcd} differs from the usual order in the bracket notation. This stems from a discrepancy in the notation used by physicists and chemists in this field. While chemists prefer to order the elements in the two-particle integrals as

$$V_{abcd} = \left(ab \left| \frac{1}{|\mathbf{r}_2 - \mathbf{r}_1|} \right| cd \right) = \iint \frac{\phi_a^*(\mathbf{r}_1)\phi_b(\mathbf{r}_1)\phi_c^*(\mathbf{r}_2)\phi_d(\mathbf{r}_2)}{|\mathbf{r}_2 - \mathbf{r}_1|} d\mathbf{r}_1 d\mathbf{r}_2, \quad (5.29)$$

physicists prefer the ordering

$$\left\langle ab \left| \frac{1}{|\mathbf{r}_2 - \mathbf{r}_1|} \right| cd \right\rangle = \iint \frac{\phi_a^*(\mathbf{r}_1)\phi_b^*(\mathbf{r}_2)\phi_c(\mathbf{r}_1)\phi_d(\mathbf{r}_2)}{|\mathbf{r}_2 - \mathbf{r}_1|} d\mathbf{r}_1 d\mathbf{r}_2. \quad (5.30)$$

This leads to some confusion generally, and in this project we mix the two notations (sorry about that!). In Hartree-Fock theory, we use the physicist's notation, while in the integral evaluations, we use the chemist's notation. Beware that `HartreeFock-Solver` therefore requests integrals from `GaussianSystem` in the order p, r, q, s to define the element Q_{pqrs} .

5.3 Solving the Eigenvalue Equations

Once all the integrals are set up, we may use an eigenvalue solver to solve the Roothaan equations or Pople-Nesbet equations. However, these are on the form of a *generalized*

eigenvalue problem. For instance, the Roothaan equations are on the form:

$$\mathbf{FC}_k = \epsilon_k \mathbf{SC}_k \quad (5.31)$$

It is the overlap \mathbf{S} that makes this differ from a *regular* eigenvalue problem. Fortunately, this may be transformed into a regular eigenvalue equation by a basis transformation which brings \mathbf{S} into unit form.

For this derivation, we will follow closely the text of J.M. Thijssen [12]. We assume there is such a \mathbf{V} that it will transform \mathbf{S} into the unit matrix:

$$\mathbf{V}^\dagger \mathbf{SV} = \mathbf{I}. \quad (5.32)$$

Then the Roothaan equation may be rewritten as

$$\mathbf{V}^\dagger \mathbf{FV} \mathbf{V}^{-1} \mathbf{C}_k = \epsilon_k \mathbf{V}^\dagger \mathbf{SV} \mathbf{V}^{-1} \mathbf{C}_k. \quad (5.33)$$

We further introduce a new set of transformed eigenvectors,

$$\mathbf{C}'_k = \mathbf{V}^{-1} \mathbf{C}_k, \quad (5.34)$$

and a transformed spatial Fock operator,

$$\mathbf{F}' = \mathbf{V}^\dagger \mathbf{FV}. \quad (5.35)$$

The result is a regular eigenvalue equation that may be solved easily with an available eigenvalue library:

$$\mathbf{F}' \mathbf{C}'_k = \epsilon_k \mathbf{C}'_k. \quad (5.36)$$

To find a candidate for \mathbf{V} , we search for a matrix \mathbf{U} that diagonalizes \mathbf{S} ,

$$\mathbf{U}^\dagger \mathbf{SU} = \mathbf{s}, \quad (5.37)$$

where \mathbf{s} is the diagonalized version of \mathbf{S} , or in other words, a matrix with the eigenvalues of \mathbf{S} on the diagonal. We may now introduce $\mathbf{s}^{-1/2}$ as a matrix with the inverse of the square root of the values of \mathbf{s} on its diagonal. Since \mathbf{S} is an overlap matrix, its eigenvalues must be positive. If we define $\mathbf{V} = \mathbf{Us}^{-1/2}$, we will find that this definition of \mathbf{V} has exactly the property we want:

$$\mathbf{V}^\dagger \mathbf{SV} = \mathbf{s}^{-1/2} \mathbf{U}^\dagger \mathbf{SU} \mathbf{s}^{-1/2} = \mathbf{I}. \quad (5.38)$$

However, this leads us to a new challenge: Namely how to find such a matrix \mathbf{U} . Fortunately, this is exactly the matrix we get by solving the eigenvalue equation in (5.37) with a standard Givens-Householder *QR* procedure.

5.3.1 The Hartree-Fock Solver Class

The `HartreeFockSolver` class is the base of both the RHF and the UHF implementation. They differ in that the UHF implementation needs to solve two eigenvalue equations simultaneously, while the RHF implementation only needs to solve one. In the following, we will focus on the RHF solver for brevity, and refer to the source code for implementation details on UHF.

Armadillo provides the easy-to-use wrapper function `eig_sym` for solving symmetric eigenvalue problems. This, and other Armadillo functions, call the LAPACK [62] Fortran library behind the scenes. We use this function in `HartreeFockSolver::setupTransformationMatrix` to find \mathbf{V} :

```
vec s;
mat U;
eig_sym(s, U, m_overlapMatrix);
m_transformationMatrix = U*diagmat(1.0/sqrt(s));
```

Next, we use `eig_sym` in the `HartreeFockSolver::advance` function to solve equation (5.36):

```
const mat V = transformationMatrix();
vector<mat> test;
mat &F = m_fockMatrix;
F = V.t() * F * V;
mat Cprime;
eig_sym(m_fockEnergies, Cprime, m_fockMatrix);
```

Once the result now is stored in `Cprime`, which is the programmatic version of \mathbf{C}'_k , we transform it back with \mathbf{V} to obtain \mathbf{C}_k , which may now be normalized again before setting up the density matrix:

```
C = V*Cprime.submat(0, 0, no - 1, nk - 1);
normalizeCoefficientMatrix();
setupDensityMatrix();
```

In both RHF and UHF, the rest of the `advance` function calculates the energy and stores this locally for the `solve` function to use in its convergence check, and possibly to be returned if this iterative procedure has converged.

5.3.2 Convergence Problems

Just like many other self-consistent iterative methods, the Hartree-Fock method suffers from convergence problems. There is always a risk that the above steps result in an oscillation between a set of eigenvalues and eigenvectors that are not solutions of the Hartree-Fock equations.

There are, however, many available schemes to aid convergence. One option is to keep the solver from producing two very different solutions between iterations. This is known as mixing [12], and is done by setting the density matrix to a weighted average of the newly obtained solution and the previous one:

$$\mathbf{P} = a\mathbf{P}^{\text{previous}} + (1 - a)\mathbf{P}^{\text{new}}. \quad (5.39)$$

This sometimes helps to achieve convergence when the initial guess for the density matrix is far away from the right one. In the subclasses of `HartreeFockSolver`, this has been implemented, and the mixing factor can be controlled by the user.

Other alternatives include extrapolation schemes, such as the DIIS procedure [63], and the use of semi-empirical methods that provide better initial guesses for the coefficients [8].

5.4 Symmetries in the Electron-Electron Integrals

There are symmetries in the two-particle matrix elements

$$Q_{prqs} = \langle pr | g | qs \rangle = \iint \varphi_p^*(\mathbf{r}_1) \varphi_r^*(\mathbf{r}_2) \frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} \varphi_q(\mathbf{r}_1) \varphi_s(\mathbf{r}_2) d\mathbf{r}_1 d\mathbf{r}_2. \quad (5.40)$$

Note the order of the indices, which is such that p and q share the spatial variable \mathbf{r}_1 , while r and s share \mathbf{r}_2 .

Because we are working with real-valued spatial functions, we may take the complex-conjugate any of the ϕ s without changing the integral. By interchanging the terms sharing the same spatial variable, we find that the following three integrals are equal to the one above:

$$Q_{qrps}, Q_{psqr} \text{ and } Q_{qspr}. \quad (5.41)$$

In addition, because the integration variables are \mathbf{r}_1 and \mathbf{r}_2 , which are dummy variables, they may be interchanged without loss of generality. Thus Q_{prqs} is also equal to the following four integrals:

$$Q_{rpsq}, Q_{sprq}, Q_{rqsp} \text{ and } Q_{sqrp}. \quad (5.42)$$

We now have a total of eight permutations of the indices p , q , r , and s , that result in the same integral. This symmetry has been implemented as an optimization in the `HartreeFockSolver` class in the Kindfield program.

5.5 Testing the Hartree-Fock Solver

Testing the Hartree-Fock solver is essential before we move on to calculate the potential energy surfaces we will use in molecular dynamics. The tests we will perform, range from basic testing of the individual components, to verification of the entire machinery by comparison to results in the literature. This way we may rest confident that our solver is correctly implemented. We will be using the `UnitTest++` library to implement our tests in C++. For further details on how to use this library, see Appendix B.

5.5.1 Integrator Verification

A central part of our Hartree-Fock program, Kindfield, are the integral calculations. To verify the implementation, it may be useful to compare the results against another integral solver. For our tests, we will use Python to calculate the same integrals numerically. There are two main approaches we will use: The first is to use the SciPy package, with functions like `integrate.trapz` and `integrate.quad`, which calculate integrals by the trapezoidal rule and by use of the QUADPACK Fortran library, respectively. The other option is to use the SymPy package, which can evaluate integrals symbolically.

5.5.2 Overlap Integrals and Kinetic Integrals

Sympy will be used for verification of the overlap and kinetic integrals. It is able to both typeset and evaluate math, which makes it a suitable tool for this purpose. We begin by loading the necessary libraries:

```
from numpy import array, dot
from sympy.abc import *
from sympy import *
init_printing(use_latex=True)
```

Next, define a primitive Gaussian function of any order, accepting the polynomial exponents i , j and k , together with the electron position \mathbf{r} , the exponent a and the nucleus position \mathbf{A} as arguments:

```
def G(i,j,k,a,r,A):
    rA = r - A
    xA = rA[0]
    yA = rA[1]
    zA = rA[2]
    rA2 = dot(rA,rA)
    return xA**i * yA**j * zA**k * exp(-a * rA2)
```

The vectors are assumed to be NumPy arrays. We verify this implementation by defining a Gaussian G_a and print it to screen:

```
Ax, Ay, Az = symbols("A_x A_y A_z")
A = array([Ax, Ay, Az])
r = array([x, y, z])
alpha = symbols('alpha')
G_a = G(1,2,0,alpha,r,A)
G_a # Trigger printing of G_a in IPython Notebook or IPython Qt Console
```

This results in the typeset output:

$$(-A_x + x) (-A_y + y)^2 e^{-\alpha((-A_x + x)^2 + (-A_y + y)^2 + (-A_z + z)^2)}$$

We can further define two Gaussians with numerical values and calculate the integral by use of the `integrate` function and the `N`-function that evaluates a SymPy expression numerically. Note that we explicitly have to request a triple integral, over the x , y and z -coordinates, and that the limits are determined to be from $-\infty$ to ∞ by the use of the “`oo`” variable:

```
product = G(0,0,0,0.2,r,array([1.2,2.3,3.4])) * G(0,0,0,0.3,r,array([0.0, 0.0,
0.0]))
N(integrate(integrate(integrate(product, (x,-oo, oo)), (y,-oo,oo)), (z,-oo,oo)))
```

The output is the number $0.1191723\dots$, which may now be plugged directly into our C++ unit tests to verify the overlap integrals (see Appendix B for more information on unit testing):

```
#include <unittest++/UnitTest++.h>
#include "math/vector3.h"
#include "basisfunctions/gaussian/integrals/gaussianoverlapintegral.h"
#include "basisfunctions/gaussian/gaussianprimitiveorbital.h"

SUITE(GaussianIntegral) {
    TEST(GaussianOverlapIntegralTest) {
        Vector3 posA = {1.2,2.3,3.4};
        Vector3 posB = {-1.3,1.4,-2.4};
        GaussianPrimitiveOrbital primitiveA(1.0, 2, 2, 2, 0.2);
        GaussianPrimitiveOrbital primitiveB(1.0, 2, 2, 2, 0.3);
        GaussianOverlapIntegral integrator(posA, posB, primitiveA, primitiveB);
        CHECK_CLOSE(integrator.overlapIntegral(0,0,0,0,0,0), 0.119172363580852,
                   1e-6);
    }
}
```

The test passes, and thus verifies the implementation for this case. The procedure for kinetic integrals is equivalent, and these test have been implemented in the tests-subproject of the Hartree-Fock program.

With this testing framework at hand, we may even generate the C++ code for our tests from Python. A IPython Notebook in the source code of Kindfield, named `notebooks/Symbolic Integrals.ipynb`, does exactly this. In this notebook, several unit tests are generated for different combinations of the exponents i, j, k, l, m , and n .

5.5.3 Coulomb Integrals

Both the nuclei-electron and the electron-electron integrals depend on the Boys function. A good start here is to test our Boys function implementation first.

The Boys-function is itself just a simple integral,

$$F_n(x) = \int_0^1 \exp(-xt^2)t^{2n} dt, \quad (5.43)$$

which may be calculated directly using either SymPy or SciPy. In the case where $n = 0$ and $x = 0.2$, we find the numerical value as follows:

```
def boys_integrand(t, x):
    return exp(-x*t**2)

quad(boys_integrand, 0, 1, args=(0.2))
# output: 0.937150028798
```

The output may again be used directly in the unit tests, implemented in our C++ library.

Since the Boys function is implemented as a composition of multiple parts, with some values pre-tabulated, some calculated on the fly, and some by recursive formulas, each part needs to be tested separately. This is all implemented in the tests-subproject, in the `boysfunction.cpp` source file.

Once the Boys-function is working properly, the integrals may be tested. Unfortunately, the SymPy library does at the time of writing not support Gaussian integrals symbolically. We therefore have to turn to pure numerical integration with NumPy. The principle is the same as with overlap integrals and kinetic integrals, and the functionality to generate tests is available in the same IPython Notebook as stated above.

For the electron-electron repulsion, we have to perform six-dimensional integrals. Although these can be evaluated numerically, they have turned out to be too expensive to calculate with our current methods. So in this case, the tests are based on the results of H. Mobarhan [1] and Eiding [2], who have implemented their own versions of a Hartree-Fock code, with the same starting point. However, in the next subsection, we will ensure that the complete solver gives comparable results to what we find in the literature. This is a good indicating that the electron-electron integrals are correctly implemented.

5.6 Results of the Hartree-Fock Solver

With tests working, it is about time to verify the complete Hartree-Fock solver. In this section, we will discuss ground state results of the simple H_2 molecule, compare RHF and UHF, calculate the ground state energies of the “10-electron series” and visualize electron densities and the electrostatic potential.

5.6.1 Ground State Energy of H_2

Our first test is to check whether the implementations of the RHF method and the UHF method result in the correct ground state energy for the hydrogen molecule, H_2 . To test this, we will compare our result to that of Szabo and Ostlund [45], which for an 6-31G** basis results in the energy $E_{HF} = -1.131$ at a distance $R = 1.40$ between the two hydrogen atoms. In this test, we set up a system with two hydrogen atoms this distance apart on the x -axis:

```
vector<GaussianCore> cores;
cores.push_back(GaussianCore({0,0,0}, "atom_1_basis_6-31Gdsds.tm"));
cores.push_back(GaussianCore({1.4,0.0,0}, "atom_1_basis_6-31Gdsds.tm"));
GaussianSystem system;
for(const GaussianCore &core : cores) {
    system.addCore(core);
}
```

and then set up either the `RestrictedHartreeFockSolver` or the `UnrestrictedHartreeFockSolver` to calculate the energy of the system:

```
RestrictedHartreeFockSolver solver(&system);
solver.setConvergenceThreshold(1e-12);
solver.setNIterationsMax(1e3);
solver.setDensityMixFactor(0.5);
solver.solve();
```

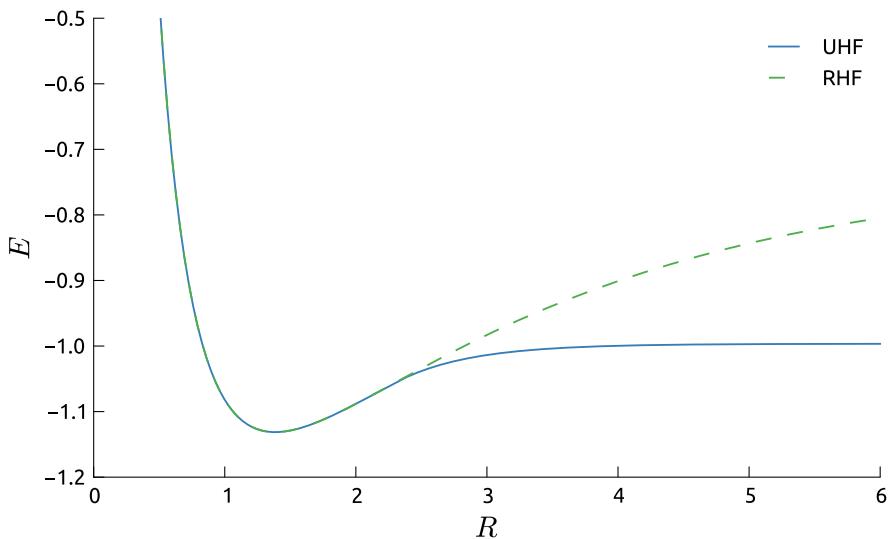


Figure 5.4: Energy of H_2 plotted as a function of the bond length (both in atomic units). Shown are the energy profiles for RHF and UHF with a 6-311++G** basis set. Only the UHF energy converges to two times the energy of an individual hydrogen atom.

Once the energy is calculated by the solve function, we check this against the expected value by use of the `CHECK_CLOSE` macro found in the `UnitTest++` library.

```
CHECK_CLOSE(-1.13128434930047, solver.energy(), 1e-6);
```

Note that this function includes many more digits after the decimal point than what we found from Szabo and Ostlund [45]. The rest of the digits are added from a previous run with this solver to create what is known as a *regression test*. It not only ensures that we are in agreement with existing results in the literature, but also ensures that future changes in the code won't introduce small perturbations to this result.

5.6.2 Pulling the Hydrogen Molecule Apart: Restricted vs. Unrestricted Hartree-Fock

Dissociation of molecules, such as H_2 , is an example of why the restricted Hartree-Fock (RHF) method is not sufficient, and the unrestricted Hartree-Fock (UHF) method is needed. If the hydrogen molecule is in the ground state, the two single-particle states will share the same spatial orbital, but with opposite spin. As we pull them apart, however, the lowest energy configuration is found when each of the two single-particle states are concentrated on one of the nuclei. The latter is only possible to describe with UHF.

We will verify that UHF provides better results than RHF for these cases. In the previous section, we described a test that verified both the RHF and the UHF

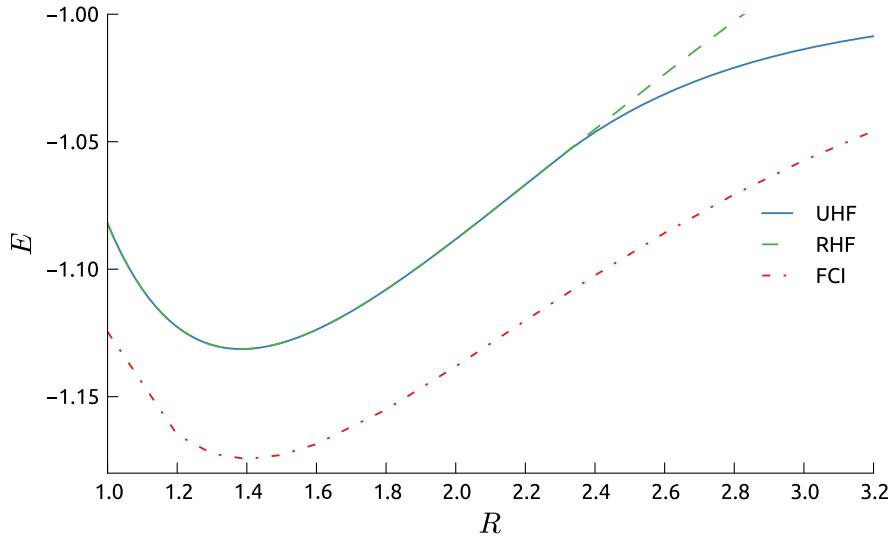


Figure 5.5: Energy of H_2 plotted as a function of the bond length (both in atomic units). Shown are the energy profiles for RHF and UHF with a 6-311++G** basis set. The Full Configuration Interaction (FCI) curve from Kolos and Wolniewicz [64] is shown as reference.

implementations for the ground state of hydrogen. The next step is to verify that UHF returns twice the energy of a single hydrogen atom once the two atoms are separated by a wide distance, such as $R = 6.0$. The latter should be close to $E_{\text{HF}} = -1.0$, because the energy of a single hydrogen atom is $E = -0.5 = -13.6 \text{ eV}$. This test is found in the file `tests/unrestricted.cpp` of the Kindfield source code (see page 3 for information on how to obtain the source code).

It is also useful to do a visual check of the values between the ground state energy, given at $R = 1.40$ [45], and the energy of a more dissociated state, with $R = 6.0$. This is calculated by producing a number of states on the interval $R = [0.5, 6.0]$, and plotting the result, as shown in Figure 5.4. Here we see that the UHF energy converges to a finite value as the two atoms are separated, while the RHF energy diverges.

In Figure 5.5 the UHF and RHF calculations are compared to the results of a ground state FCI calculation by Kolos and Wolniewicz [64]. An FCI calculation should be close to the true theoretical ground state of the system (albeit under the assumption that an infinite basis is used). This is a good benchmark for the precision of Hartree-Fock theory. As we see from Figure 5.5, there is a difference in both the numerical value of the energy, in addition to the slope of the curve. The latter will have consequences for our upcoming force calculations. Using Hartree-Fock as our *ab initio* foundation for molecular dynamics potentials will likely lead to results that are a bit off from the exact theory. However, the main features of the potential are kept intact, and Hartree-Fock should provide a decent starting point.

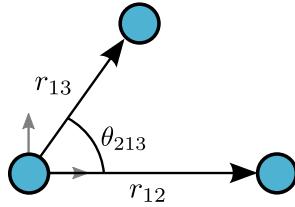


Figure 5.6: Defining the configuration of three atoms may be done by placing the first atom at the origin, and specifying the positions of the other two by the angle θ_{213} , and the distances r_{12} and r_{13} .

5.6.3 Ground State Energies of Molecules

With the results for the simple H₂ molecule tested and verified, we are now ready to see if our machinery is capable of handling even larger molecules, with more electrons and more degrees of freedom.

The first tests are done for H₂O and CO₂. They are both trimers; molecules with three atoms. With three atoms, we get three degrees of freedom for specifying their positions. A reasonable choice is to place the first atom in the origin of our system, the second atom on the x axis, a distance r_{12} away from the first atom, and the third atom in the xy -plane, a distance r_{13} away from the first, in order to form a triangle with an angle θ_{213} between the three, as shown in Figure 5.6. In fact, because both H₂O and CO₂ are symmetric molecules, both having two equal atoms, we will find that if the first atom is oxygen for H₂O, or the first is carbon for CO₂, the distances r_{12} and r_{13} will be equal.

If we wanted to search for the ground state configuration, we could now vary these three variables, r_{12} , r_{13} and θ_{213} , (or set $r_{12} = r_{13}$ in the H₂O and CO₂ cases), and search for the minimum energy expectation value with any chosen minimization scheme. However, for the sake of simplicity and brevity, we will only test that we get the correct energies for the ground state by setting the values of r_{12} , r_{13} and θ_{213} to the ground state configurations found in the text of Szabo and Ostlund [45]. The results are shown in Table 5.1 and are in perfect agreement with those found in the literature.

5.6.4 Electron Density

The modulus squared of the total wave function is interpreted as the probability density of finding the particles in a given configuration $\mathbf{x}_1, \dots, \mathbf{x}_N$ [12]:

$$P(\mathbf{x}_1, \dots, \mathbf{x}_N) = |\Psi(\mathbf{x}_1, \dots, \mathbf{x}_N)|^2. \quad (5.44)$$

If we take the integral of the total wave function over all but one coordinate, \mathbf{x}_1 , we get the probability of finding one particle with the coordinate \mathbf{x}_1 :

$$P(\mathbf{x}_1) = \int |\Psi(\mathbf{x}_1, \dots, \mathbf{x}_N)|^2 d\mathbf{x}_2 \dots d\mathbf{x}_N. \quad (5.45)$$

Because all electrons are indistinguishable, and the symmetry/anti-symmetry of the wave function, this is the same for all electrons. This means that we may drop the

Molecule	RHF ^(a)	UHF ^(a)	UHF ^(b)	HF _{limit}	Expt.
H ₂	-1.13248	-1.13248	-1.13128 ^(c)	-1.1336 ^(d)	-1.1746 ^(e)
CH ₄	-40.2092	-40.2092	-40.1985 ^(c)	-40.225 ^(d)	-
NH ₃	-56.2145	-56.2145	-56.1954 ^(c)	-56.225 ^(d)	-
H ₂ O	-76.0532	-76.0532	-76.0234 ^(c)	-76.065 ^(d)	-
FH	-100.053	-100.053	-100.011 ^(c)	-100.071 ^(d)	-
O ₂	-149.574	-149.660	-149.615	-	-150.3268 ^(f)
CO ₂	-187.689	-187.689	-187.633	-	-

Table 5.1: Table of energies resulting from calculations with the Kindfield program using (a) the 6-311++G** basis set and (b) the 6-31G** basis set. The results (c) are in exact correspondence with those found by Szabo and Ostlund [45]. Also listed are (d) the Hartree-Fock limit energies from Hariharan and Pople [65] and the best possible (experimental) results, from (e) Moskowitz and Kalos [66] and (f) Filippi and Umrigar [67]. All values are in Hartrees (E_h).

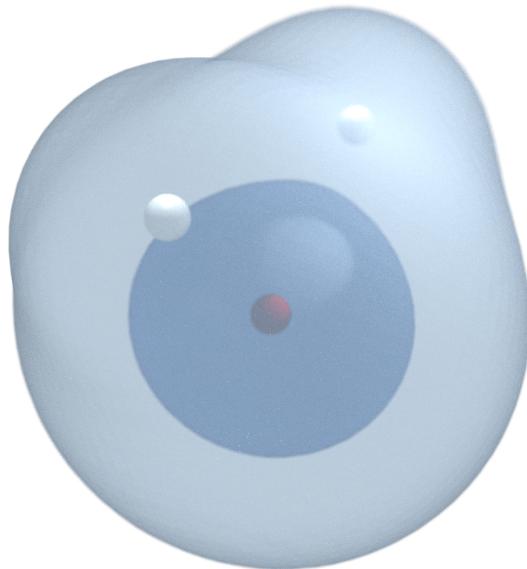


Figure 5.7: Electron density of the H₂O molecule. Two isosurfaces are shown. The outermost one, in light blue, has the lowest electron density ($\rho = 0.05$). The innermost one, in dark blue, has a higher electron density ($\rho = 0.4$).

subscript index and just write $P(\mathbf{x})$ to show that this goes for any electron coordinate.

By multiplying $P(\mathbf{x})$ by the number of electrons, we get the electron density [45]:

$$\rho(\mathbf{x}) = N_e P(\mathbf{x}) = N_e \int |\Psi(\mathbf{x}, \dots, \mathbf{x}_N)|^2 d\mathbf{x}_2 \dots d\mathbf{x}_N. \quad (5.46)$$

It is a bit unusual to write the electron density as a function of a spin-coordinate \mathbf{x} , but we will soon replace it by the spatial coordinate \mathbf{r} by integrating away (i.e. summing over) spin. However, for the sake of clarity, we will keep the spin-coordinate until we know more about the wave function.

We can use the above expression to find the electron density of a trial wave function as well. In the case of a Slater determinant, we find that the probability density is just the sum of all the single-particle wave functions, i.e. the spin orbitals:

$$\rho(\mathbf{x}) = \int |\Psi_{SD}(\mathbf{x}, \dots, \mathbf{x}_N)|^2 d\mathbf{x}_2 \dots d\mathbf{x}_N = \sum_k^{N_e} |\psi_k(\mathbf{x})|^2. \quad (5.47)$$

If you look back at the normalization check of the Slater determinant in Section 3.2, you will see that the factor $1/N$ in the above equation comes from the fact that we now integrate over one less coordinate, resulting in $(N - 1)!$ products instead of $N!$.

