Full Configuration Interaction Simulation of Quantum Dots

F. Berglyd Olsen

Thesis Submitted for the Degree of
Master of Science in Computational Physics

Department of Physics
University of Oslo

December 2012

"Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less."

– Marie Curie

"Our imagination is stretched to the utmost, not, as in fiction, to imagine things which are not really there, but just to comprehend those things which are there."

– Richard Feynman

Acknowledgements

This thesis marks the end of, so far at least, five and a half years of studies at the institute of physics at the University of Oslo. With a life long fascination for science, and love of nature itself, it has always been a dream to invest a few years of my life into studying one of the many interesting subjects in modern science. It wasn't anything in particular that made me choose physics over biology, or some of the other natural sciences, other than a suggestion by my girlfriend at the time, a geologist, to choose physics over biology for the job opportunities. That does not mean I haven't enjoyed studying physics, I certainly have; especially astrophysics and particle physics, but I could just as easily have chosen another field.

Nevertheless, here I am – five and a half years later. Moving to a new city, Oslo, back in 2007, where I knew no one, student organisations have been a very important part of my social life. I would never have enjoyed my time here like I did without them. I spent two and a half years as an active board member of *Fysikkforeningen*. Spending every day in our own private study hall. I got to meet many new people, and among other things, enjoy trips to Bergen and the Netherlands. Even more, *Realistforeningen* has been an important part of my social life; having been an active member for nearly five years now. I met some of my best friends there, or through people there. I want to thank both organisations for letting me be a part of their, respectively, 70+ and 150+ year history.

I also want to thank my thesis supervisor, professor Morten Hjorth-Jensen, for all the help and motivation as well as the occasional glass of whisky. I want to thank the Cyclotron lab at the Michigan State University for the weeks I spent there earlier this year coding the core of my project. These were the most productive weeks of this entire year. I want to thank Christoffer Hirth and Karl Leikanger for all our productive discussions about code implementation and code structure, and for their help and input when I have been stuck. Gustav Jansen and Andreas Ekström also deserves an extra mention and thanks for taking the time to discuss with me, and help me with, various coding troubles along the way. I also want to thank all the grad students at the Computational Physics department, and especially the three people I share office with: Jørgen Høgberget, Sarah Reimann and Mathilde Kamperud. I wish you all good luck with your own projects.

Last but not least I want to thank my friends Line Nateland and Julie Loen who have been there when I needed someone to talk to about life, the universe and everything else. You are both awesome.

F. Berglyd Olsen
Oslo, December 2012

Contents

Introduction **1**

I THEORY **3**

1 Quantum Mechanics **5**
1.1 The Basics . . . 6
1.1.1 The Wave Function . . . 6
1.1.2 The Postulates of Quantum Mechanics . . . 8
1.1.3 Operators and Observables . . . 9
1.1.4 Dirac Notation . . . 11
1.2 The Harmonic Oscillator . . . 12
1.2.1 Harmonic Oscillators in Quantum Mechanics . . . 13
1.2.2 The N-Dimensional Harmonic Oscillator . . . 15

2 Many-Body Theory **19**
2.1 The Many-Body Problem . . . 19
2.1.1 The General Many-Body Case . . . 19
2.1.2 The Identical Particle Case . . . 20
2.2 Second Quantisation . . . 21
2.2.1 Slater Determinants . . . 21
2.2.2 Creation and Annihilation Operators . . . 21
2.2.3 Wick's Theorem . . . 23
2.3 Quantum Dots . . . 24
2.3.1 A Hamiltonian for Quantum Dots . . . 24
2.3.2 A Hamiltonian Using Second Quantisation . . . 26
2.4 The Shell Model and the Effective Hamiltonian . . . 27
2.4.1 The Single-Particle Shell Modell . . . 27
2.4.2 The Effective Hamiltonian . . . 29

3 Many-Body Methods **31**
3.1 Variational Methods . . . 31
3.2 The Configuration Interaction Method . . . 32
3.3 The Coupled Cluster Method . . . 33
3.4 The Monte Carlo Methods . . . 34
3.4.1 Variational Monte Carlo . . . 34
3.4.2 Diffusion Monte Carlo . . . 35

II CODE DEVELOPMENT 37

4 Code and Hardware Setup 39
4.1 Hardware . . . 39
4.1.1 Single-Node Computers . . . 40
4.1.2 Multi-Node Computers . . . 41
4.2 Software . . . 41
4.2.1 Linear Algebra Libraries . . . 41
4.2.2 Parallelisation and Large Scale Usage . . . 43

5 Algorithms and Data Structure 45
5.1 The Lanczos Method . . . 45
5.1.1 Krylov Space . . . 45
5.1.2 Derivation . . . 46
5.1.3 Practical Implementatiom . . . 47
5.1.4 Ritz Eigenvectors . . . 48
5.1.5 Eigenvalues and Errors . . . 48
5.2 Computer Representation of the Shell Model . . . 49
5.2.1 Binary Slater Determinants . . . 49

6 The Code 51
6.1 Organisation of the Code . . . 53
6.2 The System Classes . . . 53
6.2.1 The Slater Class . . . 54
6.2.2 The Basis Class . . . 55
6.2.3 The Log Class . . . 60
6.2.4 The System Class . . . 60
6.3 The Potential Classes . . . 61
6.3.1 The Potential Super Class . . . 61
6.3.2 The QDot2D Class . . . 61
6.4 The Solver Classes . . . 64
6.4.1 The Diag Class . . . 65
6.4.2 The Lanczos Class . . . 65
6.5 Python Wrapper and Scripts . . . 67
6.5.1 The Job Generating Script . . . 67
6.5.2 The Sequential Run Script . . . 67
6.6 Code Usage Example . . . 67

7 Code Performance and Benchmarks 69
7.1 Convergence Properties of the Lanczos Algorithm . . . 69
7.1.1 The Lowest Eigenstate and Error Estimates . . . 69
7.1.2 Effective Interactions and Convergence . . . 72
7.2 Parallelisation and Scalability . . . 73
7.2.1 OpenMP Parallelisation . . . 73
7.2.2 OpenMPI Parallelisation . . . 75
7.3 Computation Times . . . 76

III RESULTS **81**

8 Results for Quantum Dots **83**
8.1 Effective Interactions and Energy Cut . . . 83
8.1.1 For 2 Electrons . . . 84
8.1.2 For 4 Electrons . . . 85
8.2 Comparison Runs . . . 86
8.2.1 Reproducing Published Results . . . 86
8.2.2 Additional Results . . . 88
8.3 Full Shell Results . . . 88
8.3.1 2 Electron Results . . . 89
8.3.2 6 Electron Results . . . 90
8.3.3 12 Electron Results . . . 91
8.3.4 20 Electron Results . . . 91
8.3.5 Convergence Rates . . . 92
8.4 Non Full Shell Results . . . 93
8.4.1 7 Electron Results . . . 93
8.4.2 11 and 13 Electron Results . . . 94

9 Additional Results for Quantum Dots **95**
9.1 Energy as a Function of Spin, M and ω . . . 95
9.1.1 Plots for 6 Electrons . . . 97
9.1.2 Plots for 7 Electrons . . . 98
9.1.3 Plots for 8 Electrons . . . 99
9.2 Coefficients . . . 100
9.2.1 Plots for 4 Electrons . . . 102
9.2.2 Plots for 5 Electrons . . . 103
9.2.3 Plots for 6 Electrons . . . 103
9.2.4 Plots for 7 Electrons . . . 104

Conclusion and Prospects **105**

IV APPENDICES **109**

A Additional Source Code **111**
A.1 Single-Node Usage Example of libTardis . . . 111
A.2 Multi-Node Usage Example of libTardis . . . 112

B Data Points and CPU Time for Plots **117**
B.1 CPU Time . . . 117
B.2 6 Electron Data . . . 118
B.2.1 Dimension of Basis . . . 118
B.2.2 For Omega = 0.01 . . . 119
B.2.3 For Omega = 0.1 . . . 120
B.2.4 For Omega = 1.0 . . . 121

B.2.5 For Omega = 5.0 . . . 122
B.3 7 Electron Data . . . 123
B.3.1 Dimension of Basis . . . 123
B.3.2 For Omega = 0.01 . . . 124
B.3.3 For Omega = 0.1 . . . 125
B.3.4 For Omega = 1.0 . . . 126
B.3.5 For Omega = 5.0 . . . 127
B.4 8 Electron Data . . . 128
B.4.1 Dimension of Basis . . . 128
B.4.2 For Omega = 0.01 . . . 129
B.4.3 For Omega = 0.1 . . . 130
B.4.4 For Omega = 1.0 . . . 131
B.4.5 For Omega = 5.0 . . . 132

Bibliography **133**

List of Tables **136**

List of Figures **139**

Introduction

One of the main goals of this master's project is to develop a flexible and efficient code for solving many-body problems in quantum mechanics. We will use the full configuration interaction method (FCI) to achieve this goal. It is essential that the code is structured in a modular way, ensuring flexibility and expandability. To achieve this, we will take advantage of the flexibility of the modern object oriented programming language `c++`.

The benefit of using an object oriented approach is that each class or object is an enclosed entity with an interface through which we pass commands. This ensures that the external usage of any given class does not depend on its internal structure, running the risk of "breaking the code" when making alterations. The only time this is a factor is when the interface itself is altered.

From the start it has also been a point to cooperate with other master students, especially Christoffer Hirth, in order to lay down a basic structure that could be a template for future many-body codes at our department at the University of Oslo.

The Subject of Study

The other goal of this project is to use the developed code to study quantum dots in two dimensions. A quantum dot is a piece of technology, often wrapped in a semiconductor, that is essentially an electron trap. There are several possible layouts of these, but we will be studying the single-potential two-dimensional quantum dot. This device can be approximated by a simple harmonic oscillator potential. One benefit of looking at such a system is that the code can relatively easily be expanded to three-dimensional harmonic oscillators and used for study of electrons in atoms. A quantum dot can in principle be described as an artificial atom. The quantum dot will be discussed in more detail in Chapter 2.

In order to lay out the theoretical foundation for studying quantum dots, we will go quickly through basic quantum mechanics and single-particle harmonic oscillators in Chapter 1, for then to take a look at how the theory can be expanded to include many-particle systems in Chapter 2. In Chapter 3 we will look at some of the numerical techniques commonly employed to solve many-particle problems, including the one we will be using in this project, namely the FCI method. In addition we will also introduce briefly the two other methods that have been used by others to produce the results that we will use as a basis of comparison for the results we produce and present in this thesis.

Programming and Code Structure

In order to reach the goal of building an efficient and flexible FCI many-body code, great care must be taken to ensure the algorithms and methods used are as efficient as possible. The main theory behind the core algorithm we will use, the Lanczos algorithm, is covered

in detail in Chapter 5, as well as the basic structure of how our many-body system is represented numerically on the computer. We will look at the software and hardware utilised during the development as well as the testing of the code in Chapter 4, and then go through the classes of the code in detail in Chapter 6. Since a great deal of attention has been paid to performance, a whole chapter will be dedicated to this, which is Chapter 7.

Producing Results

The goal is not only to write an efficient code, but also to use the code to do medium and large scale calculations for quantum dots. Chapter 8 contains several tables of both reproduced and new results. Chapter 9 contains some more preliminary results as we explore other ways to use the code developed during this project.

Part I

THEORY

1
Quantum Mechanics

"In fact, the mere act of opening the box will determine the state of the cat, although in this case there were three determinate states the cat could be in: these being Alive, Dead, and Bloody Furious."

– Terry Pratchett

Quantum mechanics, or *quantum physics*, is a branch of physics dealing with physical phenomena on the microscopic scale. Quantum mechanics departs from classical mechanics at the quantum realm of atomic and subatomic length scales. Quantum mechanics provides a mathematical framework for the dual particle-like and wave-like behavior and interactions of energy and matter.

A Brief Historical Overview

Already in the 17th and 18th century physicists started to study the quantum world, they just did not realise it yet. The wave-like nature of light was investigated, and a key experiment of this time was the double-slit experiment performed by Thomas Young in 1803, later published under the title "On the nature of light and colours". This experiment, where a light pattern was projected onto a board behind another board with two narrow slits, was essential to establish the early wave-theory of light.

Then, a series of experiments in the mid to late 19th century led many physicists to theorise that energy states of physical systems could in fact be discrete. Both Ludwig Boltzman (1877) and Max Planck (1900) hypothesised this. Experiments and developing theories about black-body radiation by Wilhelm Wien and Max Planck led to the law known as Planck's Law, which again led to the early stages of the development of quantum theory.

Scientists like Arthur Compton, C. V. Raman, Pieter Zeeman, Albert Einstein and Niels Bohr were all involved in the early development of quantum theory, and many of them have their research subject named after them. They were not alone, the list of, now, famous names is long.

A few years later, Max Planck suggested that the energy of light may be proportional to its frequency:

$$E = h\nu \tag{1.1}$$

where h is known as Planck's constant. Planck insisted that this was an aspect of radiation, not actual physical reality, however Einstein showed in his paper on the photoelectric effect, which he received a Nobel prize for, that these energy spectra were indeed quantised.

Thus quantum mechanics was born, and the first half of the 20th century went to establish its main framework. Its implications were much wider than the realm of physics though. It also became a philosophical question about how deterministic nature really is and whether the order of the world we observe in day-to-day life is merely an illusion, a shadow of the weirdness of the quantum world.[1]

1.1 The Basics

In classical Newtonian mechanics, objects have well-defined observables. We generally are able to measure their properties, and predict their behaviour. Classical mechanics is in principle deterministic. Quantum mechanics, on the other hand, is more fuzzy. Heisenberg's uncertainty principle tells us that you may for instance not know a particle's position and velocity at the same time; nor can we measure or observe an individual particle without disturbing it or altering its state. The act of measurement itself implies an interaction with at least one other fundamental particle. It is like measuring the velocity of a car by crashing another car into it in order to see what happens.

Instead, quantum mechanics uses a formalism based on statistical probabilities; or *probabilistic determinism*. We will now take a brief look at the central concepts of one-particle quantum mechanics. The simplest way we can look at it.

1.1.1 The Wave Function

Werner Heisenberg was the first to replicate the observed quantisation of atomic spectra with his matrix mechanics. The same year Erwin Schrödinger created his wave mechanics. Schrödinger's approach of wave mechanics led to a differential equation, something that was familiar and considered easier to work with. It turned out that these two representations were equivalent.

Early quantum mechanics started from known classical mechanics, and from there, similarities were explored.

In that spirit, let us imagine an object of mass m moving along a single axis x. In classical mechanics its position changes over time as $x(t)$. Once we know its position, we can work out its velocity $v = dx/dt$. We can also work out its momentum $p = mv$ and kinetic energy $T = mv^2/2$. Applying Newton's second law, $F = ma$, we can describe the force as the derivative of a potential energy $F = -\partial V/\partial x$, and we thus get [2]:

$$m\frac{d^2x}{dt^2} = -\frac{\partial V}{\partial x}. \tag{1.2}$$

In quantum mechanics, however, we approach these things differently.

In quantum mechanics we are looking at a particle's *wave function*, $\Psi(x,t)$, instead.

[1] See Brehm and Mullin [1] for more reading material about the history of quantum mechanics.

The wave function is a probabilistic representation of a particle's properties and how they evolve over time. Its time evolution is contained in the Schrödinger equation,

$$i\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + \hat{V}\Psi. \tag{1.3}$$

The wave function itself is a function in complex space, but its square, $|\Psi|^2$, is a real quantity that tells us something about the probability of finding a particle in a given state at a given point in time. Say we want to know how likely it is that the particle exists in an area between a and b, at a time t. The probability of this is then given as

$$P_a^b(t) = \int_a^b |\Psi(x,t)|^2 \, dx. \tag{1.4}$$

For this to represent an actual probability, in other words a value between 0 and 1, we require that the wave function itself is normalised, or that it satisfies the equation

$$\int_{-\infty}^{\infty} |\Psi(x,t)|^2 \, dx = 1. \tag{1.5}$$

Normalisation is achieved by solving equation (1.5) and then dividing the wave function by the square root of the constant satisfying the relation.

But let us go back to the Schrödinger equation (1.3). Analogous as it is to classical mechanics, the Schrödinger equation also has an energy equation that can be written as

$$\hat{H} = \hat{T} + \hat{V}, \tag{1.6}$$

where the operator $\hat{T}$ is the kinetic energy given by

$$\hat{T} = \frac{p^2}{2m} = -\frac{\hbar^2}{2m}\nabla^2, \tag{1.7}$$

where we take the momentum, p, to be the momentum operator $\hat{p} = -i\hbar\nabla$, $\hat{V}$ is the potential energy and $\hat{H}$ is our Hamilton operator. We will get back to what an operator is in a little while.

Inserting equation (1.6) into equation (1.3), we get the simplified Schrödinger equation

$$i\hbar\frac{\partial\Psi}{\partial t} = \hat{H}. \tag{1.8}$$

If we assume that the potential $\hat{V}$ is time-independent, as would be our first approach, the Schrödinger equation can be solved with separation of variables, meaning that

$$\Psi(x,t) = \psi(x)\phi(t). \tag{1.9}$$

Notice that we have split the wave function, $\Psi(x,t)$, into two functions, $\psi(x)$ and $\phi(t)$, that are decoupled.

For such a separable solution we have the following:

$$\frac{\partial \Psi}{\partial t} = \psi \frac{d\phi}{dt}, \quad \frac{\partial^2 \Psi}{\partial x^2} = \frac{d^2\psi}{dx^2}\phi. \tag{1.10}$$

If we insert these relations back into the Schrödinger equation, (1.3), we get that

$$i\hbar \frac{1}{\phi}\frac{d\phi}{dt} = -\frac{\hbar^2}{2m}\frac{1}{\psi}\frac{d^2\psi}{dx^2} + \hat{V}. \tag{1.11}$$

As we now observe, the left side is a function of t and the right side is a function of x. This is only possible if both sides are constant.

Let us then define a constant E

$$i\hbar \frac{1}{\phi}\frac{d\phi}{dt} = E, \tag{1.12}$$

which we then substitute into (1.11) gives

$$-\frac{\hbar^2}{2m}\frac{1}{\psi}\frac{d^2\psi}{dx^2} + \hat{V} = E, \tag{1.13}$$

for then to multiply with our spatial wave function, ψ, so that we arrive at

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V\psi = E\psi, \tag{1.14}$$

which, if E is the energy, leads us to the equation

$$\hat{H}\psi = E\psi, \tag{1.15}$$

or what is known as the *time-independent Schrödinger equation.*

The wave function itself can be expanded as a linear expansion of possible solutions to the Schrödinger equation, and then take the form:

$$\Psi(x,t) = \sum_i c_i \psi_i(x)\phi_i(t), \tag{1.16}$$

yielding a set of solutions, $(\psi_1(x), \psi_2(x), \psi_3(x), \cdots)$ and $(E_0, E_1, E_2, \cdots)$. Any solution can be written as such an expansion where the specific solution is determined by the coefficients c_i [2].

1.1.2 The Postulates of Quantum Mechanics

A physical system can usually be described by its states and its observables, as well as its time-evolution or dynamics. A classical physical system can be described in a phase-space model. However in quantum mechanics our systems are described in what we call *Hilbert space*. We will look at what a Hilbert space is a little later.

First, let us consider the four basic postulates of quantum mechanics.[2] The postulates of quantum mechanics are a summary of its mathematical framework.

[2]This section is a brief summary of chapter 3 in Liboff [3].

Postulate I

This postulate[3] states the following: To any well-defined observable in physics, A, there corresponds an operator, $\hat{A}$ such that measurement of A yields values a, which are eigenvalues of $\hat{A}$. That is, the values a are those values for which the equation

$$\hat{A}\psi = a\psi \tag{1.17}$$

has a solution ψ. The function ψ is thus the eigenfunction of $\hat{A}$ [3].

Postulate II

The second postulate of quantum mechanics is: Measurement of the observable A that yields the value a, leaves the system in the state ψ_a, where ψ_a is the eigenfunction of $\hat{A}$ that corresponds to the eigenvalue a [3].

Postulate III

The third postulate of quantum mechanics establishes the existence of the state function and its relevance to the properties of a system: The state of a system at any instant of time may be represented by a state or wave function Ψ which is continuous and differentiable. All information regarding the state of the system is contained in the wave function. Specifically, if a system is in the state $\Psi(\mathbf{r}, t)$, the average of any physical observable C relevant to that system at time t is

$$\langle C \rangle = \int \Psi^* \hat{C} \Psi \, d\mathbf{r}. \tag{1.18}$$

The average $\langle C \rangle$ is called the *expectation value* of $\hat{C}$. [3]

Postulate IV

The fourth postulate of quantum mechanics specifies the time development of the state function $\Psi(\mathbf{r}, t)$, the state function for a system develops in time according to the equation

$$i\hbar \frac{\partial}{\partial t} \Psi(\mathbf{r}, t) = \hat{H} \Psi(\mathbf{r}, t). \tag{1.19}$$

This equation is called the *time-dependent Schrödinger equation* [3].

1.1.3 Operators and Observables

Now that we have introduced the basic framework, we can look into how to make use of these.

In quantum mechanics observables are represented by operators. In this thesis operators are denoted with a *hat*, $\hat{O}$. An observable is a real quantity. Because of this, its

[3] There is no standardised order of these postulates.

expectation value must also be a real quantity. The expectation value of an operator $\hat{O}$ is given as

$$\langle \hat{O} \rangle = \int \Psi^* \hat{O} \Psi \, dx. \tag{1.20}$$

The notation $\langle \hat{O} \rangle$ refers back to Postulate III.

Let us then look at a few of the most common operators we encounter in quantum mechanics.

The Position and Momentum Operators

The position operator, $\hat{x}$, is simply represented by

$$\hat{x} = x. \tag{1.21}$$

It is the simplest of our operators.

The momentum operator, $\hat{p}$, on the other hand, is a little more complicated and is given as

$$\hat{p} = -i\hbar\nabla. \tag{1.22}$$

The factor $-i\hbar$ arises from the commutation relation between the the position and the momentum operator

$$[\hat{x}, \hat{p}] = i\hbar. \tag{1.23}$$

Energy Operators

The operator corresponding to the energy is the Hamiltonian. If we replace the momentum p with its operator $\hat{p}$, the Hamiltonian for a single particle with mass m in a potential $\hat{V}$ is given as

$$\hat{H} = \frac{\hat{p}^2}{2m} + \hat{V}(\mathbf{r}) = -\frac{\hbar^2}{2m}\nabla^2 + \hat{V}(\mathbf{r}). \tag{1.24}$$

The Hamiltonian has the eigenvalue equation

$$\hat{H}\psi(\mathbf{r}) = E\psi(\mathbf{r}), \tag{1.25}$$

where E, the energy, is our eigenvalue.

From this we can derive other operators, like for instance kinetic energy,

$$\hat{T} = -\frac{\hbar^2}{2m}\nabla^2, \tag{1.26}$$

which we already have introduced in (1.7).

Hermitian Operators

That an operator in quantum mechanics is real, means that

$$\langle \hat{O} \rangle = \langle \hat{O} \rangle^*, \tag{1.27}$$

and therefore all the following representations of the expectation value equation are equivalent:

$$\int \Psi^* \hat{O} \Psi \, dx = \left[\int \Psi^* \hat{O} \Psi \, dx \right]^* = \int \left[\hat{O} \Psi \right]^* \Psi \, dx. \tag{1.28}$$

All operators for observables are required to behave in this manner when acting on a wave function Ψ in Hilbert space. Such operators are referred to as *hermittian* or *self-adjoint*.

Hilbert Space

Hilbert space, named after David Hilbert, extends the notation of two and three dimensional Euclidean and Cartesian space, that we know from classical mechanics, into a generalised finite or infinite dimensional space. Hilbert space is an abstract vector space. A Hilbert space must be complete, meaning it must have enough limits that it allows for calculations producing finite results.

There is a lot of equivalent notation between classical and quantum mechanics to describe such functions and how they behave. In the same way that in a Cartesian space, a vector

$$\mathbf{V} = \mathbf{e}_x x + \mathbf{e}_y y + \mathbf{e}_z z \tag{1.29}$$

is spanned by the basis of unit vectors $\mathbf{e}_n$, a Hilbert space is spanned by functions of the form

$$\psi(x) = \sum_{n=1}^{\infty} c_n \psi_{n(x)}. \tag{1.30}$$

Our wave function is such a function, and it lives in Hilbert space.

1.1.4 Dirac Notation

In quantum mechanics we rarely write out the functions that live in Hilbert space. Instead we use a simplified notation called *bra* and *ket* notation. The notation was introduced by Dirac in 1939. The words "bra" and "ket" are derived from the English word "bracket". This is commonly referred to as *Dirac notation*.

$$\langle \Psi \mid \Psi \rangle \quad \text{Dirac Notation.} \tag{1.31}$$

In this notation, the bra is a row vector

$$\langle \Psi | = [c_0^*, c_1^*, c_2^*, \cdots], \tag{1.32}$$

and the ket is a column vector

$$|\Psi\rangle = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \end{bmatrix}. \tag{1.33}$$

Since these functions often span an infinite-dimensional space, this notation is useful and compact.