Usually, we are only interested in the one-body density for a spatial coordinate \mathbf{r} , and not the complete spin-coordinate, $\mathbf{x} = (\mathbf{r}, \xi)$. To obtain the probability density for spatial coordinates, $\rho(\mathbf{r})$, we integrate over spin as well, which is the same as doing a sum over spin up $\xi = \alpha$, and spin down, $\xi = \beta$:

$$\rho(\mathbf{r}) = \sum_{\xi \in \{\alpha, \beta\}} \sum_k^{N_e} |\psi_k(\mathbf{r}, \xi)|^2 = \sum_k^{N_\alpha} |\phi_k^\alpha(\mathbf{r})|^2 + \sum_k^{N_\beta} |\phi_k^\beta(\mathbf{r})|^2. \quad (5.48)$$

Here, the sums run to the number of electrons we have chosen to have spin up, N_α , and spin down N_β . In the case of RHF, these numbers are equal, $N_\alpha = N_\beta = N/2$, and we get only one sum. In UHF, these numbers are chosen manually.

In our case, the spatial orbitals ϕ_k are linear expansions of our basis set of contracted Gaussians, φ_p . In terms of these, the electron density is written out as

$$\rho(\mathbf{r}) = \sum_k^{N_\alpha} \sum_{pq}^M C_{kp}^\alpha C_{kq}^\alpha \varphi_p(\mathbf{r}) \varphi_q(\mathbf{r}) + \sum_k^{N_\beta} \sum_{pq}^M C_{kp}^\beta C_{kq}^\beta \varphi_p(\mathbf{r}) \varphi_q(\mathbf{r}) \quad (5.49)$$

$$= \sum_{pq}^M P_{pq}^\alpha \varphi_p(\mathbf{r}) \varphi_q(\mathbf{r}) + \sum_{pq}^M P_{pq}^\beta \varphi_p(\mathbf{r}) \varphi_q(\mathbf{r}) \quad (5.50)$$

$$= \sum_{pq}^M P_{pq} \varphi_p(\mathbf{r}) \varphi_q(\mathbf{r}). \quad (5.51)$$

Here, we have introduced the density matrix \mathbf{P} , defined as the sum of the spin-up and spin-down density matrices,

$$\mathbf{P} = \mathbf{P}^\alpha + \mathbf{P}^\beta, \quad (5.52)$$

with elements

$$P_{pq}^{N^\alpha} = \sum_k^{N_\alpha} C_{kp}^\alpha C_{kq}^\alpha, \quad (5.53)$$

$$P_{pq}^{N^\beta} = \sum_k^{N_\beta} C_{kp}^\beta C_{kq}^\beta. \quad (5.54)$$

We may interpret the two sums separately as the density of electrons with a given spin. It is more common, however, to define the *spin density*. This is given as the difference between the two densities:

$$\rho_{\text{spin}}(\mathbf{r}) = \rho_\uparrow(\mathbf{r}) - \rho_\downarrow(\mathbf{r}) = \sum_{pq}^M P_{pq}^\alpha \varphi_p(\mathbf{r}) \varphi_q(\mathbf{r}) - \sum_{pq}^M P_{pq}^\beta \varphi_p(\mathbf{r}) \varphi_q(\mathbf{r}). \quad (5.55)$$

In Figure 5.7, the electron density of an H₂O molecule is shown with two isosurfaces. The outermost surface has a lower electron density than the innermost one (see the caption for the exact values). This tells us that there is a higher electron density around the oxygen atom, which is intuitive considering its higher positive charge.

The electron density of molecules is an ideal tool when combined with experimental X-ray diffraction. Theoretically, the electron density $\rho(\mathbf{r})$ can be calculated from diffracted X-rays, after proper conversion of the experimental data (among other things, via Fourier transformation). Comparing these results with theoretical and numerical calculations of the electron density allows identification and classification of molecular and crystal structures [68]. Hydrogen atoms are not easily visible in X-ray diffraction because the electron density is low around the hydrogen atoms. In the case of the H₂O molecule shown in Figure 5.7, hydrogen atoms will not show up as clearly as the more electronegative oxygen atom. However, diffraction experiments on the structure of water have even provided evidence for the covalent nature of hydrogen bonds [69, 70].

Information about reaction pathways is also available from the electron density. Covalent bonding is most likely to happen where the gradient $\nabla\rho(\mathbf{r})$ is steepest [71]. Such paths usually begin at a nucleus and continue out of the molecule to end at infinity. However, there are equivalent paths between already bonded nuclei inside the molecule. The electron density thus provides a view of molecular structure that coincides with chemical intuition.

5.6.5 Electrostatic Potential

While the electron density is a measure of the spatial distribution of the electrons in the molecule, the electrostatic potential represents the potential energy of a test charge in the vicinity of said molecule. This is a generalization of the classical electrostatic potential, under the assumption that the test charge is point-like and not interacting with the molecule. It tells us to what degree a positive test charge will be attracted by or repelled from the molecule [8].

The electrostatic potential is calculated by summing the Coulomb repulsion of a positive test charge with the nuclei, and subtracting the attraction to the electrons.



Figure 5.8: The electrostatic potential of the H_2O molecule. Two isosurfaces are shown. The outermost one, in dark blue, has a negative electrostatic potential of $V_E = -0.09$. The innermost one, in light blue, has a positive electrostatic potential of $V_E = 0.8$. A positive charge, would prefer to approach the H_2O from the bottom, where the electrostatic potential is negative.



Figure 5.9: The electrostatic potential of the NH_3 molecule. Two isosurfaces are shown. The outermost one, in dark blue, has a negative electrostatic potential of $V_E = -0.03$. The innermost one, in light blue, has a positive electrostatic potential of $V_E = 0.7$.

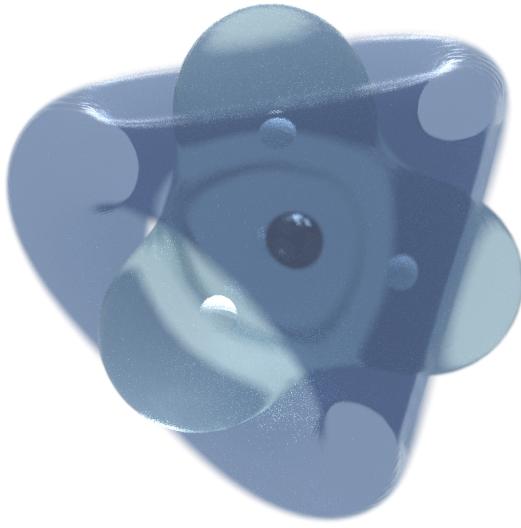


Figure 5.10: The electrostatic potential of the CH_4 molecule. Two isosurfaces are shown. The outermost one, in dark blue, has a negative electrostatic potential of $V_E = -0.002$. The innermost one, in light blue, has a positive electrostatic potential of $V_E = 0.02$.

The latter is computed by taking the integral of the electron density divided by the distance to the test charge. In total, the electrostatic potential is defined as:

$$V_{\text{ES}}(\mathbf{r}) = \sum_I^{N_n} \frac{Z_I}{|\mathbf{r} - \mathbf{r}_I|} - \int \frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} d\mathbf{r}'. \quad (5.56)$$

As above, we are using a Slater determinant Ψ_{SD} , and in this case, the electron integral can be written in terms of our contracted basis functions:

$$V_{\text{ES}}(\mathbf{r}) = \sum_I^{N_n} \frac{Z_I}{|\mathbf{r} - \mathbf{r}_I|} - \sum_{pq}^M P_{pq} \int \varphi_p(\mathbf{r}') \frac{1}{|\mathbf{r} - \mathbf{r}'|} \varphi_q(\mathbf{r}') d\mathbf{r}'. \quad (5.57)$$

Electrostatic potentials of the molecules H_2O , NH_3 and CH_4 are shown in Figures 5.8 to 5.10. To visualize the electrostatic potential, two isosurfaces are drawn, one with a slightly positive value and one with a slightly negative value. This is similar to the density, where we used isosurfaces to represent the regions of space with the same density. (The visualization method is explained in detail in Section 9.2.) The volume enclosed by the darker surface has negative electrostatic potential, while the lighter surface encloses a volume of positive electrostatic potential. All other areas of space have values that can be either positive or negative. Note that the values of the isosurfaces are not shared between the visualizations, but chosen to show as clear a picture of the negative and positive areas of space.

The illustrated negative isosurface of the CH_4 molecule has a lower value than that of the H_2O molecule, which may be used to explain the differences in chemistry between

methane and water. While the water molecule is both a dipole and has a region of higher stronger negative electric potential, the methane molecule has no overall polarization (it is symmetric) and only a weak region of negative electric potential. Thus, the main source of attraction between methane molecules are dispersive forces, as opposed to the interactions of water molecules, where dipole-dipole attractions dominate - better known as hydrogen bonds.

In total, we now have a Hartree-Fock code that works as expected, and are ready to move on to the next chapter on ANNs. Here we will fit an ANN to a PES from our Hartree-Fock calculations. In the following chapters, this will be made use of in efficient molecular dynamics simulations.

Chapter 6

Artificial Neural Networks

With the Kindfield program available, we are ready to calculate the potential energy surface (PES) of molecular systems with Hartree-Fock. This can be used to parameterize two-, three- and many-body potentials to be used in molecular dynamics (MD). We discussed some common potentials in Section 2.1. In the following, we will combine this with the theory of artificial neural networks (ANNs), as discussed in Chapter 4.

This has been implemented in form of a collection of scripts, named name FANN-MD, that use the Fast Artificial Neural Network Library (FANN) library to train and validate an ANN (see page 3 for information on how to obtain the source code).

6.1 Configurations for Two- and Three-Body Potentials

To fit the two- and three-body potentials, we first sample the energy of a number of configurations of two or three atoms, respectively. In the two-body case, we have only one degree of freedom: The distance between the two atoms. In the three-body case, we have three degrees of freedom. We can fix one atom in the origin, vary the distance to the second and third atom, r_{12} and r_{13} , respectively, and the angle between the three, θ_{213} .

6.1.1 Defining the Ranges of Particle Configurations

We need to define the range of distances and angles for which we would like to calculate the potential. There is, for instance, no reason to include calculations where two atoms are very close to each other. The repulsive forces are too high for the atoms are going to end up in such a state. There is also no need for energy calculation for states where the two atoms are far apart. In these cases, there are virtually no interacting forces.

Upper Cutoff

Let's begin by look at the case where the atoms are far apart. If we are to exclude those configurations, we should ensure that their force has close to no influence on the system. It is sufficient to require that the atom has moved no more than $\Delta x = 1.0$ during the time Δt . However, the choice of Δt is open for discussion, and may have to

be adjusted in different simulations. For now, we will choose $\Delta t = 10^5$, equivalent to about $\Delta t = 2.4 \times 10^{-12}$ s.

If this is the only acting force in the system, and we assume that the particles are so far apart that the force practically stays constant throughout the simulation, we can use the simple relation for the change in position,

$$\Delta x = \frac{1}{2} \frac{F}{m} \Delta t^2, \quad (6.1)$$

leading to

$$F > \frac{m}{2} \frac{\Delta x}{\Delta t^2}. \quad (6.2)$$

In the case of hydrogen atoms, we find (in atomic units):

$$F > \frac{1837}{2} \frac{1.0}{(10^5)^2} \approx 10^{-7}. \quad (6.3)$$

This will be our general criterion. Forces so small that atoms don't move more than $\Delta x = 1.0$ under their influence, will be neglected. And this will define the upper range for our two-body forces. For two hydrogen atoms, this is the case for distances of $r > 12$.

Lower Cutoff

A limit for the lower range is easier to define because the potential grows exponentially as the distance between two atoms goes to zero. Here it is more important to ensure that the proper functional form of a potential well is obtained by training the ANN potential. For values smaller than the lower cutoff, we may in any case approximate the functional form by a typical exponential function that is continuous in the transition.

ANNs are typically trained to return normalized values between -1 and 1 (before scaling them back). In my experience, the fitting error of the FANN library is on the order of 10^{-6} . Therefore, it seems reasonable to set the lower cutoff where the potential is about 10 times larger than the potential difference in any other characteristic feature. In simple words, for the H_2 molecule, the cutoff is set where the potential energy has a value about 10 times larger than the depth of the potential well. This should suffice to describe the characteristics of the potential. Further, it ensures that we are in the region of exponential growth before transitioning to a pure exponential function.

6.1.2 Creating Two-Body Configurations

With a decision on the upper and lower range, the two-body configurations may be created by a Python script and stored to file for calculation with the Hartree-Fock code:

```
r12s = linspace(1, 12, 10)
for r12 in r12s:
    hydrogen_1.position = array([0.0, 0.0, 0.0])
    hydrogen_2.position = array([r12, 0.0, 0.0])
    # ... create configuration
```

6.1.3 Creating Three-Body Configurations

We create configurations for energy calculations of three atoms by varying the two distances r_{12} and r_{13} , and the angle θ_{213} . This can be done programmatically with a Python script:

```
r12s = linspace(1, 6, 10)
r13s = linspace(1, 6, 10)
angles = linspace(pi/3, pi, 10)
```

When the three atoms are of equal type, we only need to create configurations with angles above 60° . At least one corner in the triangle will have an angle larger than this.

Further, we use a triple loop and generate configurations for H₂O:

```
for r12 in r12s:
    for r13 in r13s:
        for angle in angles:
            hydrogen_1.position = array([0.0, 0.0, 0.0])
            hydrogen_2.position = array([r12, 0.0, 0.0])
            oxygen.position = array([r13*cos(angle), r13*sin(angle), 0.0])
            # ... create configuration
```

Once these configurations are created and stored to file, they may be used as input for Hartree-Fock calculations in the Kindfield program. The results are then read back into Python, and used as training data.

6.1.4 Creating Many-Body Configurations

Due to the high number of possible configurations of four or more atoms, some clever method should be applied to select those that are most relevant. One approach used by Raff et al. [72] is to collect configurations from a molecular dynamics simulation with a simple potential, such as the Tersoff potential or the Stillinger-Weber potential.

Next, the potential energies of these configurations are calculated using *ab initio* methods, and the resulting energies used to train, test and validate the ANN potential. Further, the ANN potential can then be used in the same molecular dynamics simulation to find even better configurations. This results in an iterative scheme, where the potential is improved in each step. Once converged, the potential can be used in a final simulation where statistical properties are sampled.

In this thesis, however, we will limit ourselves to two- and three-body terms.

6.2 Training an Artificial Neural Network with FANN

Training the neural networks is done by a combination of the backpropagation method and the CASCADE2 algorithm. The latter has not been discussed in this text, but is a more advanced training algorithm available in the FANN library. The theoretical details of CASCADE2 are described in the thesis of Nissen [73].

The neural network is set up with two hidden layers, each with 5 neurons. FANNTool [74] has been used to find the optimal activation functions. These were found to be Gaussian activation functions for the hidden layers, and a linear output function.

FANN provides bindings to the Python language. These have been used for training the network. See the FANN documentation for details on these bindings [52]. First, we create a neural network, set the training algorithm and the network topology:

```
ann = libfann.neural_net()
ann.set_training_algorithm(libfann.TRAIN_INCREMENTAL)
ann.create_shortcut_array((1,5,5,1))
ann.set_activation_function_hidden(libfann.GAUSSIAN)
ann.set_activation_function_output(libfann.LINEAR)
ann.set_cascade_weight_multiplier(0.001)
```

Then the network is trained in iterations of backpropagation epochs. After a given number of epochs, in our case usually 2000, the network is tested against the validation data and training stopped if overfitting occurs. Otherwise, the network is saved as the currently best network. If the result since the last validation is only slightly worse, training is continued, but the current network is not saved:

```
best_result = inf
for i in range(20):
    ann.train_on_data(train_data, 2000, 500, 0.00000001)
    ann.reset_MSE()
    validate_result = ann.test_data(validate_data)
    if validate_result < best_result:
        best_result = validate_result
        ann.save(network_pre_filename)
    else:
        break
```

Once finished, the backpropagation training of the network has been completed.

There is still a chance that introducing new neurons will result in even better results. Therefore, we continue with the CASCADE2 algorithm. This trains the network iteratively by inserting one new neuron at the time. A group of neurons are compared with each other to find the best candidate. The validation is then performed for each inserted neuron:

```
for i in range(10):
    ann.cascadetrain_on_data(train_data, 1, 1, 1e-5)
    ann.reset_MSE()
    validate_result = ann.test_data(validate_data)
    if validate_result < best_result:
        best_result = validate_result
        ann.save(network_filename)
    elif validate_result < best_result*1.01:
        print "Validation almost failed..."
    else:
        print "Validation failed: Stopping training!"
        break
```

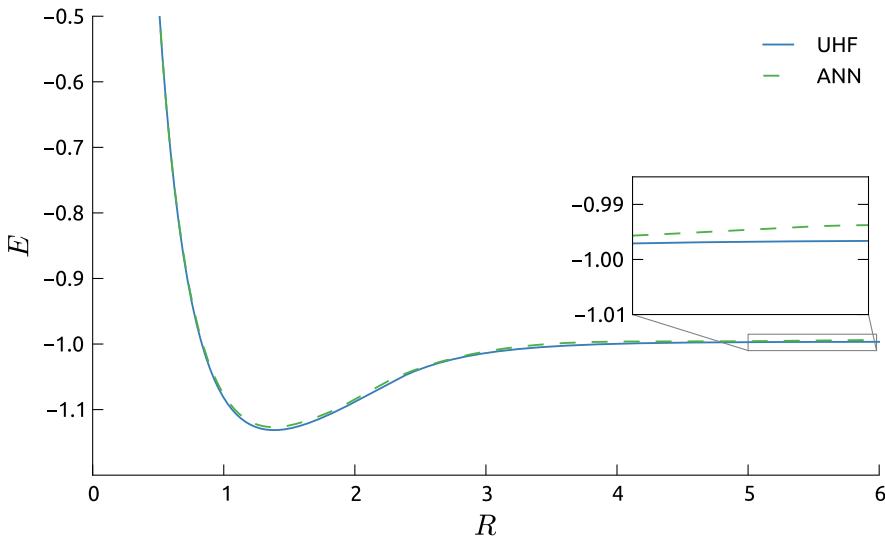


Figure 6.1: H_2 energy plotted as function of bond length (both in atomic units). The plot compares the result of an unrestricted Hartree-Fock (UHF) calculation using a 6-311++G** basis set and an ANN approximation. For details on the training method of the ANN, see Section 6.2. There is a striking similarity between the two, but closer inspection shows that there are some minor differences (inset).

Multiple training sessions like the ones above are performed sequentially or in parallel. The best network is chosen by testing each network on all the validation and training data. The complete training method is implemented in the Python scripts `fann_train_two_particles.py` and `fann_train_three_particles.py`.

6.3 Results and Verification of the Artificial Neural Network

Training the neural network should result in a good approximation of the desired function. To verify this, the mean square error (MSE) is calculated immediately after training. Further, it may be useful to perform a visual verification of the functional form.

H₂ Potential Approximation

In Figure 6.1, the UHF potential is compared to an ANN potential for the H_2 molecule. The ANN has been trained with the default method discussed in Section 6.2. The figure shows that the approximation is good, with a function body that looks much like the desired result. It is likely that many properties of the H_2 interaction will be reproduced.

There are a few issues to watch out for, however. ANNs approximations tend to introduce slight errors at the potential tail. This results in a non-physical repulsive force at wide separations (see inset of Figure 6.1). The cure for this is to train two or

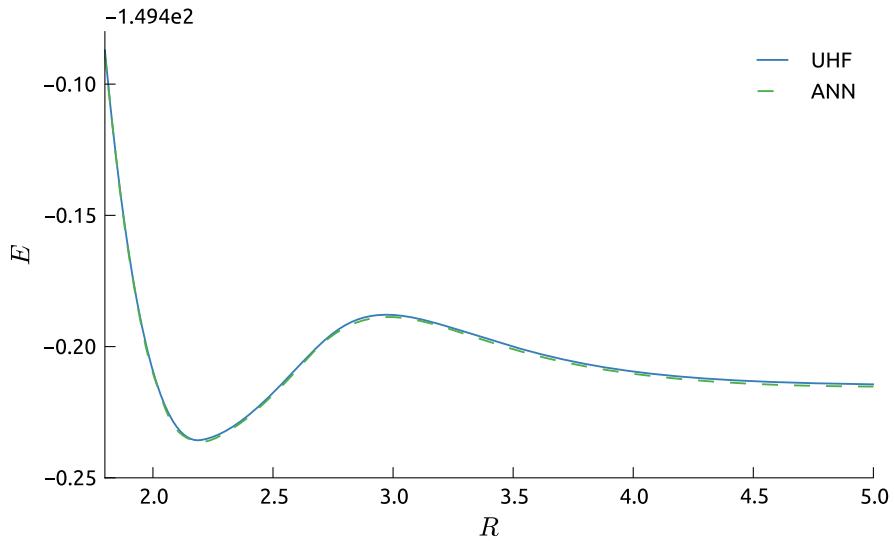


Figure 6.2: O_2 energy plotted as a function of bond length (both in atomic units). The plot compares the result of a UHF calculation using a 6-311++G** basis set with an ANN approximation. For details on the training method of the ANN, see Section 6.2. The ANN is able to pick up the characteristic bump at $R = 3.0$, showing how flexible the ANN can be at adapting to general functional forms without further intervention from the user.

more ANNs, each responsible for a different region of the potential. For instance, we could use one ANN approximating the potential for distances $r \in [0.6, 4]$, and another responsible for values of $r \in [4, 12]$. Ideally, we should smoothly interpolate between the two networks in a common region. The interpolation has not been done in this project, however, which might lead to slight energy conservation problems in our final simulations.

The same comparison is done for the O_2 molecule in Figure 6.2. This shows the strong flexibility of ANNs for function approximations. O_2 has a bump in its potential at a distance of about $R = 3.0$ between the two oxygen atoms. This feature is picked up by the ANN. While it is possible to approximate the H_2 potential with simple functions, such as the Lennard-Jones potential, the O_2 potential would require a more sophisticated functional form. Using an ANN does on the other hand require no intervention from the user. The training procedure is the same for both H_2 and O_2 , as described in Section 6.2).

H₃ Potential Approximation

In Figure 6.3, a comparison is shown for the UHF potential of a system with three hydrogen atoms, H_3 , and the approximation of this potential with an ANN. In fact, this is the sum of two ANNs, where one is used to calculate the two-body energy terms, while the other is adding a three-body correction term on top of that. See Section 7.2 for details on how the potential is built up by two- and three-body terms. The ANN

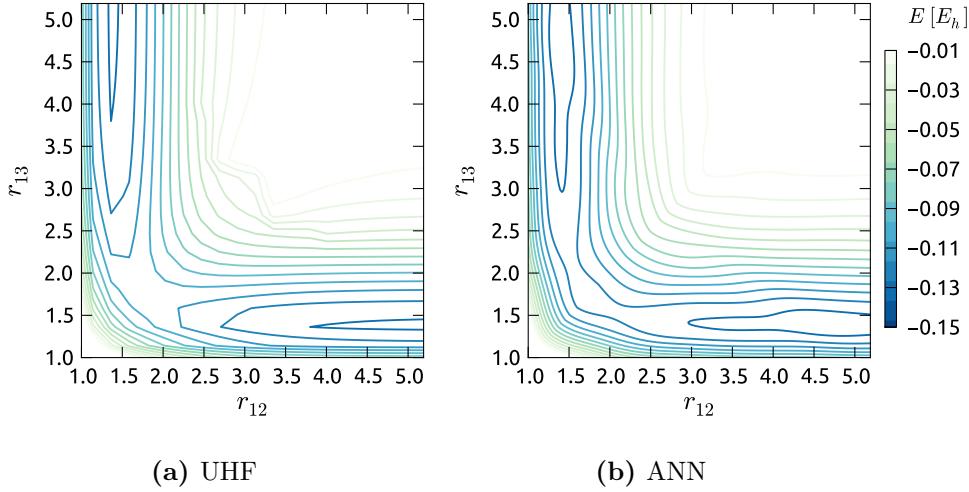


Figure 6.3: Comparison of the potential energy in a system of three hydrogen atoms, H_3 , at an angle of 180° and a range of distances, r_{12} and r_{13} , as (a) calculated with UHF and (b) approximated with an ANN. The approximation is quite good and picks up much of the characteristic shapes of the potential, which shows a clear preference for only one of atom 2 and 3 to be close to atom 1 at this angle. The energy values are offset, such that $E = 0$ corresponds to the configuration where the atoms are infinitely far apart. All values are in atomic units.

is able to find a good approximation of the potential.

Figure 6.3 also shows that the lowest energy is obtained by only having two hydrogen atoms close to each other. Considering that hydrogen usually appears as the H_2 molecule in nature, this is expected.

It should be noted that the potential of an H_3 system is symmetric. For a given angle, swapping r_{12} with r_{13} should result in the exact same potential energy. The ANN has therefore been trained for an unnecessarily large number of configurations. Further, this introduces a for the ANN to return an asymmetric function. The molecular dynamics simulations that use these potentials should therefore enforce symmetry by swapping r_{12} with r_{13} when $r_{12} < r_{13}$.

H₂O Potential Approximation

For H_2O , we expect there to be a minimum energy when both the hydrogen atoms are close to the oxygen atom. H_2O is a stable molecule, unlike H_3 . This is exactly what we get with a UHF calculation, shown in Figure 6.4a around $r_{OH_1} = r_{OH_2} \approx 1.809$. Fortunately, the feature is also visible in the ANN potential in Figure 6.4b. The ANN is able to reproduce the important features of the potential functional form.

Figure 6.4b shows the result of three ANNs. One ANN is used to calculate the two-body energy term between the two hydrogen atoms. Another calculates the two-body energy term for the two OH bonds, while the third is adding a three-body correction term on top of that.

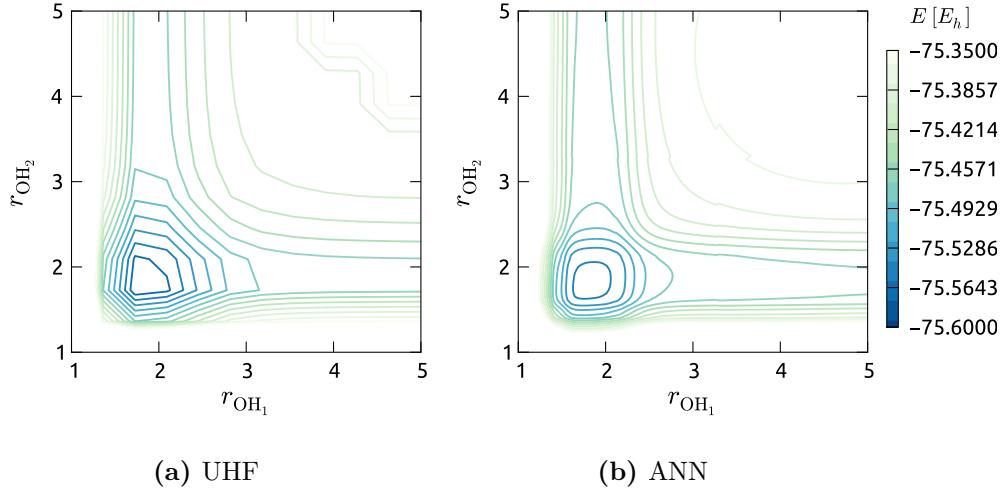


Figure 6.4: Potential energy of H_2O at the equilibrium angle (104.5°) and different distances of the OH bonds (all values are in atomic units). The plot in (a) is the result of a UHF calculation and (b) is an approximation to that with an ANN. The ANN reproduces the equilibrium length of about $r_{\text{OH}_1} = r_{\text{OH}_2} \approx 1.809$. Note that relatively few data points have been used in the UHF calculation, leading to the sharp edges in figure (a).

6.4 Implementing the Missing Gradient Function in FANN

We are also interested in the forces, which are essential in molecular dynamics simulations. We may take the derivative of the output of the ANN to find the forces because of the relation between the force and the potential,

$$\mathbf{F}_i = -\nabla V(\mathbf{R}_1, \dots, \mathbf{R}_i, \dots, \mathbf{R}_N). \quad (6.4)$$

We discussed how the derivatives of a feed forward ANN are calculated in Section 4.3. The evaluation of partial derivatives of ANNs with respect to the weights are necessary to perform training methods such as backpropagation. The FANN library therefore calculates said derivatives to train the networks. However, the derivative calculations are not available outside the training algorithm of FANN. We therefore need to reimplement this for force calculations.

Because the derivative calculations used in backpropagation training are conceptually the same as the derivatives of the network, we will base our code on the backpropagation code of FANN. In Section 4.3, we showed how the desired change δ_k could be propagated back from the last layer to the previous layer by the equation:

$$\delta_j = g'_k(x_j) \sum_k w_{jk} \delta_k. \quad (6.5)$$

In the training case, the desired change would be the error multiplied by the gradient of the output activation function, $\delta_k = e_k g'_k(x_k)$. To find the derivatives, we simply set $\delta_k = g'(x_k)$ to be the outermost element in the chain rule. We will therefore make use

of the `fann` property `train_errors` to store backpropagated derivatives. In FANN, this is used in backpropagation to store the errors.

The following is implemented in the molecular dynamics code, in the `FannDerivative` class. The code below is a stripped down version of this implementation, where all definitions have been removed and are assumed to be understood from the context. We begin by defining a few variables, and set `ann->train_errors[k]` to the derivative of the output activation functions:

```
void FannDerivative::backpropagateDerivative(struct fann *ann)
{
    // definitions
    // ...
    first_neuron = ann->first_layer->first_neuron;
    last_layer = ann->last_layer;
    output_neuron = (ann->last_layer-1)->first_neuron;
    ann->train_errors[output_neuron - first_neuron]
        = activationDerived(output_neuron->activation_function,
                            output_neuron->activation_steeplness,
                            output_neuron->value,
                            output_neuron->sum);
```

Next up, we go through all the layers from last to first and propagate the derivative backwards.

```
for(layer_it = last_layer - 1; layer_it > ann->first_layer; --layer_it)
```

For each layer, the derivative of each neuron is summed:

```
last_neuron = layer_it->last_neuron;
derivative_prev_layer = derivative_begin + ((layer_it - 1)->first_neuron
    - first_neuron);
for(neuron_it = layer_it->first_neuron; neuron_it != last_neuron;
    neuron_it++)
{
    tmp_derivative = derivative_begin[neuron_it - first_neuron];
    weights = ann->weights + neuron_it->first_con;
    for(i = neuron_it->last_con - neuron_it->first_con; i--)
    {
        derivative_prev_layer[i] += tmp_derivative * weights[i];
    }
}
```

Once we have the sum, we calculate derivatives of the activation functions of the neurons in the layer below and multiply the above sum by these:

```
derivative_prev_layer = derivative_begin + ((layer_it - 1)->first_neuron
    - first_neuron);
last_neuron = (layer_it - 1)->last_neuron;
if(layer_it - 1 > ann->first_layer) {
```

```

        for(neuron_it = (layer_it - 1)->first_neuron; neuron_it != last_neuron; neuron_it++)
    {
        *derivative_prev_layer
            *= activationDerived(neuron_it->activation_function,
                                neuron_it->activation_stEEPNESS,
                                neuron_it->value,
                                neuron_it->sum);
        derivative_prev_layer++;
    }
}
}
}
}
}

```

This implementation calculates δ_j layer by layer (see equation (6.5)). As a result, the partial derivative of the output with respect to the input is stored in the first elements of the `ann->train_errors` array. To obtain all the partial derivatives, the above method only needs to run once per output value.

The differences between the above implementation and the backpropagation algorithm used for training are subtle. In short, the derivatives of the output activation function are used instead of the errors, and we backpropagate all the way down to the first layer, instead of stopping at the second layer.

6.5 Using the Artificial Neural Network in Molecular Dynamics

To include the ANN potential in molecular dynamics, we must subclass the `*Particle-Force` classes of the Emdee program. For more information on these classes, please refer to the source code. In this section, we will focus on how the implementation is done for the two-body potential. Extending this to three-body forces requires some more work. The partial derivatives with respect to the particle positions have to be written in terms of derivatives with respect the distances and angles.

There is one function that we will focus on for this purpose, namely the overloaded `calculateAndApplyForce` of the `FannTwoParticleForce` class. We assume that the network has already been loaded from file and stored in the `network` object. Thereafter, we continue by finding the distance between the two atoms and check that it is smaller than the cutoff radius. Again, we will only list a stripped down version of the code:

```

void FannTwoParticleForce::calculateAndApplyForce(Atom *atom1, Atom *atom2)
{
    // definitions
    // ...
    Vector3 r12 = atom2->position() - atom1->position();
    double l12Squared = dot(r12, r12);
    if(l12Squared > cutoffRadius()*cutoffRadius()) {
        return;
    }
    double l12 = sqrt(l12Squared);
}

```

Next, we set up an input array with only one value, the distance r_{12} , that will be passed to the FANN network. The distance must be rescaled to the interval $[-1, 1]$, because FANN expects values within this range:

```
double dEdr12 = 0.0;
double potentialEnergy = 0;
fann_type input[2];
input[0] = network->rescaleDistance(l12);
```

Further, we call the `fann_run` function, defined in the FANN library, with a pointer to our `fann` network with the `input` array. The output is then rescaled back from the interval $[-1, 1]$:

```
fann_type *output = fann_run(network->ann, input);
double potentialEnergy = network->rescaleEnergy(output[0]);
```

Then we call the `backpropagateDerivate` function, defined in the previous section. After this, the derivative of the output with respect to the first (and only) input is stored in `network->ann->train_errors[0]`:

```
FannDerivative::backpropagateDerivative(network->ann, 0);
double dEdr12 =
    network->rescaleEnergyDerivative(network->ann->train_errors[0]);
```

However, this is the partial derivative with respect to the distance between the two atoms. To get the force, we need to find the partial derivatives with respect to the atoms' position components. This is done by multiplying $\partial E / \partial r_{12}$ with $\partial r_{12} / \partial x_1$, and equivalently for the other components:

```
double force = -1.0*(dEdr12);
double dEdr12Normalized = force / l12;

atom1->addForce(0, -r12.x() * dEdr12Normalized);
atom1->addForce(1, -r12.y() * dEdr12Normalized);
atom1->addForce(2, -r12.z() * dEdr12Normalized);

atom2->addForce(0, r12.x() * dEdr12Normalized);
atom2->addForce(1, r12.y() * dEdr12Normalized);
atom2->addForce(2, r12.z() * dEdr12Normalized);
```

Finally, half the potential energy of this interaction is contributed to each of the atoms:

```
atom1->addPotential(potentialEnergy / 2.0);
atom2->addPotential(potentialEnergy / 2.0);
}
```

For more details on the implementation, please refer to the source code (see page 3 for information on how to obtain the source code).

In this section, we have discussed the training of ANNs, specifically for two- and three-body potentials. Their ability to fit the results from our Hartree-Fock code is promising. Further, we discussed how to implement the missing gradient function in FANN. This enables fast force calculations. We may now move on to advanced molecular dynamics, which is the topic of the next chapter, before we apply the whole framework to full-scale simulations.

Chapter 7

Advanced Molecular Dynamics

With potential functions available from artificial neural networks (ANNs), we may start building a molecular dynamics (MD) code to perform our simulations. Some of the concepts of molecular dynamics were discussed in Chapter 2. In this chapter, we will build upon that theory and look at some implementation details. These are used to create the Emdee program. In addition, we will look into some advanced techniques used in molecular dynamics for improved performance. This is important because molecular dynamics scales with the square of the number of atoms ($\mathcal{O}(N^2)$) in its simplest form. However, if we cleverly use the cutoff radius of our forces and divide the system into separate domains, we can bring this scaling relation down to a linear order ($\mathcal{O}(N)$). Topping this with parallelization of the code, in our case with the Boost MPI library [75], allows for further performance improvements.

How to separate the total potential into two-, three- and many-body forces will also be addressed, and completes the earlier discussions of Sections 2.1 and 6.1. Tail corrections, used to ensure that the potentials go to zero at the cutoff, are necessary for energy conservation. Combining this with arbitrary potentials, such as ANN potentials, is tricky, but doable.

To obtain interesting physical properties, we are also in need of tools to control the system environment, such as the temperature, pressure and volume. For this purpose, we will use *system modifiers*, such as thermostats. The Berendsen thermostat and the Andersen thermostat are easily implemented in our code, which will be discussed in some detail. The more advanced Nosé-Hoover thermostat will also be mentioned, but not implemented.

An overview of the main flow in the Emdee program is shown in Figure 7.1 for reference. The source code of the Emdee program is available online (see page 3 for information on how to obtain the source code).

7.1 Subdividing the System

In the most crude implementation, molecular dynamics simulations scale badly with the number of atoms. Even with two-body forces only, optimizations are necessary to improve scaling. With two-body forces, there are $N(N - 1)/2$ interactions, where N is

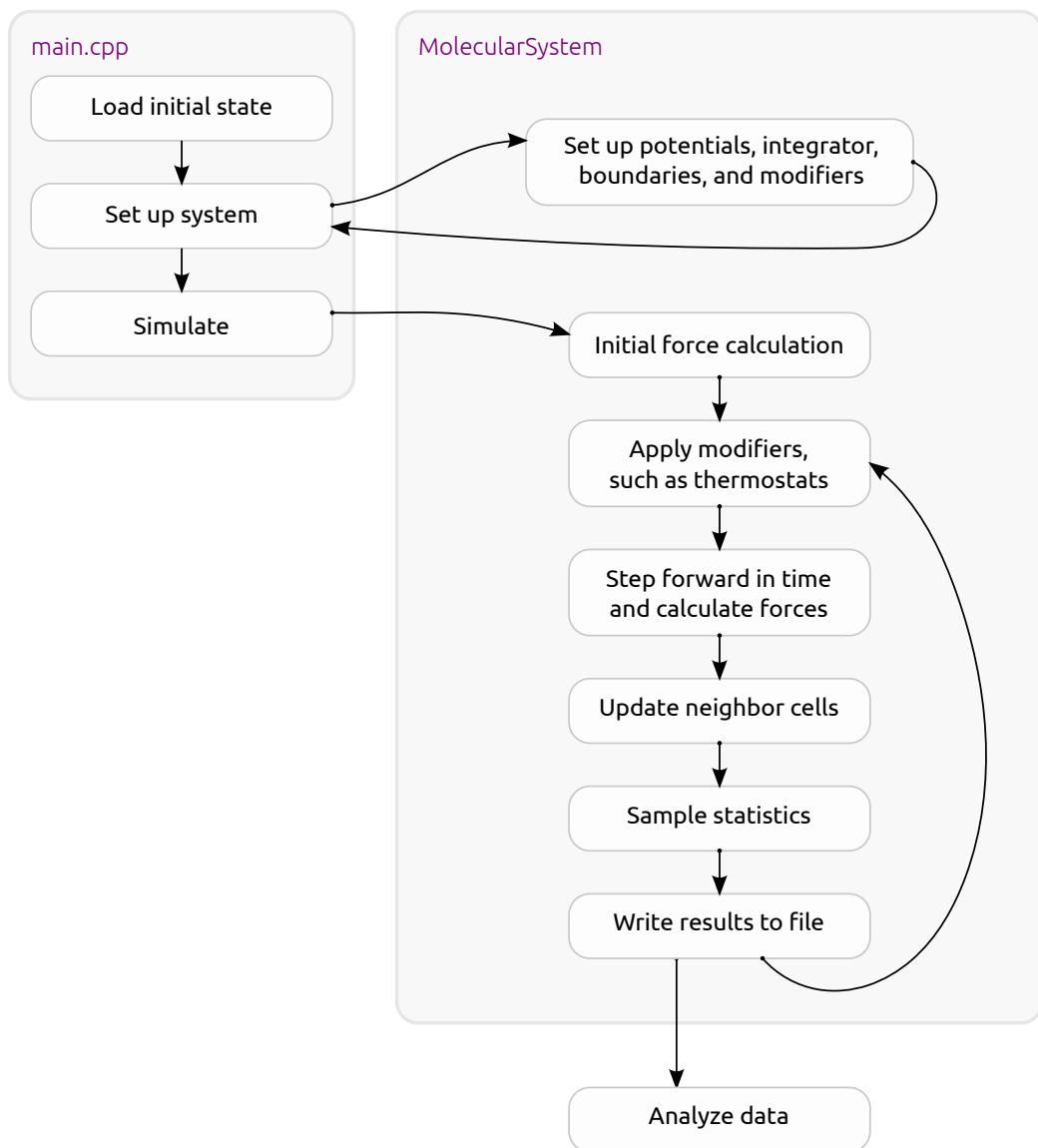


Figure 7.1: Flow chart of the Emdee program. The program loads an initial state of atoms and passes this along with the potential, integrator and modifier objects to the `MolecularSystem` class. Once the simulation begins, the forces are calculated and modifiers applied. Then the integrator is called to step forward in time. This will in turn request a force update. Further, the atoms are added to the correct neighbor cells and statistics are sampled. The results are saved to file and the loop is repeated. While the simulation is running, or once completed, the data may be analyzed by an external program.

the number of atoms. A two-fold increase in the number of atoms leads to a four-fold increase in the number of force calculations.

There are two common ways to reduce the number of calculations, assuming that the interaction falls off after a cutoff radius r_c : We may either calculate the distance between all atoms at one point in time, and only include force calculations between atoms that are less than the cutoff radius r_c apart in a few subsequent time steps. This is known as the method of Verlet lists. We may also divide the system into separate cells with sides of the same length as the cutoff radius r_c . Forces are calculated only between atoms in neighboring cells.

As it turns out, the neighbor cell method results in an algorithm that scales with N , while the Verlet lists method still scales with N^2 . The combination sometimes scales better than just using neighbor cells [22].

7.1.1 Verlet Lists

For Verlet lists to be useful, we need a cutoff r_c that is smaller than our simulation box. The cutoff also needs to be such that it does not have any adverse effects on what we want to measure. In other words, the cutoff radius varies from simulation to simulation. For a general approach, see the definition used for the “upper cutoff” in Section 6.1. For detailed information about the optimal cutoff radius for a given density, see Frenkel and Smit [22].

With the cutoff in place, we iterate over all pairs of atoms in our system and calculate the square of their distance, r_{ij}^2 . If the square of their distance is less than the square of the cutoff radius, the force also needs to be calculated. We save some time by not calculating the force between all atoms, but we still need to calculate the distances. To reduce the number of calculations, we create a list of neighbors within a bigger radius r_v , and store this for a few subsequent time steps. We further assume that these are the only particles that may appear within the original cutoff radius r_c during this time. If we also collect some information about the velocities of these particles, it will be possible to predict (to a certain degree) how much larger r_v should be than r_c .

This will reduce the number of distance and force calculations by a great amount, but every time we update these lists, we still will have to iterate over all pairs of atoms. This means that the method still scales with N^2 , although with a smaller factor.

7.1.2 Neighbor Cells

The other option is to divide the system into separate cells. This will result in a scheme that scales closely to N . The idea is to divide the system into cells with sides of about the same length as the cutoff distance r_c . Atoms in one cell only interact with atoms in its own and neighboring cells. In two dimensions, each cell has 8 neighbors, while in three dimensions the number is 26.

In the Emdee program, cell division is performed in the `MolecularSystem::setupCells` function (see page 3 for information on how to obtain the source code). A stripped version of this function is shown below. Here, the `m_boundaries` member is a 2×3 matrix, where the first row is the lower boundary in x , y and z -directions, while the second row is the upper boundary:

```

void MoleculeSystem::setupCells(double requestedCellLength) {
    // ...
    for(int iDim = 0; iDim < 3; iDim++) { // Calculate the number of cells
        double totalLength = m_boundaries(1,iDim) - m_boundaries(0,iDim);
        m_globalCellsPerDimension(iDim) = totalLength / cutoffRadius;
        m_cellLengths(iDim) = totalLength / m_globalCellsPerDimension(iDim);
        nCellsTotal *= m_globalCellsPerDimension(iDim);
    }
    // ...
    irowvec indices = zeros<irowvec>(3);
    for(int i = 0; i < nCellsTotal; i++) {
        MoleculeSystemCell* cell = new MoleculeSystemCell(this);
        cell->setID(i);
        mat cellBoundaries = m_boundaries;
        rowvec shiftVector = m_cellLengths % indices;
        cellBoundaries.row(0) = shiftVector;
        cellBoundaries.row(1) = shiftVector + m_cellLengths;
        cell->setBoundaries(cellBoundaries);
        cell->setIndices(indices);
        m_globalCells.push_back(cell);
        indices(0) += 1;
        for(uint iDim = 1; iDim < indices.size(); iDim++) {
            if(indices(iDim - 1) > m_globalCellsPerDimension(iDim - 1) - 1) {
                indices(iDim - 1) = 0;
                indices(iDim) += 1;
            }
        }
    }
    int nNeighbors;
    for(MoleculeSystemCell *cell1 : globalCells()) { // Find neighboring cells
        nNeighbors = 0;
        for(int i = -1; i <= 1; i++) {
            for(int j = -1; j <= 1; j++) {
                for(int k = -1; k <= 1; k++) {
                    irowvec direction = {i, j, k};
                    irowvec shiftVec = (cell1->indices() + direction);
                    rowvec offsetVec = zeros<rowvec>(3);
                    // ... correct direction and offset if system is periodic
                    // ... and the cell is outside the boundaries
                    // ... (see source code for details)
                    MoleculeSystemCell* cell2 =
                        m_globalCells.at(cellIndex(shiftVec(0), shiftVec(1),
                        shiftVec(2)));
                    cell1->addNeighbor(cell2, offsetVec, direction);
                    nNeighbors++;
                }
            }
        }
    }
    addAtomsToCorrectCells(m_atoms);
}

```

The final call is to a function that iterates over all atoms in the system and places them in the correct cells. Here is a snippet from `MolecularSystem::addAtomsToCorrectCells`:

```

void MoleculeSystem::addAtomsToCorrectCells(vector<Atom *> &atoms) {
    for(Atom* atom : atoms) {
        Vector3 position = atom->position();
        for(int iDim = 0; iDim < m_nDimensions; iDim++) {
            if(m_isPeriodicDimension[iDim]) {
                double sideLength = (m_boundaries(1,iDim) -
                    m_boundaries(0,iDim));
                position(iDim) = fmod(position(iDim) + sideLength * 10,
                    sideLength);
            }
        }
        int i = position(0) / m_cellLengths(0);
        int j = position(1) / m_cellLengths(1);
        int k = position(2) / m_cellLengths(2);

        // ... check and report if for some reason an atom ends up outside all
        // cells ...

        MoleculeSystemCell* cell = m_globalCells.at(cellIndex(i,j,k));
        cell->addAtom(atom);
    }
}

```

At each time step, we need to iterate over all atoms in each cell, and for each atom, we calculate the two-body forces and create a neighbor list of atoms (not cells). The list is used in three-body force calculations. The following code snippet is a stripped down version of the function `MoleculeSystemCell::updateTwoParticleForceAndNeighborAtoms`:

```

// ...
for(uint iNeighbor = 0; iNeighbor < m_neighborCells.size(); iNeighbor++) {
    MoleculeSystemCell* neighborCell = m_neighborCells[iNeighbor];
    const vector<Atom*>& neighborCellAtoms = neighborCell->atoms();
    for(Atom* atom1 : m_atoms) {
        for(Atom* atom2 : neighborCellAtoms) {
            if(atom1->id() >= atom2->id()) {
                continue;
            }
            double distanceSquared = Vector3::distanceSquared(atom1->position(),
                atom2->position());
            if(distanceSquared > cutoffRadiusSquared) {
                continue;
            }
            atom1->addNeighborAtom(atom2);
            atom2->addNeighborAtom(atom1);
            twoParticleForce->calculateAndApplyForce(atom1, atom2);
        }
    }
}
// ...

```

7.1.3 Neighbor Cells and Verlet Lists Combined

Now that we have explored two options to subdivide the system into smaller units, we may maximize our benefit by combining the two. According to [22], “the use of cell lists removes the main disadvantage of the Verlet list for a large number of particles – scales as N^2 – but keeps the advantage of an efficient energy calculation”. However, the Verlet lists have not yet been implemented in Emdee.

7.2 What Goes into the n -Body Terms?

In the following, we assume that the inherent energy of each atom has been subtracted. The inherent energy is the one we find if we perform our *ab initio* calculation on the atom alone in space. This is the energy contribution we also would get from this atom if it was separated by an infinitely large distance from the rest of the system. In other words, it is a shift in potential energy that makes it a bit harder to work with our energy terms whenever we sum them up, because each atom raises the overall potential energy. Considering the cutoff distance of the potentials in molecular dynamics, this would introduce an energy whenever two particles came within this cutoff distance, even though they were too far away to interact. So in short, this is just a term that can be removed by choice. In our case, the potential energy will now be zero when the atoms are infinitely far apart.

In Section 2.1, we noted that the potential energy surface (PES) could be written as a sum of n -body terms:

$$V(\mathbf{r}) \approx \sum_{k=1}^N V_1(\mathbf{r}_k) + \sum_{k < l}^N V_2(\mathbf{r}_k, \mathbf{r}_l) + \sum_{k < l < m}^N V_3(\mathbf{r}_k, \mathbf{r}_l, \mathbf{r}_m) + \dots . \quad (7.1)$$

Some comments on the quality of this approximation were also made. In this section, we'll look at the details of the different terms.

There is no unique definition to what goes into the other terms. We have only required that they should sum up to the total potential energy. If the sum over V_3 terms equals the total potential energy, then the sum over V_2 terms could be rendered superfluous and set to zero. However, there are good reasons not to put everything into the V_3 terms. The V_2 terms are generally less expensive to compute, because only two atoms are involved. The best option is therefore to limit both the number of terms, and the order of the included terms, as much as possible.

7.2.1 One-Body Term

The term V_1 is used to describe external forces, such as gravity. This may be added to the final potential straightforwardly after parameterizing the other terms. We therefore leave V_1 out for now.

7.2.2 Two-Body Term

The two-body interaction terms should be such that they at least take care of the cases where only two atoms are in the proximity of each other. It should therefore be ideal to

fit the two-body terms to an *ab initio* calculation with only two atoms. The three-body term will then be used as a correction to the two-body term for configurations with three atoms, and higher-order terms as corrections to their corresponding number of atoms.

7.2.3 Three-Body Term

Let's consider the case with only three atoms. The energy E calculated from *ab initio* methods describes the total potential energy $V(\mathbf{r})$ of the molecular dynamics system. The three particle term should therefore be the correction to the sum of V_2 terms

$$V_{123} = E - (V_{12} + V_{13} + V_{23}). \quad (7.2)$$

The triple for loop for three-body force calculations is shown below:

```
for(Atom* atom1 : atoms) {
    for(int i = 0; i < atom1.neighbors().size(); i++) {
        Atom* atom2 = atom1.neighbor(i);
        for(int j = i + 1; j < atom1.neighbors().size(); j++) {
            Atom* atom3 = atom1.neighbor(j);
            calculatePotential(atom1, atom2, atom3);
        }
    }
}
```

Here, a combination of three atoms will be passed to the `calculatePotential` three times. Each time, the ordering of the atoms will be different. However, we only need one configuration to determine the potential and calculate the forces. Therefore, we have to divide the force and potential contributions by three inside this function.

Alternatively, we could calculate the potential for only one of the three combinations. In the case of heterogeneous systems, where we have multiple particle types, the easiest way to do this is by figuring out which atom is the central atom in the specific potential term. For H₂O, this would typically be the oxygen atom. For the case of homogeneous systems, we may select the combination where `atom1` has a lower ID than both `atom2` and `atom3`. As mentioned in Section 6.1, for ANN potentials it may be useful to ensure that the angle of the central atom in the three-body calculation is above $\pi/3$. That way, fewer configurations are needed in training the ANN.

7.2.4 Many-Body Term

If the potential should include terms of higher order, the procedure is the same as for the three-body term. Start with setting up a system of n particles and calculate the *ab initio* energy, but now look at the n -body contribution as a correction to the energy from $n - 1$ -body terms and lower. This means that the contribution from all four-body terms should be the difference from the energy calculated in a configuration with four particles to what we had with two- and three-body terms:

$$\sum V_4(\mathbf{r}) = E(\mathbf{r}) - \sum V_3(\mathbf{r}) - \sum V_2(\mathbf{r}). \quad (7.3)$$

In this thesis, however, we will limit ourselves to two- and three-body terms.

7.3 Tail Correction

A potential that does not go to zero at the cutoff can be hazardous to energy conservation. Anytime a group of particles go across a cutoff, we will experience a jump in energy. This section is devoted to describing some methods for correcting the potential around the cutoff.

7.3.1 Adding a Shift to the Two-Particle Potentials

One option is to shift the original potential $u_o(r)$ such that it goes to zero at the cutoff radius r_c :

$$u(r) = \begin{cases} u_o(r) - u_o(r_c), & r \leq r_c, \\ 0, & r > r_c. \end{cases} \quad (7.4)$$

In the case of a Lennard-Jones potential, we would get

$$u_{\text{LJ}}(r) = \begin{cases} 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6 \right] - 4\epsilon \left[\left(\frac{\sigma}{r_c}\right)^{12} - \left(\frac{\sigma}{r_c}\right)^6 \right], & r \leq r_c, \\ 0, & r > r_c. \end{cases} \quad (7.5)$$

This takes the potential to zero at the cutoff, but leaves the force unaltered.

7.3.2 Damping the Potential Near the Cutoff

For potentials with more than two particles, the shift method does no longer fix the issue, unless the potential converges to the same value for all the limits of its variables, r_{12}, r_{13}, \dots , which is unlikely. We therefore need another approach when working with these potentials, which is to add a damping factor.

The damping factor should be a function goes from 1 to 0 on the interval of the tail correction, $r_{ij} \in [r_d, r_c]$. We get the corrected potential u from the original potential u_o in the following manner:

$$u(\mathbf{r}) = \begin{cases} u_o(\mathbf{r}), & r_{ij} \in [0, r_d], \\ u_o(\mathbf{r})f(r_{ij}), & r_{ij} \in [r_d, r_c], \\ 0, & r_{ij} > r_c. \end{cases} \quad (7.6)$$

There are several possible forms of the damping function, but it should hold a few important properties. As already mentioned, it should not alter the potential at r_d and take it to zero at r_c . So we have

$$f(r_d) = 1.0, \quad (7.7a)$$

$$f(r_c) = 0.0. \quad (7.7b)$$

Adding the damping factor to the potential will also alter the force. The force depends on the derivatives of the potential with respect to any of the variables of the

potential. On the tail correction interval, where any $r_{ij}, r_{ik}, \dots \in [r_d, r_c]$, we get the derivative as:

$$\frac{\partial}{\partial r_{ij}} u(\mathbf{r}) = u_o(\mathbf{r}) \frac{\partial}{\partial r_{ij}} f(r_{ij}) + f(r_{ij}) \frac{\partial}{\partial r_{ij}} u_o(\mathbf{r}) \quad (7.8)$$

Ideally, the derivative of the potential should remain unaltered at r_d and become zero at r_c . Combining this requirement with (7.7) results in the following requirements on the derivative of the damping function:

$$\frac{\partial}{\partial r_{ij}} f(r_d) = 0.0, \quad (7.9a)$$

$$\frac{\partial}{\partial r_{ij}} f(r_c) = 0.0. \quad (7.9b)$$

One damping function that has these properties is the following, which has been inspired by the three-particle potential invented for SiO₂ by Vashishta et.al[36]:

$$f(r_{ij}) = \exp \left(\beta \frac{r_{ij} - r_d}{r_{ij} - r_c} \right) \left(\frac{r_{ij} - r_d}{r_c - r_d} + 1 \right). \quad (7.10)$$

The derivative of this is

$$\frac{\partial}{\partial r_{ij}} f(r_{ij}) = \beta \exp \left(\beta \frac{r_{ij} - r_d}{r_{ij} - r_c} \right) \left(\frac{(r_{ij} - r_d)(r_{ij} + 2r_d - 3r_c)}{(r_c - r_d)(r_c - r_{ij})^2} \right). \quad (7.11)$$

This fulfills the requirements of equations (7.7) and (7.9), which is seen by inserting either r_c or r_d into the above.