Even if these functions can be infinite in their expansion, they still need to be normalisable. In other words the inner product needs to be finite, meaning

$$\langle \Psi_a \mid \Psi_b \rangle = \int_{-\infty}^{\infty} \Psi_a^* \Psi_b \, dx < \infty. \tag{1.34}$$

From this it follows that

$$\langle \Psi_a \mid \Psi_a \rangle = 1, \tag{1.35}$$

when we are assuming normalised wave functions.

This gives us the following relations

$$\langle \Psi | \hat{O} | \Psi \rangle = \langle \hat{O} \rangle \tag{1.36}$$

$$\langle \Psi | \hat{O} | \Psi \rangle = \langle \Psi | \hat{O}^\dagger | \Psi \rangle \qquad \text{if } \hat{O} \text{ is Hermitian} \tag{1.37}$$

$$\langle \Psi | c | \Psi \rangle = c \qquad \text{if } c \text{ is a constant.} \tag{1.38}$$

Dirac notation is the standardised notation of quantum mechanics, and is used throughout this thesis. Generally a basis of wave functions is denoted with a capital case Greek "psi", $|\Psi\rangle$, and its elements with a lower case "psi", $|\psi\rangle$. Sometimes the Greek letter "phi" is used too, but we will get back to that later when we discuss Slater determinants in 2.2.1.

1.2 The Harmonic Oscillator

In classical mechanics, a harmonic oscillator is a system where a displaced mass m experience a restoring force. The common example is a weight attached to a spring that exerts a constant force k on the mass, for instance a pendulum. The motion of such a system is governed by *Hooke's law*

$$F = -kx = m\frac{d^2x}{dt^2}. \tag{1.39}$$

Such a system has a solution which time development is described by a periodic function

$$x(t) = A \sin \omega t + B \cos \omega t. \tag{1.40}$$

The periodicity of the function, ω, is the oscillator frequency and is given as

$$\omega = \sqrt{\frac{k}{m}}. \tag{1.41}$$

The system's potential energy is given as

$$V(x) = \frac{1}{2}kx^2 \tag{1.42}$$

1.2.1 Harmonic Oscillators in Quantum Mechanics

The classical harmonic oscillator has an equivalent in quantum mechanics. A system that has a stable equilibrium point can be approximated by a harmonic oscillator potential in the vicinity of this equilibrium point.

The quantum mechanical equivalent harmonic oscillator potential is, in the one-dimensional case, given as

$$\hat{V}(x) = \frac{1}{2}m\omega^2\hat{x}^2. \tag{1.43}$$

We can take the Hamiltonian of a system with the aforementioned properties and approximate it by inserting the harmonic oscillator potential equation, giving us

$$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + \frac{1}{2}m\omega^2\hat{x}^2, \tag{1.44}$$

where the first term is the kinetic energy and the second term is our harmonic oscillator potential.

If we now take our time-independent Schrödinger equation from Postulate IV (1.19) on the eigenvalue form, and apply our harmonic oscillator approximation of the Hamiltonian, we get

$$\hat{H}\left|\psi\right\rangle = E\left|\psi\right\rangle, \tag{1.45}$$

where E is the unknown time-independent energy eigenstate. This is in fact the basics of what this entire thesis is doing. We are solving this equation for complex systems in order to determine the quantity E. The system's energy states.

Solving the Harmonic Oscillator Equation

The equation (1.45) is a differential equation, and one common method of solving such equations is to use a *spectral method*. A spectral method is a numerical way to solve such equations by expanding them into a basis of simpler functions. An example of such an expansion is a Fourier expansion which can be a series of cosine and sine functions. This expansion is in fact a very common one in physics.

Our wave function $|\psi\rangle$ can be expanded in position space as

$$\psi_n(\xi) = \frac{1}{\sqrt{2^n n!}} \sqrt[4]{\frac{m\omega}{\pi\hbar}} e^{-\frac{1}{2}\xi^2} H_n(\xi), \tag{1.46}$$

where H_n are the Hermite polynomials

$$H_n(y) = (-1)^n e^{y^2} \frac{d^n}{dy^n} e^{-y^2}, \tag{1.47}$$

and the quantity ξ is defined as

$$\xi = \sqrt{\frac{m\omega}{\hbar}} x, \tag{1.48}$$

and $n = 0, 1, 2, \cdots$ is the principal quantum number.

The eigenvalues, or energies E_n, corresponding to these functions are given by

$$E_n = \hbar\omega \left(n + \frac{1}{2} \right), \tag{1.49}$$

where, again, n is our principal quantum number.

This is the one-particle harmonic oscillator energy spectrum. There are three noteworthy points to this solution.

1. The energies are quantised. Meaning there is not a continuous spectrum of energies, but an infinite set of discrete values.

2. These discrete values are equally spaced, increasing with a factor $\frac{1}{2}\hbar\omega$ for each increasing value of n.

3. The lowest energy state, E_0, is not 0, but $\frac{1}{2}\hbar\omega$. In other words there exist a ground state energy that is non-zero, which is the lowest possible energy state of the system.

Ladder Operators

The properties of the one-dimensional harmonic oscillator potential lets us define another simplified notation, namely the *ladder operators*. It is a much simpler way to get our energy states than the method leading up to (1.49). This method is credited to Paul Dirac, and consists of two operators $\hat{a}$, and its adjoint $\hat{a}^\dagger$, defined as

$$\hat{a} = \sqrt{\frac{m\omega}{2\hbar}} \left(\hat{x} + \frac{i}{m\omega} \hat{p} \right) \tag{1.50}$$

$$\hat{a}^\dagger = \sqrt{\frac{m\omega}{2\hbar}} \left(\hat{x} - \frac{i}{m\omega} \hat{p} \right). \tag{1.51}$$

From this we can rewrite the momentum operator $\hat{x}$ and the position operator $\hat{p}$ as

$$\hat{x} = \sqrt{\frac{\hbar}{2m\omega}}(\hat{a} + \hat{a}^\dagger) \tag{1.52}$$

$$\hat{p} = i\sqrt{\frac{\hbar m\omega}{2}}(\hat{a}^\dagger - \hat{a}). \tag{1.53}$$

Note that the ladder operators are not Hermittian as $\hat{a} \neq \hat{a}^\dagger$.

We can also define a number operator

$$\hat{N} = \hat{a}^\dagger\hat{a}, \qquad \hat{N}\left|\psi\right\rangle = n\left|\psi\right\rangle, \tag{1.54}$$

where now n represents an integer eigenvalue.

These equations have the commutator relations

$$[\hat{a}, \hat{a}^\dagger] = 1, \qquad [\hat{N}, \hat{a}^\dagger] = \hat{a}^\dagger, \qquad [\hat{N}, \hat{a}] = -\hat{a}. \tag{1.55}$$

We can then then express the Hamilton operator as

$$\hat{H} = \left(\hat{N} + \frac{1}{2}\right)\hbar\omega, \tag{1.56}$$

so that the eigenstates of $\hat{N}$ are also the eigenstates of energy.

This simplifies the notation considerably as we can now apply the operators to our wave function directly

$$\hat{N}\hat{a}^\dagger\left|\psi\right\rangle = (n+1)\hat{a}^\dagger\left|\psi\right\rangle \tag{1.57}$$

$$\hat{N}\hat{a}\left|\psi\right\rangle = (n-1)\hat{a}\left|\psi\right\rangle. \tag{1.58}$$

Here $\hat{a}$ acts on $|n\rangle$ to produce a new state $|n-1\rangle$ and $\hat{a}^\dagger$ acts on $|n\rangle$ to produce a new state $|n+1\rangle$. Because of this $\hat{a}^\dagger$ is often referred to as a *rising operator* and $\hat{a}$ a *lowering operator*. This notation is also used a lot in quantum field theory (QFT) where $\hat{a}^\dagger$ is called the creation operator and $\hat{a}$ the annihilation operator, where we can for example create or annihilate particles in a complex field. Also in our representation, moving a particle can be interpreted as an annihilation event in one quantum state, followed by a creation event in another.

We will get back to this point in the next chapter when we discuss second quantisation.

1.2.2 The N-Dimensional Harmonic Oscillator

So far we have been looking at a single particle trapped in a one-dimensional harmonic oscillator potential. As we very well know, the real world consists of (at least you may say) three physical dimensions. So does the quantum world. In principal it is relatively straight

forward to generalise the one-dimensional harmonic oscillator into $N_D = 1, 2, 3, \cdots$ dimensions:

$$\hat{H} = \sum_{i=1}^{N_D} \left(\frac{\hat{p}_i^2}{2m} + \frac{1}{2} m\omega^2 \hat{x}_i^2 \right). \tag{1.59}$$

The N_D-dimensional position and momentum operators have the following commutation relations:

$$[\hat{x}_i, \hat{p}_j] = i\hbar\delta_{ij}, \qquad [\hat{x}_i, \hat{x}_j] = 0, \qquad [\hat{p}_i, \hat{p}_j] = 0, \tag{1.60}$$

where the indices i and j run from 1 to N_D.

As we would expect from this simple expansion of the one-dimensional problem, the energy levels of such a system is

$$E = \hbar\omega \left[(n_1 + n_2 + \cdots + n_{N_D}) + \frac{N_D}{2} \right] \tag{1.61}$$

where n_i are again our principal quantum numbers.

The Three-Dimensional Spherical Symmetrical Case

Let us now take a look at the more common three-dimensional system.

The Schrödinger equation for the three-dimensional harmonic oscillator can be also solved by separation of variables. The spherical symmetrical potential for such a system is

$$\hat{V}(r) = \frac{1}{2}\mu\omega^2 \mathbf{r}^2. \tag{1.62}$$

Here the mass m used earlier has been replaced with μ to avoid confusion with the quantum number m that we will discuss shortly.

The solution to the Schrödinger equation for this potential is rather long, so we are just going to state it.

$$\psi_{nlm}(r, \theta, \phi) = A_{nl} r^l e^{-\nu r^2} L_n^{l+\frac{1}{2}}(2\nu r^2) Y_{lm}(\theta, \phi), \tag{1.63}$$

where

$$A_{nl} \quad \text{is a normalisation constant,} \tag{1.64}$$

$$\nu \equiv \frac{\mu\omega}{2\hbar}, \tag{1.65}$$

$$L_n^{l+\frac{1}{2}}(2\nu r^2) \quad \text{are generalised Laguerre polynomials, and} \tag{1.66}$$

$$Y_{lm}(\theta, \phi) \quad \text{is a spherical harmonic function.} \tag{1.67}$$

The energy eigenstates are then

$$E = \hbar\omega \left(2n + l + \frac{3}{2} \right). \tag{1.68}$$

As before $n = 1, 2, \cdots$ is the principal quantum number. The angular momentum quantum number $l = 0, 1, 2, \cdots, n-1$ and the magnetic quantum number $-l \leq m \leq l$.

The Circular Harmonic Oscillator

In this thesis we will mainly look at the two-dimensional case, or the circular harmonic oscillator potential. It is simpler than the three-dimensional case, naturally, but the potential is defined in the same way as we can surmise from equation (1.59). It reads:

$$\hat{V}(r) = \frac{1}{2}\mu\omega^2\mathbf{r}^2 = \frac{1}{2}\mu\omega^2(\hat{x}^2 + \hat{y}^2). \tag{1.69}$$

Also, as we can see from equation (1.60), the eigenstates of this potential are simply [3]

$$E = \hbar\omega\left(n_x + n_y + 1\right). \tag{1.70}$$

In this thesis we will be looking at a circular harmonic oscillator potential. The wave function can then be expressed on radial form, related to the spherical form for three dimensions:

$$\psi_{nm_l}(r,\phi) = Ae^{im_l\phi}r^{|m_l|}e^{-\nu r^2}L_n^{|m_l|}(2\nu r^2), \tag{1.71}$$

where A is still the normalisation constant and ν is defined as in Equation (1.65) [4].

Lastly, the eigenstates of the two-dimensional harmonic oscillator are given as

$$E = \hbar\omega\left(2n + |m_l| + 1\right), \tag{1.72}$$

where the quantum number m_l is now a projection of l onto m. Although $m_l \neq m$, we will in the future refer to m_l as m when we talk about two-dimensional harmonic oscillators.

We will revisit this potential in the next chapter when we discuss actual systems that can be approximated by these equations.

2
Many-Body Theory

"Had I known that we were not going to get rid of this damned quantum jumping, I never would have involved myself in this business!"

– Erwin Schrödinger

In single-particle quantum mechanics we deal with systems of only one particle. It is of course a simple place to start when we want to understand how these systems behave, but in reality the world is full of particles interacting all the time. The real systems of many particles are called *many-body* systems, and our simple formalism cannot deal with such complicated systems.

This chapter will give a brief introduction to the main concepts behind many-body theory that are relevant for this thesis, as well as the formalism used in many-body theory in quantum mechanics. We will also look at the physical system that we are simulating, namely *quantum dots*.

2.1 The Many-Body Problem

First of all, we will take a quick look at how we can build a theory of many-body systems based on what we know of one-body systems.

2.1.1 The General Many-Body Case

Let us consider a system of N_p particles. The properties of this system is given by the Schrödinger equation as we have already seen it in Chapter 1. It takes the familiar form

$$i\hbar\frac{\partial}{\partial t}\left|\Psi\right\rangle = \hat{H}\left|\Psi\right\rangle, \tag{2.1}$$

where $\hat{H}$ is the Hamiltonian of the system we are studying.

The Hamiltonian can be split up into its components, as introduced in Section 1.1.1:

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

where $\hat{T}$ is the total kinetic energy operator for the whole N_P-body system, given as

$$\hat{T} = \sum_{i=1}^{N_P} \hat{t}_i, \tag{2.3}$$

where $\hat{t}_i$ is the kinetic energy of particle i. The operator $\hat{T}$ is just a sum of one-body operators.

The operator $\hat{V}$ is the total potential energy operator, given as

$$\hat{V} = \sum_{i=1}^{N_P} \hat{V}_i. \tag{2.4}$$

This, however, is not just a sum of one-body operators $\hat{v}_i$. The expansion of the total potential operator is on a generalised form

$$\hat{V} = \hat{V}_1 + \hat{V}_2 + \cdots + \hat{V}_{N_P} \tag{2.5}$$

$$= \sum_{k=1}^{N_P} \hat{v}_k^{(1)} + \frac{1}{2!} \sum_{kl=1}^{N_P} \hat{v}_{kl}^{(2)} + \cdots + \frac{1}{N_P!} \sum_{kl\cdots=1}^{N_P} \hat{v}_{kl\ldots}^{(N_P)}. \tag{2.6}$$

2.1.2 The Identical Particle Case

In quantum mechanics, particles of the same species, for example electrons, are identical and inseparable. This means they are *indistinguishable in principle* from one another [2]. Identical particles are only defined by which state they occupy. The consequence of this is that the expectation value of a system will have to be the same if we exchange two particles in coordination space

$$|\Psi_\alpha(r_i, r_j)|^2 = |\Psi_\alpha(r_j, r_i)|^2. \tag{2.7}$$

The possible solutions to this equation is

$$\Psi_\alpha(r_i, r_j) = \pm\Psi_\alpha(r_j, r_i), \tag{2.8}$$

meaning we have a positive and a negative solution. These represent the symmetric and anti-symmetric solution, respectively [5].

Let us look at a two particle wave function

$$\psi_\pm(\mathbf{r}_1, \mathbf{r}_2) = A[\psi_a(\mathbf{r}_1)\psi_b(\mathbf{r}_2) \pm \psi_b(\mathbf{r}_1)\psi_a(\mathbf{r}_2)]. \tag{2.9}$$

It is then obvious that in the anti-symmetric case, if two particles are identical, which is to say that $\psi_a = \psi_b$, this cannot be allowed as

$$\psi_-(\mathbf{r}_1, \mathbf{r}_2) = A[\psi_a(\mathbf{r}_1)\psi_a(\mathbf{r}_2) - \psi_a(\mathbf{r}_1)\psi_a(\mathbf{r}_2)] = 0. \tag{2.10}$$

This is simply not a state that exists. Its probability is thus 0. This is the result of the well known *Pauli exclusion principle*.

In quantum physics and particle physics we distinguish between *fermions* and *bosons*. Bosons have integer spins, like 0, 1 and 2; fermions have half integer spins like $\frac{1}{2}$ and $\frac{3}{2}$. Bosons, like photons and gluons, have symmetric wave functions; while fermions, like electrons and protons, have anti-symmetric wave functions. Thus the Pauli exclusion principle applies to electrons, which are the main focus of this thesis.

2.2 Second Quantisation

When we want to work with many-body systems, we soon realise that these are relatively complex to handle mathematically. As is customary in physics, we attempt to define a simpler representation for our complex problem. *Second quantisation* is a method for representing independent-particle-model wave functions (Slater determinants) and operators in a compact and convenient way. It also provides an efficient way of manipulating such functions and operators [6].

2.2.1 Slater Determinants

Most electronic structure calculations begin with a relatively simple approximation based on the independent-particle model. The wave function for such a model is a single Slater determinant,

$$\Phi = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_1(1) & \psi_2(1) & \cdots & \psi_N(1) \\ \psi_1(2) & \psi_2(2) & \cdots & \psi_N(2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(N) & \psi_2(N) & \cdots & \psi_N(N) \end{vmatrix}, \tag{2.11}$$

where $\psi_i(\mu)$ is a single-particle state, a function of the space and spin coordinates of the μth electron [6].

The product of these functions, that is $\psi_1(1)\psi_2(2)\cdots\psi_N(N)$, gives an approximate solution to the Schrödinger equation, but it is not anti-symmetric, and it does not satisfy the Pauli exclusion principle for fermions. However, if we exchange any two electrons, $\psi_1(2)\psi_2(1)\cdots\psi_N(N)$, we still have an approximate solution, but if we change the place of two rows in a matrix, the determinant of that matrix changes sign. This property of determinants can thus be exploited. It can be shown that it is the only anti-symmetric combination of these functions.

Further, if we place two electrons in the same quantum state, the corresponding rows of the determinant will be identical, and the determinant thus vanishes [7].

Our Slater determinant is then defined by a set of single-particle states

$$\Phi = |\phi_1\phi_2\cdots\phi_N\rangle\,, \tag{2.12}$$

where each ϕ_i represents a set of quantum numbers n, m_l and s.

2.2.2 Creation and Annihilation Operators

The Slater determinant, Φ, is represented in *second-quantised form* by specifying the *occupational numbers* $n_1, n_2, \cdots, n_N$ of the basis states $\phi_1, \phi_2, \cdots, \phi_N$ in the determinant [6].

The occupational numbers are thus defined as

$$n_i(\Phi) = \begin{cases} 0 & \text{if } \phi_i \text{ is not present} \\ 1 & \text{if } \phi_i \text{ is present} \end{cases}. \tag{2.13}$$

Operations on a Slater determinant can then be represented with *creation* and *annihilation* operators, represented by $\hat{a}^\dagger$ and $\hat{a}$ respectively. This is analogous to the way we defined these operators and the occupational numbers in 1.2.

The outcomes of the four possibilities of these two operators acting on a Slater determinant are

$$\hat{a}_i^\dagger |\Phi\rangle \rightarrow n_i(\Phi) = 1 \qquad \text{if } n_i(\Phi) = 0 \tag{2.14}$$
$$\hat{a}_i^\dagger |\Phi\rangle = 0 \qquad \text{if } n_i(\Phi) = 1 \tag{2.15}$$
$$\hat{a}_i |\Phi\rangle = 0 \qquad \text{if } n_i(\Phi) = 0 \tag{2.16}$$
$$\hat{a}_i |\Phi\rangle \rightarrow n_i(\Phi) = 0 \qquad \text{if } n_i(\Phi) = 1. \tag{2.17}$$

The conditions that result in a 0 of course represent an operation that is not allowed or does not exist.

Commutator Relations

A commutator is the mathematical relation between two non-scalar objects

$$[\hat{A}, \hat{B}] = \hat{A}\hat{B} - \hat{B}\hat{A}. \tag{2.18}$$

If this is 0, we say $\hat{A}$ and $\hat{B}$ commute.

The anti-commutator relation is the opposite. It is defined as

$$\{\hat{A}, \hat{B}\} = \hat{A}\hat{B} + \hat{B}\hat{A}, \tag{2.19}$$

and is 0 if $\hat{A}$ and $\hat{B}$ anti-commute.

The creation and annihilation operators anti-commute, which is to say

$$\hat{a}_p^\dagger \hat{a}_q^\dagger = -\hat{a}_q^\dagger \hat{a}_p^\dagger \tag{2.20}$$
$$\hat{a}_p \hat{a}_q = -\hat{a}_q \hat{a}_p. \tag{2.21}$$

We also have the relations

$$\{\hat{a}_p, \hat{a}_q\} = \{\hat{a}_p^\dagger, \hat{a}_q^\dagger\} = 0 \tag{2.22}$$
$$\{\hat{a}_p^\dagger, \hat{a}_q\} = \delta_{pq}. \tag{2.23}$$

Let us consider, then, $\hat{a}_p^\dagger$ and $\hat{a}_q$:

$$\hat{a}_p^\dagger \hat{a}_q |ijk\cdots q\cdots\rangle = (\pm 1)^2 |ijk\cdots p\cdots\rangle = |ijk\cdots p\cdots\rangle . \tag{2.24}$$

This operation replaces q by p in our Slater determinant. We can also use this to represent an electron moving from single-particle state q to single-particle state p, which is in fact how we "move" electrons around in our model [6].

2.2.3 Wick's Theorem

Wick's theorem is named after Gian-Carlo Wick and was published in a paper in 1950.

It is a method to rearrange products of creation and annihilation operators such that all creation operators stand to the left of all annihilation operators. Thus one may transform a product of n creation and annihilation operators into the ordered product of the same factors, plus extra terms in which some pairs of factors have been replaced by their commutators or anti-commutators while the remaining factors are ordered in the above sense [8].

In other words, any state can be rewritten as a string of creation and annihilation operators on a vacuum state, this string can be manipulated further by taking advantage of the anti-commutator relations of these operators that we discussed in the previous section.

The first concept of Wick's theorem is normal ordering. Consider a product of creation and annihilation operators

$$\hat{A}\hat{B}\cdots\hat{X}\hat{Y}. \tag{2.25}$$

The normal-order form is then defined as

$$\left\{\hat{A}\hat{B}\cdots\hat{X}\hat{Y}\right\} = (-1)^p[\text{creation}]\cdot[\text{annihilation}], \tag{2.26}$$

where p is the number of permutations needed to transform the original string of operators into this normal ordered form [9].

Further, we define the contraction of two operators as

$$\overbracket{\hat{A}\hat{B}} = \langle 0|\,\hat{A}\hat{B}\,|0\rangle\,. \tag{2.27}$$

If we want to contract two terms separated by other operators in the string, we move the right-most operator to the left and then perform the contraction

$$\left\{\overbracket{\hat{A}\hat{B}\cdots\hat{X}\hat{Y}}\right\} = (-1)^p\left\{\overbracket{\hat{A}\hat{X}}\hat{B}\cdots\hat{Y}\right\}. \tag{2.28}$$

Wick's theorem states that any string can be written as a sum of normal-ordered products with all possible combinations of contractions. Written out it means that

$$\begin{aligned}\hat{A}\hat{B}\hat{C}\hat{D}\cdots\hat{W}\hat{X}\hat{Y}\hat{Z} = (-1)^p&\left\{\hat{A}\hat{B}\hat{C}\hat{D}\cdots\hat{W}\hat{X}\hat{Y}\hat{Z}\right\}\\ &+\sum_{(1)}\left\{\overbracket{\hat{A}\hat{B}}\hat{C}\hat{D}\cdots\hat{W}\hat{X}\hat{Y}\hat{Z}\right\}\\ &+\sum_{(2)}\left\{\overbracket{\hat{A}\hat{B}\hat{C}\hat{D}}\cdots\hat{W}\hat{X}\hat{Y}\hat{Z}\right\}\\ &+\cdots\\ &+\sum_{(n/2)}\left\{\overbracket{\hat{A}\hat{B}\hat{C}\hat{D}}\cdots\overbracket{\hat{W}\hat{X}\hat{Y}\hat{Z}}\right\},\end{aligned} \tag{2.29}$$

where $n/2$ means the largest integer value that does not exceed $n/2$ [9].

2.3 Quantum Dots

Quantum dots are a "hot" topic. Apparently the number of publications with quantum dots as the topic goes as $n \approx 0.4(Y - 1985)^{3.1}$, or thereabout!