Note that when using multiple damping functions for several parameters, the force is also indirectly dependent on the other damping factors. For the case where we have both $f(r_{ij})$ and $g(r_{ik})$ as damping factors, the derivative with respect to r_{ij} of the potential becomes:

$$\frac{\partial}{\partial r_{ij}} u(\mathbf{r}) = u_o(\mathbf{r}) g(r_{ik}) \frac{\partial}{\partial r_{ij}} f(r_{ij}) + g(r_{ik}) f(r_{ij}) \frac{\partial}{\partial r_{ij}} u_o(\mathbf{r}). \quad (7.12)$$

This has been implemented in the `FannThreeParticleForce` class (see page 3 for information on how to obtain the source code).

There are other damping functions that might work equally well (see Griebel, Knapek, and Zumbusch [31] and Tersoff [37] for some examples), but this holds a few nice properties and is not very hard to implement. It also works even if $r_c < r_d$, which is useful when applied to angles with bounds.

7.4 Thermostats

Molecular dynamics simulations typically have a constant number of particles, constant volume and constant energy, known as the microcanonical (NVE) ensemble. Although we may modify either of these three deliberately during a simulation, they will not vary

if the system is left to itself. However, other properties, such as temperature, should be expected to vary unless the system is in equilibrium to begin with.

Other ensembles, such as the canonical (NVT) ensemble, can be sampled by introducing system modifiers, such as thermostats. The idea of a thermostat is to simulate exchange of energy with a heat bath. If the system is non-periodic, such a thermostat can be modeled by fixing a number of atoms to the walls that absorb or dissipate energy upon collision with the atoms of the system. If the system is periodic, the energy transfer needs to be modeled by fake collisions throughout the system, or by scaling the velocities, known as the *Andersen thermostat* and the *Berendsen thermostat*, respectively. A third alternative is to introduce a friction term in the equations of motion, known as the Nosé-Hoover thermostat. The Andersen and Berendsen thermostats and their implementation details are discussed in this section, while we will only give a brief review of the more advanced Nosé-Hoover thermostat.

7.4.1 Measuring Temperature in Molecular Dynamics

Before introducing the details of the thermostats, we should first define temperature in molecular dynamics. From the equipartition theorem of thermodynamics [76], the temperature of the system and its kinetic energy are related as

$$\langle E_k \rangle = \frac{3N}{2} k_B T, \quad (7.13)$$

where N is the total number of atoms in the system, such that there are $3N$ degrees of freedom. The constant k_B is the Boltzmann's constant. Note that if the center of gravity of the system is assumed to be constant, three degrees of freedom should be subtracted. Additionally, if the rotation of the system can be ignored, another three degrees of freedom should be subtracted [31].

This equation can be rewritten to obtain an expression for the temperature:

$$T = \frac{2}{3Nk_B} \langle E_k \rangle = \frac{2}{3Nk_B} \sum_{i=1}^N \left\langle \frac{1}{2} m_i v_i^2 \right\rangle. \quad (7.14)$$

Here, m_i is the mass, and v_i is the velocity of atom i . The sum represents the expectation value of the kinetic energy of the system, which is obtained by averaging the kinetic energy of the system over many time steps.

To initialize the system at a given *instantaneous* target temperature T_{target} , we set the magnitudes of the velocities of all particles to random numbers picked from a Maxwell-Boltzmann distribution with the target temperature. This is done by picking a velocity vector whose components are normally distributed with standard deviation $\sqrt{k_B T_{\text{bath}} / m}$. By insertion in the above equation, T will equal the target temperature T_{target} . However, this only sets the kinetic energy of the system such that the temperature of that first time step is the target temperature. Because the initial configuration of the positions can give a large range of potential energies, the temperature may vary when we start the proper measurement by averaging over time. Thus, we have not strictly chosen an ensemble with the temperature T_{target} [8]. To generate the correct ensemble, we turn to the use of thermostats.

7.4.2 Andersen Thermostat

The Andersen thermostat simulates the coupling to a heat bath by occasional collisions with randomly selected atoms in the system [22]. To implement this thermostat, we need to decide on the temperature of the heat bath $T_{\text{bath}} = T_{\text{target}}$, and the strength of the coupling between the system and the heat bath. The strength is measured by the mean collision frequency $\nu = 1/\tau$, where τ is the mean time between collisions.

For each particle in the system, a random uniformly distributed number is picked from the interval $z \in [0, 1]$. If the number is below $\nu\Delta t$, where Δt is the time step of the simulation, the atom is assumed to have collided with the heat bath and is assigned a new random velocity. The velocity magnitude is chosen from a Maxwell-Boltzmann distribution, as in the temperature initialization above. Over time, this will drive the system to an ensemble with the target temperature.

One disadvantage of the Andersen thermostat is that it will disturb the trajectories of the system. Although it ensures sampling from the canonical ensemble, it may perturb chemical reactions due to the rapid changes in the velocities. The collision time must therefore be chosen with some care. Alternatively, the system can be equilibrated at a given temperature with the thermostat enabled, before it is disabled for sampling of other observables.

An implementation of the Andersen thermostat is found in the [AndersenThermostat](#) class.

7.4.3 Berendsen Thermostat

An alternative to the occasional collisions of the Andersen thermostat, is to use velocity scaling to drive all velocities towards the target temperature. The Berendsen thermostat does this by introducing a factor

$$\beta = \sqrt{1 + \gamma \left(\frac{T_{\text{bath}}}{T} - 1 \right)}, \quad (7.15)$$

that all the velocities are multiplied by:

$$\mathbf{v}_i = \beta \mathbf{v}_i. \quad (7.16)$$

The relaxation constant $\gamma = \Delta t/\tau$ defines the strength of the thermostat, where τ is the relaxation time. If $\gamma = 1$, the instantaneous temperature T is immediately changed to the temperature of the heat bath T_{bath} . If $\gamma = 0$, the thermostat is effectively disabled.

The benefit of the Berendsen thermostat is that it will not cause abrupt changes in the trajectories of the atoms, but it does not lead to a canonical ensemble for small systems. Some care should also be made to ensure that the Berendsen thermostat does not end up transferring all the kinetic energy of the system from internal vibrations to overall translational or rotational motion, which is a known issue of velocity scaling schemes [77].

An implementation of the Berendsen thermostat is found in the [BerendsenThermostat](#) class.

7.4.4 Nosé-Hoover Thermostat

It works by incorporation of a friction term directly into the time integration scheme. In the velocity-Verlet algorithm, this accounts to replacing the force calculations by

$$\mathbf{F}_i^n = \mathbf{F}_i^n - \xi^n m_i \mathbf{v}_i^n. \quad (7.17)$$

The Nosé-Hoover is a bit harder to implement, but, contrary to the Andersen thermostat, it does not alter the trajectories significantly. The downside of the Nosé-Hoover thermostat is that it may suffer from non-ergodic behavior, meaning that it will not sample the whole ensemble properly. To alleviate this, there are extensions to the scheme, such as Nosé-Hoover chains [22].

7.5 Testing the Molecular Dynamics Simulator

As with the other components in this thesis project, testing and verifying the integrity of the molecular dynamics code is very important. One challenge is that many particles and components are involved, making it a very complex system to verify solely by looking at the numbers.

Creating tests for molecular dynamics is a somewhat more creative process than for the other parts of this project. In Hartree-Fock, there are several simple numbers that may be measured and compared for test cases, such as checking that the integrator does its job correctly compared to a brute force numerical integrator, or that the resulting energy for the H₂ system in ground state matches that of other published results. For molecular dynamics, however, there are only a few simple tests, such as checking that the force implementation returns the right value for two atoms at a given distance.

In addition to these simple tests, there are a few more complex tests that can be performed. Some of these tests require setting up a whole system, and may therefore diverge from the simplicity of regular unit tests. However, they are quite useful to verify the code. A few examples are listed:

- Create a "counting" force class that does nothing but add 1 to the potential energy of each atom it is applied to. This way the potential energy stored on an atom represents the number of atoms it has interacted with.
- Set up a system with one atom in the center of each cell and use the counting force class to check that it actually interacts with the right number of neighbors. If the cells are cubic, this requires the counting force class to have a cutoff equal to the cell side length to obtain 6 neighbors (one for each side of the cell) or $\sqrt{2}$ times the cell side length to get 26 neighbors.
- Set up a system with periodic boundary conditions, and a constant velocity in a random direction. The magnitude should be such that an integer number of time steps should take all the particles back to their initial positions.
- Create one or a few atoms in what should be their equilibrium configuration and observe that they stay still for a long period of time.

- Count the number of atoms in the system before and after a simulation to ensure that nothing goes wrong when moving atoms between cells. This is especially important when transferring atoms between processors in parallel molecular dynamics code.

This list is of course not comprehensive, and there are certainly more tests that could be performed. Some, but not all, of the above tests are available in the tests directory of the Emdee source code. These have been implemented with the `UnitTest++` library, which is discussed in Appendix B.

7.6 Verification: Crystallization of Argon

To verify that the molecular dynamics simulator works as expected, we will attempt to reproduce the results of Griebel, Knappek, and Zumbusch [31] for argon crystallization. Our setup will be slightly different, with more atoms, a smaller time step and a lower target temperature, but otherwise equivalent to their simulation.

The instantaneous pressure of a system can be measured as

$$P_{\text{inst}} = \rho_n k_B T + \frac{1}{3\Omega} \left\langle \sum_{i < j} \mathbf{F}_{ij} \cdot \mathbf{r}_{ij} \right\rangle, \quad (7.18)$$

where Ω is the volume of the system, ρ_n is the number density, k_B is Boltzmann's constant, \mathbf{v}_i is the velocity of particle i , \mathbf{F}_{ij} is the interacting force between particles i and j , and $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$. This follows from the ideal gas pressure and the virial theorem. For a full derivation, see the texts of Ercolessi [78] and Hafreager [3].

The interaction between the atoms is described by a standard, unshifted Lennard-Jones potential,

$$V_{\text{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right], \quad (7.19)$$

where, in the case of argon atoms, $\sigma = 3.4 \text{ \AA}$ and $\epsilon = 1.65 \times 10^{-21} \text{ J}$. The time step is chosen to be $\Delta t = 1.0$ in atomic units, or $\Delta t = 0.2419 \times 10^{-4} \text{ ps}$. The simulation starts at a boiling temperature of $T_{\text{boiling}} = 360 \text{ K}$, with $N = 16^3 = 4096$ atoms uniformly distributed with cubic symmetry in a simulation box with sides $L = 63.92 \text{ \AA}$. The system will quickly drift from the initial configuration and become randomized due to the high temperature.

To keep the system at the boiling temperature, a Berendsen thermostat is applied for the first 10 000 time steps with a relaxation time of $\tau = 2.419 \times 10^{-2} \text{ ps} = 100\Delta t$. Thereafter, the system is cooled linearly to $T_{\text{target}} = 20 \text{ K}$ by repeatedly applying the thermostat with slightly lower target temperatures for 100 time steps, before equilibrating for another 100 time steps. The linear cooling is performed in 1000 such stages, adding up to 200 000 time steps in total.

In Figure 7.2, the potential energy and the pressure is shown as a function of time in the cooling period of this simulation. Here, we see that as the temperature is being lowered, the potential energy falls, while the pressure decreases. However, after about 372 ps, there is an abrupt change in the slope of both curves. The potential energy

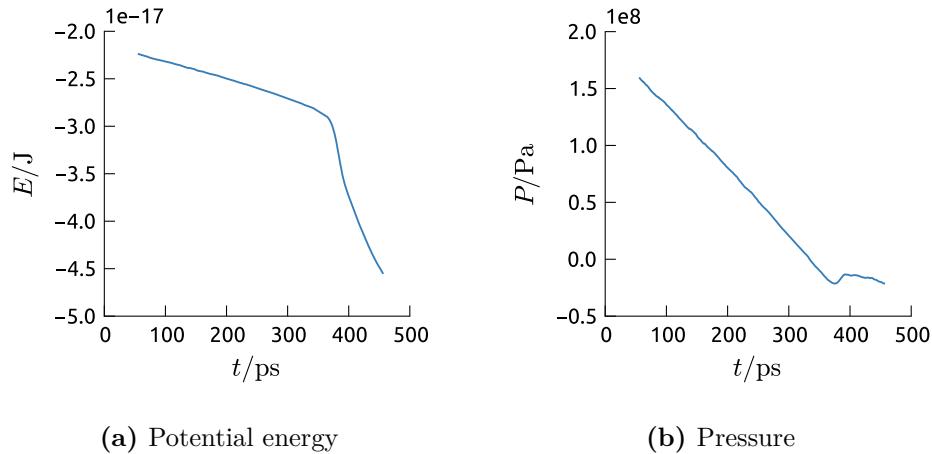


Figure 7.2: Potential energy and pressure in a simulation of crystallization of argon. The system is cooled linearly by a Berendsen thermostat. After about 372 ps, the temperature drops below 90 K, and the system goes through a phase transition, as can be seen by the sudden slope changes in both plots. This is not far from the boiling and melting points of argon, at 87 K and 84 K, respectively.

suddenly drops drastically, and the pressure flattens out. This is an indication of a phase change, and happens right after the temperature of the system goes below 90 K. The physical boiling point of argon is 87 K, and the melting point is 84 K. This event corresponds closely to these phase changes. Additionally, the same behavior was reported by Griebel, Knapek, and Zumbusch [31], based on their molecular dynamics simulations.¹

An abrupt change is also apparent when visualizing the system. At the boiling temperature of 360 K, the atoms are freely moving around, but below 90 K, the atoms get stuck in a crystal structure. Figure 7.3 shows a visualization of the solid structure at the end of this simulation. We also see that a large hole has manifested itself after the phase transition. This is a non-physical artifact that occurs because the volume is kept fixed. In a physical system, the volume would have adapted to temperature change. This can, on the other hand, be modeled by methods that enable simulations of the NPT ensemble [31].

7.7 High-Performance Computing: Parallelization

With neighbor lists, molecular dynamics simulations scale as $\mathcal{O}(N)$, which means that we should be able to increase the system size linearly by throwing in extra processing

¹Griebel, Knapek, and Zumbusch [31] find that the pressure is increasing with decreasing temperature before hitting a maximum at the phase change. This seems unlikely to be correct, and their plot can be reproduced by (wrongly) changing the sign of the virial term in equation (7.18). This has lead me to believe that there might be a typo in their code. To verify this, the same system was simulated with the LAMMPS molecular dynamics package [79]. This simulation matched our results, with decreasing pressure for decreasing temperature.

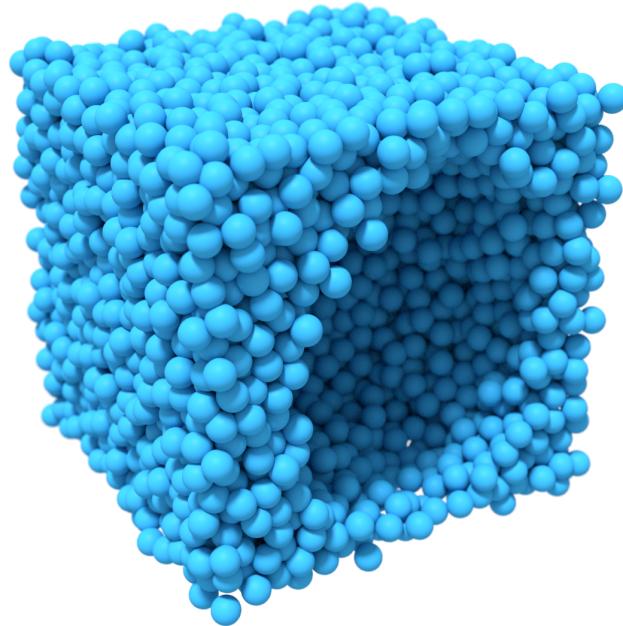


Figure 7.3: Snapshot at the end of a simulation of argon crystallization. After a phase change, a non-physical hole appears. This happens because the volume is kept fixed throughout the simulation, while it would have been able to adapt in a physical system.

power. However, to make extra power available, we need to parallelize the code such that every CPU on our computer or computing cluster can be made use of.

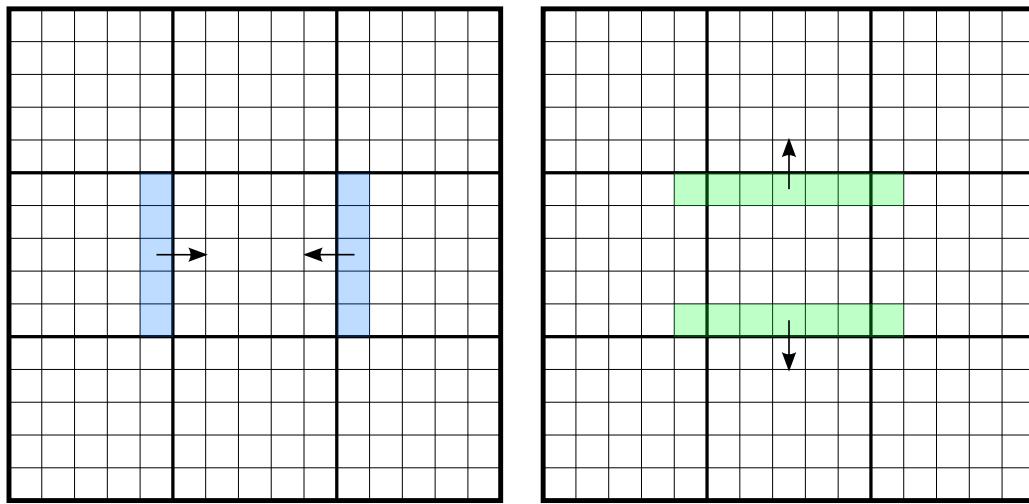
In this section, we will discuss how we can make use of the Boost MPI library to create a parallel version of our molecular dynamics code.

7.7.1 An Argument for Boost MPI

Although the MPI library is well designed, it doesn't sit well in a C++ application. The MPI library is very C-centric, in the sense that it works with low-level data structures and function calls. This means that it is easy to make mistakes when passing large data structures between processors. Fortunately, the Boost MPI library [75] fits the C++ design much better, with wrappers for the most common MPI calls. The documentation for Boost MPI is thorough and should provide a good starting point for anyone who wants to learn MPI for the first time. In addition, the function names are similar to those found in the original C library. If you are moving from C MPI to Boost MPI, you are likely to experience an easy transition. However, if you are an advanced MPI user requiring subtle and cutting-edge features of the library, you might want to check if these are available in Boost MPI before transitioning.

To give some weight to this argument, let's look at an example of how to receive data to a C++ `vector<int>` in C MPI:

```
vector<int> recvVector;
```



(a) Cells received from left and right

(b) Cells sent to front and back

Figure 7.4: To reduce the number of communications, the cells (a) received from the processors to the left and right are (b) passed on to the processors in front and back. The thick lines show processor division, while thin lines show neighbor cell division. In the next step (not shown here), all the received cells would also be passed on to the neighbors above and below. This results in a total of six communication stages per time step.

```
int recvSize = 0;
MPI_Status status;
MPI_Recv(&recvSize, 1, MPI_INT, fromRank, tag1, MPI_COMM_WORLD, &status);
recvVector.resize(recvSize, 0);
MPI_Recv(recvVector.data(), recvVector.size(), MPI_INT, fromRank, tag2,
         MPI_COMM_WORLD, &status);
```

The equivalent version using Boost MPI is reduced to two lines:

```
vector<int> recvVector;
world.recv(fromRank, tag, recvVector);
```

The fact that Boost.MPI takes care of resizing the receiving vector is just one of the many benefits of the C++ version of the MPI library.

7.7.2 Parallel Molecular Dynamics

The main idea in parallel molecular dynamics is that each processor is responsible for a volume of neighbor cells. Each processor takes care of calculating the forces and stepping forward in time for all atoms in its interior cells. Because positions of the atoms in cells on other processors are needed for the force calculations, each processor communicates the cells on its edges with adjacent processors every time step. These cells are named *ghost cells*, and are involved in force calculations.

In three dimensions, each processor has 26 neighbors. If the number of processors in use is lower than 26, some neighbors may appear twice or even be the same processor. However, there is no need to perform communications with all 26 neighbors. Communication with the six nearest neighbors is enough, if these are used as intermediate information carriers. If a processor first communicates with neighbors to its left and right, it may pass this information on to the neighbors to its front and back. This information may in turn be passed on to the neighbors above and below. This is illustrated in Figure 7.4 for the cells received from left and right neighbors, and for the cells passed on to the front and back neighbors.

Parallel molecular dynamics has been implemented in this project using Boost MPI. The main logic of this implementation is found in the `Processor` class. Upon initialization of the system, the function `Processor::setupProcessors()` sets up the cells on the current processor and maps the neighboring processors to the different spatial directions (left, right, front, back, up and down). The `MoleculeSystem` may now use `Processor` to get a list of the local and global cells whenever needed. For each time step, after the atoms have been moved by the integrator, the `Processor::communicateAtoms()` function is called, and takes care of sending and receiving data to and from the neighboring processors, in the order outlined above. Once all atoms have been sent and received, the `MoleculeSystem::refreshCellContents()` places the atoms in the correct cells and the simulation is continued. With this method, there is no need for an explicit stage where the atoms are moved between the processors if they cross the cell boundary. If, during the course of one time step, an atom has moved out of the domain of a processor, it must have been in a cell on the edge of the processor and will be communicated anyways, thus being moved as a side effect of this communication and the call to `MoleculeSystem::refreshCellContents()`.

Due to the large amount of communication involved, a parallel implementation is only beneficial if the system size is large enough for each processor to have a significant number of internal cells. At some point, the communication time becomes longer than the calculation time on a single processor, rendering a parallel implementation unnecessary. This especially holds true when network communication is involved, such as on a computing cluster. Performance benchmarks should therefore be performed to find the optimal number of processors to use before running full-scale simulations.

Chapter 8

Hydrogen Molecules: Results of the Complete Workflow

With all pieces at hand, it is time to gather results from the complete workflow. With the Emdee program, we are ready to run molecular dynamics (MD) simulations on atoms with artificial neural network (ANN) potentials. These ANNs are parameterized from *ab initio* unrestricted Hartree-Fock (UHF) calculations by the Kindfield program. The resulting simulation should have much of the predictive power of an *ab initio* molecular dynamics simulation. However, by avoiding the full quantum mechanics machinery at each time step, we are able to model systems with thousands of atoms at an affordable rate.

In this chapter, we will study the dissociation of hydrogen, $\text{H}_2 \rightleftharpoons 2\text{H}$, at a high density. We will estimate the radial distribution function $g(r)$, the dissociation N_H , and the standard reaction enthalpy $\Delta_r H^\circ$. The results are compared to those of a recently published study by Skorpa et al. [26]. In their study, the Kohen-Tully-Stillinger potential was used (see Section 2.3).

We find that $g(r)$ is similar in both studies, but differs because the potentials have disparate properties. There also appears to be a strong dependency on the chosen cutoff radius r_c . The dissociation is a bit higher in our study. Yet the standard reaction enthalpies are found to be almost equal ($\Delta_r H^\circ = 385 \text{ kJ mol}^{-1}$ in our study, versus $\Delta_r H^\circ = 380 \text{ kJ mol}^{-1}$ in their study).

Additionally, we report that the ANN potential is about 10 – 20 times as expensive to calculate as the Kohen-Tully-Stillinger potential. Considering the possibilities of introducing more optimizations, this cost can, however, be reduced even further.

8.1 Simulation Details

First, we will simulate a small system to verify the ANN potential. We set up a system of $N = 50$ hydrogen atoms in a cubic structure. The simulation box is rectangular, with sides $L_x = 2L_y = 2L_z$ and density $\rho = 19.1 \text{ kg/m}^3$. The time step is $\Delta t = 1.0$ in atomic units, or $\Delta t = 0.2419 \times 10^{-4} \text{ ps}$. The initial temperature is controlled by a thermostat and kept at $T = 300 \text{ K}$ for 10^5 time steps. Then the system is equilibrated



Figure 8.1: Snapshot of a molecular dynamics simulation, showing 50 hydrogen atoms forming 25 H_2 molecules, due to an ANN potential. The system is periodic in all directions with $L_x = 2L_y = 2L_z$. In this simulation, the temperature was $T = 14\text{ K}$ and the density $\rho = 19.1\text{ kg/m}^3$.

without a thermostat for 10^5 time steps. Next, the temperature is lowered to $T = 14\text{ K}$ for 10^5 time steps. Finally, the system is equilibrated for 10^6 time steps. In Figure 8.1, we show a snapshot at the end of the simulation. Here, we see that the simulation has turned our initial system of individual hydrogen atoms, into a system of hydrogen molecules. This verifies that the ANN potential is capable of reproducing the formation of hydrogen molecules.

We continue with a larger system that can be used for statistical sampling. We will try to reproduce a study [26] of hydrogen dissociation at high density and a range of temperatures. They used the classical molecular dynamics potential derived by Kohen, Tully, and Stillinger [27] to model two- and three-body forces between hydrogen atoms. Their results were in good correspondence with experimental data for the dissociation reaction.

The system is set up with $N = 1000$ hydrogen atoms in a rectangular simulation box of size $L_x = 2L_y = 2L_z$. We will run multiple simulations at temperatures from $T = 14\text{ K}$ to $T = 15\,600\text{ K}$. The time step is $\Delta t = 10.0$ for the simulation with temperature $T = 14\text{ K}$ and $\Delta t = 1.0$ otherwise. We target the lowest density studied by Skorpa et al. [26], namely $\rho_1 = 19.1\text{ kg/m}^3$. In comparison, liquid hydrogen has a density of $\rho_{\text{H}_2} = 70.85\text{ kg/m}^3$. Hydrogen is in a liquid state between $T_m = 14.0\text{ K}$ and $T_b = 20.3\text{ K}$ at atmospheric pressure [80]. However, we will simulate a liquid state even at high temperatures because the density is high and the volume is fixed.

The interatomic potential is approximated by an ANN, and consists of two- and three-body terms. The potential energy surface is first calculated with the Kindfield program, using UHF and a 6-311++G** basis set. The ANN is trained as discussed for H_3 in Section 6.2.

From the initial configuration, the system is heated to $T > 15\,600\text{ K}$ to ensure a random configuration before sampling. The heating is performed for 2×10^4 time steps.

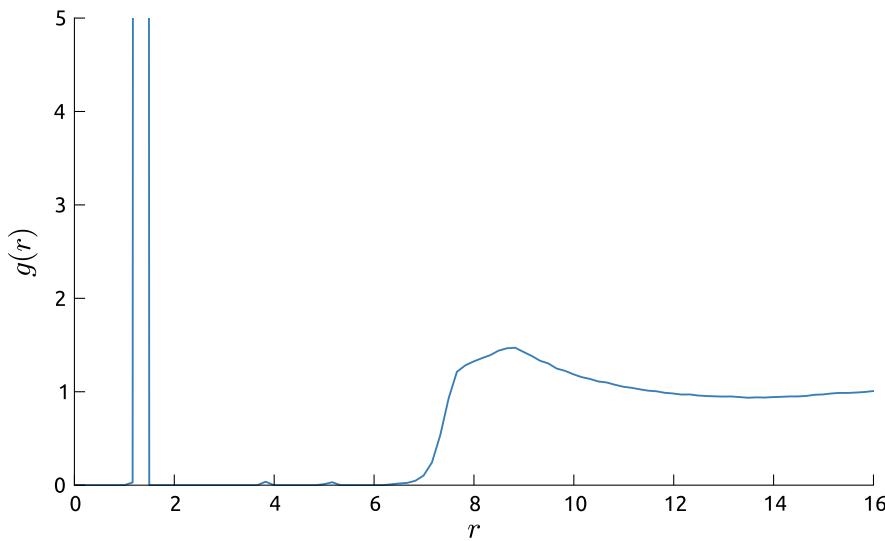


Figure 8.2: Radial distribution function for hydrogen with $N = 1000$ atoms at temperature $T \approx 14\text{ K}$ and density $\rho = 19.12\text{ kg/m}^3$. The distance r is in atomic units.

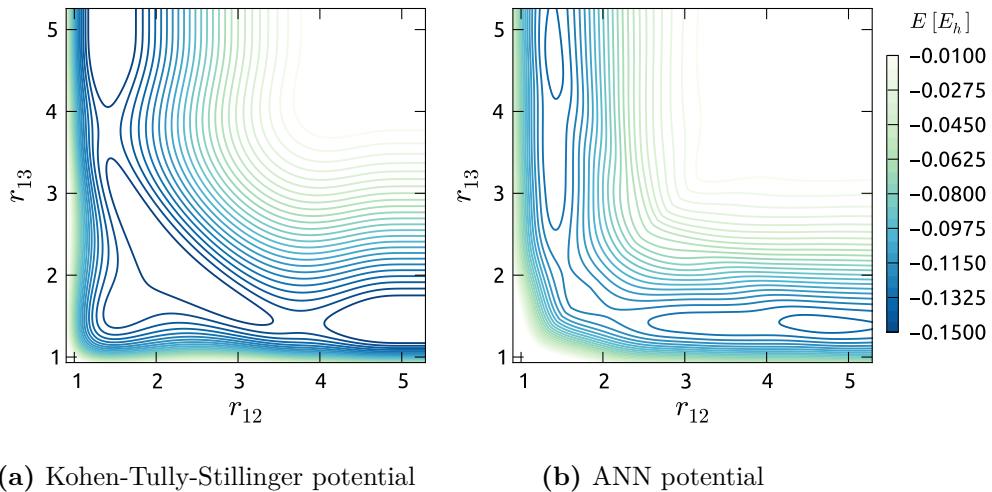
If the target temperature is below 300 K, the system is then cooled to this temperature for another 10^5 time steps. The system is then equilibrated at the current temperature for 10^5 time steps, before the temperature is brought to the target temperature for another 10^5 time steps. Finally, data collection begins over the course of 10^6 time steps.

Skorpa et al. [26] used a total of 5×10^6 time steps, where 3×10^6 were used for equilibration. We assume that this difference won't have a big impact on the results.

8.2 Pair Correlation Function

The pair correlation function $g(r)$ (also known as radial distribution function) is a measure of correlations in distances between the atoms. If the pair correlation function is unity for all distances, the system is in a randomly distributed state where all distances between atoms are equally probable. Any deviation from unity in $g(r)$ indicates correlation. A typical feature evident in $g(r)$ for most systems is the repulsion between atoms, taking $g(r)$ to zero for small r . Bond lengths are peaks in $g(r)$. The height of each peak is a measure of the number of bonds with the given bond length.

The radial distribution at $\rho = 19.1\text{ kg/m}^3$ and $T = 14\text{ K}$ is shown in Figure 8.2. The values of $g(r)$ for $r < 1.4$ are zero. This is due to the repulsive forces between the atoms. There is a peak at $r \approx 1.4$ followed by a gap up to $r \approx 7.5$. The bond length of hydrogen molecules is about $r \approx 1.4$, explaining the peak at this distance. The gap following the peak is due to the repulsion between hydrogen atoms and other molecules, which is a contribution from the three-body term in our potential. After this, there is a slight bulge before $g(r)$ goes to unity.



(a) Kohen-Tully-Stillinger potential (b) ANN potential

Figure 8.3: Comparison of (a) the Kohen-Tully-Stillinger potential and (b) our ANN potential. The plot shows the energy in a system of three hydrogen atoms (H_3) at an angle of 180° and a range of distances, r_{12} and r_{13} . Although the overall shapes of the potentials are similar, they differ in that the Kohen-Tully-Stillinger potential is deeper and wider. Differences are expected because the Kohen-Tully-Stillinger is based on a combination of theoretical and empirical data, while the ANN potential is based solely on a UHF calculation. All values are in atomic units.

These results are similar to those of Skorpa et al. [26], but there are differences. They found three minor peaks at $r \approx 1.90$, $r \approx 2.6$ and $r \approx 4.32$, which are not found in our results. However, we have some even smaller peaks at other values of r , barely visible in Figure 8.2. Their peaks were directly related to minima in the Kohen-Tully-Stillinger potential.

The differences are not very surprising, considering that their potential is formed from chemical intuition and parameterized from experimental data. Ours is an ANN potential fitted to a Hartree-Fock calculation. The two potentials are shown in Figure 8.3 for comparison. Although the overall shape is similar, it is evident that the Kohen-Tully-Stillinger potential is both deeper and wider than our potential.

The plateau starting from $r \approx 7.5$ is located closer to $r \approx 5.19$ in their study, corresponding to their cutoff radius $r_c = 5.29$. We have used a cutoff radius of $r_c = 8.0$ for the three-body term, while our two-body term has a cutoff radius of $r_c = 12.0$. This indicates that the location of the plateau could be an artifact of the chosen cutoff radius. However, Skorpa et al. [26] argue that their results are in agreement with the literature [81]. We should also remember that our Hartree-Fock approximation could be too crude for the hydrogen potential. However, as long as the characteristics of $g(r)$ appear to be related to the cutoff radius, we should be careful to make conclusions based on these results. Some of these characteristics may be entirely unphysical.

An important remark made by Skorpa et al. [26] is that the results at temperatures below 156 K are not adequate because the de Broglie wavelength of hydrogen λ_H is too large. At 14 K, the de Broglie wavelength is $\lambda_H = 8.81$, while at 156 K it is $\lambda_H = 2.64$. This indicates that full-scale *ab initio* calculations should be employed to

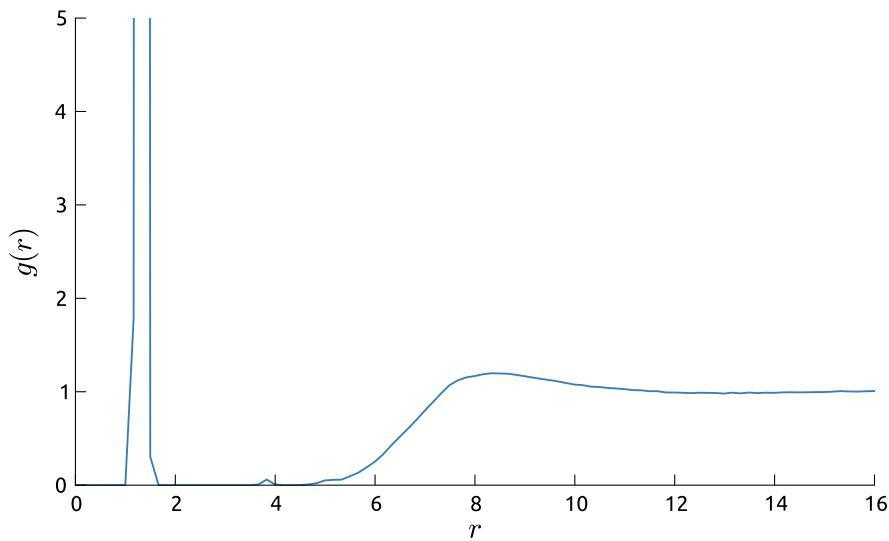


Figure 8.4: Radial distribution function for hydrogen with $N = 1000$ atoms at temperature $T \approx 156\text{ K}$ and density $\rho = 19.12\text{ kg/m}^3$. The distance r is in atomic units.

take all quantum effects into account. However, the above comparison is still a useful benchmark of ANN potentials against the Kohen-Tully-Stillinger potential.

In Figure 8.4, the radial distribution at a higher temperature of $T = 156\text{ K}$ is shown. The peak around $r \approx 1.4$ is now broader, while the plateau step has been significantly smoothed. Both are likely caused by the increased kinetic energy of the atoms. This results in higher vibrational amplitudes in the hydrogen molecules. Additionally, with higher velocities, the hydrogen molecules are pushing further into each others' exclusion zones.

In Figure 8.5, the radial distribution at $T = 15\,990\text{ K}$ is shown. Now the exclusion zone has been broken down and the plateau is extended towards the bond length of $r = 1.4$. This indicates that dissociation is taking place, with hydrogen molecules being torn apart to become separate atoms. This is why there are non-zero values for $g(r)$ in what was previously the exclusion zone of the molecules. The changes to $g(r)$ obtained by raising the temperature from 14 K to 156 K and $15\,600\text{ K}$ correspond closely to those observed in the above study [26].

8.3 Dissociation

Following the methodology of Skorpa et al. [26], we label hydrogen pairs with distances below $r = 3.46$ as H_2 molecules. The number of molecules and dissociated hydrogen atoms are counted every 1000 time steps and averaged over all collected time steps.

The mole fraction is the ratio of a constituent to the total number of all constituents in the system. The mole fractions of individual hydrogen atoms, and hydrogen

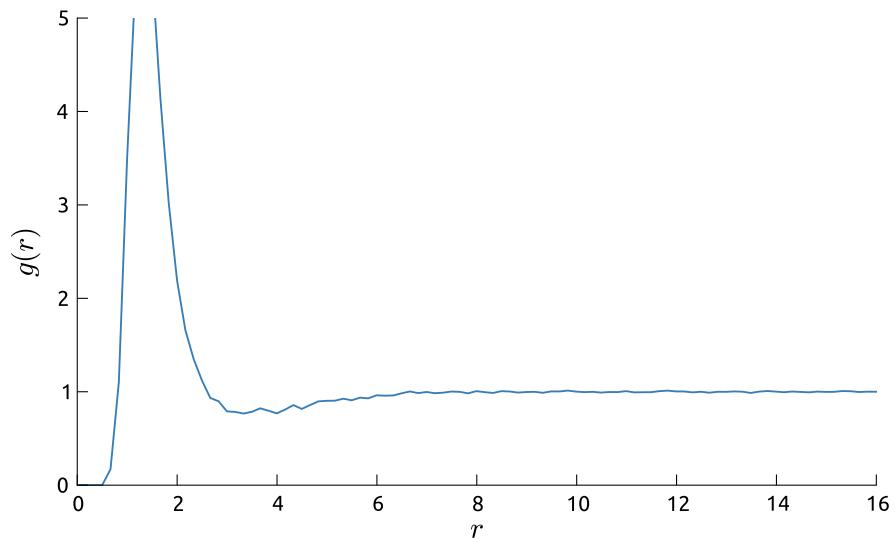


Figure 8.5: Radial distribution function for hydrogen with $N = 1000$ atoms at temperature $T \approx 15\,990$ K and density $\rho = 19.12$ kg/m³. The distance r is in atomic units.

ANN					Kohen-Tully-Stillinger ⁽¹⁾				
T /K	N_H	N_{H_2}	x_{H_2}	K_x	T /K	N_H	N_{H_2}	x_{H_2}	K_x
14	0.00	500.00	1.000	—	14	0.00	500.00	1.000	—
156	0.00	500.00	1.000	—	156	0.00	500.00	1.000	—
2718	1.04	499.48	0.998	0.000	2600	0.00	500.00	1.000	—
4847	34.28	482.86	0.934	0.005	4679	15.77	492.11	0.969	0.001
15 990	776.46	111.77	0.126	6.072	15 600	468.74	265.63	0.362	1.126

Table 8.1: Comparison of hydrogen dissociation results for the ANN potential and the Kohen-Tully-Stillinger potential at a density of $\rho = 19.12$ kg/m³. We have listed the number of dissociated atoms N_H , number of molecules N_{H_2} , mole fraction x_{H_2} , and dissociation constant K_x at given temperatures T . The numbers (1) for the Kohen-Tully-Stillinger potential are from the study of Skorpa et al. [26]. There are clearly differences in the results of the two simulations. These are likely caused by the differences between the potentials.

molecules are, respectively:

$$x_{\text{H}} = \frac{N_{\text{H}}}{N_{\text{H}_2} + N_{\text{H}}}, \quad (8.1)$$

$$x_{\text{H}_2} = \frac{N_{\text{H}_2}}{N_{\text{H}_2} + N_{\text{H}}}, \quad (8.2)$$

The mole fractions can be used to define the dissociation constant,

$$K_x = \frac{x_{\text{H}}^2}{x_{\text{H}_2}}, \quad (8.3)$$

which is related to the thermodynamic equilibrium constant of the $2 \text{ H} \rightleftharpoons \text{ H}_2$ reaction,

$$K_{\text{th}} = K_x \frac{\gamma_{\text{H}}^2}{\gamma_{\text{H}_2}}, \quad (8.4)$$

where γ_{H} and γ_{H_2} are the activity coefficients. For ideal mixtures, the ratio of the activity coefficients is unity, and $K_{\text{th}} = K_x$ [26]. In the following, we will assume that this is the case. This can further be used to find the reaction enthalpy, as given by the van 't Hoff equation:

$$\left[\frac{d \ln K_{\text{th}}}{d T^{-1}} \right]_P = -\frac{\Delta_r H^\circ}{R}, \quad (8.5)$$

where the subscript P denotes that the derivative is to be taken with pressure kept constant. We have not ensured that the pressure is kept constant between our simulations. The deviation from this criterion, however, is expected to have a small effect on the standard enthalpy of reaction [26].

Table 8.1 shows the dissociation N_{H} , the number of molecules N_{H_2} , the mole fraction x_{H_2} and the dissociation constant K_x . There is no dissociation for temperatures measured at $T = 156 \text{ K}$ and below. However, at 2718 K , we begin to see traces of slight dissociation, with 0.1% of the hydrogen atoms in a dissociated state. At 4847 K , we have 3.4% dissociation, meaning that the covalent bonds are breaking up due to the high temperature. Finally, at $T = 15\,600 \text{ K}$ there is a high dissociation at 77.07%. As expected, we find that there is a higher dissociation with increasing temperature.

The exact temperatures for which dissociation occurs, however, differs from the results in the study of Skorpa et al. [26]. Again, this is likely caused by the difference in our potentials.

In Figure 8.6, we have plotted the logarithm of the dissociation constant $\ln K_x$ as a function of the inverse temperature. We have limited the range of temperatures to the cases where dissociation occurs. There is an approximate linear trend. From this, we can estimate the standard enthalpy of reaction to be $\Delta_r H^\circ = 385 \text{ kJ mol}^{-1}$. Skorpa et al. [26] found a value of $\Delta_r H^\circ = 380 \text{ kJ mol}^{-1}$. The binding energy of H_2 is $\Delta_r H^\circ = 436 \text{ kJ mol}^{-1}$ at 298 K and 101 kPa [82]. Note that only three data points have been used to estimate the standard enthalpy of reaction in our study. This is a crude approximation that could be improved to obtain a more accurate result. Nevertheless, the result is promisingly close to that of the other study.

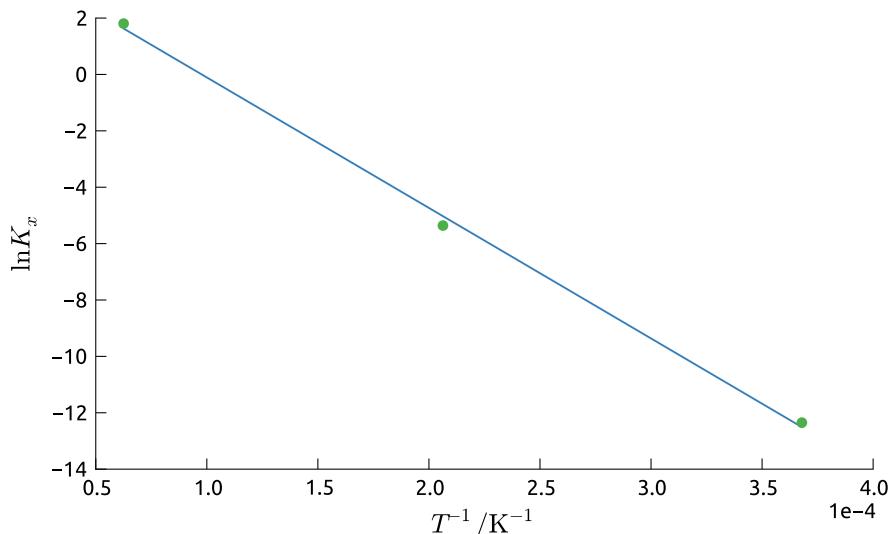


Figure 8.6: Logarithm of the dissociation constant $\ln K_x$ plotted as a function of the inverse temperature T^{-1} . The approximate linear trend gives an estimated standard enthalpy of the reaction $\Delta_r H^\circ = 385 \text{ kJ mol}^{-1}$. This is close to the value found by Skorpa et al. [26] ($\Delta_r H^\circ = 380 \text{ kJ mol}^{-1}$) and can be compared to the binding energy of H_2 ($\Delta_r H^\circ = 436 \text{ kJ mol}^{-1}$ at 298 K and 101 kPa [82]).

8.4 Performance Benchmark

We have also measured the performance of the ANN potential in comparison to the Kohen-Tully-Stillinger potential. To measure this, a simple benchmark has been set up with three hydrogen atoms. The two potentials are called 10^6 times, and the run time evaluated. For each call, the same three atoms, in fixed positions, are issued as parameters to the functions.

The resulting run times are $t_K = 2 \text{ s}$ for the Kohen-Tully-Stillinger potential and $t_A = 21 \text{ s}$ for the ANN potential. This means that the ANN potential uses approximately 11 times more CPU-time to complete the task.

There are, however, a few caveats with this comparison. There are likely optimizations available to both implementations, such as cache optimizations, precalculating certain values, applying clever trigonometric, etc. The impact of such optimizations is unknown. We therefore have an unknown systematic error in this measure that depends solely on the implementation. A reference implementation in an existing molecular dynamics library would therefore be a more proper benchmark than the test above.

Another missing point is that the number of neurons used in the ANN is central to the performance. It could be that a network with fewer neurons is able to properly approximate the potential energy surface (PES), at the benefit of more affordable computations. This has not been tested in this work, but is an interesting area to research in the future.

Even so, I would argue that this benchmark gives a decent indication of the cost

of using ANN potentials in comparison to classical potentials. We could have found that the Kohen-Tully-Stillinger potential was several magnitudes faster than the ANN potential. The above result does on the other hand indicate that the ANN potential could be a true competitor to its traditional counterparts. Further, the increased computational time is negligible compared to the many hours usually spent on crafting a good potential function.

Part III

Visualization

Chapter 9

Visualization of Molecules

Any visualization of an atom is artificial. After all, in everyday life we don't really see atoms; only the effect they have on the photons that are emitted from or bounce off them. However, certain representations can help us get a better understanding of the physics and chemistry of the systems we're working with. Visualizing electron densities and electrostatic potentials reveals information about bond strengths, molecular orbitals and nucleophilic regions. By looking at molecular dynamics (MD) simulations of crystals and fluids, we can observe microscopic phenomena such as bond breaking, atomic dislocations and porosity, which can explain macroscopic properties like fracture formation, tensile strength, elasticity and permeability.

In this chapter we will explore a few different methods used for visualization of molecules and molecular systems. We will focus on a few techniques that are useful for molecular data representation, such as the volumetric and isosurface rendering of electron densities, and a technique used for high performance rendering of millions of particles, namely the billboard technique.

The reader is assumed to have a basic understanding of Qt, Qt3D and OpenGL. Appendix C provides some information about these topics.

9.1 Densities with Volumetric Rendering

One way to present density data, such as the electron density (see Section 5.6.4), is by use of volumetric rendering. Volumetric rendering is done by tracing a path from the camera to a point infinitely far away. Along this path, the values of the volume data is accumulated, and the sum is used to decide the color of the pixels on the screen. This is illustrated in Figure 9.1. In the following, we will make use of a vertex shader program and a fragment shader program to achieve this effect (see Appendix C for details on shader programming).

9.1.1 Volumetric Vertex Shader

The vertex shader defines the geometry to hold the density data. We will use a cube with local vertex coordinates $[0, 0, 0]$, $[0, 0, 1]$, $[0, 1, 0]$, $[0, 1, 1]$, $[1, 0, 0]$, $[1, 0, 1]$ and $[1, 1, 1]$. These coordinates are associated to the `EntryPoint` variable. The `gl_Position`

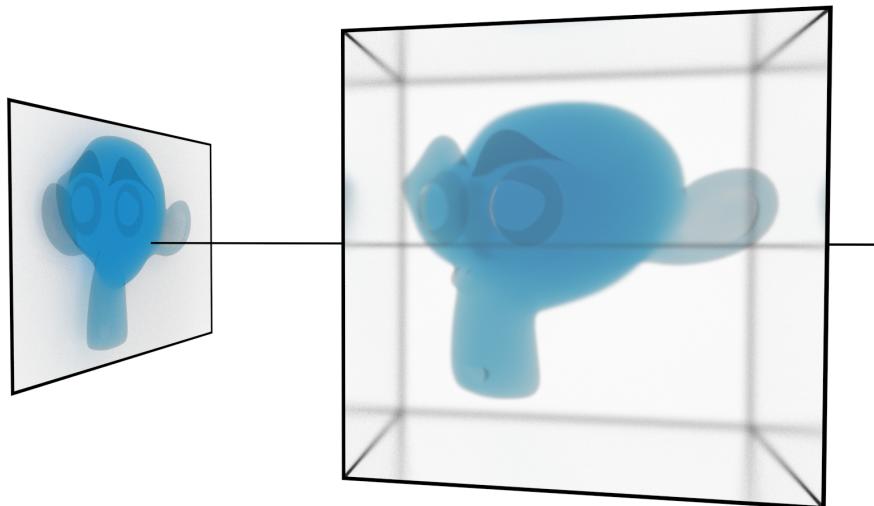


Figure 9.1: In volumetric rendering, rays are traced (black line) through a cube of volume data (right). Along its path, values of the data are accumulated to set the intensity and color of a pixel on the screen (left). Each pixel has its value set by a separate ray.

vertex is transformed by the model-view-projection matrix (see Appendix C), and passed along with `entryPoint` to the fragment shader:

```
#version 330 core
uniform mat4 qt_ModelViewProjectionMatrix;
in vec4 qt_Vertex;
out vec4 entryPoint;

void main(void)
{
    gl_Position = qt_ModelViewProjectionMatrix * qt_Vertex;
    entryPoint = qt_Vertex;
}
```

9.1.2 Volumetric Fragment Shader

In the fragment shader, the `entryPoint` will be used as the starting point of our ray in the volume data. The volume data is loaded into a 3D texture and a `uniform float` is used to control the quality of the rendering:

```
#version 330 core
uniform sampler3D volumeData;
uniform vec4 ve_eyePosition; // last column of the inverse modelViewMatrix
uniform float quality;
in vec4 entryPoint; // = EntryPoint
```

```
out vec4 outColor;
```

Next, we set the initial conditions for the tracing, such as the start and exit points, the direction of the ray and the step size:

```
void main(void)
{
    float stepSize = 1.0 / quality;
    vec3 exitPoint = ve_eyePosition.xyz;
    vec3 direction = exitPoint - entryPoint.xyz;
    vec3 deltaDir = normalize(direction) * stepSize;
    vec3 voxelCoord = entryPoint.xyz;
    vec4 standardColor = vec4(0.0, 0.0, 1.0, 1.0);
    vec4 accumulatedColor = vec4(0.0, 0.0, 0.0, 0.0);
```

The following main loop performs the actual tracing. On its way, it will sample the 3D data by use of the `texture` function:

```
for(int i = 0; i < int(1.732 / stepSize); i++) { // 1.732 = cube diagonal
    if(any(lessThan(voxelCoord, vec3(0,0,0)))
        || any(greaterThan(voxelCoord, vec3(1,1,1)))) {
        break;
    }
    voxelCoord += deltaDir;
    float value = stepSize * texture(volumeData, voxelCoord).x;
```

Once the value has been retrieved, the color of the pixel is updated by what is known as *alpha blending*. This can be mathematically expressed if we define the colors as vectors of RGB and alpha values. If $\mathbf{d} = [d_r, d_g, d_b, d_a]$ is the current color value of the pixel and $\mathbf{s} = [s_r, s_g, s_b, s_a]$ is the color value to be added on top of this, the result $\mathbf{r} = [r_r, r_g, r_b, r_a]$ is defined as

$$\mathbf{r} = s_a \mathbf{s} + (1 - s_a) \mathbf{d}. \quad (9.1)$$

This is implemented in the next step of the shader program, before the loop continues. Finally, when the loop is complete, the color of the pixel is set to the final value:

```
float newAlpha = standardColor.a * value;
acmmulatedColor = newAlpha * standardColor + (1 - newAlpha) *
    acmmulatedColor;
acmmulatedColor = clamp(acmmulatedColor, 0.0, 1.0);
}
outColor = acmmulatedColor;
```

An example of this technique is applied in the Denseness application, which we will discuss in the next section.

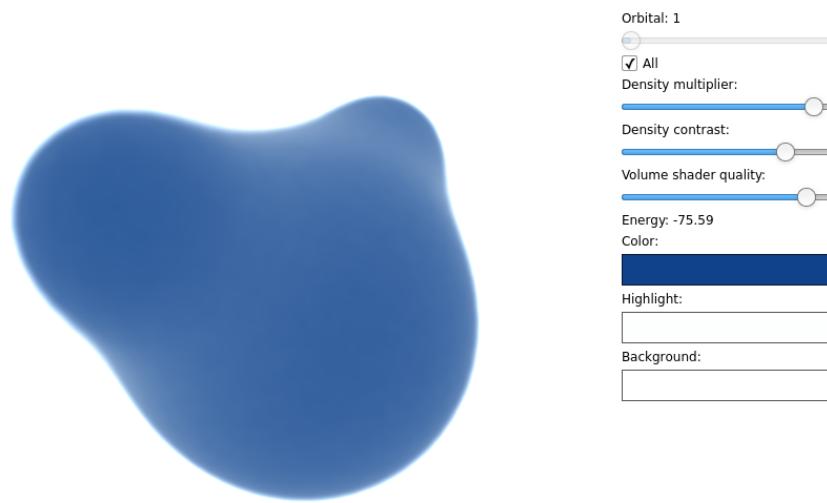


Figure 9.2: Screenshot of the Denseness application. Here showing the total electron density of the H_2O molecule. The controls to the right may be used to select the rendered molecular orbital, or adjust the quality and contrast of the rendering.

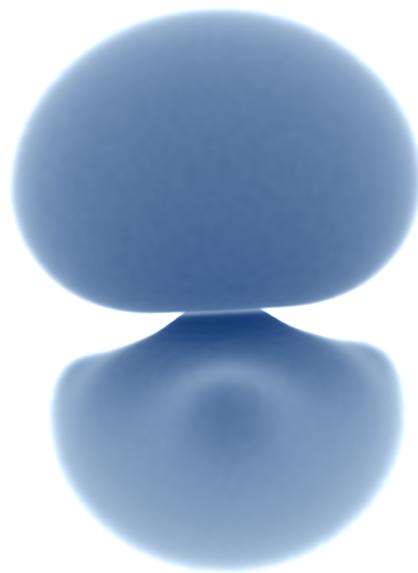


Figure 9.3: The NH_3 molecular orbital with the highest orbital energy as rendered by the Denseness application.

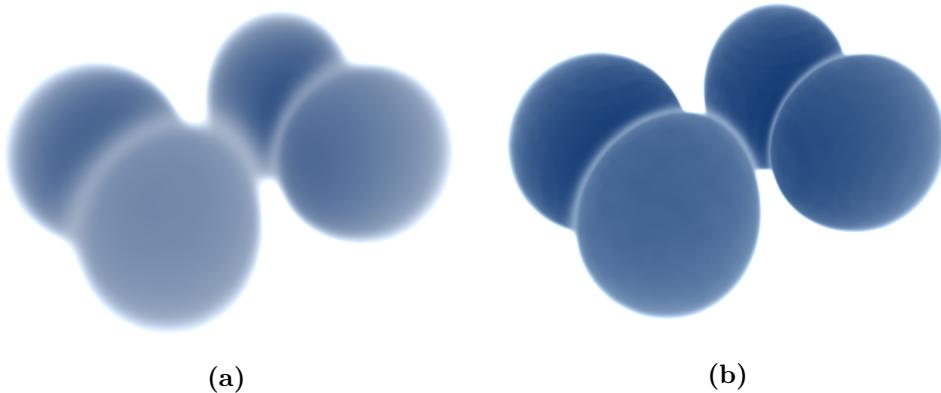


Figure 9.4: Difference between (a) low contrast and (b) high contrast in the Denseness application. Shown is the O₂ molecular orbital with highest orbital energy.