A quantum dot is a nano-scale system, usually wrapped in a semiconductor, that confines a few electrons in a potential. Since quantum dots are made with semiconductor etching techniques, they are macroscopic in the quantum mechanical sense. Often with a diameter in the 5-50 nm area. Still, these tiny devices have quantum mechanical properties that we can exploit for a variety of technologies [10].

Applications

There are numerous applications for quantum dots. Here is a little selection:

- Quantum dots are the most promising candidates for use in quantum computing. For instance a single electron trapped in a quantum dot can fulfil the full set of basic logic operations that are necessary for quantum computations [11].
- Quantum dots can also be designed to emit light in LEDs (Light Emitting Diodes). These LEDs can be used in displays or they can for instance be used for imaging to produce self-illuminating probes that have no need for conventional optics for light coupling [12].
- Another article states in its abstract that: Quantum dots have tunable optical properties that have proved useful in a wide range of applications from multiplexed analysis such as DNA detection and cell sorting and tracking, to most recently demonstrating promise for in vivo imaging and diagnostics [13].
- Semiconductor quantum dots may also be used for solar cells. As another abstract goes: They have the potential to greatly increase the photon conversion efficiency by the production of multiple excitons from a single photon of sufficient energy and the formation of intermediate bands in the band-gap that use sub-band-gap photons to form separable electron-hole pairs [14].

2.3.1 A Hamiltonian for Quantum Dots

The system we are looking at in this thesis project is a simplified quantum dot in two dimensions. A quantum dot can be produced that is restricted in the third dimension so that it behaves more like a two-dimensional quantum dot or a *lateral quantum dot* [15]. Such a quantum dot can for instance be simulated numerically as a set of electrons trapped in a two-dimensional harmonic oscillator potential. A sort of "flat atom" if you will. See Figure 2.1 for an example.

A quantum dot may have several potential wells, but we will look at the single potential well variant only. This quantum dot can be approximated by a single harmonic oscillator potential.

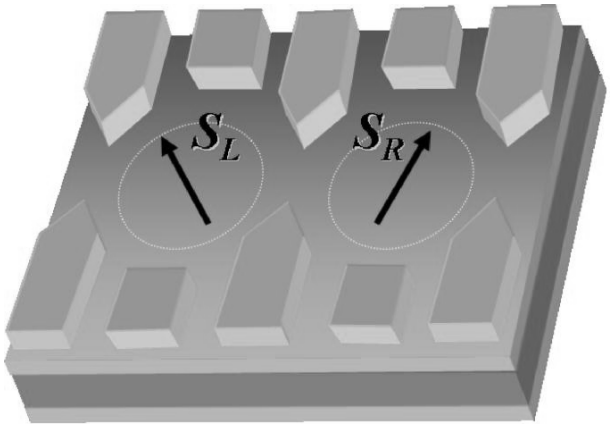

Figure 2.1: A double lateral quantum dot system. Electrons or holes are confined in the middle layer using the electrostatic gates (structures on the surface). The illustration is from a public domain source.

We described briefly the harmonic oscillator potential in two dimensions in 1.2.2. Our Schrödinger equation for such a system reads

$$-\frac{\hbar^2}{2m_e}\nabla^2\psi + \frac{1}{2}m_e\omega^2\mathbf{r}^2\psi = E\psi. \tag{2.30}$$

Since our lateral quantum dots are circular, we can use the convenient polar coordinates $\mathbf{r} = (r, \phi)$ [16].

This is the one-body equation for our system, but we are interested in solving the many-electron equation. When we introduce additional electrons into our potential, these electrons, as they are charged particles, will not only depend on the potential, but also interact with each other. Particles of equal charge will repel each other as a function of $1/(\mathbf{r}_1 - \mathbf{r}_2)$.

In reality, all electrons will interact in such a manner with all other electrons, but we can approximate this interaction by summing over all possible two-electron interactions. This is a relatively good approximation for the many-electron case and exact for the two-electron case.

With these corrections we can extend the N-body Hamiltonian we introduced in equation (1.59) in 1.2.2

$$\hat{H} = \sum_{i=1}^{N_e}\left(\frac{\mathbf{p}_i^2}{2m_e^*} + \frac{1}{2}m_e^*\omega^2\mathbf{r}_i^2\right) + \frac{e^2}{4\pi\epsilon_0\epsilon_r}\sum_{i<j}\frac{1}{\mathbf{r}_i - \mathbf{r}_j}, \tag{2.31}$$

where N_e is the number of electrons, e is the charge of the electron, ϵ_0 and ϵ_r are the free space permittivity and the relative permittivity of the host material, respectively [17].

Also note the mass of the electron, m_e^*, is given as an effective mass. This mass differs from the free-electron mass m_e. In semiconductors this mass may be as little as a factor of 0.1 to 0.01 of the free-electron mass [18].

Dimensionless Form

In order to simplify the computations, it is convenient to write our equations on a dimensionless form.

We define $\hbar\omega$ as our unit of energy, and we define a length unit $l = \sqrt{\hbar/(m_e^*\omega)}$, or the characteristic length unit [17].

This lets us rewrite $\mathbf{r} \to \mathbf{r}/l$ and $\nabla \to l\nabla$.

Our rewritten Hamiltonian now reads

$$\hat{H} = \sum_{i=1}^{N_e}\left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\mathbf{r}_i^2\right) + \frac{e^2}{4\pi\epsilon_0\epsilon_r}\frac{1}{\hbar\omega l}\sum_{i<j}\frac{1}{\mathbf{r}_{ij}}, \tag{2.32}$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$.

We also want to introduce a parameter called the interaction strength parameter

$$\lambda = \frac{e^2}{4\pi\epsilon_0\epsilon_r}\frac{1}{\hbar\omega l}. \tag{2.33}$$

Our final Hamiltonian the reads

$$\hat{H} = \sum_{i=1}^{N_e}\left(-\frac{1}{2}\nabla_i^2 + \frac{1}{2}\mathbf{r}_i^2\right) + \lambda\sum_{i<j}\frac{1}{\mathbf{r}_{ij}}, \tag{2.34}$$

or written simply as a non-interaction part, $\hat{H}_0$, and an interacting part, $\hat{V}$

$$\hat{H} = \hat{H}_0 + \lambda\hat{V}. \tag{2.35}$$

2.3.2 A Hamiltonian Using Second Quantisation

Now that we have established the form of this potential in 2.3.1, and we have previously established a convenient notation for many-body systems in 2.2, it is time to put these together and look at how we can rewrite our Hamiltonian even further.

Since we are using a basis $|\Psi\rangle$ of Slater determinants $|\Phi\rangle$, we need to use an appropriate form for the Hamiltonian. We will be using the occupational number representation, which can be written in the following manner

$$\hat{H} = \sum_{ik} H_{ik}^{(1)}\hat{a}_i^\dagger\hat{a}_k + \frac{1}{4}\sum_{ijkl} H_{ijkl}^{(2)}\hat{a}_i^\dagger\hat{a}_j^\dagger\hat{a}_l\hat{a}_k, \tag{2.36}$$

where $H_{ij}^{(1)}$ is the one-body Hamiltonian matrix containing the one-paticle interaction elements and $H_{ijkl}^{(2)}$ is the two-body Hamiltonian matrix containing the two-particle interaction elements. We will discuss these matrices further in 2.4.

It is computationally a little inconvenient to have a double set of matrices to iterate over when we want to run through our creation and annihilation operators on our basis, so a more computationally convenient simplification is to normal-order the operators and combine the two operators into a single two-body operator. This simplification takes the form

$$\hat{H} = \sum_{\substack{i<j \\ k>l}} \left[\frac{1}{N_e - 1} H^{(1)}_{ik} \delta_{jl} + H^{(2)}_{ijkl} \right] \hat{a}^\dagger_i \hat{a}^\dagger_j \hat{a}_l \hat{a}_k, \tag{2.37}$$

where N_e is the number of electrons of our shell-model system and $\frac{1}{N_e - 1}$ is an algebraic factor that arises from rewriting the one-particle Hamiltonian into two-particle form [19].

Our Hamiltonian now only contains one quadruple sum to run through. Now we just need to acquire the elements of the $H^{(1)}$ and $H^{(2)}$ matrices.

2.4 The Shell Model and the Effective Hamiltonian

So far in this chapter we have looked at how we can approach our many-body problem for quantum dots by using a harmonic oscillator potential and extending our Hamiltonian to include two-body Coulomb interactions. We have also looked at how we can apply second quantisation with creation and annihilation operators to make our problem easier to approach numerically.

Now we will attack the problem of how to solve a system that is in principle infinite in Hilbert space and approximate a finite subspace that is possible to solve numerically.

2.4.1 The Single-Particle Shell Modell

The unperturbed part of our Hamiltonian yields single-particle energies

$$\epsilon_i = \omega(2n + |m| + 1), \tag{2.38}$$

where n is our principal quantum number and m is our m_l quantum number from before, we just dropped the index l. Our quantum numbers are given as $n = 0, 1, 2, \cdots$ and $m = 0, \pm 1, \pm 2, \cdots$. For each set of n and m we also have two possible spin states, $|\uparrow\rangle$ and $|\downarrow\rangle$. It is convenient to represent such a system in a shell-like manner as illustrated in Figure 2.2.

Each shell has an index $R = 1, 2, 3, \cdot$ where

$$R_i \equiv \frac{\epsilon_{(i-1)}}{\omega}. \tag{2.39}$$

As we can clearly see from the figure, the number of available quantum states goes as

$$N_\phi = R(R + 1). \tag{2.40}$$

This is the notation we will be using to represent the number of single-particle states in our Slater determinants.

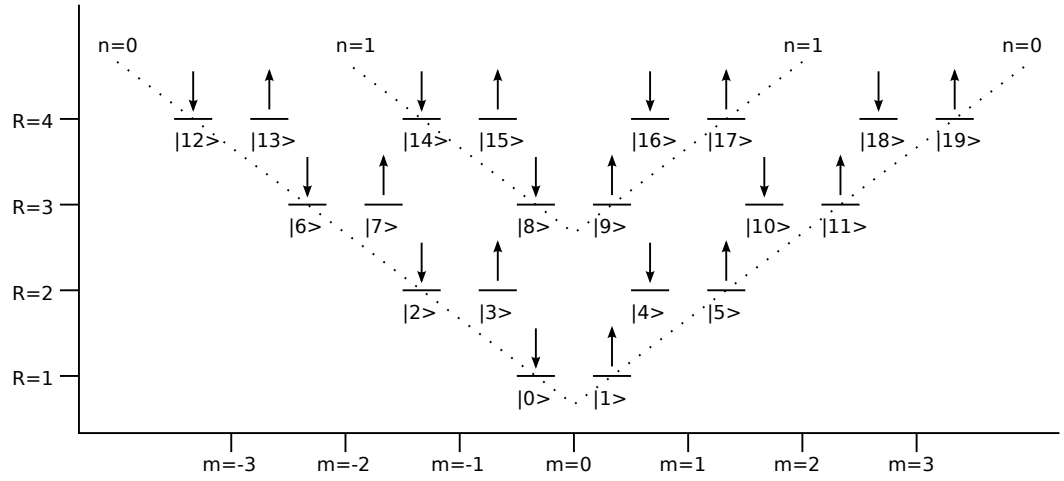

Figure 2.2: The structure of single-particle states for a harmonic oscillator potential in two dimensions. Here illustrated as a system of four shells, R, with single-particle states from $|0\rangle$ to $|19\rangle$. Each state represents one unique set of quantum numbers n and m as well as spins $\uparrow$ and $\downarrow$.

When we use a single-particle basis to carry out many-body calculations, this basis must be truncated. We have to limit our set of possible single-particle states for our electrons as they are in principle infinite. As we just saw, the number of states grows exponentially as $\mathcal{O}(R^2 + R)$. However this is for the single-particle case. For the N_P-particle case our Hilbert space grows as

$$\dim(\mathcal{H}) = \binom{N_\phi}{N_P}, \tag{2.41}$$

where N_ϕ is again the dimensionality of our single-particle basis, and N_P is the number of particles in our many-particle basis [20].

The upper limit of our numerical capability is, as you can imagine, quickly reached. The largest system we have looked at in this project is one of 13 electrons in 6 shells, giving us a total dimensionality of 2.55×10^{10} Slater determinants in our basis.

Our Schrödinger equation can be solved as an eigenvalue equation, as we have already looked at. By building the full Hamiltonian matrix, we can find the energy eigenstates by simply diagonalising this matrix. The larger this matrix is, or the larger the basis is, the more accurate the result is. However since the dimensionality of this matrix grows with the square of the basis, this becomes impossible relatively fast.

In Chapter 5 we will look at a way to solve this without having to diagonalise the full matrix, but we will still have to work with a large basis. There are however ways to reduce this problem. There are ways to reduce the number of shells and thus dramatically reduce the dimensionality of the basis without losing all the accuracy of a larger basis. This is the topic for the next section.

2.4.2 The Effective Hamiltonian

Effective interactions are a vital part of many-body physics and chemistry. In nuclear physics it is used because of the complicated nature of the interactions between nucleons where one has to rely on experimental data in order to approximate these interactions [21]. In nuclei it is not only the Coulomb interaction that plays a role, but also the strong interaction (QCD). The strong interaction cannot be represented by low order approximations as there are significant contributions from higher order terms. Gluons are not at all well behaved particles.

A way to circumvent the dimensionality problem in quantum dots is to take a similar approach, even though our interaction is already relatively simple in nature. This is done by introducing a renormalised Coulomb interaction $\hat{V}_e$ for a limited number of low-lying shells [20]. For quantum dots, this was first used for configuration interaction calculations by Navratil *et al.* [22], and have been extensively studied by Kvaal [21,23,24].

We will not go into a lengthy derivation of the effective Hamiltonian, and we will look at how it behaves numerically later in Chapter 5, but we will describe briefly what this approach entails.

Our Hilbert space, $\mathcal{H}$, is divided into two parts, $P\mathcal{H}$ and $Q\mathcal{H}$, where P is the orthogonal projection onto the aforementioned smaller, effective model space, and $Q = 1 - P$. $P\mathcal{H}$ is then the model space where we will be doing our computations, while $\mathcal{H}$ is in principle the full untruncated Hilbert space.

We consider the interaction operator, $\hat{V}$, a perturbation and we introduce a parameter, z, and instead study $\hat{H}(z) = \hat{H}_0 + z\hat{V}$. If we set $z = 1$, we recover our original bare Hamiltonian.

Let us now define a similarity transform

$$\hat{H}'(z) \equiv e^{-X(z)}\hat{H}(z)e^{X(z)}, \tag{2.42}$$

where the operator $X(z)$ is such that

$$Q\hat{H}'(z)P = 0 \tag{2.43}$$

is orthogonal over $\hat{H}$.

The function $X(z)$ needs to be determined so that the criterion in (2.43) holds and that $X(0) = 0$.

The consequence of these equations is that $\hat{H}'$ has identical eigenvalues with $\hat{H}$ and there are $D = \dim(P\mathcal{H})$ eigenvectors that are entirely in the model space $P\mathcal{H}$. The effective Hamiltonian is thus defined by

$$\hat{H}_e(z) \equiv P\hat{H}'(z)P, \tag{2.44}$$

with D eigenvalues [20].

3
Many-Body Methods

"It would indeed be remarkable if Nature fortified herself against further advances in knowledge behind the analytical difficulties of the many-body problem."

– Max Born, 1960

Many-body problems represent a vast category of physical problems concerning the properties of systems on the quantum scale made of a large number of interacting particles. A *large number* can be anywhere from three particles to an infinite number in some cases (that is, practically infinite, as in a crystal for example).

The method used in this project is the *full scale interaction method*, often abbreviated *FCI*, which we will look at soon, but first a word or two about the variational principle that is the foundation of so many numerical many-body methods.

3.1 Variational Methods

Many of the most used methods for approximating solutions to the Schrödinger equation are based on the *variational principle*.

The variational principle states that for any wave function satisfying the requirements of continuity, differentiability, single-valuedness and normalisability, the expectation value, $\langle \hat{H} \rangle$, is greater than or equal to the ground state energy [25].

Or in other words

$$E_0 \leq \langle \psi | \, \hat{H} \, | \psi \rangle \equiv \langle \hat{H} \rangle. \tag{3.1}$$

That is to say that the expectation value of $\hat{H}$ in the presumably incorrect state ψ is certain to overestimate the ground state energy [2].

Proof: Since the eigenfunctions of $\hat{H}$ form a complete set, ψ can be expressed as a linear combination

$$\psi = \sum_n c_n \psi_n, \tag{3.2}$$

and since ψ is normalised

$$\langle \psi \mid \psi \rangle = \left\langle \sum_m c_m \psi_m \,\middle|\, \sum_n c_n \psi_n \right\rangle = \sum_m \sum_n c_m^* c_n \langle \psi_m \mid \psi_n \rangle = \sum_n |c_n|^2 = 1, \tag{3.3}$$

assuming orthonormal eigenfunctions $\langle \psi_m \mid \psi_n \rangle = \delta_{mn}$.

Now, since

$$\langle \hat{H} \rangle = \left\langle \sum_m c_m \psi_m \middle| \hat{H} \middle| \sum_n c_n \psi_n \right\rangle = \sum_m \sum_n c_m^* E_{nc_n} \langle \psi_m \mid \psi_n \rangle = \sum_n E_n |c_n|^2, \tag{3.4}$$

the ground state energy, E_0, is by definition the smallest eigenvalue, so $E_0 < E_n$ and thus

$$\langle \hat{H} \rangle \geq E_0 \sum_n |c_n|^2 = E_0, \tag{3.5}$$

which is the proof we are looking for [2].

A Usable Algorithm

In order to use the variational method numerically, we need a usable algorithm. It involves three main steps:

1. Construct a trial many-body wave function $\psi_\alpha(R)$, depending on S variational parameters $\alpha = (\alpha_1, \alpha_2, \cdots, \alpha_N)$. The wave function, ψ_α, depends on the position, $R = \mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_N$, of all of the N particles.

2. Evaluate the expectation value of the energy

$$\langle E \rangle = \frac{\langle \psi_\alpha | \, \hat{H} \, | \psi_\alpha \rangle}{\langle \psi_\alpha \mid \psi_\alpha \rangle}. \tag{3.6}$$

3. Vary the paramter α according to some minimisation algorithm and loop back to step 1.

The loop stops when the energy reach some convergence criterion [26].

3.2 The Configuration Interaction Method

The configuration interaction method is the mathematically simplest of the many-body methods, it is also the most accurate in that it converges to the exact solution where other methods are approximations to the FCI method [10]. The method involves expanding an independent-particle wave function of Slater determinants where the coefficients are determined by application of the variational principle [25].

The motivation for this method is that we want to solve the time-independent Schrödinger equation

$$\hat{H} \, |\Psi\rangle = E \, |\Psi\rangle \tag{3.7}$$

that we introduced in Chapter 1.

The expansion of the wave function $|\Psi\rangle$ into N_e-electron basis functions, or Slater determinants, takes the following form

$$|\Psi\rangle = \sum_i c_i |\Phi\rangle , \tag{3.8}$$

where the coefficients c_i are the CI-coefficients.

We can then proceed to rewrite our Schrödinger equation on a matrix form suitable for computation

$$\mathbf{Hc} = E\mathbf{Sc}, \tag{3.9}$$

where the Hamiltonian, $\hat{H}$, has been replaced by a matrix $\mathbf{H}$ and the wave function, $|\Psi\rangle$, has been replaced by a column vector of coefficients, $\mathbf{c}$. The matrix $\mathbf{S}$ is the overlap matrix with elements $s_{ij} = \langle \Phi_i \mid \Phi_j \rangle$.

In principle this is an exact solution to the Schrödinger equation, but in practice, the dimensionality of the matrices must be finite, so we must truncate our expansion to a finite number of terms [27].

However, by introducing the effective Hamiltonian from 2.4, we can regain some of the accuracy we lose by truncating our model space.

3.3 The Coupled Cluster Method

The Coupled Cluster method was originally introduced by Coester [28, 29] in nuclear physics, and introduced in quantum chemistry between 1966 and 1971 by Čížek and Paldus [30]. It is probably the most reliable and computationally affordable method for the approximate solution of the electronic Schrödinger equation [31].

The method aims to solve the time-independent Schrödinger equation (1.15)

$$\hat{H} |\psi\rangle = E_0 |\psi\rangle , \tag{3.10}$$

where $\hat{H}$ is the Hamiltonian and E_0 is the ground state energy.

The central equation of the coupled cluster method is the exponential ansatz of the wave function $|\psi\rangle$,

$$|\psi\rangle \equiv e^{\hat{T}} |\Phi_0\rangle , \tag{3.11}$$

where $|\Phi\rangle_0$ is a Slater determinant (see 2.2.1) usually in a Hartree-Fock basis[1], and $\hat{T}$ is a multi-particle and multi-hole excitation operator.

A common approach is to approximate the multi-excitation operator $\hat{T}$ by including only one-particle one-hole (single excitations) and two-particle two-hole excitations (double excitations)

$$|\psi\rangle \approx e^{\hat{T}_1 + \hat{T}_2} |\Phi_0\rangle . \tag{3.12}$$

[1] Hartree-Fock theory is not covered in this thesis. See Shavitt and Bartlett [6] for more details.

The $\hat{T}_1$ operator represents all possible one-electron excitations of our basis, and $\hat{T}_2$ represent all two-electron excitations.

These can be expanded as:

$$\hat{T}_1 = \sum_i \sum_a t_i^a \hat{a}_a^\dagger \hat{a}_i \tag{3.13}$$

$$\hat{T}_2 = \frac{1}{4} \sum_{i,j} \sum_{a,b} t_{ij}^{ab} \hat{a}_a^\dagger \hat{a}_b^\dagger \hat{a}_i \hat{a}_j, \tag{3.14}$$

where $\hat{a}^\dagger$ and $\hat{a}$ are our creation and annihilation operators.

The expansion of the $\hat{T}$ operator goes to $m \leq N_e$ electrons, and on the generalised form, reads

$$\hat{T}_m = \frac{1}{(m!)^2} \sum_{i\ldots} \sum_{a\ldots} t_{i\ldots}^{a\ldots} \hat{a}_{a\ldots}^\dagger \hat{a}_{i\ldots}. \tag{3.15}$$

The factor $\frac{1}{(m!)^2}$ accounts for redundancy in the summations since permutations of the indices do not produce distinct contributions [6]. It is customary to suffix CC (for coupled cluster) with S, D, T and/or Q, which stands for single, double, triple and quadruple excitations, respectively.

For further insight into the coupled cluster method, see for instance Shavitt and Bartlett [6] or Crawford and Schaefer [31].

3.4 The Monte Carlo Methods

Any numrical technique involving random numbers can be called a *Monthe Carlo* method, named after the famous casino town.

This thesis project does not use any Monte Carlo techniques either, but we are referring to results calculated by such methods, so we will look briefly into how these techniques are used in many-body quantum mechanics.

3.4.1 Variational Monte Carlo

In Section 3.1 we introduced the variational principle. The variational Monte Carlo method, or VMC, consists of choosing a trial wave function, $\psi_T(R)$, which we want to be as realistic as possible for our system. The trial wave function serves as a way to define the quantal probability distribution

$$P(\mathbf{R};\alpha) = \frac{|\psi_T(\mathbf{R};\alpha)|^2}{\int |\psi_T(\mathbf{R};\alpha)|^2 \, d\mathbf{R}}. \tag{3.16}$$

This is out probability distribution function, or PDF.

The expectation value of the Hamiltonian is given as

$$\langle \hat{H} \rangle = \frac{\psi^*(\mathbf{R}) H(\mathbf{R}) \psi(\mathbf{R}) \, d\mathbf{R}}{\int \psi^*(\mathbf{R}) \psi(\mathbf{R}) \, d\mathbf{R}}, \tag{3.17}$$

where ψ is the exact eigenfunction. Using our trial wave function we define a local energy operator

$$\hat{E}_L(\mathbf{R}; \alpha) = \frac{1}{\psi_T(\mathbf{R}; \alpha)} \hat{H} \psi_T(\mathbf{R}; \alpha), \tag{3.18}$$

which, together with our trial PDF, lets us compute the expectation value of the local energy

$$\langle E_L(\alpha) \rangle = \int P(\mathbf{R}; \alpha) \hat{E}_L(\mathbf{R}; \alpha) \, d\mathbf{R}. \tag{3.19}$$

By computing this equation for a set of α values and possible trial wave functions, and then selecting the minimum of $E_L(\alpha)$, the trial wave function should ideally be close to the true wave function, and $\langle E_L(\alpha) \rangle$ should be close to the exact eigenvalue [32].

3.4.2 Diffusion Monte Carlo

The diffusion Monte Carlo method, or DMC, can be used after the VMC method to produce more accurate results [33]. The DMC method is in principle an exact method for finding the ground state of the Schrödinger equation.