9.1.3 Example Application: Denseness

To test volumetric renderer, a visualization program was created to view densities from Hartree-Fock calculations. The program, called Denseness (see Figure 9.2), uses the Kindfield application as a library to calculate densities on the fly (see Appendix A for details on how to write combined applications and libraries). Once the density is calculated, the user may choose to view a specific orbital or the total electron density. In Figure 9.3, one of the NH₃ orbitals is rendered with Denseness.

The program uses a more advanced version of the fragment shader in the previous section, which also allows modifications of the volumetric data visualized, including strength, contrast and color representation. The user may alter the contrast by raising all data values x to the power of a :

$$x_{\text{new}} = x^a \quad (9.2)$$

If the power is 1, the data remains unaltered. If it is a fraction less than 1, the contrast will be reduced: Low values will be raised and high values lowered. This is similar to taking the square root, or the logarithm, of the values to even out the extremes. If the fraction is more than 1, the contrast will be raised: High values are raised even further, and low values become negligible. Two visualizations with different contrasts are shown in Figure 9.4.

Additionally, a slightly different blend function is used in the Denseness program:

$$r_a = s_a + (1 - s_a)d_a \quad (9.3)$$

$$\mathbf{r}_{\text{rgb}} = \frac{s_a \mathbf{s}_{\text{rgb}}}{d_a} + (1 - s_a)d_r \quad (9.4)$$

where $\mathbf{r} = [r_r, r_g, r_b]$ are the RGB components of \mathbf{r} . The difference is subtle, but this version is a bit less “foggy” when rendered.

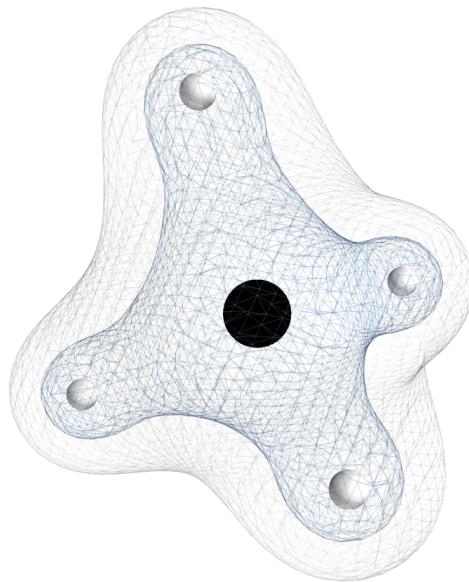


Figure 9.5: Example of isosurface rendering with Mayavi. Shown is the density of the CH₄ molecule, with two isosurfaces. The innermost one has the highest electron density.

9.2 Densities with Isosurfaces

There is another common way to render volumetric data that we have only mentioned briefly in earlier chapters, and that is by drawing isosurfaces. As its name indicates, this is a method where a surface is drawn in the area of the volume having the same scalar value. There are many ways to calculate the structure of such isosurfaces, where the marching cubes algorithm is a popular choice, and the marching tetrahedron algorithm is an improving extension on top of that. We won't go into the details of the algorithm here, but rather explore how we can make use of an existing implementation in the Python package Mayavi [28]. Further, the result will be exported to the 3D graphics program Blender [29] to add effects and otherwise improve the rendered result.

9.2.1 Calculating Isosurfaces in Mayavi

Let's assume that we have already performed a Hartree-Fock calculation on a molecule, and have stored the density data in an Armadillo [61] `cube` object. In C++, we can make use of the `save` function of the `cube` object to store the data to a HDF5 [83] file:

```
cube density;
// calculate density and store data in the density cube
density.save("density.h5", hdf5_binary);
```

In Python, this file can be loaded with the h5py library, and the data visualized with Mayavi:



Figure 9.6: Example of isosurface rendering with Blender. Shown is the density of the CH₄ molecule, with two isosurfaces. The innermost holding the highest electron density.

```
import h5py
from mayavi import *
density_file = h5py.File("density.h5")
data = density_file.get("dataset")[:] # copies the data to keep after file closes
density_file.close()
iso = contour3d(data)
```

The atom positions may also be loaded from file and rendered with the `points3d` function:

```
atoms_data_file = h5py.File(join(args.results_path, "atoms.h5"))
atoms = atoms_data_file.get("state")[:]
atoms_data_file.close()
for atom in atoms:
    points3d(atom[0], atom[1], atom[2])
```

After tweaking the settings for Mayavi a bit, you might end up with a rendering similar to the one shown in Figure 9.5.

9.2.2 Final Rendering in Blender

After calculating the isosurface in Mayavi, we may export the data for further post-processing in Blender. To export the data, we make use of the `savefig` function of Mayavi and the X3D file format:

```
savefig("density.x3d")
```

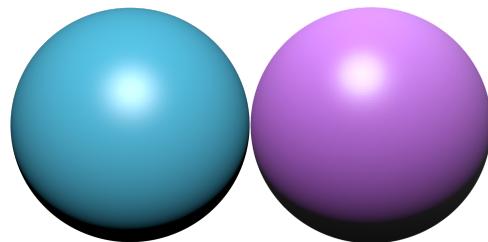


Figure 9.7: The sphere to the left is the true geometric shape of a sphere, while the one to the right is just a billboard. It is not easy to spot the difference.

This file may then be loaded into Blender by clicking File → Import → X3D Extensible 3D. Once loaded, a new camera and a few lights will have been imported in addition to the already existing cameras and lights in the scene. Unless you want these, you may safely delete them.

Blender has a range of options for creating visually appealing renderings. For the interested reader, I would recommend the Tutorials section of the Blender website as a good starting point [29]. As for our current isosurface rendering, it suffices to change to the Cycles renderer, and choose a glass-material for the densities and a glossy material for the atoms. Because the atoms are inside the isosurface object, a spotlight may be added for each atom, pointing directly towards it, such that the light reflects back to the camera. If the isosurface is a bit coarse, a Subdivision surface modifier may be added to smooth it. A resulting rendering with this setup is shown in Figure 9.6.

9.3 Millions of Atoms with Billboarding

The most technique with most impact on efficiency used in this thesis is the billboarding technique. This is technique dating back to the earliest of 3D video games. The perception of 3D is given by varying the size of each billboard, which is based on the distance from the camera.

The technique is to represent any 3D object with an image of a 3D object, placed on a plane that always faces towards the camera [84]. The plane is usually centered at the position of the object one wants to represent.

With billboarding, the number of vertices sent from the central processing unit (CPU) to the graphics processing unit (GPU) can be dramatically reduced. A sphere may consist of an arbitrary number of vertices, depending on the level of detail needed. A billboard will on the other hand consist of only 4 vertices, (or just 1 vertex in combination with a geometry shader, as will be discussed in Appendix C.2.3), no matter the level of detail in the image used on the billboard. This makes it easy to optimize visualization of spheres, by tricking the eye into believing it sees a sphere when what it really sees is just a flat image of a sphere on a billboard.

The visual differences between a sphere and a billboard of a sphere are subtle. In Figure 9.7, a sphere and its billboarded counterpart are rendered in Blender. It is not

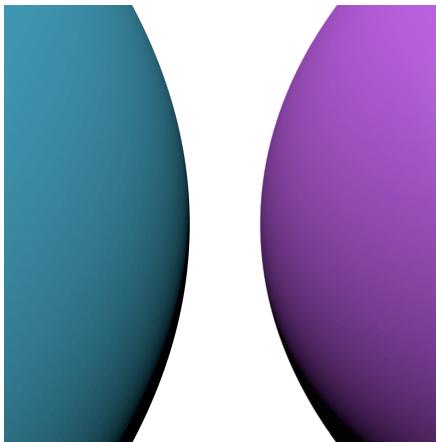


Figure 9.8: A closer look at the two spheres. The one to the left is the true geometric shape of a sphere, while the one to the right is just a billboard. The curvature is slightly different, but it is hard to pinpoint which curvature is the correct one.

easy to see which is which, but the one to the left that is the true sphere. In Figure 9.8 the same spheres are rendered close-up.

9.3.1 Billboard with a Geometry Shader

With OpenGL version 3.2 the geometry shader was introduced, and allowed implementation of the billboard technique on the GPU. The alternative is to implement billboard on the CPU and pass the final billboards to the GPU for rendering. Because the geometry shader is not yet available in OpenGL ES¹, the standard used on mobile devices, this implementation of the technique is only available for desktop computers. The alternative approach, which is to implement everything on the CPU, is described in Section 9.3.2.

First of all, we need to define the positions of the billboards. We will be implementing this in Qt and have made a subclass of `QQuickItem3D`, named `MultiBillboard`. This class will be available in QML, and holds a list of positions of all the atoms, stored in a `QArray< QVector3D >`. We need to pass this list to our GPU in the `drawItem` virtual method of the class. This is done by creating a `QGLVertexBuffer` (better known as a vertex buffer object (VBO) in OpenGL), an index buffer and send both of these to the `QGLPainter` object before calling its draw method:

```
void MultiBillboard::drawItem(QGLPainter *painter) {
    QGLVertexBuffer vertexBundle;
    QGLIndexBuffer indexBuffer;
    vertexBundle.addAttribute(QGL::Position, m_points);
    indexBuffer.setIndexes(indexes);

    painter->clearAttributes();

    // Set the rest of the vertex bundle (basically only positions)
```

¹As of today, the current version of OpenGL ES is 3.0.

```

    painter->setVertexBundle(vertexBundle);
    painter->setUserEffect(effect);
    painter->draw(QGL::DrawingMode(QGL::Points), indexBuffer, 0,
                  indexBuffer.indexCount());
}

```

Note that we are also setting the `userEffect` property of the painter object, which will hold our shader programs - the vertex shader, fragment shader and of course, the geometry shader.

The `effect` object is defined as a subclass of the `QGLShaderProgramEffect` class. The reason for this is that we can set the vertex and fragment shader programs of any `QGLShaderProgramEffect` object, but as of the current version of Qt3D, the geometry shader must be set manually in the `beforeLink()` method of the object:

```

bool CustomEffect::beforeLink() {
    QString geometryShaderFile = "geometryshader.glsl";
    if(!program()->addShaderFromSourceFile(QOpenGLShader::Geometry,
                                             geometryShaderFile)) {
        qCritical() << "Could not compile geometry shader! Log: \n"
                      << program()->log();
    }
    return true;
}

```

The geometry shader is defined as follows:

```

#version 330
layout(points) in;
layout(triangle_strip, max_vertices = 4) out;
uniform mat4 qt_ProjectionMatrix;
out vec2 texCoord;
void main(void) {
    float scale = 0.1;
    vec4 pos = gl_in[0].gl_Position;
    gl_Position = pos + qt_ProjectionMatrix*vec4(-scale, -scale, 0.0, 0.0);
    texCoord = vec2(0.0, 0.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(-scale, scale, 0.0, 0.0);
    texCoord = vec2(0.0, 1.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(scale, -scale, 0.0, 0.0);
    texCoord = vec2(1.0, 0.0);
    EmitVertex();
    gl_Position = pos + qt_ProjectionMatrix*vec4(scale, scale, 0.0, 0.0);
    texCoord = vec2(1.0, 1.0);
    EmitVertex();
    EndPrimitive();
}

```

Note that we need the projection matrix to multiply this with the vectors used to offset each vertex from the center of the billboard. The reason is that the projection matrix takes us from the world space to the right projection onto the screen. Without this, all

billboards would be of equal size and depth on the screen, and this is of course not the effect we really want. We rather want billboards that are far away to appear smaller than those that are close to the camera.

For our upcoming purposes, it may be useful to introduce periodic images of the billboards. In systems with periodic boundary conditions, this will help visualize the system properly, as there are technically no edges. A common way to do this is to add 27 identical copies of all the atoms in the system, each offset $-1, 0$ or 1 times the system length in each Cartesian coordinate. However, this will add a tremendous amount of new data that needs to be stored in memory and copied to the graphics card. Instead, we may add periodic copies of the billboards by using geometry shader *instancing*. This is a feature that was added to OpenGL 4.0, which allows the geometry shader to work on the same vertex multiple times with a slightly different output. To do run the geometry shader 27 times, we add `layout(invocations=27) in;` to the top of the shader program. Then we can make use of the `gl_InvocationID` variable to decide which of the 27 instances is currently being executed:

```
#version 400
layout(invocations=27) in;
layout(points) in;
layout(triangle_strip, max_vertices = 4) out;
uniform mat4 qt_ProjectionMatrix;
uniform vec3 systemSize;
out vec2 texCoord;
void main(void) {
    int x = gl_InvocationID % 3 - 1;
    int y = (gl_InvocationID/3) % 3 - 1;
    int z = (gl_InvocationID/9) % 3 - 1;

    vec4 displacement = vec4(x*systemSize.x, y*systemSize.y, z*systemSize.z, 0);
    vec4 pos = gl_in[0].gl_Position + qt_modelViewProjectionMatrx * displacement;
    gl_Position = pos + qt_ProjectionMatrix*vec4(-scale, -scale, 0.0, 0.0);
    texCoord = vec2(0.0, 0.0);
    EmitVertex();
    // ... equivalently as above for the remaining 3 vertices
}
```

Note that the displacement must be multiplied by the model-view-projection matrix, because it is a displacement vector defined in the global coordinate system, just like the original vertex. The offsets used to position the four vertices in the billboard are still multiplied with only the projection matrix, because these are supposed to be shifted only in the camera coordinate system. This method is implemented in the MultiBillboard project.

9.3.2 Billboard without a Geometry Shader

On mobile devices where the geometry shader is not available, we need to create the billboards with regular vertices on the CPU and then pass them to the GPU. This requires much more communication between the two, because the vertices need to be sent to the GPU not only when the particles move, but also when the camera moves

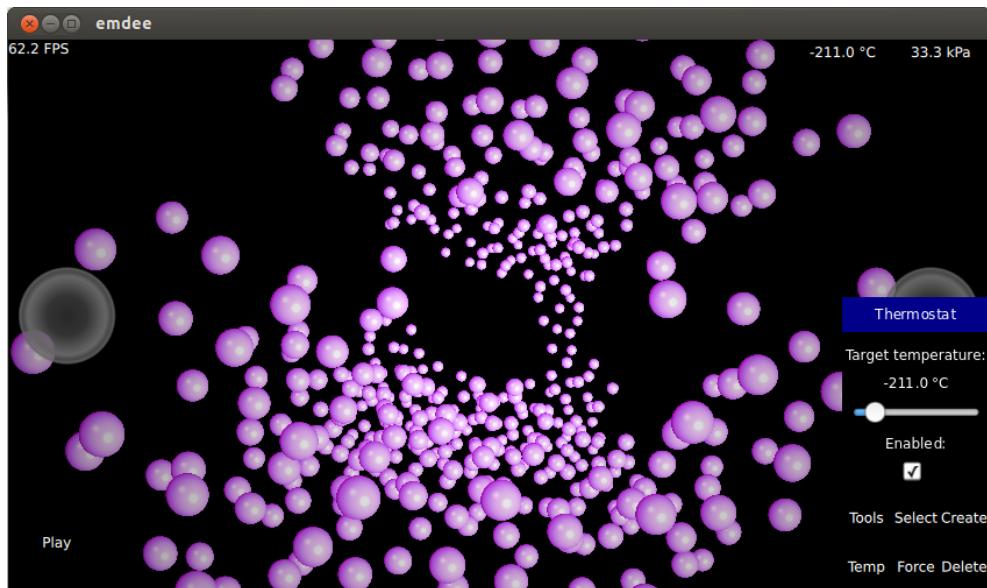


Figure 9.9: Screenshot of the Emdee application, showing a live simulation of argon crystallization, where a non-physical hole has appeared in the structure (see Section 7.6 for details on argon crystallization). The user can control the temperature by using the slider to the right and observe the current temperature and pressure in the upper right corner. The system is periodic, but only one simulation box is shown in this figure.

to make the billboards face the camera. On a desktop system, this slows down the rendering by a factor of about 100.

Constructing the billboard on the CPU is done in the same way as on the GPU. We still define the center of the particle as the center of the billboard, and construct four vertices around this center, but we now have to explicitly create the vertices on the CPU and pass them on to the GPU. Because OpenGL builds up larger objects by drawing triangles, we could send six vertices to construct a billboard, because two triangles are needed to make a rectangle. However, we may reduce the amount of data slightly by sending four vertices that are placed in the corners of the billboard, in addition to six indices, each referring to one of the four vertices. The three first indices could point to the vertices making up the upper triangle, while the last three indices could point to the vertices making up the lower triangle. By such a reuse of the coordinates, we are able to reduce the amount of data to transfer.

9.3.3 Example Application: Emdee

The Emdee application is an interactive molecular dynamics simulation program and visualization tool that makes use of the cross-platform capabilities of Qt to run on everything from desktop computers to Android tablets. The application is currently in early development, but already features a live molecular dynamics simulation of argon with temperature control. In the future, this application is planned to support more advanced potentials, in addition to more interactions by the user, such as adding or

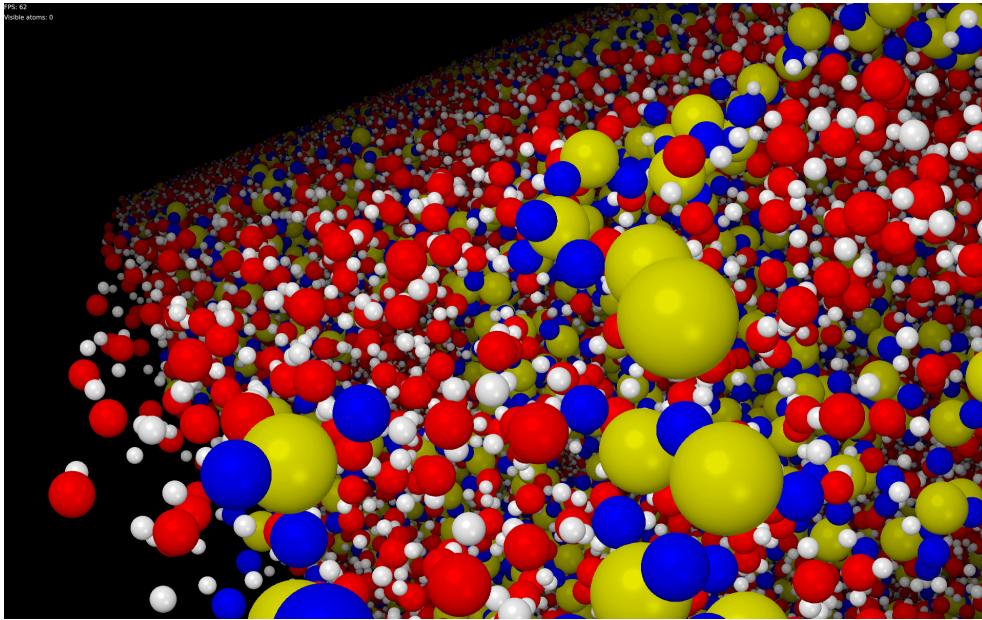


Figure 9.10: Screenshot of the Poson application, showing a snapshot of a system of nanoporous SiO_2 (yellow and blue), and H_2O (white and red), just outside the simulation box (without periodic images added). The molecular dynamics simulation was performed by Hafreager [3].

removing atoms, resizing the volume of the simulation box, inducing drift and more. To render on mobile devices, this application makes use of the CPU billboarding method outlined in Section 9.3.2.

In Figure 9.9, a screenshot of Emdee shows a live molecular dynamics simulation of argon crystallization. The Lennard-Jones potential is used to model the interactions between the argon atoms and the temperature of the simulation is controlled by the user with a Berendsen thermostat (see Section 7.4 for details). When driven to a low temperature, the system of argon atoms will crystallize. While the system is periodic in all direction, only one simulation box is rendered in this figure.

9.3.4 Example Application: Poson

In collaboration with Hafreager [3], I have used the billboard technique to develop Poson: a large-scale molecular dynamics visualization tool. It has support for the Oculus Rift virtual reality headset, built with the Oculus VR software development kit (SDK) [30]. This in turn allows the user to control the camera by moving his or her head. Further, the Oculus Rift headset (we have been working with the Oculus Rift Development Kit 1) has a screen that covers the entire field of view, placing the user in the middle of the visualization. The camera is moved forward, backwards and sideways by clicking the buttons on the keyboard, and the result is almost like flying a spacecraft through a molecular structure (except for the obvious size-difference).

The application currently features loading XYZ-files, disabling visibility of water molecules, adding periodic images of the simulation box and setting the application to

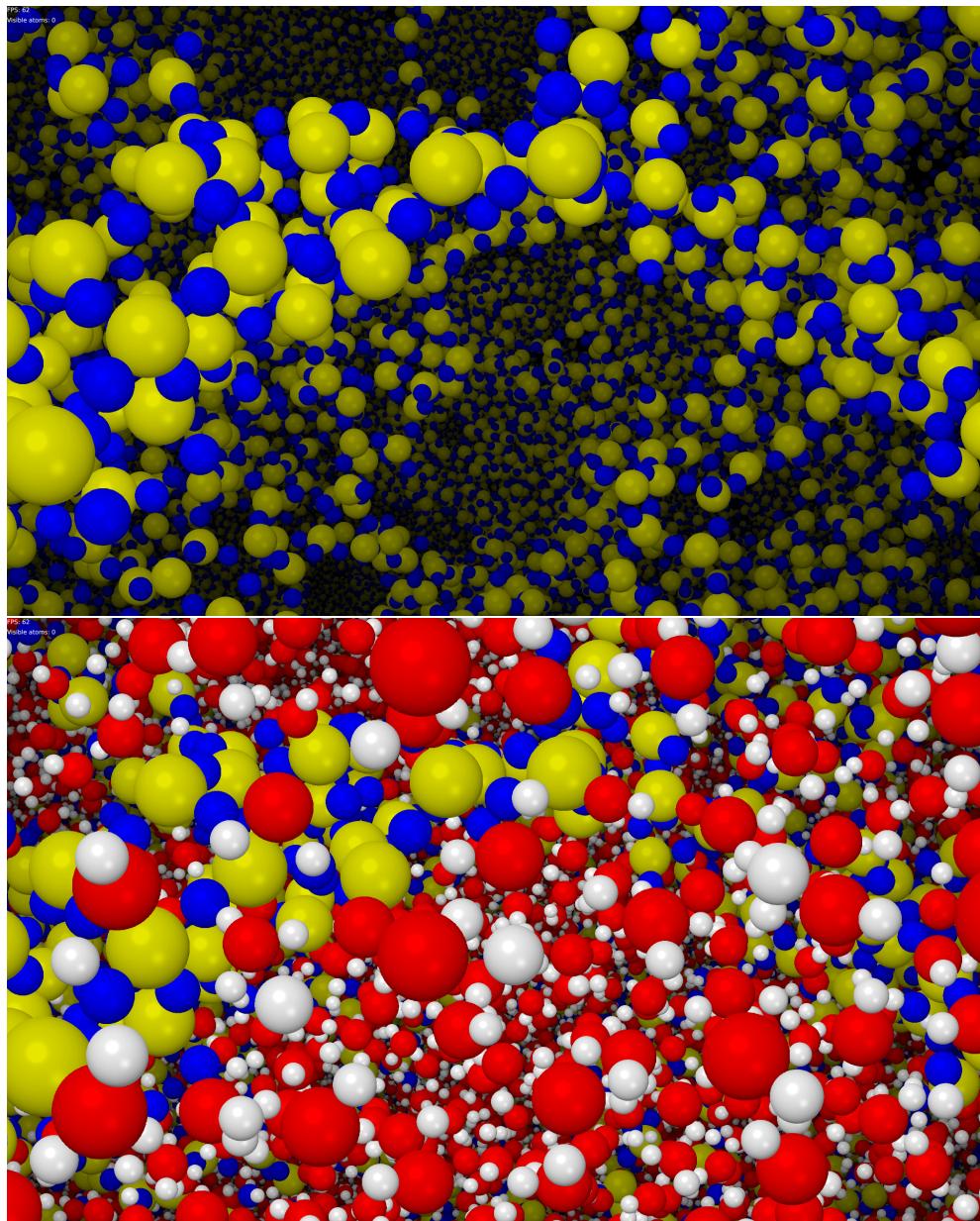


Figure 9.11: Screenshot of the Poson application, showing a snapshot of a system of nanoporous SiO_2 (yellow and blue), and H_2O (white and red). The camera is placed inside the simulation box, and periodic images are added. The water molecules are invisible in the top image and visible in the bottom image. The molecular dynamics simulation was performed by Hafreager [3].

full screen over a four-display video-wall. This application is capable of rendering 45.5 million particles at 20.8 frames per second on our reference visualization machine.² For details and more values from the performance benchmark, see Hafreager [3].

In Figure 9.10, a screenshot of the Poson application is shown where a molecular dynamics system of SiO_2 and H_2O has been loaded. The simulation was performed by Hafreager [3]. In this screenshot, the camera was placed just outside the simulation box and periodic images disabled. In Figure 9.11, the same system is shown, but this time periodic images are included. Additionally, this figure shows the difference between visualizing the system with and without the water molecules. As can be seen in this figure, the water molecules obscure much of the view. Hiding them reveals the underlying nanoporous SiO_2 structure.

One may ask why we should combine virtual reality toolkits with molecular dynamics visualizations. I must admit that at the current state of the Oculus Rift Development Kit 1, the resolution is a bit too low for use with particle visualizations. Particles that are far away are rendered to small to tell apart properly. A regular monitor is therefore still better suited for such visualizations. However, in my experience from testing molecular dynamics visualizations on 3D TVs, the added depth is very helpful when you need to distinguish particles that are overlapping in the image. This is especially true in simulations of particle flow, where it can be easier to see on which side of a wall the flow is actually occurring. With new and upcoming versions of virtual reality hardware, I expect higher resolutions to make this experience as good as on today's 3D TVs, with the additional benefits of controlling camera movement with your head, and having your entire field of view covered by the screen.

²See Appendix D for the technical specifications of this machine.

Part IV

Conclusions and Future Work

Chapter 10

Conclusion

In this thesis, we aimed to automate the parameterization of classical molecular dynamics (MD) potentials by training artificial neural networks (ANNs) to fit the potential energy surface (PES). The theoretical grounds on which we built the PES was founded on Hartree-Fock theory.

While laying down the groundwork of an Hartree-Fock code implementation, we also studied other properties accessible from Hartree-Fock theory, such as electron densities and electrostatic potentials of molecules. The results from our Hartree-Fock benchmarks were found to be in good correspondence with the literature.

The PES of H₂ and H₂O were calculated for two- and three-body interactions, before an ANN was applied to find a proper function approximation. For this purpose, the Fast Artificial Neural Network Library (FANN) was used. Visually and numerically, the approximation matched the PES well.

Further, we developed an efficient and flexible molecular dynamics code, which could make use of said ANN potential. To make this possible, an extension to the FANN library was made to calculate the analytical derivatives of the PES approximation. The molecular dynamics code was benchmarked against a recent study on dissociation of hydrogen [26], and was found to reproduce the main features of the system, such as hydrogen molecule formation and separation. However, certain characteristics in the radial distribution function differed, and there were discrepancies in the dissociation factors. These differences are attributed to the potentials used, in their case the Kohen-Tully-Stillinger potential. This may be caused by the Hartree-Fock method returning an incomplete PES, while the Kohen-Tully-Stillinger potential has been fitted to experimental values of hydrogen interactions. Nonetheless, the radial distribution function is strongly correlated with the cutoff distance chosen for the simulation. The differences to the above study should therefore be considered with some care. To improve the reliability of the results, a new ANN potential could be trained based on post-Hartree-Fock methods.

Finally, the performance of the ANN potential was compared to that of the Kohen-Tully-Stillinger potential by monitoring the function call times. The extra cost of the ANN potential was found to be just 11 times higher. Considering that training the ANN potential can be done with no or little supervision from a user, this bodes well for the future of ANNs in molecular dynamics. For projects where computational resources

are abundant, but the cost of crafting a potential function by hand is high, the ANN potentials can turn out to be very useful.

In the visualization part of this thesis, we looked into rendering volume data with isosurfaces and volumetric rendering. The latter was also successfully implemented in Hartree-Fock visualization program, Denseness. This is capable of rendering the total electron densities for molecules and their molecular orbitals. Further, it serves as a demonstration of how a graphical user interface (GUI) can be combined with graphics programming to modify the parameters used in .

Further, we developed an efficient billboard rendering method which uses the OpenGL geometry shader. With this, millions of atoms are rendered to screen at acceptable frame rates on high-end graphics hardware. This was combined with the Oculus VR SDK [30] to create a high-performance virtual reality application for molecular dynamics visualizations. The quality of todays virtual reality hardware is still not quite able to compete with traditional monitors due to low resolutions. Nevertheless, the future of combining scientific visualization with virtual reality toolkits looks promising.

Chapter 11

Future Implementations and Optimizations

While much effort has been put into the implementations and optimizations in this thesis, there is still much room for improvement. In the overall scheme of using artificial neural network (ANN) potentials in molecular dynamics (MD), the improvements range from calculating the potential energy surface with more accurate methods than Hartree-Fock, through replacing the FANN library with code tailored for molecular dynamics, to pure numerical optimizations for increased performance. For the visualization part, much can be done to improve the design and usability of all the programs.

In this chapter, I have listed the details of the above suggested improvements. Finally, in the last section, we'll discuss the possible future applications of the programs, libraries and methods developed in this thesis.

11.1 Improved Potential Energy Surface Calculations

Hartree-Fock is a good starting point for many calculations on molecular systems. However, with a single Slater determinant approximation to the wave function, it does not include correlations apart from exchange. This in turn results in an underestimation of equilibrium bond lengths and generally poor energy determination [45]. To alleviate this, we may turn to post-Hartree-Fock methods such as perturbation theory [2, 13] coupled cluster theory [13, 21] and density functional theory (DFT) [14]. Alternatively, methods like Full Configuration Interaction (FCI) [13, 15–17], variational Monte Carlo (VMC) [18, 19] and diffusion Monte Carlo (DMC) [19, 20] may also provide more accurate results. To construct a potential that can be used for truly predictive molecular dynamics simulations, Hartree-Fock should be replaced by one of the above for calculations of the potential energy surface. Fortunately, the only increased cost would occur in the computation of the potential energy surface, because the ANN training process remains unchanged. This is therefore likely a large improvement to the overall scheme.

11.2 A Customized Artificial Neural Network Library

The Fast Artificial Neural Network Library (FANN) is a very flexible and fast ANN library. However, it is also very general, and there are likely many improvements that can be made by tailoring a new ANN library to the problem of fitting potential energy surfaces (PESs).

FANN also lacks a function that calculates derivatives of the network outputs with respect to the inputs. Although we have proven that such a function can be implemented as an extension to FANN, it requires a complete reimplementation of an already existing function. This depends on internal parts of the library, which makes it prone to errors if changes are made in future versions of FANN. Additionally, because FANN is a highly-optimized C library, this extension is a bit out of place when implemented in a C++ application.

A customized ANN library tailored for molecular dynamics simulations could, for the above reasons, be an interesting future project.

11.3 Improved Convergence in Hartree-Fock

As we discussed in Sections 3.4 and 5.3, the Hartree-Fock method is not guaranteed to converge if our initial guess is far off. We mentioned briefly a few methods used to aid convergence, one of which was to use a mixing factor a to combine the previous and the newly found density matrix,

$$\mathbf{P} = a\mathbf{P}^{\text{previous}} + (1 - a)\mathbf{P}^{\text{new}}. \quad (11.1)$$

We did however not discuss what the mixing factor a should be. This is not a trivial problem, as a large mixing factor would lead to slowing down convergence, and a small value won't introduce any change. Setting up an algorithm that tries out different mixing factors and chooses the best, could therefore be a future possibility.

Other options include implementing the DIIS procedure [63] and other extrapolation methods. In fact, a preliminary implementation of the DIIS procedure is already present in the code, based on a version written by H. Mobarhan [1].

11.4 Future Visualizations

In future versions of the Hartree-Fock program, other properties such as the electrostatic potential and the electron localization function (ELF) could be visualized. While we already have looked into visualizing the electrostatic potential with Mayavi [28] and Blender [29], it could be useful to have this as part of the Hartree-Fock program too.

The ELF has not been discussed previously in this thesis, but is another Hartree-Fock visualization that is much used. It is a different method used to identify the localization of electrons, published by Becke and Edgecombe [85] in 1990. According to the authors, the method "easily reveals atomic shell structure and core, binding, and lone electron pairs in simple molecular systems". It is however a bit harder to implement than the visualization we've worked with in this thesis; among other things,

it requires a calculation of the gradient of the electron density. Visualizations of the ELF has therefore been left as a future implementation possibility. However, due to its popularity, this method would be very interesting to study further.

11.5 Applications to New Systems

In this thesis, we tested the complete workflow, including ANN potentials from Hartree-Fock in molecular dynamics simulations, on a system of hydrogen molecules. We studied the dissociation of $\text{H}_2 \rightarrow 2 \text{H}$ and measured the radial distribution function of the system. While our simulation was found to reproduce most of the properties of hydrogen dissociation found in the literature [26], we also discovered differences that were attributed to the use of different potential functions. Whether the differences are a consequence of lacking accuracy in our potential energy surface calculations, due to the limits of Hartree-Fock (as discussed in Chapter 8), or the results of actual physical phenomena, remains to be investigated.

There is also an interesting possibility in applying this workflow to other systems. Here, the choice is open, but personally I would like to see how well it bodes with systems of H_2O , due to importance and complexity of water. Additionally, I would like to see how good ANN potentials are at predicting chemical reactions. Our potential energy surface calculations imply that the system collapses to the ground at each time step. However, it will still be interesting to see just how much of the chemistry is still kept intact under this assumption. Therefore, energetic reactions such as $2 \text{H}_2 + \text{O}_2 \rightarrow 2 \text{H}_2\text{O}$ and $\text{CH}_4 + 2 \text{O}_2 \rightarrow \text{H}_2\text{O} + \text{C}$ would be very interesting to study. Their reaction rates depend on the physics of the involved components, and we have already done some calculations on these molecules in this thesis.

Further, at the Computational Physics research group at the University Oslo, we have ongoing projects on CO_2 storage and shale gas systems, and a recently launched project on neuroplasticity in the brain (CINPLA). Therefore, applications to systems of silicon, water, CO_2 and methane would be of high interest, in addition to biological systems, including carbon, water, salts and complex structures. Both these projects have a strong focus on general multiscale modeling. ANNs are promisingly flexible, and it may be that other gaps in such multiscale models can be bridged by ANN approximations.

Appendices

Appendix A

Libraries

In C++ , and most other languages, it is possible to extend the language by writing libraries or modules that other may use. In this context, one often talks about linking or loading such libraries or modules, either at compile-time or at run-time. Some languages, such as Python, usually load modules at run-time, while C++ and other languages with compilation usually do linking at compile time. It is also possible to load libraries at run-time with C++ , usually in the form of plugins.

A.1 Plugins

Plugins in C++ are platform-dependent and usually built as a shared library object (.so on UNIX-systems and .dll on Windows). They need to adhere to the application programming interface (API) of the system they are used on. In addition, they must adhere to an API defined by the application that will load the plugin. If you want to make an application for which others can write plugins, you must first define how and when they should be able to modify the execution of your application.

A.2 Libraries

In contrast to plugins, libraries are intended for use when you are writing your application. They extend the language, in our case C++ , with new features and functionality that you as a developer may use. To use libraries in C++ , you are usually provided with a binary library file (.so on UNIX-systems and .dll on Windows) and some header files (.h) that define the classes or functions available in the library. When compiling your code, you must link the library which you are using, and tell the compiler where to search for the library and header files. This is usually done by adding the -I, -L and -l flags to your compile command:

```
g++ mylib.cpp -L/library/path -lnameoflibrary -I/header/path -o mylib
```

This will look for the `lnameoflibrary.so` file in `/library/path` and the header files in `/header/path` will be available whenever you use the `#include` directive in your source

files. Note the order of the parameters to the `g++` command; the source file name comes before any linking commands. This is important, because the linker will only make use of the parameters in this certain order.¹

A.3 Project Structure for Library Based Programs

If you are using qmake and Qt Creator, a great way to work on a program that is separated into its own library, some tests and the actual application, is to use subprojects. This will give you one main project with three subprojects, one for the app itself, one for the library, and one for the tests. In addition, it is useful to set up a helper project file, named `defaults.pri`. The structure of the project is like this on my file system:²

```
MyProject
|- MyProject.pro
|- defaults.pri
|- app/
|   |- app.pro
|   |- main.cpp
|- src/
|   |- src.pro
|   |- myclass.cpp
- tests/
  |- tests.pro
  - main.cpp
```

The main project file, `MyProject.pro` will now be based on a `subdirs` template, and may look like this:

```
TEMPLATE = subdirs
CONFIG+=ordered
SUBDIRS = \
    src \
    app \
    tests
app.depends = src
tests.depends = src
```

The `app.depends` and `tests.depends` statements makes sure that the `src` project is compiled before the application and tests, because the `src` directory contains the library that will be used by both the app and the tests.

A.3.1 defaults.pri

Each of the other `.pro` files will include `defaults.pri` to have all the headers available in a useful path and set other common options for compiling the different projects. As

¹In fact, there is a separate program `ld` that is being called by `g++` to do the actual linking. It is `g++` that needs to be served the parameters in this order to be able to pass them on to `ld`.

²An example project using this code structure has been posted on Github by Filip Sund, at github.com/FSund/qtcreator-project-structure. (Thanks to Filip for doing this!)

an example, defaults.pri may contain the following:

```
INCLUDEPATH += $$PWD/src
SRC_DIR = $$PWD
```

If the library, main program and tests use common libraries, it is very useful to have the defaults.pri define these dependencies too.

A.3.2 src/

In the src folder, we'll put myclass.cpp, which is the class that we want to use and test. The src project will be compiled to a library that may be used both by the app and tests projects. To achieve this, we set the `TEMPLATE` variable in the .pro file to `lib` and specify the name of our library with the `TARGET` variable:

```
include(../defaults.pri)
CONFIG -= qt

TARGET = myapp
TEMPLATE = lib

SOURCES += myclass.cpp
HEADERS += myclass.h
```

An example class with a function that takes two doubles as parameters and returns the sum, is shown below:

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    double addition(double a, double b);
};

#endif // MYCLASS_H
```

The implementation would look like this in the source file:

```
#include "myclass.h"

double MyClass::addition(double a, double b) {
    return a + b;
}
```

A.3.3 app/

The app project needs to be set up to compile to an executable. We set the `TEMPLATE` variable to `app` for qmake to do this for us. This project can now be extremely small,

because it will only be the entry point to functionality that is implemented in the src project. The app project will depend on the shared library compiled from src. Therefore, we set the `LIBS` variable in the project file to make sure we link app to the library:

```
include(..../defaults.pri)

TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt

SOURCES += main.cpp

LIBS += -L..../src -lmyapp
```

The main.cpp file could be a simple program that uses MyClass to add 10 and 20 and prints the result:

```
#include <myclass.h>
#include <iostream>

using namespace std;

int main()
{
    MyClass adder;
    cout << adder.addition(10, 20) << endl;
    return 0;
}
```

A.3.4 tests/

The test project will be set up much like the app project. It will link to the shared library, in addition to the unit testing library:

```
include(..../defaults.pri)
TEMPLATE = app

CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt

SOURCES += main.cpp

LIBS += -lunitest++ -L..../src -lmyapp
```

The main.cpp in tests file which could contain the following test to check that the addition function works as expected:

```
#include <unittest++/UnitTest++.h>
#include <myclass.h>

TEST(MyMath) {
    MyClass my;
    CHECK_CLOSE(7.0, my.addition(3.0, 4.0), 1e-14);
}

int main()
{
    return UnitTest::RunAllTests();
}
```

See Appendix B for details on unit testing.

Appendix B

Unit Testing

While developing, it is a good idea to implement unit tests to make sure that each part of the code does its job properly. This helps tracking down bugs, prevents you from introducing errors and may even make you sleep better at night, knowing that at least your 3D vector class does what it is supposed to.

In fact, unit tests are a great tool for the development process as well.. In so-called test-driven development, the idea is that you should write tests before you even write your functions. It is not to say that you should take it to the extremes and let every line of code be preceded by a unit test implementation, but thinking about how your new feature may be tested could give you some ideas about what its interface should look like.

B.1 Using a Simple Unit Test Library

The UnitTest++ framework is a lightweight library for unit testing in C++ which is extremely simple to use. To download and install it in Ubuntu, all you have to do is

```
sudo apt-get install libunitest++-dev
```

Now you can create a simple test program to check that it works and play around with its interface:

```
#include <unitest++/UnitTest++.h>
TEST(TrivialExample) {
    int a = 2;
    int b = 3;
    int c = a + b;
    CHECK_EQUAL(5, c);
}
int main()
{
    return UnitTest::RunAllTests();
}
```

When you compile, all you need to do is to link to the library, which is done by adding the following to your .pro file if you are using qmake:

```
LIBS += -lunittest++
```

You may write as many tests as you like and check that they are successful with the CHECK macro. The above example is trivial, and will check if the sum of the numbers 2 and 3 results in 5. This is also a pointless test to implement because we can expect that simple arithmetic operations in C++ just work. However, it may be useful to test such simple operations in our own classes, such as the `Vector3` implementation we use in both the Hartree-Fock code and the molecular dynamics (MD) code:

```
#include <unittest++/UnitTest++.h>
#include "vector3.h"
TEST(VectorAddition) {
    Vector3 a( 1.0, 2.0, -3.0);
    Vector3 b( -3.0, 2.0, -1.5);
    Vector3 c = a + b;
    CHECK_CLOSE(-2.0, c.x(), 1e-14);
    CHECK_CLOSE( 4.0, c.y(), 1e-14);
    CHECK_CLOSE(-4.5, c.z(), 1e-14);
}
int main()
{
    return UnitTest::RunAllTests();
}
```

Note the use of `CHECK_CLOSE` to compare doubles. Because of the floating point precision, we are not guaranteed that doubles are exactly equal after arithmetic operations and direct initialization. We instead have to check that they differ at most by numerical precision, namely with a maximum error of 10^{-14} .

A good idea is also to separate the tests from your main code, by rewriting the code as a library, as discussed in Appendix A, and creating a separate project that links to this library. This way you don't have to depend on your users having UnitTest++ installed to use your program.

B.2 Automatic Unit Testing with Jenkins

At one point you are likely to be in a position where you find it tiresome to have to go into that folder where the tests are defined and run them manually. This is where Jenkins comes in to play.

Jenkins is a build bot that is designed to fetch your code from wherever, download, build and run the commands you need to test your code. It also provides a very nice web interface that will allow you to keep track of which builds are working and which are not.

This has the advantage that you don't have to run your tests manually all the time, or add some obscure way to run the tests before running your application. With

Jenkins, everything happens in the background. And Jenkins can be configured to run builds periodically, whenever your source code changes or is pushed to Git.

Installing Jenkins in the Ubuntu operating system is done the usual way:

```
sudo apt-get install jenkins
```

After installing Jenkins, you should immediately be able to access it by opening your browser and entering `localhost:8000` in your address bar.

Setting up your first project is done through the web interface. Simply click Jenkins → New job and give it a name of your choice. Select Build a free-style software project as your setting and click OK. Your project is now ready to be set up, and you may add an “execute shell” build step to input the commands it will run to pull your project files from your Git server, compile the code and run the tests. After doing so, click Apply and test your build by clicking Build now.

You may check the result by looking at the Build History on the left hand side of the screen. Red means unstable (crashed) build, blue means stable build. Hover the pointer over a build and click Console output to see the exact output of the build. This is very useful if the build should fail and you want to know why.

If all goes well, you may explore the options to build whenever your source code changes or periodically, by looking at the Build triggers.

There are also plenty of other features in Jenkins, but I'll leave it up to you to explore them all. The last thing you should do is to add Jenkins as your browser's start up page or always keep it visible on a screen in your office, so that you and your co-workers at any given time may know how your builds perform.

Appendix C

Visualization Tools and Graphics Programming

In this chapter, we will discuss some components of modern shader programming.

C.1 Graphics Tools

In this thesis project, we have focused on graphics programming with Qt, OpenGL and Qt3D [86]. In this section, we will give a quick overview of the different frameworks and argue for why they have been chosen.

C.1.1 Qt Application Framework

Qt is an open source application framework consisting of multiple modules for graphical user interface (GUI) programming. It has a strong focus on cross-platform support, and thus provides tools for creating GUIs both for desktop and mobile devices. We will use Qt to create controls for user interaction, such as buttons, sliders, checkboxes, etc.

C.1.2 OpenGL

OpenGL is one of the industry standards for the graphics specification used by hardware vendors. It defines how the computer software should communicate with the graphics hardware and has several implementations for multiple programming languages. OpenGL has a C API, with set of extensions that implemented by a range of libraries.

C.1.3 Qt3D

Qt3D is an experimental module that is currently not included in the main Qt framework, but it is officially distributed through Qt Project's Git repository. However, some Linux distributions provide binary packages.

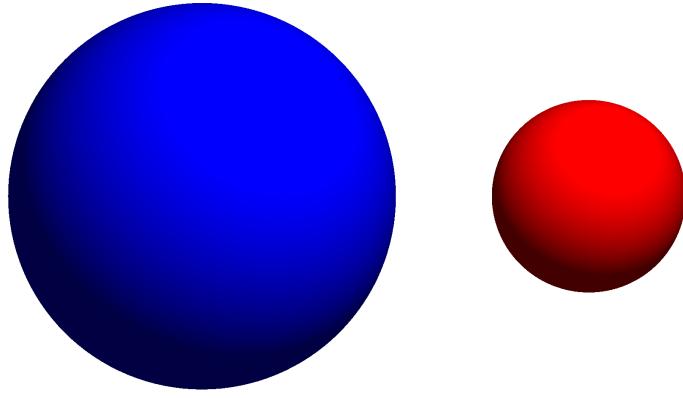


Figure C.1: Two spheres rendered in Qt3D.

The Qt3D module provides a simple interface for programmers to write OpenGL code. A basic example showing a blue and a red sphere in a 3D viewport can easily be written in QML:

```
import Qt3D 2.0
import Qt3D.Shapes 2.0

Viewport {
    Sphere {
        x: -5.0
        y: 0
        z: 0
        radius: 5.0
        effect: Effect {
            color: "blue"
        }
    }
    Sphere {
        x: 5.0
        y: 0
        z: 0
        radius: 2.5
        effect: Effect {
            color: "red"
        }
    }
}
```

With a few adjustments to the camera position and the lighting, the resulting rendering will look something like the one in Figure C.1.

More complex shapes are easily included by using the `Mesh` type and pointing its `source` property to a file containing a 3D mesh. Dynamic objects, such as the ones we have used to render many particles efficiently, may be written in C++ by extending Qt3D classes like `QQuickItem3D`. More information is found in the Qt3D documentation [86].

C.2 Shader Programming

Up until April 2004, OpenGL was explicitly programmed in what is known as a *fixed function pipeline*. This allowed for a limited number of operations that could be performed by the graphics card. When an instruction had been executed, the control was returned to the main program, which would continue to send new instructions to the graphics card. Custom computations had to be performed on the CPU in-between the instructions sent to the GPU.

In OpenGL version 2.0, the OpenGL shader language (GLSL) was formally introduced and enabled a *programmable pipeline*. This allowed the programmer to create separate programs that are compiled and executed on the graphics card. This is known as *shader programming*.

Shader programming opened up for more possibilities and higher performance in graphics programming. The graphics card is now able to optimize the instructions in the shader programs because the whole instruction set is known before it is executed. The drawback of programmable pipelines is the higher complexity, making them harder to learn for newcomers.

The programmable pipeline typically consists of a number of shader programs that are executed in order. Each program defines input and output variables used to pass information between the stages. In the following, we will describe the use of the vertex, fragment geometry, and tessellation shader programs.

C.2.1 Vertex Shader

The vertex shader is a program that is executed for each vertex in the scene. A vertex is a point in 3D space and all objects are usually built up from collections of such vertices. Triangles or triangle strips are in OpenGL the common construction units used to build up geometries. A triangle of course has three vertices, while a triangle strip may consist of a large number of vertices.

Each vertex is passed through the vertex shader and the shader may modify the properties of each vertex, but it may not remove vertices or emit new vertices. Only the tessellation shader or the geometry shader may modify the number of vertices, which we will discuss in Appendix C.2.3. The vertex shader is on the other hand first and foremost used to transform the vertex position from the world space to the screen space. As all objects have their geometry and position defined in a model space, that is, a 3D space that is relative to some origin point in the 3D model, they must be transformed to the screen space, which is a 2D space fixed to our computer screen or the screen on a mobile phone.

Model-View-Projection Matrix

This transformation is performed in a set of steps, resulting in a transformation from the model space, with an origin point usually placed in the center of the model, through a world space with its origin placed where we decide, and finally projected down onto our screen.

4-component Position Vector

The first thing to notice before delving into the details of the matrices used in this projection is that in 3D graphics, 4-component vectors are used to define positions and directions. The reasoning behind this becomes evident when we start working with these vectors and matrices, but in short, the idea is that the first three components represent the x , y and z components of the position or direction, while the fourth component, denoted w , is used to define the vector as a position if $w = 1$ and a direction if $w = 0$:

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}. \quad (\text{C.1})$$

The benefit of using four components is that we now may transform the position of a vector by a matrix multiplication:

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + a \\ y + b \\ z + c \\ 1 \end{bmatrix}. \quad (\text{C.2})$$

However, if we apply the same matrix multiplication to a direction (the w -component is then $w = 0$) we see that there is no change - which is good, because it makes no sense to perform a translation on a direction:

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}. \quad (\text{C.3})$$

Model Matrix

The model matrix is used to transform the coordinates of the vertices in the model from the model space (with the origin usually placed in the center of the model) to the world space. This means that if the object is located at a specific position, with a specific scaling and rotation, in our world space, the model matrix will take it to that position by translation, rotate it and scale it. All these operations are baked into the model matrix.

View Matrix

With the object placed properly in our world, it is about time to place our camera correctly. In 3D graphics programming, it is easier to move the world, rather than to move the camera. We therefore move the object to its correct position in the world, before we move and rotate the entire world relative to the camera. This is the purpose of the view matrix.

Projection Matrix

Finally, we need to project everything onto our screen properly. We now have all objects placed such that the coordinates they have relative to the camera (the current origin) coincide with the coordinate they will have on our screen. Further, we want the z -component of their position to be taken into account. That way, the appearance of the object on the screen is dependent on the distance from the camera to the object. The projection matrix takes care of this.

Setting up the Model-View-Projection Matrix

The final model-view-projection matrix is constructed by multiplying these matrices. The resulting matrix can be applied directly to any vector:

$$PVM\mathbf{v} = (PVM)\mathbf{v} = T\mathbf{v}, \quad (\text{C.4})$$

where P is the projection matrix, V is the view matrix and M is the model matrix. (Note that the ordering of the operators is the opposite of the verbal ordering in model-view-projection. This is because the order of operations is reversed in matrix-vector multiplication.)

Qt3D and the Model-View-Projection Matrix

Fortunately, there is not much we need to do to implement the model-view-projection matrix in Qt3D. All we need to do is decide the placement of our objects and the camera, and settings such as the field-of-view (FOV) and clipping distances of the camera. Qt3D does all the hard work with constructing the three matrices, and makes them available to the vertex shader through the variables `qt_ProjectionMatrix`, `qt_ModelViewProjectionMatrix` and alike [86].

Note that every object in our 3D scene will have its unique model-view-projection matrix. The view and projection matrices are the same for all objects, but the model and the model-view-projection matrices differ.

Pass-through Vertex Shader

A *pass-through* is a shader that doesn't alter the incoming data before passing it on to the next step in the pipeline. There is no such thing as an actual default shader in OpenGL. This is likely because the developer in any case needs to tell the graphics processing unit (GPU) what variables are being passed from the central processing unit (CPU) to the shader. Because there are no constraints on the names of these variables, the developer needs to write even the simplest shader programs by hand.

Qt3D eases this by defining some default variables that are automatically passed from the CPU to the shader program. The pass-through vertex shader in Qt3D could look something like:

```
#version 330 core
in vec4 qt_Vertex;
uniform mat4 qt_ModelViewProjectionMatrix;
```

```

void main(void)
{
    gl_Position = qt_ModelViewProjectionMatrix * qt_Vertex;
}

```

All this shader does is to take in each vertex defined in an object together with the model-view-projection matrix defined by Qt3D, and tell the GPU that the vertex position on our screen is the vertex position in model space, multiplied by the model-view-projections matrix. This is done by setting the `gl_Position` variable, which is expected to be set by the vertex shader. This shader contains no color or texture information and will therefore give a very plain result.

Vertex Shader with Texture Information

This is pretty much the same as the pass-through shader, but for each vertex, we include some information about what part of the texture should be aligned to the position of this vertex.