In DMC the wave function of a state is given by the density of random walkers. For bosons the total wave function is symmetric, so there are in principal no conceptual problems. However, in the fermion case, the wave function is anti-symmetric and this leads to problems since the density of random walkers, which defines the probablity distribution, is always larger than or equal to zero. This leads to interpretation problems and the so-called fermion sign problems [34].

By use of an approximation called *fixed-node* method, accurate results can still be obtained. The fixed-node method introduces a restriction on the algorithm that it can not cross any of the nodes of the trial wave function, thus effectively dividing the configuration space into volumes connected by these nodes [26]. Hence it depends on a good trial wave function. Variational Monte Carlo can for instance provide such a wave function. In the limit that the trial wave function has the correct nodes, fixed-node DMC produces the exact energy with a statistical error that can be made smaller by increasing the number of Monte Carlo steps [33].

Part II

CODE DEVELOPMENT

4
Code and Hardware Setup

> *"Computer science also differs from physics in that it is not actually a science. It does not study natural objects. Neither is it, as you might think, mathematics; although it does use mathematical reasoning pretty extensively. Rather, computer science is like engineering; it is all about getting something to do something, rather than just dealing with abstractions, as in the pre-Smith geology."*
>
> – Richard Feynman, 1970

The core of this project is the full scale configuration interaction library named `libTardis`. The library name *Tardis* is a reference to the popular British science fiction television series *Doctor Who*. The library is designed to be a high-performance, parallelised and versatile solver for many-body systems using the Lanczos algorithm as its core algorithm.

The current version of `libTardis` only supports systems of electrons trapped in a two-dimensional harmonic oscillator potential. This type of potential is supported directly in `libTardis` by the inclusion of the effective interaction code written by Simen Kvaal [35]. Only the generator of interaction elements is used from his source code. All redundant code has been stripped away and a few tweaks and updates have been done to what remains.

The library itself is designed to support any type of potential though, not only the two-dimensional harmonic oscillator. Neither is it in theory restricted to two-particle interactions. The structure is in place, by usage of super classes and sub classes in the source code, to extend the library to include other potentials as well as extend the Lanczos algorithm to account for three-particle interactions. This, however, is a project for another time.

In the rest of this short chapter we will provide a quick overview of the hardware and software used to develop this project as well as to produce the results presented in Chapters 8 and 9.

4.1 Hardware

As mentioned already, `libTardis` is a high-performance, multi-core and multi-node implementation of the FCI method using the Lanczos algorithm as its core algorithm. In order to develop and test, as well as produce results, the library has been tested and deployed on several desktop computers, servers and also the new super computer at the University of Oslo named *Abel*.

Let us take a quick look at the hardware specifications of these computers and servers in order to have some basis of references when we later look at the computation times for the larger jobs.

4.1.1 Single-Node Computers

The test and development environments for this project consists of mainly four computers. Two computers belong to the Computational Physics department; one has generously been made available by the Institute of Informatics and has primarily been used for core-scale testing as it has 16 physical cores (and 32 when enabling hyper threading); and one is privately owned by the author of this thesis.

Below is a quick description of these computers, and table 4.1 lists the hardware specifications for each of these.

- **Sigma:** Dell Precision T5400 Workstation. This is the office workstation computer which the code is developed on. It has also been used to run smaller jobs. It performs relatively well given that it only has 4 CPU cores.

- **Gizmo:** Dell Precision R5500 Server. This server was bought for the purpose of running larger jobs for master students at the Computational Physics department. The priority when buying this work horse was memory. It was set up with 128GB, but has room for 192GB. It also has a high performance GPU card that was used for another project [16], but this project does not utilise the GPU.

- **Lincoln:** Dell PowerEdge T620. This is a server owned by the Institute of Informatics and has primarily been used for core-scale testing as it is often in use by others. A few calculation jobs have also been run on this computer when it has been available.

- **TheBeast:** This is a custom built high-performance computer based on an ASUS M4A89TD mainboard with the hex core AMD Phenom II X6 1090T Processor. The AMD Phenom II runs at 3.2GHz on all six cores, but can run at 3.6GHz on three cores if necessary. The CPU has 6MB of shared L3 cache and each core has 512kb of L2 cache. The computer is over 2 years old, but the CPU still performs very well and each core is faster than those found in *Gizmo*.

In later references to computation jobs, these names will be used to refer to the computer or server on which that given job was executed. The work horses for the main body of calculations have been *Gizmo*, and to an extent *TheBeast*. Some of the calculations have taken between two and three weeks to complete, and these two have often been running jobs in parallel to reduce the individual work loads.

Name	CPU	Cores	L2	L3	RAM	OS
Sigma	Intel Xeon 5450	4 @3.00GHz	12MB	–	8GB	Ubuntu
TheBeast	AMD Phenom II X6 1090T	6 @3.20GHz	3MB	6MB	8GB	Ubuntu
Gizmo	2×Intel Xeon 5620	8 @2.40GHz	1MB	12MB	128GB	Ubuntu
Lincoln	2×Intel Xeon E5-2650	16 @2.00GHz	2MB	20MB	64GB	Debian

Table 4.1: The hardware specifications for the four main computers used for development and testing of `libTardis`. These are also the main computers used for the majority of the calculations where the jobs have been small enough to not require the use of a super computer.

4.1.2 Multi-Node Computers

The major jobs, requiring up to as much as 92 000 CPU hours, have been run on the new super computer recently installed by the University of Oslo and The Research Council of Norway. It is currently the 96th fastest super computer in the world. The super computer is named *Abel* after the famous Norwegian mathematician Niels Henrik Abel (1802 – 1829).

Abel has a computing capacity of 258 teraflops per second distributed over 10080 CPU cores with a total of 40 terabytes of memory. Each node has 16 physical Intel cores in two Intel Xeon E5-2670 CPUs running at 2.6 gigahertz. Each node has 64 gigabytes of memory and the inter-node communication runs at 56 gigabits per second. Thus making it very efficient even when a lot of inter-node communication is needed.

For more information about *Abel*, see the official website at:
http://www.uio.no/english/services/it/research/hpc/abel

For an overview of the calculations run on *Abel* and the CPU hours used, see table 7.3. There is also a table showing the percentage of pure computation time and communication time for most of the larger calculations. These are available in table 7.4. Both tables are located in Chapter 7 together with a discussion of the performance of the code on the super computer.

4.2 Software

The library `libTardis` is not a stand-alone library. It depends on several other shared libraries and is not likely to be multi-platform without modification. The code has been developed on Ubuntu 11.04 and 12.04 and will most likely work on most, if not all, Linux distributions, but is not likely to run on Windows or MacOS unmodified.

4.2.1 Linear Algebra Libraries

The core linear algebra library that `libTardis` depends on is Armadillo [36]. Armadillo is essentially a wrapper for the more commonly known linear algebra libraries Lapack and Blas, but also provides a set of useful classes that greatly simplifies their usage. Since `libTardis` only uses relatively few linear algebra operations, performance is not an issue

although the creators of Armadillo has put a lot of work into making the library as effcient as possible.

Armadillo

The main feature of Armadillo used in `libTardis`' source code is the simplified matrix notation. A double precision matrix can be created easily in a few lines. Let us for example create a 10 by 10 matrix of double precision values and set them all to random numbers:

```
Mat<double> mMatrix;
mMatrix.randu(10,10);
```

If we, for instance, then want to extract the second column into a new vector object, we simply call

```
Col<double> mVector = mMatrix.col(1);
```

Another nifty feature that is used in `libTardis` to store the block-diagonal Hamiltonian matrix, is the ability to stack multiple Armadillo objects into a collector object. If we for instance want to store a Hamiltonian matrix of 5 diagonal blocks with dimensions (5×5, 4×4, 3×3, 2×2, 1×1), we could do this:

```
field<Mat<double> > mHamiltonian(1,5);
for(i=0; i<5; i++) {
    mHamiltonian(0,i).set_size(i,i);
    [ Some code to fill the blocks ]
}
```

In other words, as these examples show, Aramdillo makes matrix implementation easy and time and energy can be used for more important tasks. It is also worth mentioning that all Armadillo objects support a simple `save()` and `load()` function that will stream the data to or from memory from or to a file, either as binary data, or as a readable ASCII file. These are highly efficient functions.

The other main advantage of Armadillo is how it accesses linear algebra operations. Under the hood, Armadillo will either solve the problem itself, or pass the job on to Lapack, Blas, MKL or a list of other libraries it supports. Armadillo will choose the most efficient method for each type of problem and consider its dimensionality, and do the optimisation for you. This does of course remove that control from you as the programmer. A point worth taking into consideration.

At this point it may also be worth mentioning the main disadvantage of Armadillo. Its use significantly increase compiling time for small projects. It also bloats the executable. This is however not a major issue with `libTardis` as it only uses a few linear algebra operations.[1]

[1] See Armadillo's API manual for further details [37].

Lapack, Blas and MKL

As mentioned, Armadillo also functions as a wrapper for Lapack, Blas or the Intel implementation of these, the MKL library. During compile time Armadillo should discover the existence of these and incorporate them into its function set.

Aside from this, these libraries are not in use directly by `libTardis`. However it is worth mentioning that the stripped down code from Kvaal, that implements effective interactioin elements, does use Lapack directly. The relevant header files for this have been included within `libTardis` in the `OpenFCI` folder, but it still requires the development package for Lapack to be installed in order for this to work.

So far, no issues have been discovered by using the MKL library instead of Lapack and Blas.

Compilers

The main compiler used for development and testing, and the one used by the Python wrapper scripts, is `g++` or in the case of MPI, `mpic++`. On the super computer *Abel*, the code compiles fine with the Intel compiler `icc` and the Intel OpenMPI compiler `mpicc`.

4.2.2 Parallelisation and Large Scale Usage

Since `libTardis` is designed as a library, only containing a set of classes relevant to the FCI method, it cannot on its own be compiled and run. In order for it to run and produce a result, the library needs to be included, and its classes called, from a wrapper code. There are examples of this provided for both an OpenMPI implementation and a single-node OpenMP implementation. There is also included a Python scripts that will generate a set of the latter based on arrays of variables, and another Python script to compile and run these in sequence, enabling a user to easily run the code for a set of configurations defined by arrays by simply running these two scripts. The outputs and results are then neatly organised into log files and output folders. An example of such an autogenerated file can be found in Appendix A, Listing A.1.

The library has been optimised for parallelisation an utilises OpenMP to split its core algorithm, the Lanczos algorithm, across multiple threads on a single-node computer. Multi-node parallelisation is not directly implemented. but the Lanczos algorithm has been implemented both as a single function call and as a set of functions for master and slave nodes so that a relatively straight forward wrapper code can utilise packages like OpenMPI or MPICH2 to distribute the code on multiple nodes. The wrapper code used for the large scale calculations in this project, using OpenMPI, has been included in Appendix A, Listing A.2.

We will discuss the performance boost of the parallelisation of `libTardis` further in 7.2.

5
Algorithms and Data Structure

"An infinite number of monkeys typing into GNU emacs would never make a good program."

– Linus Torvalds, 1995

The core functionality of the code for this project, `libTardis`, is based on an article from 1977 by Whitehead, Watt, Cole and Morrison [19] which discusses an implementation of the Lanczos algorithm to solve the Hermitian eigenproblem.

The original article discusses an implementation for a nuclear shell model, but we are setting it up to handle, in principle, both nuclear shell models and electron shell models of various types of potentials. The only implementation provided at this time, though, is the one for quantum dots in two dimensions using a harmonic oscillator potential.

This chapter discusses the core algorithms, mainly the Lanczos algorithm, and a few other key implementations and solutions. This chapter also includes a description of the data structure and how certain challenges have been overcome.

5.1 The Lanczos Method

The Lanczos algorithm is an iterative algorithm attributed to Cornelius Lanczos (1950). It is an algorithm well suited for finding eigenvalues and eigenvectors of square matrices of very large dimensionality.

5.1.1 Krylov Space

Let us first take a quick look at the Krylov space as some of its properties will be needed later.

A Krylov matrix, $\mathbf{K}$, is a matrix where each column represents an iteration of

$$\mathbf{K}^m(\mathbf{x}, \mathbf{A}) \equiv [\mathbf{x}, \mathbf{A}\mathbf{x}, \mathbf{A}^2\mathbf{x}, \cdots, \mathbf{A}^{m-1}\mathbf{x}] \in \mathbb{F}^{n\times m}, \tag{5.1}$$

generated by the vector $\mathbf{x} \in \mathbb{F}^n$.

Its columns span the Krylov subspace

$$\mathcal{K}^m(\mathbf{A}, \mathbf{x}) \equiv (\mathbf{x}, \mathbf{A}\mathbf{x}, \mathbf{A}^2\mathbf{x}, \cdots, \mathbf{A}^{m-1}\mathbf{x}). \tag{5.2}$$

The Lanczos method computes an orthonormal basis of the Krylov space.

Given a matrix $\mathbf{A}$ that we want to find some or all of the eigenvalues for, let

$$[\mathbf{x}, \mathbf{A}\mathbf{x}, \cdots, \mathbf{A}^{k-1}\mathbf{x}] = \mathbf{Q}^{(k)}\mathbf{R}^{(k)}, \tag{5.3}$$

be the QR-factorisation of the Krylov matrix $K^m(\mathbf{x})$. The Ritz values and Ritz vectors of $\mathbf{A}$ in this space are then obtained by the $k \times k$ eigenvalue problem

$$\mathbf{Q}^{(k)*}\mathbf{A}\mathbf{Q}^{(k)}\mathbf{y} = \vartheta^{(k)}\mathbf{y}. \tag{5.4}$$

If $(\vartheta_j^k, \mathbf{y}_j)$ is an eigenpair of (5.4), then $(\vartheta_j, \mathbf{Q}^{(k)}\mathbf{y}_j)$ is a Ritz pair of $\mathbf{A}$ in $\mathbf{K}^m(\mathbf{A}, \mathbf{x})$ [38].

5.1.2 Derivation

Suppose $\mathbf{A} \in \mathbb{R}^{n\times n}$ is large, sparse and symmetric, and assume that a few of its largest and/or smallest eigenvalues are desired. This problem can be solved by the Lanczos method. The method generates a sequence of tri-diagonal matrices, $\mathbf{T}_k$, with the property that the extremal eigenvalues of $\mathbf{T}_k \in \mathbb{R}^{k\times k}$ are progressively better estimates of $\mathbf{A}$'s extremal eigenvalues [39].

The tridiagonal matrix $\mathbf{T}_k$ is represented by the vectors $\alpha(1:k)$ and $\beta(1:k-1)$

$$\mathbf{T}_k = \begin{bmatrix} \alpha_1 & \beta_1 & \dots & 0 \\ \beta_1 & \alpha_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \beta_{k-1} \\ 0 & \dots & \beta_{k-1} & \alpha_k \end{bmatrix}. \tag{5.5}$$

The Lanczos iterations consists of looping over

$$\mathbf{T}_k^{(k\times k)} = \mathbf{Q}_k^T \mathbf{A}^{(n\times n)} \mathbf{Q}_k \tag{5.6}$$

until the extremal eigenvalues converge or until $k = n$, in which case $\mathbf{T}_k$ should have the same eigenvalues as $\mathbf{A}$. The transformation matrix, $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_k]$, contains the Lanczos vectors $\mathbf{q}_i$, which needs to be orthonormal. Orthonormality of $\mathbf{q}_i$ implies that

$$\alpha_k = \mathbf{q}_k^T \mathbf{A} \mathbf{q}_k. \tag{5.7}$$

If, then,

$$\mathbf{r}_k = (\mathbf{A} - \alpha_k \mathbf{I})\mathbf{q}_k - \beta_{k-1}\mathbf{q}_{k-1} \tag{5.8}$$

is non-zero, then

$$\mathbf{q}_{k+1} = \frac{\mathbf{r}_k}{\beta_k}, \tag{5.9}$$

where

$$\beta_k = ||\mathbf{r}_k||_2, \tag{5.10}$$

we get the straight forward Lanczos algorithm in listing 5.1.

```
r[0] = q[1]
beta[0] = 1
q[0] = 0
k = 0
while beta[k] != 0
    q[k+1] = r[k]/beta[k]
    k = k + 1
    alpha[k] = trans(q[k])*A*q[k]
    r[k] = (A - alpha[k]*diag(1))*q[k] - beta[k-1]*q[k-1]
    beta[k] = dot(r[k],r[k])^2
end
```

Listing 5.1: Simple Lanczos algorithm pseudo-code [39].

5.1.3 Practical Implementatiom

There is a cost to doing it this way though. The $\mathbf{q}_i$ vectors must remain orthonormal, so these must be kept in memory for that purpose. Since $\mathbf{q}_i$ has length n, the dimensionality of $\mathbf{A}$, these vectors can run relatively large for systems with a large basis.

The solution to this problem is to exploit the formula

$$\alpha_k = \mathbf{q}_k^T(\mathbf{A}\mathbf{q}_k - \beta_{k-1}\mathbf{q}_{k-1}). \tag{5.11}$$

Doing so, we can reduce the algorithm to two Lanczos vectors and overwrite them for each iteration. This new iteration loop is shown in pseudo code in listing 5.2.

```
v(1:n) = 0
w(1:n) = rand()
beta[0] = 1
k = 0
while beta[k] != 0
    if k != 0
        for i = 1:n
            t = w[i]
            w[i] = v[i]/beta[k]
            v[i] = - beta[k]
        end
    end
    v = v + A*w
    k = k + 1
    alpha[k] = w*v
    v = v - alpha[k]*w
    beta[k] = v*v
end
```

Listing 5.2: Improved Lanczos pseudo-code [39].

For each Lanczos iteration, the eigenvectors of $\mathbf{T}_k$ are calculated and also the relative change, or convergence, of the lowest eigenvalue. It is still important that the vectors $\mathbf{v}$ and $\mathbf{w}$ remain orthonormal throughout the process. If β_k becomes small, orthogonality may be lost. In the implementation of the algorithm in `libTardis`, orthogonality is checked for each iteration, and the vectors corrected if the dot product drifts too far off numerical zero. This ensures orthonormality of the current Lanczos vectors, but may not ensure their orthonormality to all the previous vectors.

The implementation of the Lanczos algorithm in `libTardis` is based on the implementation described in the 1977 article by Whitehead *et al.* [19]. The computation power of the average computer has increased significantly since then, but the principle is still the

same. The core of their implementation is looping over all possible creation and annihilation operators on each element of the bases, check its change in energy, and update the vector index corresponding to the new basis element created by the sequence of creation and annihilation operations.

The two-particle problem can easily be solved by calculating the eigenvalues of the Hamilton matrix trough straight forward diagonalisation, so the Lanczos algorithm is redundant for such a small system, but it still serves as a good control of the accuracy and convergence properties of the Lanczos algorithm itself. The simple diagonalisation implementation is retained for this purpose in the class `classDiag` in `libTardis`.

The big bonus for our purpose, however, is that for a much larger system of particles, by using the Lanczos algorithm, we do not need to build the matrix $\mathbf{A}$ (in our case, the Hamiltonian), which for the largest calculations in this project has been in the order of $n = 3.13\text{e}8$, which is a matrix that would consume in the area of 700 petabytes of memory ($n^2 \times 8$ bytes). We instead do as Whitehead *et al.* and loop over all the possible single-element multiplications and sums contained in the line "`v = v - alpha[k]*w`" in Listing 5.2. Because of this, a serial (one core) implementation of this algorithm uses $2n \times 8$ bytes as the bulk memory requirement.

5.1.4 Ritz Eigenvectors

In addition to finding the lowest eigenvalues by using the Lanczos algorithm, we may also want to find the corresponding eigenvector. However the eigenvectors of $\mathbf{T}_k$, let us call them $\mathbf{s}_i$, are obviously not the eigenvectors of $\mathbf{A}$ unless $k = n$. In order to retrieve the extremal eigenvectors of $\mathbf{A}$, we need to transform $\mathbf{s}_i$ by using the matrix $\mathbf{W}_k = [\mathbf{w}_1, \mathbf{w}_2, \cdots, \mathbf{w}_k]$ from our Lanczos iteration, see (5.4).

$$\mathbf{y}_i = \mathbf{W}_k \mathbf{s}_i^{(k)}, \tag{5.12}$$

where the vector $\mathbf{y}_i$ is the i-th Ritz vector, an approximate eigenvector of $\mathbf{A}$.

The need to do this causes us again to have to keep all the Lanczos vectors around after all. We no longer need to use them in our Lanczos iterations though, so they can be dumped to file for each iteration and we then perform this calculation as a separate job.

5.1.5 Eigenvalues and Errors

Now, since our eigenvalues are approximate eigenvalues, Ritz values ϑ_i, our actual eigenvalues are given as

$$\lambda_i \in [\vartheta_i^{(k)} - \tau_i^{(k)}, \vartheta_i^{(k)} + \tau_i^{(k)}], \tag{5.13}$$

where $\tau_i^{(k)}$ is our error of the k-th Lanczos iteration

$$\tau_i^{(k)} = ||(\mathbf{A} - \vartheta_i^{(k)}\mathbf{I})\mathbf{y}_i^{(k)}|| = |\beta_k| \times |\mathbf{e}_k^\dagger \mathbf{s}_i^{(k)}|. \tag{5.14}$$

In our calculations we are usually only interested in the lowest eigenvalue, λ_0, which is our ground state energy, E_0. However, all (5.13) promises us is that there is an eigenvalue

within the interval $[\vartheta_i^{(k)} - \tau_i^{(k)}, \vartheta_i^{(k)} + \tau_i^{(k)}]$ when the Lanczos algorithm has "stabilised" to a convergence δ_C. [40]

We will look more at what this uncertainty entails in Chapter 7.

5.2 Computer Representation of the Shell Model

Our model space is represented by a set of single-particle states ϕ_i. The model space is in reality infinite, but we need to truncate it to a manageable number in order to be able to run calculations on it. Preferably we want the number as low as possible as the number of single-particle states goes as

$$N_\phi = R(R+1), \tag{5.15}$$

where R is the number of shells in our model space.

The shell structure of single-particle states was shown earlier in Figure 2.2 in 2.4, showing an example for a system of 4 shells.

This shell structure is represented in computer memory by a $N_\phi \times 3$ matrix where the column indices represent the single-particle state, and the quantum numbers n, m and spin are stored in each row. After this point, the states are referred to only by their column index, or $|\phi_i\rangle = \mathrm{Row}(i)$. The order of these is irrelevant at this point, but for the sake of readability of terminal outputs, the ordering is from left to right, bottom to top.

5.2.1 Binary Slater Determinants

One of the key features of `libTardis` is the binary representation of Slater determinants. In this representation a Slater determinant is a binary word of length N_ϕ such that

$$|0\rangle_{R=4} = |0000000000000000000\rangle \quad \text{is the vacuum state} \tag{5.16}$$

$$\hat{a}_4^\dagger \hat{a}_9^\dagger |0\rangle_{R=4} = |0000100001000000000\rangle = |4,9\rangle_{R=4} \quad \text{is a creation operation} \tag{5.17}$$

$$\hat{a}_4 |4,9\rangle_{R=4} = |0000000010000000000\rangle \quad \text{is an annihilation operation.} \tag{5.18}$$

From a coding point-of-view, this representation means that each operation on the Slater determinant is one or more binary operations on a binary word. The benefit of this is a very fast code as binary operations is what the CPU does best. A creation operation is represented by setting a bit that was formerly 0, to 1; and vice versa for annihilation (see 6.2.1 for details on the implementation in the code).

The object used for storing this information in the code is a `bitset`. The key benefit of using the `bitset` class, over simply storing the data in a long variable, is that the `bitset` class will scale beyond 64 bit (assuming a 64 bit architecture) by using an array of long variables. Since the size of the word is set at compile time, the `bitset` class will only use the array representation if it is strictly necessary[1]. There is, not surprisingly, a cost to this usage of increasingly longer arrays. The two plots in figure 5.1 show the time consumption of 10^8 creation and annihilation operations as a function of shells, R, on the left side, and

[1] See the STL source code [41].

as a function of states (bits) on the right side. There is a non-negligible increase in computation time for each time the `bitset` class needs to increase the dimensionality of this array. It is therefore beneficial to keep the variable `SLATER_WORD` in the config file of `libTardis` set as low as possible to ensure that as little time as possible is used to loop through array elements that are not in use.

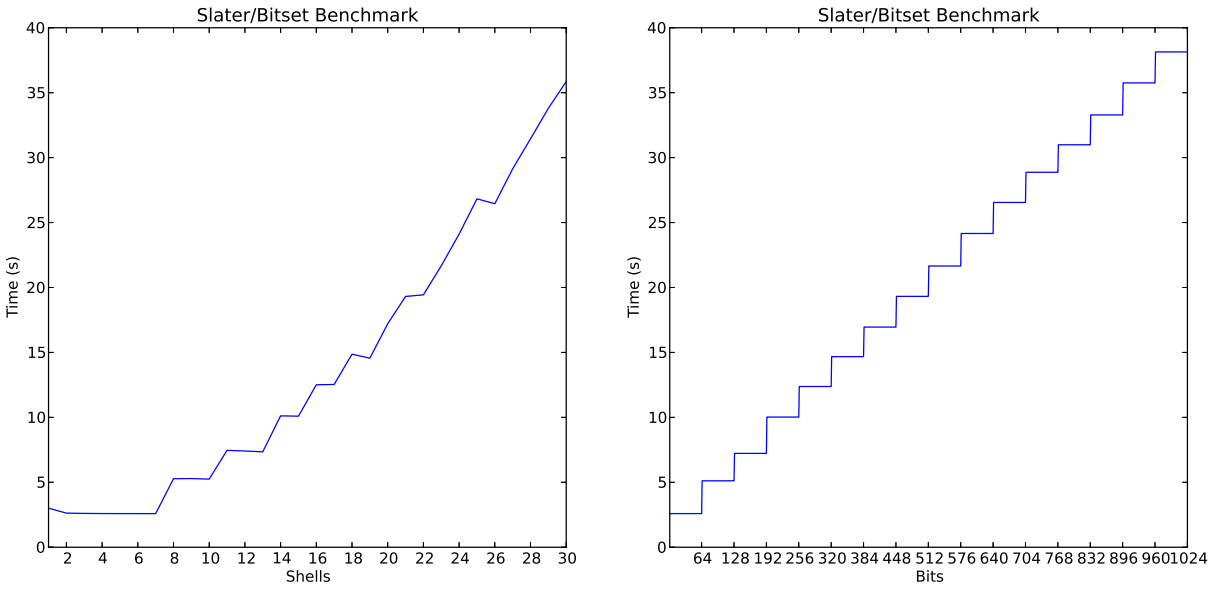

Figure 5.1: Two plots showing computation time of 10^8 creation operations and 10^8 annihilation operations on a `Slater` object using a binary represenation of single-particle states via the `bitset` class. The `bitset` class will use an array of long variables (64-bit in our case) to store these states in the event more than one long variable is needed. The left plot shows the increase in computation time as a function of number of shells, R. As can be seen from this plot, the number of shells only affect the speed whenever the `bitset` object needs to add another long variable of storage to its internal array. A step-like structure can be seen, but as the number of states blow up rapidly, the steps smooth out. The right plot shows the same situation, but now as a function of states (bits). The step-structure for each new 64-but variable is clearly visible.

The downside to this approach is that the maximum number of single-particle states needs to be set manually at compile time. There does exist a dynamic version of the `bitset` class that will allow this number to be set at run time, but the performance of this class is significantly lower than that of the `bitset` class as one can imagine. Since the manipulation of Slater determinants lay at the very core of the matrix-vector multiplication section of the implementation of the Lanczos algorithm, it matters a great deal that they are as efficient as possible.

6
The Code

"The most important thing in a programming language is the name. A language will not succeed without a good name. I have recently invented a very good name, and now I am looking for a suitable language."

– Donald Knuth

The source code of `libTardis` is developed in `c++` with a set of accompanying Python scripts to generate and run jobs using this library. `c++` was an obvious choice for this project as it is an object oriented and fast programming language. `libTardis` is a purely object oriented source code. There are no parts of the library that is not a class, so in order for it to run, the library needs to be called from a wrapper code.

In this chapter we will look at the class structure of `libTardis` and go through each of them and detail their core functionality and discuss their implementation. We will also look at how to set up a wrapper for a simple many-body calculation.

Dependencies

The library `libTardis` depends on a few other libraries. Most importantly it utilises the efficient and straight forward linear algebra library Armadillo [36]. Armadillo is a wrapper for well known linear algebra libraries like Lapack and Blas. It also uses Boost. Armadillo was chosen because of its efficiency in choosing the best underlying method for solving a give linear algebra problem. It also provides easy-to-use containers for vectors and matrices as well as multi-dimensional objects called "fields".

In addition to this, Armadillo contains easy-to-use load and save functionality that will dump or load a given dataset to or from file at as high speed as hardware will allow. This is utilised if the cache functionality is enabled in the config file.

In `libTardis` there is also included a stripped down version of `OpenFCI` by Simen Kvaal [10, 35]. The code uses two classes from the `OpenFCI` package to generate both the effective and the bare two-particle interaction elements used for quantum dots in two dimensions.

The author of the `OpenFCI` package used their own classes for matrices and vectors, which again relies on Lapack. The code folder should have all the necessary Lapack header files included and it is designed to be stand-alone. This is all organised into a folder, under the main project folder, named "OpenFCI". This code can be extracted in its entirety and used directly in other projects where generating interaction elements for large two-particle systems is needed. The content if the `OpenFCI` folder does not depend on any of the other code in `libTardis` although a few header file references may need to be altered.

The implementation of `OpenFCI` is slightly modified from the original version. The files have been renamed to conform to the naming convention used for the library as a whole, and minor modifications to the code have been made to eliminate numerous compiler warnings.

Parallelisation

In addition to these libraries, `libTardis` also uses OpenMP to distribute the computation load of the Lanczos algorithm on all cores of the computer on which it is running. OpenMP implementation can be turned off entirely in the configuration file `config.hpp`. This will also eliminate the dependency of OpenMP during compilation.

It is also possible to use OpenMPI to distribute the code across multiple nodes. Although master and slave functions are provided for the Lanczos solver as an alternative to the single-node function calls, there are no communication routines included in `libTardis`. Any multi-node implementation needs to be coded into the wrapper code.

Philosophy

When developing `libTardis`, future possible extensions were kept in mind. The structure has been planned in cooperation with other students, especially Christoffer Hirth [16], in order to be as flexible as practically possible without affecting performance.

The `System` object is intended to hold all the necessary information about the system which the simulation is currently running on. This means it contains the basis as well as the potential of that given system. The philosophy here is that the object, after it is initialised and the interaction elements loaded or generated, can be passed through several other processes which perform calculations on it and return the results to the system object itself. Example being calculating the coefficients of the basis through the Lanczos algorithm and then pass them on for further analysis, or simply save them to disk.

As it is now, `libTardis` only contains functions for generating interaction elements for a harmonic oscillator potential in two dimensions intended for performing calculations on quantum dots (QDOT2D). The code is prepared for implementation of at least four more types of potential. Namely quantum dots in three dimensions (QDOT3D), electrons in atoms (ATOM), nucleons in normal nuclei (NUCLEUS) and hyper-nuclei with strangeness different than one (HNUCLEUS). (The code words in the bracket are the names of the potential to pass on to the build command.)

Note

Most input variables are intuitive, but note that the input variables for spin and total spin (usually denoted ξ and $2s$ in the text) are integer values, not float or double. In other words, spin 1/2 is represented by a 1.

6.1 Organisation of the Code

All of the source code of `libTardis` is contained within the folder by the same name. The root folder contains the main header file, `libTardis.hpp` and a config file, `config.hpp`. The rest of the source code is organised in sub folders containing one cpp and one hpp file for each class and a few additional module files that are included into classes where needed.

Folder Structure

The folder structure is roughly divided into categories with the relevant classes located in each folder.

- **System**: Contains the classes for setting up quantum mechanical systems.
  - Class `Log`: A class that handles runtime output and logging.
  - Class `Basis`: A class to contain the basis that the solver will run over.
  - Class `Slater`: A class to contain and manipulate Slater determinants.
  - Class `System`: A wrapper class that contains the full system for the solver.
- **Potential**: Contains the classes for quantum mechanical potentials.
  - Class `Potential`: A super class for the available potentials.
  - Class `QDot2D`: A class for the two-dimensional harmonic oscillator.
- **OpenFCI**: Contains the stripped down version of `OpenFCI` used by the class `QDot2D` to generate both bare and effective interaction elements.
- **Solvers**: Contains the solvers.
  - Class `Diag`: A class that calculates the lowest energy eigenvalue for 2-particle systems by use of simple diagonalization of the Hamiltonian matrix.
  - Class `Lanczos`: A class that calculates the lowest energy eigenvalue and all the coefficients for a given basis by use of the Lanczos algorithm. This is a many-body solver.

6.2 The System Classes

The `System` class is the core class of `libTardis`. It is the first object that needs to be created, and this object is passed on to any further operations.

The other classes in the *System* category are all accessed by the `System` class directly, and does not need to be created by the user. However these sub-objects can be accessed by pointer functions from `System` if so is needed.

But let us first define the three dependencies of `System` before we look at `System` itself.

6.2.1 The Slater Class

The `Slater` class is the most fundamental class of `libTardis`. It is the core data type that holds the elements of the basis. It is a binary implementation of Slater determinants based on the standard `c++` class `bitset`. The construction is straight forward. The set of single-particle states in the Slater determinant is contained in a bitset variable named `Word`. This is the only global variable of `Slater`. This is also the only time `libTardis` deviate from the standard of having Get and Set functions to access the main variables of the class. This is due to improve performance and considering the fact that a third party user is not likely to need to request this class directly.

Performance

The `bitset` class in `c++` will create a vector of binary objects of 32 or 64 bit, depending on the architecture of the computer the code is compiling on. The constructor can take any reasonable size as input, and will build the object by using several short words in a way that is transparent to the user. However there is a cost in computation time when the `bitset` object needs more than one word to construct the Slater determinant. These performance issues were discussed in more detail in 5.2 where binary Slater determinants are introduced and discussed. A plot illustrating how the computation cost of each incremental step of 64 bit (or 32 bit in case of a 32-bit architecture) is available as Figure 5.1 in the previous mentioned section.

Creation and Annihilation Operations

The main methods of the `Slater` class are the `Create()` and `Annihilate()` functions. They both work in a similar fashion. They take an integer as input, representing one single-particle state

$$\texttt{Slater.Create}(p) \rightarrow \hat{a}_p^\dagger \left|\Phi\right\rangle \tag{6.1}$$

$$\texttt{Slater.Annihilate}(p) \rightarrow \hat{a}_p \left|\Phi\right\rangle, \tag{6.2}$$

and returns a 0 if the operation is illegal; in other words, the single-particle state is not occupied in the event of an annihilation operation, or is already occupied in the event of a creation operation (see 2.2.1).

In the case that the operation is allowed, a binary bit-shift operation is performed in order to remove all bits set at the given state p or higher. This leaves only the lower-than-the-current states, they are counted, and a parity, P, is calculated and returned as the return value.

$$P = (-1)^{\sum_{i=0}^{p-1} n_i}. \tag{6.3}$$

where n_i is the occupational number, 0 or 1, of single-particle state i.

Comparing Slater Objects

There are two functions for comparing Slater objects. The `Equal()` method simply returns true if the subject Slater determinant is equal to the object. The `Compare()` method returns a 0 in the above case, but also performs a greater/lesser-than test if the Slater determinants are found not to be equal. This test is the backbone of the binary search algorithm of the `Basis` class. Without this highly efficient binary implementation of comparison, the efficiency of the Lanczos algorithm would suffer greatly.

The test works as follows (with object Slater determinant labelled as $|\Phi\rangle_O$ and subject Slater determinant labelled as $|\Phi\rangle_S$):

- Perform an XOR operation on the two Slater determinants:

$$|\Phi\rangle' = |\Phi\rangle_O \text{ XOR } |\Phi\rangle_S . \tag{6.4}$$

- The XOR operation returns a 1 for all bits that are unequal in the two Slater determinants. Therefore the first bit to be set to one in $|\Phi\rangle'$ is the bit representing the larger Slater determinant of the two compared. The position of this bit is located by use of the `_Find_first()` method of the `bitset` class. This method is an experimental one and may or may not be supported in future releases of the `bitset` class [41].

- The last step is to check whether it is the object Slater determinant that has that particular bit set. If so, it is the largest of the two and the function returns a 1. If this is not the case, i.e. the subject Slater determinant is the one with that bit set, the subject is the larger, and the function returns a -1.

Reader Friendly Output

A method called `Output()` is also available. This method is intended for printing the Slater determinants to terminal or file in a reader-friendly manner. The only input value is an integer that is used to truncate the output since the `bitset` variable is most likely longer than the actual number of single-particle states in use since they are rounded up to nearest integer multiple of 32 or 64 bit.

6.2.2 The Basis Class

The `Basis` class contains the basis of Slater determinants the solver will run over. It requires a `Potential` and a `Log` object as input and needs to know how many particles and shells the system contains. The `Basis` object is usually constructed and configured through the `System` object, but all the methods in `Basis` is available through the pointer available via `System->GetBasis()`.

The basis itself is built using an STL vector [41] of `Slater` objects. There is also an Armadillo double vector that is intended to contain the configuration interaction coefficients (see 3.2), but this vector is set to zero-length until it is needed in order to save memory.

Indexing the Basis

The basis of Slater determinants can grow very large when we use a full configuration interaction model (see 3.2). The matrix-vector product function of the Lanczos algorithm heavily relies on an efficient search algorithm in order to perform well. During development the initial implementation was a straight forward sequential search by help of a two-dimensional index. The index was built as a matrix of column vectors

$$\mathbf{I} = \begin{bmatrix} \mathbf{i}_{0,0} & \mathbf{i}_{0,1} & \cdots & \mathbf{i}_{0,N_\phi} \\ \mathbf{i}_{1,0} & \mathbf{i}_{1,1} & \cdots & \mathbf{i}_{1,N_\phi} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{i}_{N_\phi,0} & \mathbf{i}_{N_\phi,1} & \cdots & \mathbf{i}_{N_\phi,N_\phi} \end{bmatrix}, \tag{6.5}$$

where N_ϕ is the number of single-particle states and the matrix elements are vectors with indices defined by the p and q indices of the $\hat{a}_p^\dagger$ and $\hat{a}_q^\dagger$ creation operators from the Lanczos matrix-vector multiplication.

There is a switch defined in `config.hpp`, `INDEX_BASIS`, which when defined, causes this index of the basis to be built. As discussed below, the search algorithm is, after optimalisation, primarily based on the binary search algorithm, but a brute force search can be faster on smaller systems. The main disadvantage with the old method is that the dimensionality of the matrix grows as $\mathcal{O}(N_\phi^2 N_P)$ where N_P is the number of particles.

The binary search algorithm also uses and index. This index is much simpler and is basically an index, $\mathbf{i}_k$, of what interval of Slater determinants have the first single-particle state, k, occupied.

$$\begin{aligned}
\mathbf{i}_0 &= \begin{cases} |\Phi_0\rangle &= |11\ 0000\rangle \\ |\Phi_1\rangle &= |10\ 1000\rangle \\ |\Phi_2\rangle &= |10\ 0100\rangle \\ |\Phi_3\rangle &= |10\ 0010\rangle \\ |\Phi_4\rangle &= |10\ 0001\rangle \end{cases} = (0,1) \\
\mathbf{i}_1 &= \begin{cases} |\Phi_5\rangle &= |01\ 1000\rangle \\ |\Phi_6\rangle &= |01\ 0100\rangle \\ |\Phi_7\rangle &= |01\ 0010\rangle \\ |\Phi_8\rangle &= |01\ 0001\rangle \end{cases} = (5,8) \\
\mathbf{i}_2 &= \begin{cases} |\Phi_9\rangle &= |00\ 1100\rangle \\ |\Phi_{10}\rangle &= |00\ 1010\rangle \\ |\Phi_{11}\rangle &= |00\ 1001\rangle \end{cases} = (9,11) \\
\mathbf{i}_3 &= \begin{cases} |\Phi_{12}\rangle &= |00\ 0110\rangle \\ |\Phi_{13}\rangle &= |00\ 0101\rangle \end{cases} = (12,13) \\
\mathbf{i}_4 &= \begin{cases} |\Phi_{12}\rangle &= |00\ 0011\rangle \end{cases} = (14,14)
\end{aligned}, \tag{6.6}$$

where the index vectors, $\mathbf{i}_k$, are only of length 2 and the size of the index is $\mathcal{O}(2[N_\phi - 1])$ where N_ϕ is again the number of single-particle states.

Building the Basis

The key function of the `Basis` class is `Build()`, which is for building the basis based on the given configuration parameters. This function is primarily called by the `System` class, but can also be called directly. It takes no input parameters. All parameters need to be set via Set-methods first.

The number of possible unique configurations, N_Ψ, for a system of N_P particles is given as

$$N_\Psi = \binom{N_\phi}{N_P}. \tag{6.7}$$

For the `QDOT2D` case, only the configurations that conform to a given total spin ($2s$) and total m, usually denoted capital M in this thesis, are included into the basis. It is not currently possible to run over all the unique configurations, but there is no reason why the code could not be altered to do this. It is generally not a desirable thing to do as the dimensionality is significantly larger with no selections done on the full basis. It is still a "full basis" in the sense that it contains all the configurations for the diagonal block of the full Hamiltonian that we wish to study.

The code of this function loops over all possible configurations. It was a little challenge in itself to code. The complicating factor here being that it needs to run efficiently for any number of particles. The naive way of doing it is having N_P for loops within each other running over all values of N_ϕ. The obvious problem being the need for one function for each number of particles.

The first working implementation was by use of a recursive algorithm that would assign each particle of a vector of length N_P a single-particle state index i. The function would call itself until there were no more unassigned particles. It would then check the set of occupied single-particle states it had built, and check the quantum numbers for M and $2s$ and keep or reject the configuration based on this. This implementation was relatively efficient, but a more effective method was found as recursive function calls carry an overhead.

The currently implemented algorithm still uses a vector of length N_P, representing all particles, but is otherwise structured differently.

It first assigns all particles to the lowest states that are free, in incrementing order. From there, a while loop takes over and will attempt to move the right-most particle that is not in the highest available state, i.e. while $p < N_P$, into a higher state. If will then move to the left until it finds one particle it cannot move up one state. If the particle that was just moved to a higher state is not the last particle, the following particles are moved to the lowest free single-particle state. When all particles are at the highest possible state, the loop exits.

The while loop builds a basis with a structure like

$$\begin{aligned}
|\phi_0\rangle &= |11\ 1000\rangle \\
|\phi_1\rangle &= |11\ 0100\rangle \\
|\phi_2\rangle &= |11\ 0010\rangle \\
|\phi_3\rangle &= |11\ 0001\rangle \\
|\phi_4\rangle &= |10\ 1100\rangle \\
|\phi_5\rangle &= |10\ 1010\rangle \\
|\phi_6\rangle &= |10\ 1001\rangle \\
|\phi_7\rangle &= |10\ 0110\rangle \\
&\vdots \\
|\phi_{m-2}\rangle &= |00\ 1110\rangle \\
|\phi_{m-1}\rangle &= |00\ 1101\rangle \\
|\phi_m\rangle &= |00\ 0111\rangle\,.
\end{aligned} \tag{6.8}$$

The flow is best viewed by looking at the source code and comments.

```
// Stop if first particle is in highest state allowed
while(vTemp[0] < iStates-iParticles) {
    // Loop through particles from last to first
    for(i=iParticles-1; i>=0; i--) {
        // If there are any higher state available for given particle
        if(vTemp[i] < iStates-iParticles+i) {
            // Move particle up one state
            vTemp[i]++;
            // If this is not the last particle
            if(i < iParticles-1) {
                // Reset the following particles to lowest states available
                for(j=i+1; j<iParticles; j++) {
                    vTemp[j] = vTemp[j-1] + 1;
                }
            }
            break;
        }
    }
    if(fCheckQDot2D(vTemp)) {
        sdTest.Zero();
        for(i=0; i<iParticles; i++) sdTest.Create(vTemp[i]);
        vBasis.push_back(sdTest);
    }
}
```

Listing 6.1: The main while-loop of the basis generator function.

If the `INDEX_BASIS` setting is set in the config file, this function will also generate a map of all Slater determinants that have two specific single-particle states occupied as described earlier, see (6.5).

If the `INDEX_BASIS` is not set, an array containing the intervals for each state will be generated instead, see (6.6).

Searching the Basis

The search algorithm will search through the basis looking for the index corresponding to the Slater object passed to the function.

In the case that the full index is being used, two particles known to exist in the Slater determinant is passed along to the search function, and the search algorithm will only search through a list of those that it knows have those two single-particle states occupied. These are the two particles created in the matrix-vector multiplication section of the Lanczos algorithm.

One of the challenges with the Lanczos algorithm is that the innermost loop needs to look up the generated Slater determinant in the basis and retrieve the index value in order to update the relevant energy in the Lanczos vector. This is a very costly procedure.

As the basis can become very large, this search needs to be very fast. As mentioned, an index is generated when the basis is built. The index of the start and stop position of each of these occupied states are recorded in a $2 \times N_\phi$ matrix (6.6). Any non-existing state will contain a -1 in this index indicating that a lookup is redundant and will break the search, something which also saves CPU time.

The search algorithm itself is of the type "binary search". This algroithm works very well on sorted arrays, and scales on average as approximately $\mathcal{O}(ln(N_\phi))$ [42]. This method however requires an effective algorithm that compare two Slater determinants and return whether the one we are checking against exist before or after that specific point in the basis. A greater than/lesser than function is required. The `bitset` class in the STL library cannot do a numerical comparison if the `bitset` is larger than the size of unsigned long. So another approach is necessary.

The solution, as we covered in 6.2.1, is to look at the difference between the two Slater determinants by using the binary XOR operation.

The binary search itself goes like

```
1  // Binary search
2  int iCheck;
3  int iP    = sdFind.GetFirst(); // Find lowest occupied state
4  int iMin = mIndex(iP,0);       // Set search range based on lowest occupied state
5  int iMax = mIndex(iP,1);
6
7  if(iMin == -1) return -1;      // Returns "not found" if state is never occupied in
       basis
8
9  while(iMax!=iMin) {            // Otherwise do binary search until search interval is 0
10     iCheck = iMin+(iMax-iMin)/2;
11     switch(vBasis[iCheck].Compare(sdFind)) {
12         case  0: return iCheck;
13         case -1: iMax = iCheck;   break;
14         case  1: iMin = iCheck+1; break;
15     }
16 }
17
18 if(iMax == iMin) return iMin; // If only one state exists, return index value
```

Listing 6.2: The binary search function for the `Basis` class.

An example of the search procedure is shown below where a Slater determinant is found in just 4 steps in a four-electron basis.

```
1  Min   192
2  Max   206
3  Check 199
4  Check |00000010010000001100>
```

```
Find  |000000100011001000001>
XOR   |000000000110010011101>
First XOR bit: 9
Compare Result: 1

Min   200
Max   206
Check 203
Check |00000010000101000010>
Find  |00000010001001000001>
XOR   |00000000001100000011>
First XOR bit: 10
Compare Result: -1

Min   200
Max   203
Check 201
Check |00000010001000010100>
Find  |00000010001001000001>
XOR   |00000000000001010101>
First XOR bit: 13
Compare Result: -1

Min   200
Max   201
Check 200
Check |00000010001001000001>
Find  |00000010001001000001>
XOR   |00000000000000000000>
Compare Result: 0

Index found: 200
```

Listing 6.3: An example of a binary search on a basis of Slater determinants.

6.2.3 The Log Class

The `Log` class is a simple log object that makes sure all terminal outputs throughout `libTardis` can also be written to file. This is achieved by sending all outputs to a stringstream object and let the `Log` object handle printing to terminal and/or file or neither. It also provides a simple way to shut down all output from slave nodes when running the library in multi-node mode.

The class itself needs no further explanation. It contains a few set methods for file path and output enable/disable as well as the `Output()` function itself. The log object is normally created by the `System` class and passed on to all child objects and can also be accessed from the wrapper script via `System->GetLog()`.

6.2.4 The System Class

The System class is the main container for the quantum mechanical system the solver is set to work on. It will, upon initiation, create a `Log` object, a `Basis` object and a `Potential` object (see 6.3). This class will forward all system settings to these classes, and will also forward the basis and potential `Build` commands, the log `Output()` commands and so on. We will look at how the `System` object is configured for a test example at the end of this chapter in 6.6.

The class does not really do much other than that. It is essentially a container with a set of Set and Get functions and a few forward calls like `BuildBasis()` and `BuildPotential()`.

6.3 The Potential Classes

The potential classes are the classes holding and generating or loading the various potentials one would want to run `libTardis` for. At the moment only harmonic oscillator potential for two-dimensional quantum dots have been implemented, but the intention and design of this class is such that it can easily be extended to support other types of potentials.

6.3.1 The Potential Super Class

The `Potential` class is a super class holding all the methods and variables used by the subclasses. The super-/subclass structure is used so that one `Potential` object can be created in a given system and any of the available potentials can be loaded into it, or generated on the fly. This avoids unnecessary memory usage and avoids the need to have if statements to check which type of potential is in use each time a lookup is made. Since this class is called extremely often by the Lanczos algorithm, in the very innermost for loop of the matrix-vector function, it is essential that redundant if statements are avoided.