```

#version 330 core
in vec4 qt_Vertex;
in vec4 qt_MultiTexCoord0;
uniform mat4 qt_ModelViewProjectionMatrix;
out vec4 texCoord;

void main(void)
{
    gl_Position = qt_ModelViewProjectionMatrix * qt_Vertex;
    texCoord = qt_MultiTexCoord0;
}

```

This shader is the default example shown in the Qt3D documentation [86]. Note that the texture alignment information is passed from the CPU to the shader program through the `in vec4 qt_MultiTexCoord0` variable.

C.2.2 Fragment Shader

Before the fragment shader is invoked, a process known as rasterization is performed. In this process, the vertices are joined to create a 2D-image. Depending on how the vertices are interpreted, either as lines, points or polygons, their outgoing properties are interpolated for all pixels covered by the object. So if the three vertices of a triangle has different outgoing color-values, an interpolated combination of their values will be made available to the fragment shader for every pixel covered by the triangle.

The fragment shader is a program that uses the interpolated values from the vertex shader to construct the final image. The below fragment shader ignores the output of the vertex shader and renders the object in a flat color:

```

#version 330 core
out vec4 fragmentColorOut;

```

```
void main()
{
    fragmentColorOut = vec4(1.0, 0.0, 0.0, 1.0);
}
```

OpenGL 3.1 and above only expects one output value from the fragment shader. Any `out vec4` value will do.

A more interesting fragment shader would of course use the input from the vertex shader for something useful, such as loading values from a texture based on the texture coordinates (which are also interpolated between the vertices):

```
#version 330 core
uniform sampler2D myTexture;
in vec4 texCoord;
out vec4 fragmentColorOut;

void main(void)
{
    fragmentColorOut = texture(myTexture, texCoord.st);
}
```

Here, the `texture2D` function will look up the color of the texture at the 2D texture coordinate `texCoord.st`. The `vec4` member `vec4.st` is just an alias for `vec4.xy`. In fact, many of the `vec4` member variables are the same, such as `vec4.xyzw`, `vec4.rgb` and `vec4.stuv`. The aliasing is just for readability, and `vec4.st` is the standard naming of the *x*- and *y*-coordinates of a texture value, also known as *s* and *t*.

The texture will have to be passed to the fragment shader program as a `uniform sampler2D` object and uploaded to the graphics card before the draw call is invoked. As long as we are using Qt3D, this is done automatically by setting the `texture` property of a QML Effect element. The texture is available in the fragment shader as `uniform sampler2D qt_texture0;`. Details are found in the Qt3D documentation [86].

C.2.3 Geometry Shader

With version 3.2 of OpenGL, the Geometry Shader was introduced. This is a shader that allows the introduction of more vertices. In contrast to the vertex and fragment shader programs, it is optional. If no geometry shader is defined, all vertices are passed on to the fragment shader from the vertex shader.

Pass-through Geometry Shader

A pass-through geometry shader is defined as follows:

```
#version 330 core

layout(triangles) in;
layout(triangle_strip, max_vertices = 4 ) out;

void main() {
```

```
    for(int i = 0; i < 3; i++) {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}
```

This shader contains only functions and variables that are already defined in GLSL. It tells the GPU to expect triangles as input, and that it will output triangle strips. For each triangle, we make a triangle strip with only 3 vertices, which means that we are basically outputting triangles as well.

The main function in this shader loops over the vertices (there are 3 in a triangle), sets `gl_Position` to the position of each vertex in the incoming triangle and outputs an vertex with the GLSL function `EmitVertex()`. When done, the GLSL function `EndPrimitive()` is called to tell the GPU that no more vertices will be emitted.

C.2.4 Tessellation Shader

The final shader of interest is the tessellation shader. This is a shader that allows subdivision of the geometry originally passed from the CPU to the GPU, normally used to increase the level of detail or smooth out sharp corners in objects.

Because this has been of limited use in this thesis, we won't go into any detail about this shader, but simply note that it has become a useful tool in computer graphics.

Appendix D

Visualization Machine

D.1 Hardware

Our reference machine used to run all visualizations has the following hardware installed:

CPU Intel Core i7-4820K 3,70GHz

RAM Corsair Simm DDR3 PC1600 32GB CL10

GPU Gainward GeForce GTX TITAN 6GB 837MHz, 384bit

MB MSI X79A-GD45 Plus

HDD Intel SSD/530 Series 240GB 2.5" SATA 6Gb/s

D.2 Software

The following software has been used:

OS Ubuntu Linux

Graphics driver NVIDIA 313

Glossary

ANN

Artificial neural network. Inspired by neural networks in nature, artificial neural networks are used in machine learning. They learn to predict an output based on a given input. 3, 6, 7, 13, 19–21, 54, 55, 61, 62, 65, 89, 91–93, 95–98, 100, 102, 103, 109, 121, 122, 124–126, 128, 129, 151–155

API

application programming interface 159

Armadillo

Linear algebra library [61] for C++ with easy-to-use wrapper functions for LAPACK [62] and BLAS [87]. 71, 76, 138

backpropagation

Training method used in artificial neural networks. Belongs to the gradient descent family of algorithms and is implemented in FANN. 59–61, 94, 98–100, *see* ANN & FANN

Blender

3D software package for amazing 3D visualizations, animations and even movies [29]. 7, 138–140, 154

C

Low-level programming language created between 1969 and 1973 at AT&T Bell Labs. Much used in the development of high-performance software, operating systems, embedded systems and device drivers. 154

C++

Programming language that combines the low-level performance and control of C with object orientation. 5, 69, 77–79, 117, 118, 138, 154, 159, 165, 166, 170, 179, 183

CPU

Central processing unit. The main processing unit on a computer or a mobile device. 140, 141, 143, 144, 173, 174

Denseness

Visualization tool for electron densities. Developed for this thesis project. See page 3 for a link to the source code. 137, 152, *see* Hartree-Fock

DFT

Density functional theory. A quantum mechanical method similar to Hartree-Fock. Uses the Khon-Sham equations. All correlation, including exchange is handled by a energy functional in the Hamiltonian. 5, 23, 53, 153

DMC

Diffusion Monte Carlo. Similar to VMC, but extends the theory in a way that allows the resulting energy to converge to the exact ground state energy. 6, 153, *see* VMC

Emdee

Molecular dynamics library, simulator and visualizer. Developed for this thesis project. See page 3 for a link to the source code. x, 6, 15, 18, 100, 103–105, 108, 115, 121, 144, 145, *see* molecular dynamics

FANN

Fast Artificial Neural Network Library. Developed in C by Steffen Nissen [52]. 3, 55, 60, 61, 91–94, 98, 99, 101, 102, 151, 153, 154, *see* ANN

FANN-MD

A set of scripts developed for this thesis project. Developed for this thesis project. See page 3 for a link to the source code. 91, *see* Hartree-Fock, molecular dynamics & FANN

FCI

Full Configuration Interaction. A computational many-body quantum mechanics method that provides theoretically exact results when using an infinite basis set [13]. It provides high-quality results, but is computationally expensive. 6, 82, 153, *see* SCF

GLSL

OpenGL shader language. Programming language for OpenGL shader programming. Sits at the center of the programmable pipeline and defines a C-style language used by programmers to instruct the GPU. 7, 135, 152, 171, *see* GPU

GPU

Graphics processing unit. A chip or device in a computer responsible for graphical calculations and rendering to screen. 7, 140, 141, 143, 144, 173, 174

GUI

Graphical user interface. Computer interface that allows users to interact with visual feedback. 7, 152, 169, *see* Qt & OpenGL

Hartree-Fock

A quantum mechanical method that is self consistent. The Hartree-Fock equations are defined by the Fock operator, which depends on the spin-orbitals that are the solutions of the equations. This method is thus solved iteratively. 3, 5–7, 13, 20, 23, 33, 34, 39, 41, 44, 49, 53, 54, 61, 62, 65, 66, 74, 76, 77, 80, 82, 89, 91–93, 102, 114, 124, 137, 138, 151–155, 166, *see* SCF

HDF5

Hierarchical Data Format. File format used to store and organize large amounts of numerical data. Developed at the National Center of Supercomputing Applications. 138

Kindfield

Hartree-Fock program and library for atoms and molecules. Developed for this thesis project. See page 3 for a link to the source code. 6, 62, 65, 66, 70, 77, 79, 82, 84, 91, 93, 121, 122, 137, *see* Hartree-Fock

Lennard-Jones

A simple and efficient two-body potential, often used in simulations of noble gases. 6, 15–17, 96, 115, 145

Mayavi

Python package and standalone program for data visualization. Developed by Enthought, Inc. 7, 138, 139, 154

molecular dynamics

Group of simulation methods for atoms and molecules with Newtonian mechanics as the basis for time development. 3, 6, 7, 11, 12, 14, 17, 19, 20, 23, 26, 54, 55, 61, 62, 65, 77, 82, 89, 91, 93, 97–100, 102, 103, 108, 109, 111, 112, 114–119, 121, 122, 128, 133, 144–147, 151–155, 166

MSE

Mean square error. Often used as an error measurement in the training of artificial neural networks. 61, 95, *see* ANN

Oculus Rift

Virtual reality headset developed by Oculus VR Inc. They publish a SDK for developers to integrate the virtual reality technology in their applications [30]. 145, *see* SDK

OpenGL

Open graphics library. Specification for communication between applications and GPUs. May also refer to the C API for the same communication. 133, 141, 152, 169–171, 173, 175, *see* GPU

OpenGL ES

Specialized version of OpenGL for mobile devices and low-level GPUs. *see* GPU & OpenGL

perturbation theory

Extending upon certain quantum mechanical theories, such as Hartree-Fock, is possible through perturbation theory. Introduces higher order perturbations to the energy to include correlations between electrons. 23, 153, *see* Hartree-Fock

PES

Potential energy surface. Describes the potential energy hypersurface of a configuration of atoms. 3, 6, 11, 12, 26, 55, 65, 89, 91, 108, 128, 151, 154, 155

Poson

Large-scale molecular dynamics visualization tool with support for virtual reality hardware. Developed for this thesis project. See page 3 for a link to the source code. x, 145–147, *see* molecular dynamics

QMC

Quantum Monte Carlo. A group of computational quantum mechanics methods that apply Monte Carlo integration to calculate the energy expectation value. 23, 28, 29, 54, *see* VMC & DMC

QML

Qt Modeling Language. Declarative language for easy development and design of GUIs with Qt. 170, *see* Qt & GUI

Qt

Application framework designed to create graphical user interfaces on multiple platforms. Also includes libraries for networking, file-parsing, and much more. 7, 133, 144, 169, *see* GUI

Qt3D

Experimental module for Qt that gives developers easy access to OpenGL features [86]. 133, 169, 170, 173–175, *see* Qt & OpenGL

Quickprop

A backpropagation method for artificial neural networks. This method is implemented in FANN. 59, *see* ANN, FANN & backpropagation

RHF

Restricted Hartree-Fock. When we use a finite basis set in the Hartree-Fock equations, we may choose to pair particles with opposite spin and equal spatial orbitals, which is the restricted Hartree-Fock (RHF) method and results in the Roothaan equations. We could also allow all particles to have different spin and spatial orbitals, resulting in the unrestricted Hartree-Fock method (UHF) and the Pople-Nesbet equations. 33, 40, 42, 43, 65, 69, 75, 76, 80–82, 84, 85, *see* Hartree-Fock

RPROP

Resilient backpropagation. A backpropagation method for artificial neural networks. This method is implemented in FANN. 59, *see* ANN, FANN & backpropagation

SCF

Self consistent field. Some computational quantum mechanics methods are self-consistent. This means that the equation to solve depends on the solutions, and thus needs to be solved iteratively. The Hartree-Fock method was earlier known as the self consistent field method. *see* Hartree-Fock

SDK

software development kit 145

UHF

Unrestricted Hartree-Fock. 33, 42, 43, 65, 69, 75, 76, 80–82, 84, 85, 95–98, 121, 122, 124, *see* RHF

UnitTest++

C++ unit testing library. 115, *see*

VMC

Variational Monte Carlo. Computational quantum mechanics method that calculates the energy expectation value by Monte Carlo integration. 6, 153

Bibliography

- [1] M. H. Mobarhan. "From Quantum to Molecular, A Review of Gaussian Basis Sets in Ab Initio Molecular Dynamics". MA thesis. University of Oslo, In preparation, 2014.
- [2] H. M. Eiding. "Ab Initio Studies of Molecules". MA thesis. University of Oslo, In preparation, 2014.
- [3] A. Hafreager. "Flow of Dilute Gases in Complex Nanoporous Media". MA thesis. University of Oslo, 2014.
- [4] H. Shull and G. G. Hall. "Atomic Units". In: *Nature* 184. (1959), 1559. DOI: [10.1038/1841559a0](https://doi.org/10.1038/1841559a0).
- [5] A. Valdes, J. Brillet, M. Grätzel, H. Guðmundsdóttir, H. A. Hansen, H. Jónsson, P. Klüpfel, G.-J. Kroes, F. Le Formal, I. C. Man, et al. "Solar hydrogen production with semiconductor metal oxides: new directions in experiment and theory". In: *Physical Chemistry Chemical Physics* 14. (2012), 49. DOI: [10.1039/C1CP23212F](https://doi.org/10.1039/C1CP23212F).
- [6] S. Agrawalla and A. C. T. van Duin. "Development and Application of a ReaxFF Reactive Force Field for Hydrogen Combustion". In: *The Journal of Physical Chemistry A* 115. (2011), 960. DOI: [10.1021/jp108325e](https://doi.org/10.1021/jp108325e).
- [7] M. F. Russo Jr and A. C. van Duin. "Atomistic-scale simulations of chemical reactions: Bridging from quantum chemistry to engineering". In: *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* 269. (2011), 1549. DOI: [10.1016/j.nimb.2010.12.053](https://doi.org/10.1016/j.nimb.2010.12.053).
- [8] C. Cramer. *Essentials of Computational Chemistry: Theories and Models*. Wiley, 2005.
- [9] C. D. Sherrill. "Frontiers in electronic structure theory". In: *The Journal of chemical physics* 132. (2010), 110902. DOI: [10.1063/1.3369628](https://doi.org/10.1063/1.3369628).
- [10] D. E. Shaw, P. Maragakis, K. Lindorff-Larsen, S. Piana, R. O. Dror, M. P. Eastwood, J. A. Bank, J. M. Jumper, J. K. Salmon, Y. Shan, et al. "Atomic-level characterization of the structural dynamics of proteins". In: *Science* 330. (2010), 341. DOI: [10.1126/science.1187409](https://doi.org/10.1126/science.1187409).
- [11] R. Abrol, B. Kirchner, and J. Vrabec. *Multiscale Molecular Methods in Applied Chemistry*. Vol. 307. Springer, 2011.
- [12] J.M. Thijssen. *Computational Physics*. Cambridge University Press, 2007.

- [13] I. Shavitt and R. Bartlett. *Many-Body Methods in Chemistry and Physics: MBPT and Coupled-Cluster Theory*. Cambridge Molecular Science. Cambridge University Press, 2009.
- [14] K. Burke. "Perspective on density functional theory". In: *The Journal of chemical physics* 136. (2012), 150901. DOI: [10.1063/1.4704546](https://doi.org/10.1063/1.4704546).
- [15] C. D. Sherrill and H. F. Schaefer III. "The Configuration Interaction Method: Advances in Highly Correlated Approaches". In: *Advances in quantum chemistry* 34 (1999), 143. DOI: [10.1016/S0065-3276\(08\)60532-8](https://doi.org/10.1016/S0065-3276(08)60532-8).
- [16] K. R. Leikanger. "Full Configuration Interaction Monte Carlo Studies of Quantum Dots". MA thesis. 2013.
<http://urn.nb.no/URN:NBN:no-38650>.
- [17] V. K. B. Olsen. "Full Configuration Interaction Simulation of Quantum Dots". MA thesis. 2012.
<http://urn.nb.no/URN:NBN:no-32952>.
- [18] D. Ceperley, G. Chester, and M. Kalos. "Monte Carlo simulation of a many-fermion study". In: *Physical Review B* 16. (1977), 3081. DOI: [10.1103/PhysRevB.16.3081](https://doi.org/10.1103/PhysRevB.16.3081).
- [19] J. Høgberget. "Quantum Monte-Carlo Studies of Generalized Many-body Systems". MA thesis. University of Oslo, 2013.
<http://urn.nb.no/URN:NBN:no-38645>.
- [20] W. A. Lester, B. Hammond, and P. J. Reynolds. *Monte Carlo methods in ab initio quantum chemistry*. World Scientific, 1994.
- [21] C. Hirth. "Studies of quantum dots: Ab initio coupled-cluster analysis using OpenCL and GPU programming". MA thesis. 2012.
<http://urn.nb.no/URN:NBN:no-31657>.
- [22] D. Frenkel and B. Smit. *Understanding molecular simulation: from algorithms to applications*. Vol. 1. Academic press, 2001.
- [23] L. Verlet. "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules". In: *Physical Review* 159. (1967), 98. DOI: [10.1103/PhysRev.159.98](https://doi.org/10.1103/PhysRev.159.98).
- [24] F. H. Stillinger and T. A. Weber. "Computer simulation of local order in condensed phases of silicon". In: *Physical Review B* 31. (1985), 5262. DOI: [10.1103/PhysRevB.31.5262](https://doi.org/10.1103/PhysRevB.31.5262).
- [25] L. M. Raff, M. Malshe, M. Hagan, D. I. Doughan, M. G. Rockley, and R. Komanduri. "Ab initio potential-energy surfaces for complex, multichannel systems using modified novelty sampling and feedforward neural networks". In: *The Journal of Chemical Physics* 122., 084104 (2005), DOI: [10.1063/1.1850458](https://doi.org/10.1063/1.1850458).
- [26] R. Skorpa, J.-M. Simon, D. Bedeaux, and S. Kjelstrup. "Equilibrium properties of the reaction $H_2 \rightleftharpoons 2H$ by classical molecular dynamics simulations". In: *Physical Chemistry Chemical Physics* 16. (2013), 1227. DOI: [10.1039/C3CP54149E](https://doi.org/10.1039/C3CP54149E).

- [27] D. Kohen, J. C. Tully, and F. H. Stillinger. “Modeling the interaction of hydrogen with silicon surfaces”. In: *Surface science* 397. (1998), 225. DOI: [10.1016/S0039-6028\(97\)00739-5](https://doi.org/10.1016/S0039-6028(97)00739-5).
- [28] P. Ramachandran and G. Varoquaux. “Mayavi: 3D Visualization of Scientific Data”. In: *Computing in Science & Engineering* 13. (2011), 40. DOI: [10.1109/MCSE.2011.35](https://doi.org/10.1109/MCSE.2011.35).
- [29] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Blender Institute, Amsterdam, 2014.
<http://www.blender.org>.
- [30] Oculus VR, Inc. *Oculus VR*. 2014.
<http://www.oculusvr.com/>.
- [31] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical simulation in molecular dynamics*. Springer, 2007.
- [32] D. Marx and J. Hutter. “Ab initio molecular dynamics: Theory and implementation”. In: *Modern methods and algorithms of quantum chemistry* 1 (2000), 301.
- [33] H. P. Langtangen. *Introduction to computing with finite difference methods*. University of Oslo, 2013.
<http://hplgit.github.io/INF5620/doc/web/notes.html>.
- [34] J. E. Jones. “On the Determination of Molecular Fields. II. From the Equation of State of a Gas”. In: *Proceedings of the Royal Society of London. Series A* 106. (1924), 463. DOI: [10.1098/rspa.1924.0082](https://doi.org/10.1098/rspa.1924.0082).
- [35] D. M. Heyes. “Thermodynamic stability of soft-core Lennard-Jones fluids and their mixtures”. In: *The Journal of Chemical Physics* 132., 064504 (2010), DOI: [10.1063/1.3319510](https://doi.org/10.1063/1.3319510).
- [36] P. Vashishta, R. K. Kalia, J. P. Rino, and I. Ebbsjö. “Interaction potential for SiO₂: a molecular-dynamics study of structural correlations”. In: *Physical Review B* 41. (1990), 12197. DOI: [10.1103/PhysRevB.41.12197](https://doi.org/10.1103/PhysRevB.41.12197).
- [37] J. Tersoff. “New empirical approach for the structure and energy of covalent systems”. In: *Physical Review B* 37 (1988), 6991. DOI: [10.1103/PhysRevB.37.6991](https://doi.org/10.1103/PhysRevB.37.6991).
- [38] A. C. Van Duin, S. Dasgupta, F. Lorant, and W. A. Goddard. “ReaxFF: a reactive force field for hydrocarbons”. In: *The Journal of Physical Chemistry A* 105. (2001), 9396. DOI: [10.1021/jp004368u](https://doi.org/10.1021/jp004368u).
- [39] H. M. Aktulga, S. A. Pandit, A. C. van Duin, and A. Y. Grama. “Reactive molecular dynamics: Numerical methods and algorithmic techniques”. In: *SIAM Journal on Scientific Computing* 34. (2012), C1. DOI: [10.1137/100808599](https://doi.org/10.1137/100808599).
- [40] D. Griffiths. *Introduction to quantum mechanics*. Pearson Prentice Hall, 2005.
- [41] M. Hjorth-Jensen. *Computational Physics*. University of Oslo, Norway, 2011.
<http://www.uio.no/studier/emner/matnat/fys/FYS3150>.
- [42] J. R. Taylor. *Classical mechanics*. University Science Books, 2005.

- [43] D. Sholl and J. Steckel. *Density Functional Theory: A Practical Introduction*. Wiley, 2011.
- [44] J. L. Sonnenberg, H. B. Schlegel, and H. P. Hratchian. “Spin Contamination in Inorganic Chemistry Calculations”. In: *Encyclopedia of Inorganic Chemistry*. John Wiley & Sons, Ltd, 2006. doi: [10.1002/0470862106.ia617](https://doi.org/10.1002/0470862106.ia617).
- [45] A. Szabo and N. S. Ostlund. *Modern quantum chemistry: introduction to advanced electronic structure theory*. Courier Dover Publications, 1996.
- [46] R. Ditchfield, W. J. Hehre, and J. A. Pople. “Self-Consistent Molecular-Orbital Methods. IX. An Extended Gaussian-Type Basis for Molecular-Orbital Studies of Organic Molecules”. In: *The Journal of Chemical Physics* 54. (1971), 724–728. doi: [10.1063/1.1674902](https://doi.org/10.1063/1.1674902).
<http://scitation.aip.org/content/aip/journal/jcp/54/2/10.1063/1.1674902>.
- [47] T. Helgaker, P. Jorgensen, and J. Olsen. *Molecular electronic-structure theory*. Wiley, 2013.
- [48] T. Helgaker. *Quantum Chemistry and Molecular Properties: Molecular Integral Evaluation*. 2006.
http://folk.uio.no/helgaker/talks/SotrupIntegrals_06.pdf.
- [49] T. Helgaker. *Molecular Integral Evaluation*. 2010.
http://folk.uio.no/helgaker/talks/SotrupIntegrals_06.pdf.
- [50] L. E. McMurchie and E. R. Davidson. “One- and two-electron integrals over cartesian gaussian functions”. In: *Journal of Computational Physics* 26. (1978), 218. doi: [10.1016/0021-9991\(78\)90092-X](https://doi.org/10.1016/0021-9991(78)90092-X).
<http://www.sciencedirect.com/science/article/pii/002199917890092X>.
- [51] L. Raff, R. Komanduri, M. Hagan, and S. Bukkapatnam. *Neural Networks in Chemical Reaction Dynamics*. Oxford University Press, USA, 2012.
- [52] S. Nissen. *Implementation of a fast artificial neural network library (FANN)*. Tech. rep. 2003.
<http://leenissen.dk/fann/>.
- [53] R. Rojas. *Neural Networks: A Systematic Introduction*. Springer, 1996.
- [54] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals and Systems* 2. (1989), 303. doi: [10.1007/BF02551274](https://doi.org/10.1007/BF02551274).
- [55] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber. “A Novel Connectionist System for Unconstrained Handwriting Recognition”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 31. (2009), 855. doi: [10.1109/TPAMI.2008.137](https://doi.org/10.1109/TPAMI.2008.137).
- [56] MATLAB developers. *Divide Data for Optimal Neural Network Training*. MathWorks Documentation Center. 2014.
<http://www.mathworks.se/help/nnet/ug/divide-data-for-optimal-neural-network-training.html>.

- [57] YAML developers. *YAML specification*.
<http://www.yaml.org/>.
- [58] K. L. Schuchardt, B. T. Didier, T. Elsethagen, L. Sun, V. Gurumoorthi, J. Chase, J. Li, and T. L. Windus. “Basis Set Exchange: A Community Database for Computational Sciences”. In: *Journal of Chemical Information and Modeling* 47. (2007), 1045. doi: [10.1021/ci600510j](https://doi.org/10.1021/ci600510j).
- [59] *TURBOMOLE V6.5 2013, a development of University of Karlsruhe and Forschungszentrum Karlsruhe GmbH*. 1989-2007.
<http://www.turbomole.com>.
- [60] J. Maddock. *Boost.Regex Library Documentation*. 2010.
<http://www.boost.org/libs/regex>.
- [61] C. Sanderson. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*. Tech. rep. Australia: NICTA, 2010.
- [62] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. “LAPACK: A Portable Linear Algebra Library for High-performance Computers”. In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, 2.
<http://dl.acm.org/citation.cfm?id=110382.110385>.
- [63] P. Pulay. “Convergence acceleration of iterative sequences. the case of scf iteration”. In: *Chemical Physics Letters* 73. (1980), 393. doi: [10.1016/0009-2614\(80\)80396-4](https://doi.org/10.1016/0009-2614(80)80396-4).
- [64] W. Kolos and L. Wolniewicz. “Improved Theoretical Ground-State Energy of the Hydrogen Molecule”. In: *The Journal of Chemical Physics* 49. (1968), 404. doi: [10.1063/1.1669836](https://doi.org/10.1063/1.1669836).
- [65] P. Hariharan and J. Pople. “The influence of polarization functions on molecular orbital hydrogenation energies”. English. In: *Theoretica Chimica Acta* 28. (1973), 213. doi: [10.1007/BF00533485](https://doi.org/10.1007/BF00533485).
- [66] J. W. Moskowitz and M. Kalos. “A new look at correlations in atomic and molecular systems. I. Application of fermion Monte Carlo variational method”. In: *International Journal of Quantum Chemistry* 20. (1981), 1107.
- [67] C. Filippi and C. J. Umrigar. “Multiconfiguration wave functions for quantum Monte Carlo calculations of first-row diatomic molecules”. In: *The Journal of Chemical Physics* 105. (1996), 213. doi: [10.1063/1.471865](https://doi.org/10.1063/1.471865).
- [68] U. Flierler and D. Stalke. “More than Just Distances from Electron Density Studies”. English. In: *Electron Density and Chemical Bonding I*. Ed. by D. Stalke. Vol. 146. Structure and Bonding. Springer Berlin Heidelberg, 2012, 1. doi: [10.1007/430_2012_80](https://doi.org/10.1007/430_2012_80).
- [69] G. W. Stewart. “X-Ray Diffraction in Water: The Nature of Molecular Association”. In: *Physical Review* 37 (1 1931), 9. doi: [10.1103/PhysRev.37.9](https://doi.org/10.1103/PhysRev.37.9).

- [70] T. W. Martin and Z. S. Derewenda. “The name is bond – H bond”. In: *Nat Struct Mol Biol* 6. (1999). 10.1038/8195, 403. doi: [10.1038/8195](https://doi.org/10.1038/8195).
- [71] N. Sukumar. *A Matter of Density: Exploring the Electron Density Concept in the Chemical, Biological, and Materials Sciences*. Wiley, 2012.
- [72] L. M. Raff, M. Malshe, M. Hagan, D. I. Doughan, M. G. Rockley, and R. Komanduri. “Ab initio potential-energy surfaces for complex, multichannel systems using modified novelty sampling and feedforward neural networks”. In: *The Journal of Chemical Physics* 122., 084104 (2005), doi: [10.1063/1.1850458](https://doi.org/10.1063/1.1850458).
- [73] S. Nissen. “Large Scale Reinforcement Learning Using Q-SARSA (λ) and Cascading Neural Networks”. MA thesis. 2007.
- [74] FANNTool developers. *FANNTool*.
<https://code.google.com/p/fanntool/>.
- [75] D. Gregor and M. Troyer. *Boost.MPI Library Documentation*. 2007.
<http://www.boost.org/libs/mpi>.
- [76] F. Schwabl. “Equilibrium Ensembles”. In: *Statistical Mechanics*. Advanced Texts in Physics. Springer Berlin Heidelberg, 2006, 25. doi: [10.1007/3-540-36217-7_2](https://doi.org/10.1007/3-540-36217-7_2).
- [77] S. C. Harvey, R. K.-Z. Tan, and T. E. Cheatham. “The flying ice cube: velocity rescaling in molecular dynamics leads to violation of energy equipartition”. In: *Journal of Computational Chemistry* 19. (1998), 726. doi: [10.1002/\(SICI\)1096-987X\(199805\)19:7<726::AID-JCC4>3.0.CO;2-S](https://doi.org/10.1002/(SICI)1096-987X(199805)19:7<726::AID-JCC4>3.0.CO;2-S).
- [78] F. Ercolessi. *A molecular dynamics primer*. University of Udine, Italy, 1997.
<http://www.fisica.uniud.it/~ercolessi/md/md.html>.
- [79] S. Plimpton. “Fast Parallel Algorithms for Short-Range Molecular Dynamics”. In: *Journal of Computational Physics* 117. (1995), 1. doi: [10.1006/jcph.1995.1039](https://doi.org/10.1006/jcph.1995.1039).
<http://lammps.sandia.gov/>.
- [80] G. Baysinger. *CRC Handbook of Chemistry and Physics*. 2014.
- [81] V. Labet, P. Gonzalez-Morelos, R. Hoffmann, and N. W. Ashcroft. “A fresh look at dense hydrogen under pressure. I. An introduction to the problem, and an index probing equalization of H–H distances”. In: *The Journal of Chemical Physics* 136., 074501 (2012), doi: [10.1063/1.3679662](https://doi.org/10.1063/1.3679662).
- [82] E. Wiberg and N. Wiberg. *Inorganic Chemistry*. Academic Press, 2001.
<http://books.google.no/books?id=Mtth5g59dEIC>.
- [83] The HDF Group. *Hierarchical Data Format, version 5*. 1997-2014.
<http://www.hdfgroup.org/HDF5/>.
- [84] A. R. Fernandes. *Billboarding Tutorial*. 2013.
<http://www.lighthouse3d.com/opengl/billboarding>.
- [85] A. D. Becke and K. E. Edgecombe. “A simple measure of electron localization in atomic and molecular systems”. In: *The Journal of Chemical Physics* 92. (1990), 5397. doi: <http://dx.doi.org/10.1063/1.458517>.

- [86] Qt3D developers. *Qt3D Reference Documentation*. Qt Project. 2014. <http://doc-snapshot.qt-project.org/qt3d-1.0>.
- [87] L. S. Blackford et al. “An Updated Set of Basic Linear Algebra Subprograms (BLAS)”. In: *ACM Transactions on Mathematical Software* 28 (2001), 135. DOI: [10.1145/567806.567807](https://doi.org/10.1145/567806.567807).