`libTardis`, and thus the `Potential` class, is designed to support five types of potentials:

- `QDot2D`: Electrons in a quantum dot in two dimensions
- `QDot3D`: Electrons in a quantum dot in three dimensions
- `ATOM`: Electrons in an atom
- `NUCLEUS`: Nucleons (neutrons and protons) in an atomic nucleus
- `HNUCLEUS`: Heavy nucleons (strangeness $\neq 0$) in an atomic nucleus

Only the first of these is currently implemented. These classes can be extended to support other types of potentials. One caveat being that the `Potential` class may need alterations to accommodate new functions that may be needed for other potentials. At this point, only the needs for `QDot2D` have been taken into consideration.

6.3.2 The QDot2D Class

This is the class that generates or loads the interaction elements for a harmonic oscillator in two dimensions. When building the two particle Hamiltonian matrix containing all our interaction elements, we first need to choose the method for generating these and whether we want bare or effective interaction elements.

The options available are:

1. The first implementation is based on code by Morten Hjorth-Jensen, which is again based on the analytical expressions from a paper by Anisimovas and Matulis [43]. This is a very slow algorithm, and it becomes almost impossible to use for any large number of shells. It was however the first implementation used and has been left in the code for reference. It can be used by setting the potential type to `Q2D_ANALYTIC`.

2. The next implementation is the effective interaction algorithm discussed in 2.4.2. The code is developed by Simen Kvaal [10, 35] and is implemented and adapted for use with `libTardis`. This method can be used by setting the potential type to `Q2D_EFFECTIVE`.

3. The last implementation is the bare interaction based on the effective interaction algorithm by Kvaal, but where we configure the code to return just the bare Hamiltonian, see 2.4.2 for further details. This method can be used by setting the potential type to `Q2D_NORMAL`.

During development it was discovered that the output result of the Anisimovas and `OpenFCI` codes were not equal even for the same type of matrices. The difference amounts to a factor

$$P = (-1)^{n_p+n_q+n_r+n_s}, \tag{6.9}$$

where n_i is the quantum number n for the four wave functions we are looking at. This is only mentioned for reference as others have already been implementing this generator code into other projects and it may be a source of error.

The Structure of the Hamiltonian

The Hamiltonian matrix is a sparse matrix. Its full dimensionality in for instance 20 shells is $[R(R+1)]^4 \approx 31 \times 10^9$ elements. However the non-zero elements are along the diagonal in block form if we group the columns and rows by total $M = m_1 + m_2$ and total spin $2s = 2\xi + 2\xi$ (Note: spin is represented in integer values, hence the factor 2). For 20 shells, these blocks are up to 1440×1440 in dimensionality.

In order to store the Hamiltonian matrix effectively without using in the area of 250 gigabytes of memory, we only keep the blocks, reducing the required memory to approximately 520 megabytes. However we still need a way to map the full Hamiltonian matrix coordinates to the block diagonal matrix.

Before we generate the Hamiltonian matrix, we build a map of all combinations of two particles that satisfy a given set of M and $2s$. For example for three shells we have a map like the one shown in Table 6.1.

$\|M,2s\rangle$	Configurations								λ
$\|$ -4,-2$\rangle$									0
$\|$ -4, 0$\rangle$	$\|6,7\rangle$								1
$\|$ -4, 2$\rangle$									2
$\|$ -3,-2$\rangle$	$\|2,6\rangle$								3
$\|$ -3, 0$\rangle$	$\|2,7\rangle$	$\|3,6\rangle$							4
$\|$ -3, 2$\rangle$	$\|3,7\rangle$								5
$\|$ -2,-2$\rangle$	$\|0,6\rangle$	$\|6,8\rangle$							6
$\|$ -2, 0$\rangle$	$\|0,7\rangle$	$\|1,6\rangle$	$\|2,3\rangle$	$\|6,9\rangle$	$\|7,8\rangle$				7
$\|$ -2, 2$\rangle$	$\|1,7\rangle$	$\|7,9\rangle$							8
$\|$ -1,-2$\rangle$	$\|0,2\rangle$	$\|2,8\rangle$	$\|4,6\rangle$						9
$\|$ -1, 0$\rangle$	$\|0,3\rangle$	$\|1,2\rangle$	$\|2,9\rangle$	$\|3,8\rangle$	$\|4,7\rangle$	$\|5,6\rangle$			10
$\|$ -1, 2$\rangle$	$\|1,3\rangle$	$\|3,9\rangle$	$\|5,7\rangle$						11
$\|$ 0,-2$\rangle$	$\|0,8\rangle$	$\|2,4\rangle$	$\|6,10\rangle$						12
$\|$ 0, 0$\rangle$	$\|0,1\rangle$	$\|0,9\rangle$	$\|1,8\rangle$	$\|2,5\rangle$	$\|3,4\rangle$	$\|6,11\rangle$	$\|7,10\rangle$	$\|8,9\rangle$	13
$\|$ 0, 2$\rangle$	$\|1,9\rangle$	$\|3,5\rangle$	$\|7,11\rangle$						14
$\|$ 1,-2$\rangle$	$\|0,4\rangle$	$\|2,10\rangle$	$\|4,8\rangle$						15
$\|$ 1, 0$\rangle$	$\|0,5\rangle$	$\|1,4\rangle$	$\|2,11\rangle$	$\|3,10\rangle$	$\|4,9\rangle$	$\|5,8\rangle$			16
$\|$ 1, 2$\rangle$	$\|1,5\rangle$	$\|3,11\rangle$	$\|5,9\rangle$						17
$\|$ 2,-2$\rangle$	$\|0,10\rangle$	$\|8,10\rangle$							18
$\|$ 2, 0$\rangle$	$\|0,11\rangle$	$\|1,10\rangle$	$\|4,5\rangle$	$\|8,11\rangle$	$\|9,10\rangle$				19
$\|$ 2, 2$\rangle$	$\|1,11\rangle$	$\|9,11\rangle$							20
$\|$ 3,-2$\rangle$	$\|4,10\rangle$								21
$\|$ 3, 0$\rangle$	$\|4,11\rangle$	$\|5,10\rangle$							22
$\|$ 3, 2$\rangle$	$\|5,11\rangle$								23
$\|$ 4,-2$\rangle$									24
$\|$ 4, 0$\rangle$	$\|10,11\rangle$								25
$\|$ 4, 1$\rangle$									26

Table 6.1: A map of all configurations for a three-shell system. The configurations are a set of two single-particle states, $|p,q\rangle$, running from 0 to 11. See Figure 2.2 for reference for the positions of these states in our shell model.

Each of the lines in this table represents a $k \times k$ block representing those total quantum numbers. We then proceed to generate all of these sequentially and store them as matrices in an Armadillo field like we described in 4.2.1.

In order to find these elements later, we have define a lookup index, λ, representing the spectrum of possible values of M and $2s$

$$\lambda \equiv \frac{6M + 12R + 2s - 10}{2}, \tag{6.10}$$

where R is the number of shells.

This will convert any value of M and $2s$ contained in our model space to an index running from 0 to $N_{M,2s}$. Table 6.1 also lists these λ values for the example basis in its

right-most column.

We also have two indices, $\mu_1 = pR + q$ and $\mu_2 = rR + s$, that span the length and height of the full Hamiltonian matrix. Since the matrix is symmetrical, these are equal, $\mu = \mu_1 = \mu_2$ for a set of $(p, q) = (r, s)$. We therefore define a lookup matrix of dimensionality $2 \times N_\phi^2$, where N_ϕ is the number of single-particle states. The indices of the columns of this lookup matrix corresponds to our μ indices, and the first row holds a λ index for that configuration of $|p, q\rangle$ or $|r, s\rangle$ pointing to which block the interaction element is stored in. The second row contains the index, ν, of the location within that block where the element can be found.

If this seems tricky to follow, take a look at the lookup function that retrieves an interaction element for a given p, q, r and s in Listing 6.4.

```
1  double QDot2D::Get2PElement(int p, int q, int r, int s) {
2
3      if(!bCache) return fCalcElementQ2D(p, q, r, s);
4
5      int iLambda1 = mMap(p*iStates+q,0);
6      int iLambda2 = mMap(r*iStates+s,0);
7      if(iLambda1 == iLambda2) {
8          return mBlHam(iLambda1)(mMap(p*iStates+q,1),mMap(r*iStates+s,1));
9      } else {
10         return 0.0;
11     }
12
13 }
```

Listing 6.4: The interaction element lookup function.

Energy Cut

There also exists an option for the potential called energy cut. It only applies to effective interactions. When this setting is enabled, the elements in the Hamiltonian matrix for which the sum of one-particle energies exceeds the number of shells, the value is set to 0.

That is to say that if

$$2(n_1 + n_2) + |m_1| + |m_2| \geq R, \tag{6.11}$$

then the element is set to 0.

This further truncates the model space generated by the effective interaction algorithm, and for the two-particle case, it is an exact Hamiltonian matrix with lowest eigenvalue $E_0 = 3$.

6.4 The Solver Classes

The `Solver` classes are the core algorithms in the code. They diagonalise our Hamiltonians and return our eigenvalues and lowest approximate eigenvector.

6.4.1 The Diag Class

The `Diag` class is straight forward. It will take a block of our two-particle Hamiltonian matrix generated by the `QDot2D` class and diagonalise it by use of the `eig_gen()` function in Armadillo. This returns the eigenvalues and eigenvectors directly and thus solves the Schrödinger equation in one quick step.

This class serves as a reference algorithm for the Lanczos algorithm as the Lanczos algorithm should return the same results in the two-particle case.

6.4.2 The Lanczos Class

The main solver class is however the `Lanczos` class. We will not go into details as to how the algorithm works as this is already covered in Chapter 5. However, we will look briefly at a couple of functions.

The main function is ordinarily the function `Run()`. It will execute the Lanczos algorithm in its entirety and spawn the on-node parallel processes. However, there also exists a split-up version of this function intended for multi-node parallelisation that divides the `Run()` function into

- `RunInit()`, which will initiate the Lanczos process and set up all the run-time variables;
- `RunSlave()`, which will run one Lanczos iteration over an interval of the basis; and
- `RunMaster()`, which will collect all the pieces from the slave processes and combine them and finish the Lanczos iteration, check for convergence and prepare next iteration if one is needed.

A functioning wrapper code utilising these functions is provided in Appendix A, A.2.

The Matrix-Vector Product

As mentioned, the main benefit of using the Lanczos algorithm for our problem is that the Hamiltonian matrix, $\hat{H}$, does not need to be stored in memory. Instead we re-create each possible non-zero element in this matrix on the fly on a column-by-column basis. Suppose that our matrix $\mathbf{A}^{(n\times n)}$ is made up of a series of columns $[\mathbf{a}_0, \mathbf{a}_1, \cdots, \mathbf{a}_n]$ of length n. $\mathbf{w}$ is our updated vector from the previous Lanczos iteration, and $\mathbf{v}$ is the new temporary vector we are about to update. Our matrix-vector product can be expanded straight forwardly as

$$\mathbf{v}_i = \mathbf{v}_i + \sum_{j=0}^{n} a_{ij} w_j. \tag{6.12}$$

Since we do not know how $\mathbf{a}_i$ looks like, we generate this vector on the fly. Each index i represents a Slater determinant in our basis. Each element in the vector $\mathbf{a}_i$ represents the energy difference between this element and all possible configurations that can arise

by annihilating and creating two particles. This is how we calculate our sum of two-body interactions. This can easily, though at a high computational cost, be increased to three-body, or more, interactions.

All the possible two-particle changes to our basis are generated by looping over all unique set of two annihilation operators, and then all possible unique set of two creation operators

$$|\psi_j\rangle = \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_s \hat{a}_r |\psi_1\rangle , \tag{6.13}$$

where $p < q$ and $s > r$ to ensure the operations are unique.

Of course, any illegal operation will result in a factor 0, in which case the for loops will exit to avoid further, redundant iterations.

The source code of the matrix-vector function is listed in Listing 6.5 for reference.

```
void Lanczos::fMatrixVector(Col<double> &mInput, Col<double> &mReturn, double d1PFac,
    double d2PFac) {

    int    i, p, q, r, s;
    int    iS1, iS2, iS3, iS4, iL=0;
    double dV;
    Slater sdPhiPQSR, sdPhiQSR, sdPhiSR, sdPhiR;

    for(i=0; i<iDimBasis; i++) {
        for(r=0; r<iStates; r++) {
            sdPhiR = oBasis->GetSlater(i);
            iS1 = sdPhiR.Annihilate(r);
            if(iS1 == 0) continue;
            for(s=r+1; s<iStates; s++) {
                sdPhiSR = sdPhiR;
                iS2 = sdPhiSR.Annihilate(s);
                if(iS2 == 0) continue;
                for(q=0; q<iStates; q++) {
                    sdPhiQSR = sdPhiSR;
                    iS3 = sdPhiQSR.Create(q);
                    if(iS3 == 0) continue;
                    for(p=0; p<q; p++) {
                        sdPhiPQSR = sdPhiQSR;
                        iS4 = sdPhiPQSR.Create(p);
                        if(iS4 == 0) continue;
                        iL = oBasis->FindSlater(sdPhiPQSR);
                        if(iL > -1) {
                            dV = 0.0;
                            if(p == r && q == s) dV += oPot->Get1PElement(p,s)*d1PFac;
                            dV += oPot->Get2PElement(p,q,r,s)*d2PFac;
                            mReturn(iL) += iS1*iS2*iS3*iS4*dV*mInput(i);
                        }
                    }
                }
            }
        }
    }

    return;
}
```

Listing 6.5: The matrix-vector product function used for the implementation of the Lanczos algorithm in `libTardis`. The additional compiler settings and variables used for parallelisation has been edited out for the sake of readability. `oPot` is the object holding the matrix elements for the harmonic oscillator potential (or any other potential for that matter), and the variables

d1PFac and d2PFac hold the λ or ω values depending on which way one chooses to define interaction strength. The code supports both methods.

6.5 Python Wrapper and Scripts

A set of Python scripts come with libTardis. Several of these are specially designed to compile and run jobs on our super computer *Abel*, so these are of little general interest, but there are two scripts that are intended to generate and to execute a sequence of jobs on a single host computer. We will discuss these briefly.

6.5.1 The Job Generating Script

The first script we will look at is a simple script, named makeJobs.py, which in its top have all configurable variables defined as arrays.

```
aShells     = [4,5,6]
aParticles = [8]
aM          = [0]
aMs         = [0]
aOmega      = [0.1,1.0]
aLambda     = [0.0]
aEnergyCut = [False]
aEffective = [True,False]
aMethod     = [0]

bCalcCoeff = False
sCoeffPath = "/scratch/Temp/Coeff/"

for bEffective in aEffective:
    ....
```

Listing 6.6: The first part of makeJobs.py

The example in Listing 6.6 shows a setup for generating wrapper code for a job for 8 electrons in 4, 5 and 6 shells with ω-values of 0.1 and 1.0 with both bare and effective interactions. In total 12 jobs.

When executed, one cpp file will be generated for each of these 12 systems. An example of such a file can bee found in Appendix A, A.1.

6.5.2 The Sequential Run Script

The second key script, runJobs.py, will simply take all the files generated by makeJobs.py and sequentially compile them and execute them. It will output a log file with a date stamp and the results, and store the run-time output, i.e. the Lanczos output, in a separate file stored in a directory tree by year, month, day and time of day.

6.6 Code Usage Example

Finally, we will provide a simple example of how a job can be run with libTardis. The example is provided in Listing 6.7 and should be relatively self-explanatory, but let us

make a few points.

- `SetPotential` takes the number of shells, here 6, the potential type and the interaction type as input.
- `SetQNumber` will set the total M (`QN_M`) and the total spin, $2s$ (`QN_MS`).
- Either specify an ω or a λ value for the harmonic oscillator frequency or interaction strength and set the other one to 0. Setting both to 0 will disable the interaction part of the Hamiltonian and only use the non-interacting Hamiltonian with an ω-value of 1.
- Build the potential and the basis,
- and run the Lanczos algorithm.

```
#include "../libTardis/libTardis.hpp"

using namespace std;
using namespace tardis;

int main(int argc, char* argv[]) {

    System *oSystem = new System();

    oSystem->SetPotential(6, QDOT2D, Q2D_EFFECTIVE);
    oSystem->SetParticles(8);
    oSystem->SetQNumber(QN_M, 0);
    oSystem->SetQNumber(QN_MS, 0);
    oSystem->SetVariable(VAR_LAMBDA, 0);
    oSystem->SetVariable(VAR_OMEGA, 1.0);
    oSystem->EnableEnergyCut(false);
    oSystem->BuildPotential();
    oSystem->BuildBasis();

    Lanczos oLanczos(oSystem);
    double dEnergy = oLanczos.Run();

    cout << "Energy:" << dEnergy << endl;

    return 0;
}
```

Listing 6.7: Minimal example of how to run a calculation with `libTardis`.

7
Code Performance and Benchmarks

"More computing sins are committed in the name of efficiency without necessarily achieving it than for any other single reason –including blind stupidity."

– W. A. Wulf

Throughout the code development stage of `libTardis`, great attention has been given to the performance of each and every module and class. As we have seen in 5.2.1, the binary representation of the basis performs well due to the usage of binary operations. Throughout code development, several approaches to solving various programming tasks were tested and the most efficient ones chosen. Most of the development history of this is not retained, but both the original and improved basis-search algorithm have been kept in the source code and can be switched through the compile setting in the config file. This is covered in more detail in 6.2.2.

In this chapter, however, we will look at how the Lanczos algorithm converges and the various challenges related to this. We will also take a look at how the code scales when parallelised through the use of OpenMP and shared memory technology as well as OpenMPI. Lastly we will look at the computation times for the larger jobs `libTardis` has been tested with.

7.1 Convergence Properties of the Lanczos Algorithm

The Lanczos algorithm, described in Chapter 5, is an algorithm that iteratively builds a tri-diagonal matrix whose extremal eigenvalues will converge to those of the larger matrix we are attempting to analyse. It is an algorithm well suited to solving our type of problem of very large Hamiltonian matrices [19].

As the algorithm runs through its iterations, the eigenvalues, most importantly for us the lowest eigenvalue, will start to converge at a stable value. In Chapter 8 the accuracy of `libTardis` is compared to other codes performing the same type of calculations. The main reference articles, the ones by S. Kvaal [35] and M. Rontani *et al.* [44] give energy with a numerical precision of 6 and 4 figures respectively.

7.1.1 The Lowest Eigenstate and Error Estimates

In order to evaluate whether we have reached a good ground state energy for our Hamiltonian, we calculate its convergence.

The convergence criterion for the Lanczos algorithm in `libTardis` is given as

$$\delta_k = \left| \frac{E_0^{(k-1)}}{E_0^{(k)}} \right| - 1 < \delta_C, \tag{7.1}$$

where δ_C is the convergence criterion specified in `config.hpp` and $k > 1$ is the current Lanczos iteration.

The default convergence criterion for all results, unless otherwise specified, is set to $\delta_C = 1 \times 10^{-6}$.

During testing by reproducing the results of table II in the OpenFCI paper [35] (see also 8.2), one configuration caused some trouble with convergence. Specifically, 4 electrons in 6 shells with $\lambda = 6$, $M = 0$ and $2s = 0$. The initial run produced the results displayed in Table 7.1.

Source	E_0
Kvaal [35]	23.68944
Rontani *et al.* [44]	23.691
`libTardis`	23.638717

Table 7.1: The lowest eigenvalue for a test run with 4 electrons in 6 shells with $\lambda = 6$, $M = 0$ and $2s = 0$ with bare interactions.

This configuration was the only one of the initial test runs that did not immediately reproduce the results from Kvaal [35]. In order to understand why, let us first take a look at the terminal output in Listing 7.1:

```
....
Lanczos Iteration 21 : Energy = 23.68530500 : Convergence = 7.55e-05 : Error = 0.115
Lanczos Iteration 22 : Energy = 23.68417467 : Convergence = 4.77e-05 : Error = 0.0935
Lanczos Iteration 23 : Energy = 23.68340038 : Convergence = 3.27e-05 : Error = 0.0744
Lanczos Iteration 24 : Energy = 23.68292383 : Convergence = 2.01e-05 : Error = 0.0613
Lanczos Iteration 25 : Energy = 23.68259947 : Convergence = 1.37e-05 : Error = 0.0516
Lanczos Iteration 26 : Energy = 23.68235539 : Convergence = 1.03e-05 : Error = 0.0462
Lanczos Iteration 27 : Energy = 23.68214481 : Convergence = 8.89e-06 : Error = 0.0455
Lanczos Iteration 28 : Energy = 23.68191390 : Convergence = 9.75e-06 : Error = 0.0507
Lanczos Iteration 29 : Energy = 23.68159029 : Convergence = 1.37e-05 : Error = 0.0650
Lanczos Iteration 30 : Energy = 23.68104645 : Convergence = 2.30e-05 : Error = 0.0820
....
```

Listing 7.1: Segment of the terminal output for the Lanczos algorithm.

The first value to hit a convergence of $\delta_{27} = 8.89 \times 10^{-6}$ does fit the results given by both [35] and [44], however this is not the end of it, the Lanczos iterations continue, and the estimated ground state energy continues to fall.

This *may* indicate that a too relaxed convergence criterion could give too high an energy result. To investigate further, the convergence criterion was lowered to $\delta_C = 1 \times 10^{-10}$ and the problematic configuration run again. The convergence of this result with error bars included can be seen in Figure 7.1. For a description of how error is estimated, see 5.1.

Further, it is interesting to note that the ground state energy reported by Kvaal is the same as the first excited state, E_1, of the output from `libTardis` as shown in Table 7.2 below.

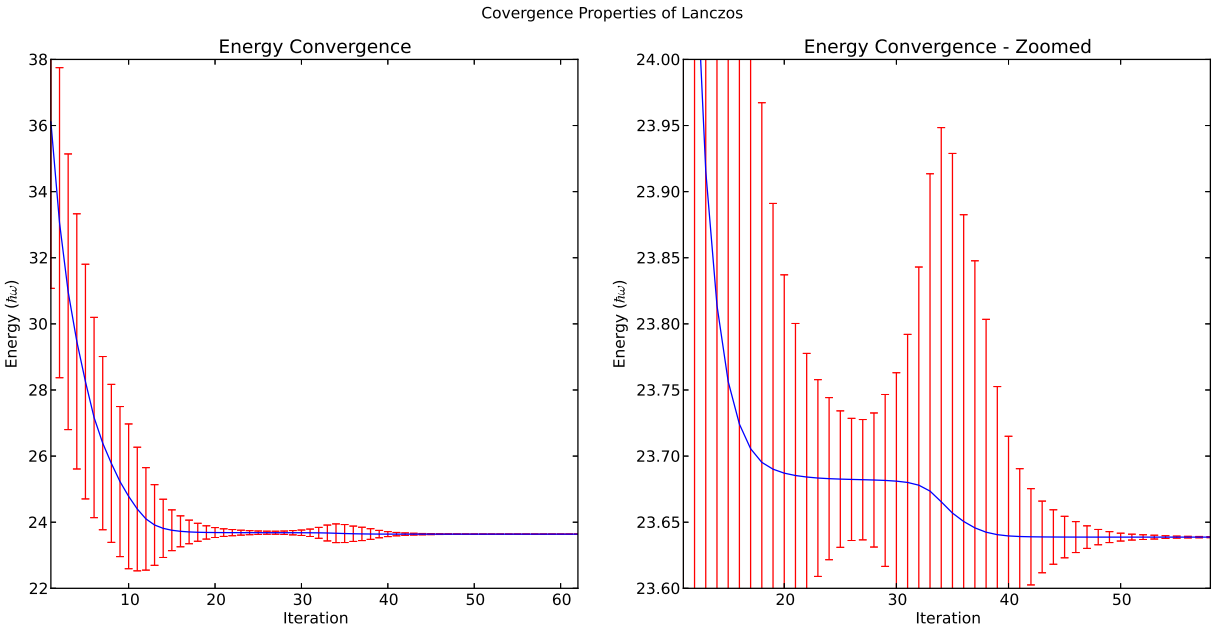

Figure 7.1: The convergence properties of a system of 4 electrons in 6 shells with $\lambda = 6$, $M = 0$ and $2s = 0$. The plot illustrates how there may be more than one "shelf" where the eigenvalues seem to converge. The higher shelf converges at $\delta_k < 1 \times 10^{-5}$, while the lower shelf converges at around $\delta_k < 1 \times 10^{-8}$ with a value of 23.638717.

E_n	Energy	Error
E_0	23.638717	1.21e−4
E_1	23.689441	6.70e−5
E_2	24.471876	3.61e−1
E_3	24.556409	2.47e−1
E_4	24.791066	2.58e−1
E_5	25.016846	3.28e−1
E_6	25.424942	2.61e−1
E_7	25.724992	7.04e−1

Table 7.2: The 8 lowest eigenstates of a system of 4 electrons in 6 shells with $\lambda = 6$, $M = 0$ and $2s = 0$. Presented with error estimates.

As we can see from Table 7.2, both E_0 and E_1 are good eigenvalues. Figure 7.1 also indicates that our E_0 is the correct ground state. It is not necessarily the case that the Lanczos algorithm will converge towards the final lowest eigenvalue at all times during its iterations. See the discussion of the convergence properties of the Lanczos algorithm in 5.1. This example is a clear case of such an incident, and throughout the numerous runs presented in this thesis, this has indeed happened several times.

7.1.2 Effective Interactions and Convergence

It is a general trend that effective interactions converge faster than bare interactions. However, this is not always the case. Temporary and partial convergence to "wrong" ground state energies, meaning in actuality an excited state, may cause the convergence rate to flatten out and cause some configurations to have worse convergence properties. These may be different for bare and effective interactions even for the same configurations.

It is also a general trend that lower values of ω converge slower than higher values. Again, this is not always the case and convergence to "wrong" eigenvalues may again interfere with this process.

Figures 7.2 and 7.3 both illustrate these behaviours. In these figures $\omega = 0.01$ tend to converge faster than $\omega = 0.1$, and in three of the four plots $\omega = 0.1$ seems to struggle more to find the lowest eigenvalue than for other values of ω and thus the algorithm requires more iterations to converge. (The lines are more wiggly.)

Based on this and the previous section, we may question the use of convergence as the sole criterion for making decision to terminate the Lanczos algorithm. This is what has been done for most of the simulations in this project, but it may be wise to look further into this in the future. The inclusion of the error estimate can clearly be helpful in determining whether we have found the true ground state or not. This is not a very large problem for this thesis though as the results are usually compared to results by other algorithms and discrepancies are spotted relatively easily.

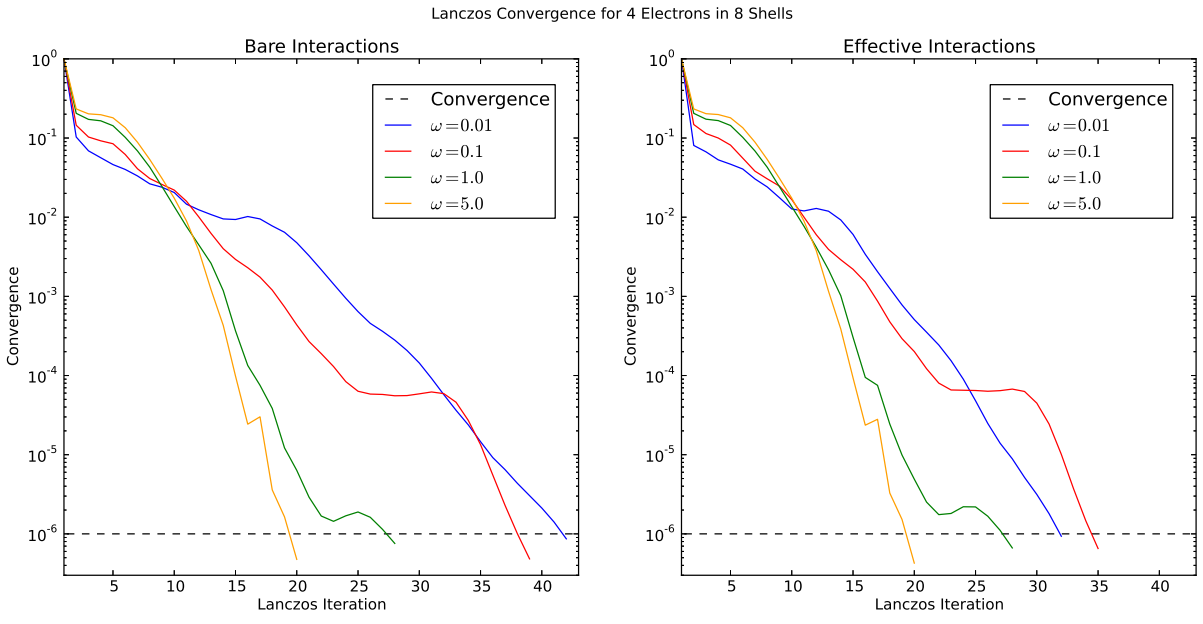

Figure 7.2: The convergence properties of the Lanczos algorithm using bare and effective interactions on a system of 4 electrons in 8 shells with a set of ω-values. Convergence is defined in the code as $\left(E_0^{(k-1)}/E_0^k\right) - 1$. The plot's y-axis displays the 10-logarithm of the convergence for each iteration of the Lanczos process.

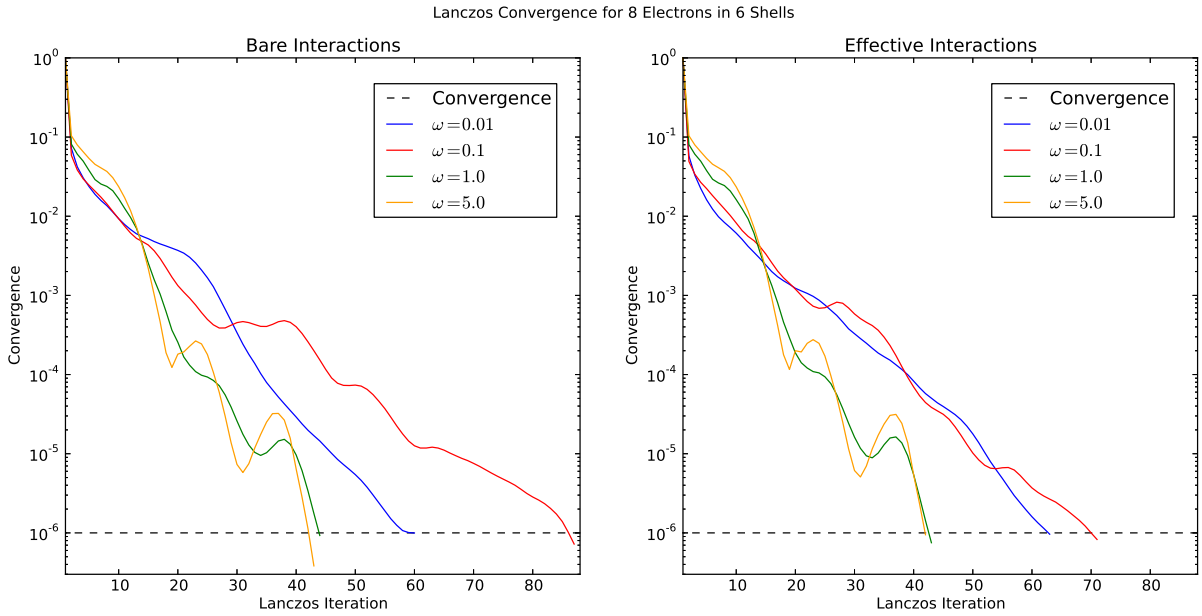

Figure 7.3: The convergence properties of the Lanczos algorithm using bare and effective interactions on a system of 8 electrons in 6 shells with a set of ω-values. Convergence is defined in the code as $\left(E_0^{(k-1)}/E_0^k\right) - 1$. The plot's y-axis displays the 10-logarithm of the convergence for each iteration of the Lanczos process.

7.2 Parallelisation and Scalability

At first it was decided against implementing a multi-node parallelisation of `libTardis` due to time constraints. Late in the project it was implemented anyway, and as a result much larger simulations have been run spending as much as 450 000 CPU hours on the new super computer at the University of Oslo.

7.2.1 OpenMP Parallelisation

The original code was parallelised by use of a straight forward implementation of OpenMP. Due to the advantages of shared memory technology, a significant boost in performance was gained. As can be seen from Figure 7.4, a near perfect linear scaling has been achieved.

Though the process of generating the two-particle interaction elements, and then building and filtering the basis, takes some processing time, the majority of the CPU time is used by the Lanczos algorithm. It is therefore there the parallelisation is implemented. Even if the code is to run on several nodes, the basis and the interaction elements needs to be available on each node. Whether the other nodes wait for these data to be generated or loaded and then distributed, or each node generates or loads them itself, makes no difference time-wise.

The implementation of the parallelisation itself is done by extracting the matrix-vector multiplication portion of the Lanczos algorithm (see Listing 5.2) and moving it

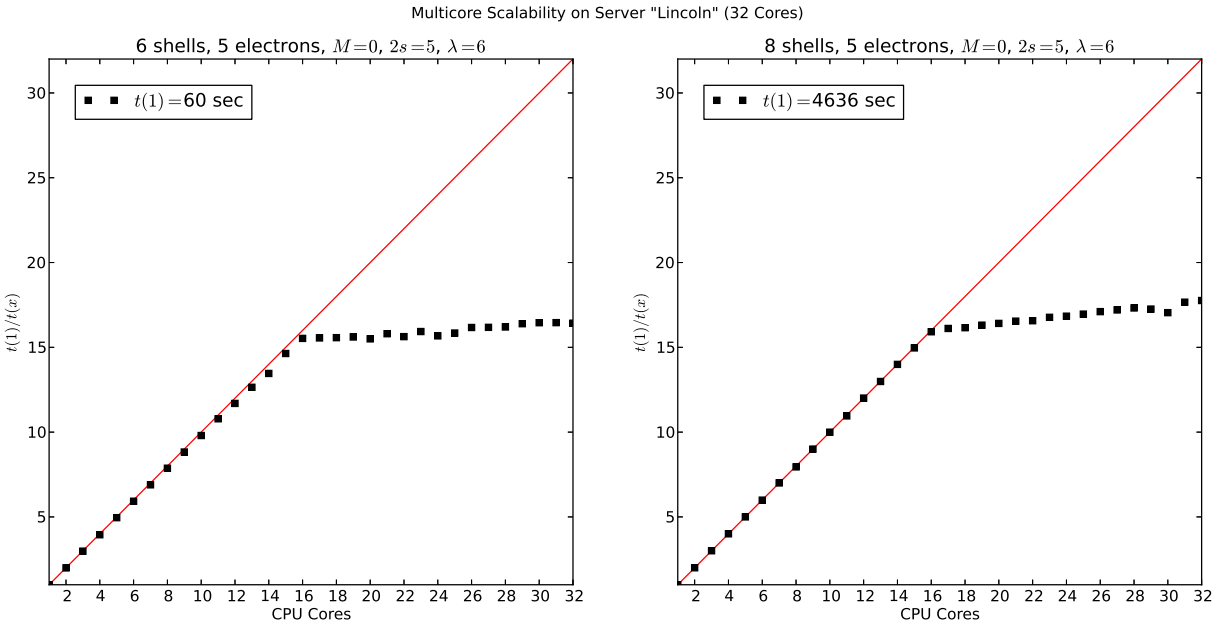

Figure 7.4: Multi-core scalability on the computer "Lincoln" with the optimised-for-speed version of the matrix-vector function of the Lanczos algorithm. The reason the curve flattens out from 17 cores is due to the hyper-threading kicking in. All 16 physical cores are running at maximum capacity, and hyper-threading is not able to squeeze any more power out of these cores. The line in the plots represents a theoretical linear scaling of the code's performance as a function of cores. The squares are the inverse of the actual computation times divided by the time for only one core, thus showing how many times faster n cores is than 1 core.

into a separate function named `fMatrixVector`. The Lanczos vector, `mW`, is passed along, and so is a temporary vector, `mT`. The temporary vector is initially zeroed out, and the values accumulated are returned and added to the other, waiting Lanczos vector `mV` upon completion of the function's internal for loops (representing one iteration of the Lanczos algorithm itself).

```
1  void Lanczos::fMatrixVector(Col<double> &mInput, Col<double> &mReturn, double d1PFac,
       double d2PFac) {
2
3      int    i, p, q, r, s;
4      int    iS1, iS2, iS3, iS4, iL=0;
5      double dV;
6      Slater sdPhiPQRS, sdPhiQRS, sdPhiSR, sdPhiR;
7
8      #ifdef OPENMP
9          #pragma omp parallel for private(p,q,r,s,iS1,iS2,iS3,iS4,sdPhiPQRS,sdPhiQRS,
               sdPhiSR,sdPhiR,iL,dV) schedule(dynamic,1)
10     #endif
11     for(i=0; i<iBasisDim; i++) {
12         for(r=0; r<iStates; r++) {
13             sdPhiR = oBasis->GetSlater(i);
14             iS1 = sdPhiR.Annihilate(r);
15             if(iS1 == 0) continue;
16             for(s=r+1; s<iStates; s++) {
17                 sdPhiSR = sdPhiR;
18                 iS2 = sdPhiSR.Annihilate(s);
19                 if(iS2 == 0) continue;
```

```
20                  for(q=0; q<iStates; q++) {
21                      sdPhiQRS = sdPhiSR;
22                      iS3 = sdPhiQRS.Create(q);
23                      if(iS3 == 0) continue;
24                      for(p=0; p<q; p++) {
25                          sdPhiPQRS = sdPhiQRS;
26                          iS4 = sdPhiPQRS.Create(p);
27                          if(iS4 == 0) continue;
28                          iL = oBasis->FindSlater(sdPhiPQRS,p,q);
29                          if(iL > -1) {
30                              dV = 0.0;
31                              if(p == r && q == s) dV += oPot->Get1PElement(p,s)*d1PFac;
32                              dV += oPot->Get2PElement(p,q,r,s)*d2PFac;
33                              #ifdef OPENMP
34                                  #pragma omp critical
35                              #endif
36                              mReturn(iL) += iS1*iS2*iS3*iS4*dV*mInput(i);
37                          }
38                      }
39                  }
40              }
41          }
42      }
43
44      return;
45  }
```

Listing 7.2: Lanczos algorithm with first implementation of `OpenMP`

The lines starting with `#pragma` are the `OpenMP` commands for the compiler. Notice especially the one occurrence at line 34. The `critical` command signifies that this is a critical point. Since all processes are writing their results back to the vector `mReturn`, there needs to be some level of control with what process writes when so that they do not erase each other's input. Even if each iteration only adds a sum, it matters what number is summed with, and it matters that this number is not changed by one process while another process is performing the summation. Initial tests without this option revealed that this conflict was indeed occurring.

However, further testing revealed that with this compiler setting, the queueing would become problematic when there was many processes queueing up to write the data at the same time. The performance of the code scaled relatively well up to three to four cores running on *Gizmo*, but performance would drop significantly with five to eight cores, often ending up under-performing even one process.

The solution to this problem was to create a vector of `mT` vectors, one for each process, thus letting each thread write to its own memory area and having the main function sum them up afterwards.

The difference in performance of these two methods are clearly visible in Figure 7.5 where a fast and a slow job is compared with processing time for one core running up to eight cores.

7.2.2 OpenMPI Parallelisation

The implementation of multi-node parallelisation is not included directly in `libTardis`, although the Lanczos algorithm method call is available in both single and multi-node form (see 6.4.2), the MPI code must be provided by a wrapper script. The script used for large scale calculations have been included as Listing A.2 in Appendix A.

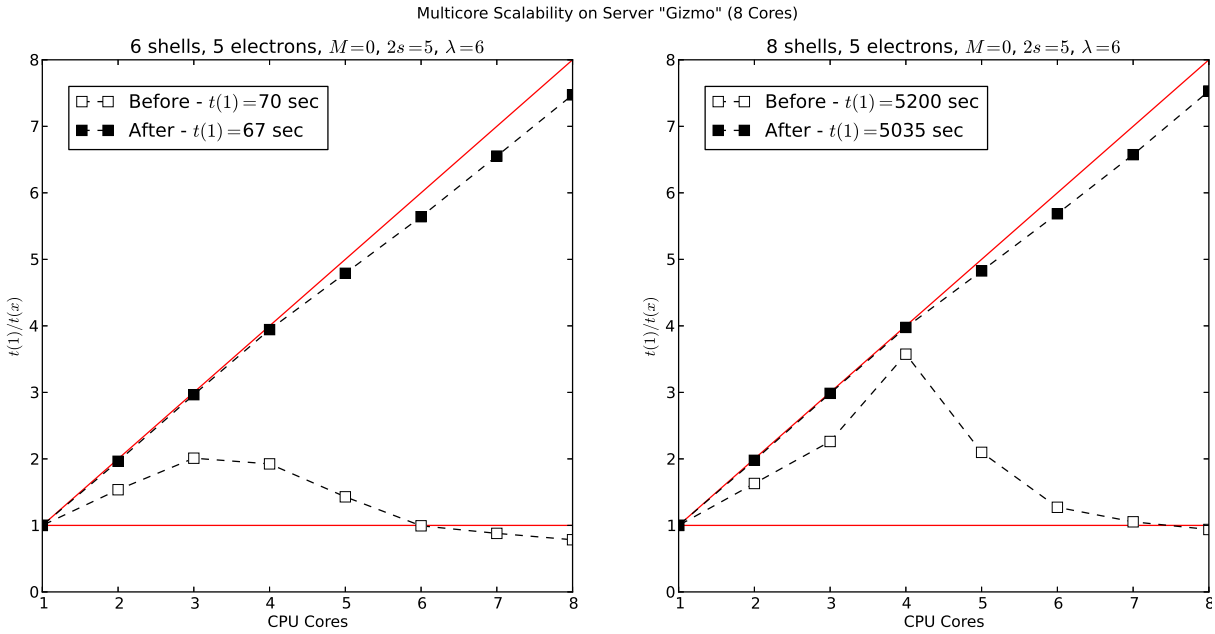

Figure 7.5: Multi-core scalability of `libTardis` on the computer *Gizmo* before and after optimisation of queueing of the main Lanczos vector. The diagonal line in the plots represents a theoretical linear scaling of the code's performance as a function of cores. The horizontal line represents a scaling of one. The squares are the inverse of the actual computation times divided by the time for only one core, thus showing how many times faster n cores is than one core.

In this chapter, however, we are interested in the performance of such a parallelisation. The plot in Figure 7.6 represents a relatively small test of the MPI implementation used for this thesis. Although the hardware specifications of the nodes on *Abel* are uniform, the actual performance seems to be slightly better than optimal. This may be due to the load of the super computer at the time of the test, but the main point is that the code scales close to linearly even on multiple nodes. It has to be said here though that the system chosen is small enough that communication time is negligible.

7.3 Computation Times

Finally, we will take a look at the actual computing times for the large scale simulations made for this project. Most of the large scale calculations have been run on the super computer *Abel*, but some of the larger calculations run on single node computers have been included as well for the sake of comparison and completeness of the list.

Table 7.3 lists all the major computations. We will get back to the actual results in Chapter 8, but for now it is the actual CPU hours that are of interest. The job labels are descriptive enough that it should be easy to cross-reference them with the results tables in Chapter 8 if that is desired.

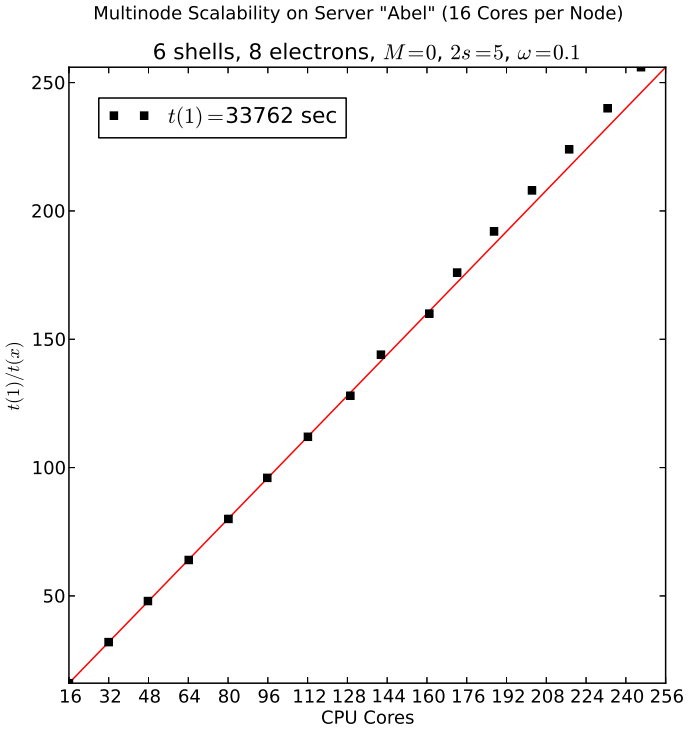

Figure 7.6: The performance of `libTardis` when distributed across multiple nodes on a super computer. The inter-node communication is very fast, and for the relatively small system of 8 electrons in 6 shells does not generate enough traffic to be a significant factor. The line in the plot indicates a linear performance scale, and the squares are the actual computation times. Each node has 16 cores, hence the step-length of 16 along the axes.

The third column lists the dimensionality of the basis (after selection, see 6.2.2) and together with column four listing the number of iterations the Lanczos algorithm needed to achieve convergence, are the main influences on the time consumed by the given calculation. One more factor that plays a role is the number of shells, or rather the number of single-particle states, given by $R(R+1)$, and how many sets of 64 bit is needed to represent the Slater determinant. As discussed in 5.2.1, this affects the performance of the operations on the Slater determinants and thus the total performance of the Lanczos algorithm.

Column five lists the number of cores used, or CPUs if you like. For the simulations run on *Abel*, the number of nodes can be extracted by dividing this number by 16 as all nodes have 16 physical cores (32 when hyper-threading is enabled). The wall times listed are the total time from start to finish of the job, and the CPU time is simple the product of wall and the number of cores. The total CPU hours on *Abel* in this table is in the 440 000 hours range. The list only contains the single configuration jobs. Others have been performed too to generate the plots in Chapter 9. They are listed in Tables B.1 and B.2 in Appendix B together with the data points for those plots.

Job Description	Computer	Dim(Basis)	It.	Cores	Wall Time	CPU Time
P11 Sh5 M2 2s1 ω0.1 E	Abel	9.45e5	56	160	00:11:22	30 h
P7 Sh6 M0 2s1 ω0.01 E	TheBeast	4.88e5	79	6	05:22:25	32 h
P7 Sh6 M0 2s1 ω0.1 E	Gizmo	4.88e5	44	8	05:44:00	46 h
P13 Sh5 M2 2s1 ω0.1 E	Abel	1.98e6	89	160	00:54:38	147 h
P7 Sh7 M2 2s1 ω0.1 E	Abel	3.37e6	52	160	01:49:18	291 h
P6 Sh8 M0 2s0 ω0.28 E	Gizmo	2.46e6	34	8	1:17:18:10	330 h
P6 Sh8 M0 2s0 ω1.0 E	Abel	2.46e6	29	320	01:10:43	377 h
P7 Sh7 M0 2s1 ω0.1 E	Abel	3.48e6	72	160	02:39:03	424 h
P6 Sh8 M0 2s0 ω0.1 E	Gizmo	2.46e6	49	8	2:22:21:36	563 h
P4 Sh14 M0 2s0 ω0.1 B	Abel	9.94e5	48	160	04:59:36	799 h
P6 Sh8 M0 2s0 ω0.01 E	Abel	2.46e6	96	160	07:28:14	1 195 h
P6 Sh8 M0 2s0 ω0.01 B	Abel	2.46e6	121	160	09:24:19	1 505 h
P6 Sh9 M0 2s0 ω1.0 E	Abel	8.62e6	31	400	06:06:58	2 446 h
P6 Sh9 M0 2s0 ω0.1 E	Abel	8.62e6	45	320	10:23:55	3 328 h
P4 Sh16 M0 2s0 ω0.1 B	Abel	2.46e6	53	320	13:09:09	4 209 h
P6 Sh9 M0 2s0 ω0.01 E	Abel	8.62e6	108	320	1:01:43:21	8 232 h
P11 Sh6 M0 2s1 ω0.1 E	Abel	5.76e7	79	800	14:09:24	11 325 h
P6 Sh10 M0 2s0 ω1.0 E	Abel	2.65e7	34	640	21:11:13	13 560 h
P12 Sh6 M0 2s0 ω1.0 E	Abel	1.49e8	37	960	17:59:35	17 273 h
P4 Sh18 M0 2s0 ω0.1 B	Abel	5.50e6	62	480	1:16:53:16	19 626 h
P6 Sh10 M0 2s0 ω0.1 E	Abel	2.65e7	53	640	1:08:08:56	20 575 h
P12 Sh6 M0 2s0 ω0.1 E	Abel	1.49e8	73	1280	1:03:42:36	35 469 h
P6 Sh11 M0 2s0 ω0.1 E	Abel	7.33e7	50	640	3:23:47:51	61 310 h
P4 Sh20 M0 2s0 ω0.1 E	Abel	1.13e7	59	640	4:13:03:41	69 799 h
P4 Sh20 M0 2s0 ω0.1 B	Abel	1.13e7	62	640	5:01:55:12	78 029 h
P13 Sh6 M2 2s1 ω0.1 E	Abel	3.13e8	79	*1600	2:17:04:01	92 894 h

Table 7.3: Run-times for some of the major jobs. The job column describes jobs by number of particles (P), number of shells (Sh), total M (M), total spin times two (2s) and omega (ω). 'E' indicates effective interactions, while 'B' indicates bare interactions. The column labelled 'It.' lists the number of Lanczos iterations needed for convergence as this number highly affects computation time. All runs have $< 1 \times 10^{-6}$ as convergence criterion.
* Ran on 1280 cores for part of the calculation.

Communication Overhead

Lastly, a few comments are needed on communication overhead.

At first the implementation of OpenMPI was straight forward. The parallelisation is as on-node by OpenMP, done in the outer for loop of the matrix-vector function by selecting an interval of the full basis to run through. The first attempt was to simply divide this number by the number of nodes. This approach worked well enough for smaller jobs, but it was discovered, not surprisingly, that for larger configurations, each interval did not require the same time to compute. In fact the slowest node could in some cases use 50% more time than the fastest node resulting in the other nodes having to wait for the last one to complete its course.

Job Description	Calc	Comm	Wait	Merge
P11 Sh5 M2 2s1 ω0.1 E	96.63%	3.37%	–	–
P13 Sh5 M2 2s1 ω0.1 E	97.20%	2.80%	–	–
P7 Sh7 M2 2s1 ω0.1 E	98.55%	0.23%	1.18%	0.05%
P6 Sh8 M0 2s0 ω1.0 E	99.46%	0.54%	–	–
P7 Sh7 M0 2s1 ω0.1 E	98.29%	0.45%	1.18%	0.08%
P4 Sh14 M0 2s0 ω0.1 B	99.31%	0.04%	0.64%	0.02%
P6 Sh8 M0 2s0 ω0.01 E	99.53%	-0.05%	0.52%	0.00%
P6 Sh8 M0 2s0 ω0.01 B	99.55%	-0.05%	0.50%	0.00%
P6 Sh9 M0 2s0 ω1.0 E	98.61%	1.39%	–	–
P6 Sh9 M0 2s0 ω0.1 E	99.21%	0.19%	0.60%	0.01%
P4 Sh16 M0 2s0 ω0.1 B	99.00%	0.09%	0.90%	0.01%
P6 Sh9 M0 2s0 ω0.01 E	99.30%	0.14%	0.56%	0.00%
P6 Sh10 M0 2s0 ω1.0 E	97.83%	2.17%	–	–
P4 Sh18 M0 2s0 ω0.1 B	98.67%	0.04%	1.29%	0.00%
P6 Sh10 M0 2s0 ω0.1 E	97.73%	2.27%	–	–
P12 Sh6 M0 2s0 ω0.1 E	82.69%	17.31%	–	–
P6 Sh11 M0 2s0 ω0.1 E	98.08%	0.35%	1.57%	0.00%
P4 Sh20 M0 2s0 ω0.1 E	94.89%	0.09%	4.02%	0.00%
P4 Sh20 M0 2s0 ω0.1 B	97.30%	0.06%	2.64%	0.00%
P13 Sh6 M2 2s1 ω0.1 E	89.71%	5.81%	4.48%	0.00%

Table 7.4: An overview of the efficiency of the OpenMPI implementation. The numbers are based on the output files. One early version of the code did not output each node's calculation time (Calc), so wait time (Wait) and merge time (Merge) is unknown. This unknown time is baked into between calculation time and communication time (Comm). For the other cases, the calculation time is accurate, so in case of negative numbers elsewhere, this is due to round-off errors when calculating the remaining quantities when these numbers are small. This is caused by the clock only counting in integer increments.

In Table 7.4, the pre-optimisation jobs are the ones that does not list a wait time. Simply because the time spent waiting is unknown.

Upon the discovery of significant wait times, a simple rewrite was done. Each node keeps a record of the time spent running through its allocated interval, at first just its equal share of the total, and returns this time back to the master node. The Master node will then gather all the individual times returned and divide the average by each node's specific time. The previously allocated interval will then be multiplied by this number, which for the slow nodes will be < 1 and for the fast nodes > 1, and thus determine the allocated intervals for the next iteration of the Lanczos algorithm. In general it was noticed that the time spent by each node would equal out more or less after eight to ten iterations, but significant improvement would be achieved after just two to three iterations.

Since some of the rows in Table 7.4 are lacking wait time, the total wait time, which may be large, is hidden within computation time (labelled "Calc") and communication time (labelled "Comm"). For the other rows, the times listed are close to the true values,

but some suffer a little from accumulated rounding errors since the clock returns integer values. That is the reason for some of the near-zero numbers being negative. They can be taken as close to zero, and have been included as-is in the table for completeness' sake.

Part III

RESULTS

8

Results for Quantum Dots

"It does not do to leave a live dragon out of your calculations, if you live near him."

– J. R. R. Tolkien

This chapter contains a listing of ground state energy, E_0, calculations for a number of full and non-full shell configurations of quantum dots in two dimensions. A set of standard ω-values have been selected, 0.01, 0.1, 0.28, 0.5 and 1.0. For large scale calculations, $\omega = 0.1$, and to an extent $\omega = 1.0$, have been prioritised.

In the tables, Ψ signifies the full basis of Slater determinants for a given configuration of particles and shells, and $\Psi_{M,2s}$ signifies the filtered basis for a given total M and Spin for any Ψ. Since `libTardis` takes spin values as integers, spin is denoted by $2s$ unless otherwise stated. See note in introduction to Chapter 6.

Due to symmetries of the Hamiltonian, both total M and total spin are symmetric around 0. We therefore only consider the positive values of these, but for $M \neq 0$ and $2s \neq 0$, an additional degeneracy of 2 is implied.

The majority of calculations in this chapter use the standard harmonic oscillator frequency ω, but a few use the interaction strength parameter λ instead (see 2.3.1). This is due to the parameters used in the results we are comparing against. Since `OpenFCI` takes λ as an input parameter for effective interactions, and will produce the wrong results if it is set to 1 for $\omega \neq 1$, we pass a $\lambda = \sqrt{\omega}$ to `OpenFCI` and multiply the two-electron interaction element with ω afterwards, giving us the correct $\sqrt{\omega}$ factor for the interaction.

8.1 Effective Interactions and Energy Cut

First of all we will take a look at the the three options we have for our two-electron interaction elements generated by the `OpenFCI` package by Kvaal [35]. These options are: effective interactions with energy cut, denoted $V_{e,cut}$; effective interactions without energy cut, denoted v_e; and bare interactions, denoted V_b.

We will also compare the Lanczos algorithm to a straight forward diagonalisation of the block of the Hamiltonian that contains the ground state. This is trivial for the two-electron case where we have these matrices stored in memory already.

8.1.1 For 2 Electrons

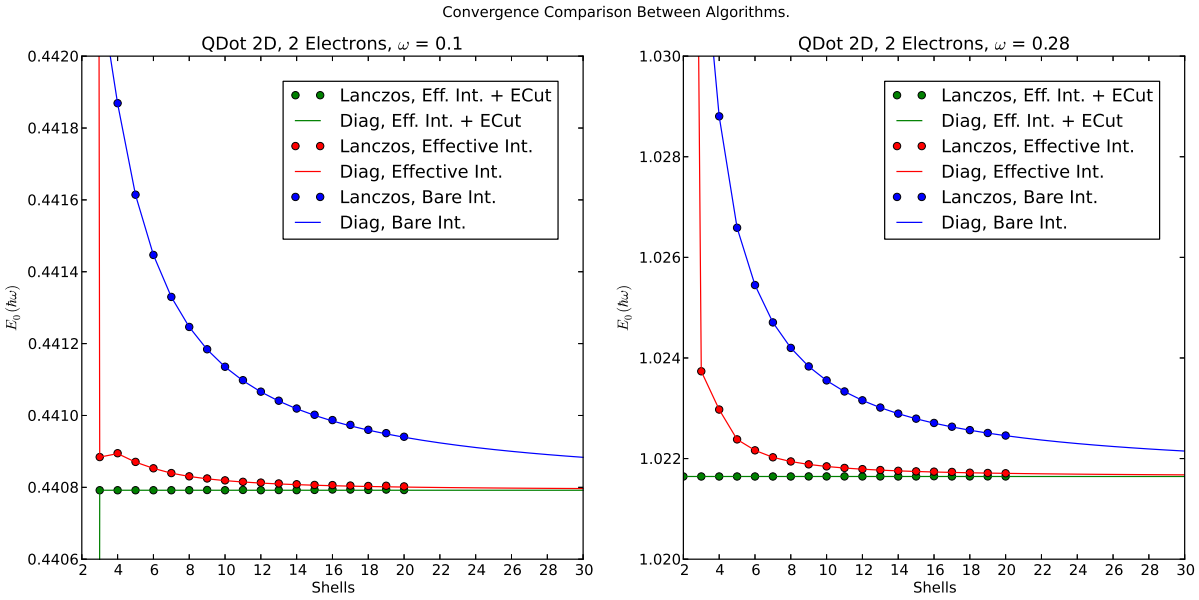

Figure 8.1: Accuracy of the different interaction elements compared to the best result, and comparison between the Lancsoz algorithm and straight forward diagonalisation of the two-electron Hamiltonian. Here for $\omega = 0.1$ and $\omega = 0.28$.

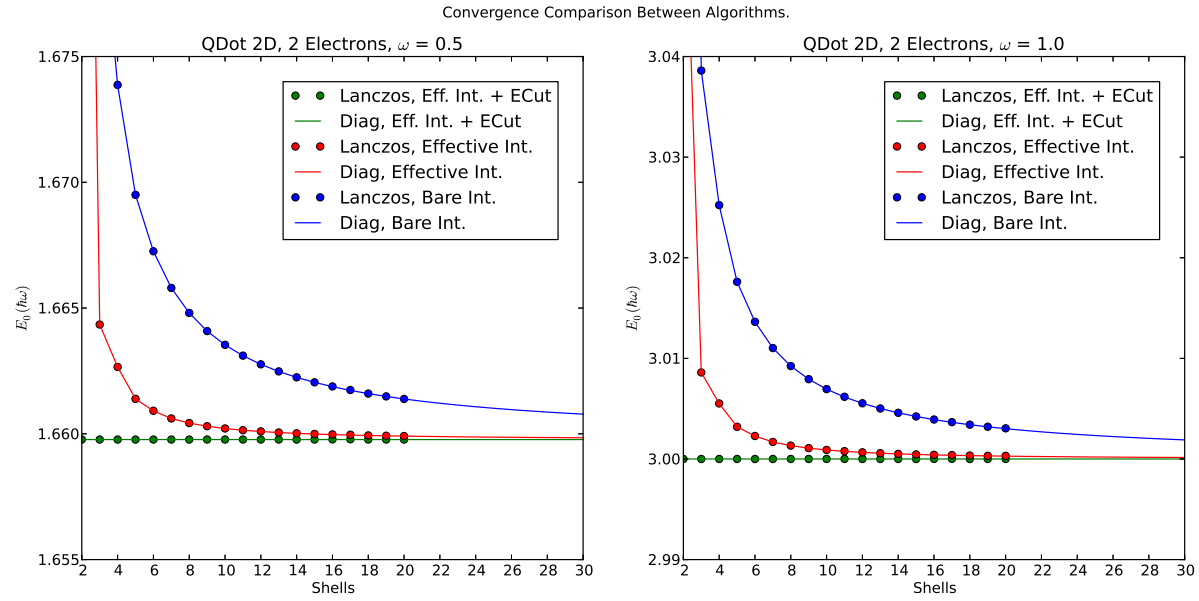

Figure 8.2: Accuracy of the different interaction elements compared to the best result, and comparison between the Lancsoz algorithm and straight forward diagonalisation of the two-electron Hamiltonian. Here for $\omega = 0.5$ and $\omega = 1.0$.

Firstly, we perform a comparison between the Lanczos algorithm and a simple diagonalisation of the two-electron Hamiltonian. Running the Lanczos algorithm for the two-electron case is of course a waste of CPU time, but it serves as a test for its accuracy as a pure diagonalisation will produce an exact result for the given Hamiltonian matrix. Within numerical precision of course.

As demonstrated by Figures 8.1 and 8.2, the Lanczos algorithm produces a lowest eigenvalue that is, as far as we can gaze, within a small error of the lowest eigenvalue produced by a direct diagonalisation of the Hamiltonian. These figures also illustrate how the three options converge to an exact result, which we can calculate analytically, and is exactly 3 for the $\omega = 1$ case.

As implied by the plots, effective interactions with energy cuts produce exact results for the two-electron case, but as table 8.1 illustrates, effective interactions without energy cut also converge relatively fast. Much faster than bare interactions do.

R	$N_{\Psi_{0,0}}$	$V_{e,cut}$	V_e	V_b
2	3	3.000000	3.063440	3.152328
3	8	3.000000	3.008602	3.038605
4	16	3.000000	3.005519	3.025231
5	29	3.000000	3.003199	3.017606
6	47	3.000000	3.002279	3.013626
7	72	3.000000	3.001692	3.011020
8	104	3.000000	3.001328	3.009236
9	145	3.000000	3.001076	3.007930
10	195	3.000000	3.000895	3.006938
11	256	3.000000	3.000760	3.006159
12	328	3.000000	3.000655	3.005532
13	413	3.000000	3.000574	3.005019
14	511	3.000000	3.000506	3.004589
15	624	3.000000	3.000453	3.004226
16	752	3.000000	3.000406	3.003912
17	897	3.000000	3.000368	3.003642
18	1 059	3.000000	3.000335	3.003406
19	1 240	3.000002	3.000309	3.003199
20	1 440	3.000000	3.000283	3.003012

Table 8.1: E_0 for 2 electrons, with $\omega = 1.0$, $M = 0$ and $2s = 0$, and with effective interactions with and without energy cut and with bare interactions.

As expected, Table 8.1 presents in its first column, near exact values for the ground state energy of the $\omega = 1$ configuration. Round-off errors do accumulate due to the iterative nature of the Lanczos algorithm, more so than pure diagonalisation.

8.1.2 For 4 Electrons

Now let us move on to a configuration of four electrons. As we saw in Table 8.1, the energy cut option appears to behave well for the two-electron case. This does not seem to

be the case for a higher number of electrons as illustrated by Table 8.2. This does require further study. It may be some disagreement between `OpenFCI` and `libTardis`. However, due to time constraints this is left for future studies. The results produced are still very good compared to other methods and to bare interactions.

As we can see from Table 8.2, the energies no longer converge neatly as functions of R, they fluctuate slightly. This illustrates the reason why we have chosen to not to enable the energy cut option for configurations with more that two electrons. Similar behaviour was noticed with six electrons as well.

R	$N_{\Psi_{0,0}}$	$V_{e,cut}$	V_e	V_b
3	37	10.265250	10.419363	10.565725
4	239	10.252124	10.272282	10.334327
5	1 025	10.245226	10.271782	10.313386
6	3 404	10.260793	10.269459	10.300211
7	9 444	10.263001	10.268195	10.292522
8	22 927	10.263718	10.267121	10.287154

Table 8.2: E_0 for 4 electrons, with $\omega = 1.0$, $M = 0$ and $2s = 0$, and with effective interactions with and without energy cut and with bare interactions.

8.2 Comparison Runs

The first mile stone of the development of `libTardis`, after the successful implementation of the Lanczos algorithm working for two electrons at least, was reproducing the results by Kvaal [35] and Rontani *et al.* [44]. Since `libTardis` performs well on larger systems, an additional set of runs have been performed that exceed the dimensionality of the referenced results. They are provided as additional results in 8.2.2.

8.2.1 Reproducing Published Results

N_P	λ	M	$2s$	6-Shell V_e	6-Shell V_b	Ref [35]	Ref [44]
2	1	0	0	3.002279	3.013626	3.013626	
	2	0	0	3.722295	3.733598	3.733598	3.7338
		1	2	4.142227	4.143592	4.143592	4.1437
3	2	1	1	8.156527	8.175038	8.175035	8.1755
	4	1	1	11.04019	11.04481	11.04480	11.046
		0	3	11.05207	11.05428	11.05428	11.055
4	6	0	0	23.63362	23.63874	23.68914	23.691
		2	4	23.86071	23.86769	23.86769	23.870
5	2	0	5	21.13605	21.15094	21.15093	21.15
	4	0	5	29.40852	29.43528	29.43528	29.44

Table 8.3: Comparison of 6-shell results with both bare and effective interaction elements with Kvaal [35] and Rontani *et al.* [44].

N_P	λ	M	$2s$	7-Shell V_e	7-Shell V_b	Ref [35]	Ref [44]
2	1	0	0	3.001692	3.011020	3.011020	
	2	0	0	3.721835	3.731058	3.731057	3.7312
		1	2	4.142043	4.142947	4.142946	4.1431
3	2	1	1	8.155446	8.169917	8.169913	
	4	1	1	11.03983	11.04339	11.04338	
		0	3	11.05181	11.05325	11.05325	
4	6	0	0	23.60110	23.60267	23.65559	
		2	4	23.80660	23.80798	23.80796	
5	2	0	5	21.12572	21.13415	21.13414	21.13
	4	0	5	29.30151	29.30900	29.30898	29.31

Table 8.4: Comparison of 7-shell results with both bare and effective interaction elements with Kvaal [35] and Rontani *et al.* [44].

N_P	λ	M	$2s$	8-Shell V_e	8-Shell V_b	Ref [35]	Ref [44]
2	1	0	0	3.001328	3.009236	3.009236	
	2	0	0	3.721556	3.729324	3.729324	3.7295
		1	2	4.141941	4.142581	4.142581	4.1427
3	2	1	1	8.154840	8.166709	8.166708	8.1671
	4	1	1	11.03965	11.04254	11.04254	11.043
		0	3	11.05162	11.05262	11.05262	11.053
4	6	0	0	23.59529	23.59636	23.64832	23.650
		2	4	23.80287	23.80374	23.80373	23.805
5	2	0	5	21.12433	21.12993	21.12992	21.12
	4	0	5	29.29778	29.30252	29.30251	29.30

Table 8.5: Comparison of 8-shell results with both bare and effective interaction elements with Kvaal [35] and Rontani *et al.* [44].

Tables 8.3, 8.4 and 8.5 list the results from `libTardis` for both effective and bare interactions, and are compared to Kvaal [35] who used only bare interactions, and to Rontani *et al.* [44] where such results are available.

Generally, the discrepancies between `libTardis`' bare interaction results and Kvaal's results are mere round-off errors. The one exception being the ground state of four electrons. The reason for this difference is due to the slow convergence of this particular configuration and that Kvaal and Rontani *et al.* most likely stopped their calculations too early and thus only found the first excited energy state. This is discussed in detail with error analysis in 7.1.1 and will not be repeated, but it is an important discussion.

Again, note that for all these configurations, effective interactions give a much better result for lower values of R than bare interactions do. We will get back to this point again in the next section.

8.2.2 Additional Results

N_P	λ	M	$2s$	9-Shell V_e	9-Shell V_b	10-Shell V_e	10-Shell V_b
2	1	0	0	3.001076	3.007930	3.000895	3.006938
	2	0	0	3.721366	3.728059	3.721228	3.727100
		1	2	4.141884	4.142360	4.141849	4.142217
3	2	1	1	8.154423	8.164461	8.154156	8.162844
	4	1	1	11.03955	11.04199	11.03950	11.04160
		0	3	11.05154	11.05228	11.05149	11.05206
4	6	0	0	23.59504	23.59595	23.59502	23.59581
		2	4	23.80201	23.80262	23.80196	23.80242
5	2	0	5	21.12339	21.12735	21.12293	21.12591
	4	0	5	29.29568	29.29888	29.29537	29.29772

Table 8.6: 9 and 10-shell results with both bare and effective interaction elements.

Lastly, the configurations by Kvaal and Rontani *et al.* have been run also for 9 and 10 shell systems. These are presented in Table 8.6 for the sake of reference. Not surprisingly the energies are lower than for the previously listed results.

8.3 Full Shell Results

Let us now consider ground state energy results for systems with fully occupied shells as illustrated by Table 8.7.

N_e	R_1	R_2	R_3	R_4	R_5
2	2	0	0	0	0
6	2	4	0	0	0
12	2	4	6	0	0
20	2	4	6	8	0
Max	2	4	6	8	10

Table 8.7: Number of total electrons, N_e, and their distribution across the shells.

Full shell systems have the benefit of having a number of Coupled Cluster (see 3.3) results for comparison. Results by former master student Christoffer Hirth [16] have been used as we have been comparing results for his thesis as well. These same results are used in the tables below, where applicable.

Also published results by M. P. Lohne *er al.* [20] have been used. These include CCSD, CCSD(T) and DMC (see 3.4.2) results and therefore provide a wider set of methods for comparison.

Lastly, DMC results by fellow master student Karl Leikanger have been used as well where they exist. These results are not publicly available as of this time.

8.3.1 2 Electron Results

R	N_Ψ	$N_{\Psi_{0,0}}$	$\omega = 0.01$	$\omega = 0.1$	$\omega = 0.28$	$\omega = 0.5$	$\omega = 1.0$
5	4.35e2	2.90e1	0.0738364	0.441614	1.026588	1.669498	3.017606
10	6.00e3	1.95e2	0.0738351	0.441135	1.023551	1.663535	3.006938
15	2.87e5	6.24e2	0.0738352	0.441000	1.022791	1.662046	3.004226
20	8.80e5	1.44e3	0.0738352	0.440940	1.022456	1.661386	3.003012
Lohne, Hagen, Hjorth-Jensen, Kvaal and Pederiva [20]							
10	CCSD		–	–	–	1.663535	3.000895
20			–	–	–	1.661378	3.000282
–	DMC		–	–	–	1.65975(2)	3.00000(3)

Table 8.8: E_0 for 2 electrons, with $M = 0$ and $2s = 0$, and with bare interactions. Comparing with results by [20].

R	N_Ψ	$N_{\Psi_{0,0}}$	$\omega = 0.01$	$\omega = 0.1$	$\omega = 0.28$	$\omega = 0.5$	$\omega = 1.0$
5	4.35e2	2.90e1	0.07383504	0.44079189	1.0216440	1.6597722	3.0000000
10	6.00e3	1.95e2	0.07383505	0.44079189	1.0216440	1.6597722	3.0000001
15	2.87e5	6.24e2	0.07383505	0.44079191	1.0216440	1.6597724	3.0000002
20	8.80e5	1.44e3	0.07383505	0.44079191	1.0216441	1.6597723	3.0000001
Lohne, Hagen, Hjorth-Jensen, Kvaal and Pederiva [20]							
20	CCSD		–	–	–	1.659722	3.000000
–	DMC		–	–	–	1.65975(2)	3.00000(3)
Leikanger [45]							
–	DMC		0.07384(2)	0.44087(3)	1.02166(3)	–	3.00000(3)

Table 8.9: E_0 for 2 electrons, with $M = 0$ and $2s = 0$, and with effective interactions and energy cut. Comparing with results by [20] and [45].

Tables 8.8 and 8.9 provide further results for the two-electron case. Not much more is to be said that was not discussed in 8.1.1. Energy cut has been used for effective interactions such that the results are within a small error of the exact results for the two-electron case. The results provided by Leikanger and Lohne *et al.* are also in agreement.

For the bare interaction case, the results deviate more, but since the dimensionalities are small, we can run for a relatively large number of shells. Here up to $R = 20$, but during testing, $R = 25$ did not cause any problems either, however, there is little to be gained from increasing the dimensionality for these calculations.

8.3.2 6 Electron Results

R	N_Ψ	$N_{\Psi_{0,0}}$	$\omega = 0.01$	$\omega = 0.1$	$\omega = 0.28$	$\omega = 0.5$	$\omega = 1.0$
3	9.24e2	6.40e1	1.009487	4.149560	8.520412	12.897229	21.420589
4	3.88e4	1.49e3	0.856380	3.797439	7.851833	12.036943	20.415833
5	5.94e5	1.65e4	0.775682	3.645755	7.710452	11.913062	20.316762
6	5.25e6	1.15e5	0.730679	3.567998	7.629581	11.840364	20.257196
7	3.25e7	5.94e5	0.708777	3.559622	7.620985	11.825317	20.232307
8	1.18e8	2.46e6	0.697161	3.555165	7.615535	11.815817	20.216428
Hirth [16]							
6	CCSD		–	3.596913	–	11.863455	20.273247
30			–	3.582050	–	11.811704	20.183493

Table 8.10: E_0 for 6 electrons, with $M = 0$ and $2s = 0$, and with bare interactions. Comparing with results by [16].

R	N_Ψ	$N_{\Psi_{0,0}}$	$\omega = 0.01$	$\omega = 0.1$	$\omega = 0.28$	$\omega = 0.5$	$\omega = 1.0$
3	9.24e2	6.40e1	0.767619	3.818664	8.086706	12.373620	20.863074
4	3.88e4	1.49e3	0.746501	3.673350	7.703043	11.865767	20.210671
5	5.94e5	1.65e4	0.735119	3.618749	7.653232	11.829380	20.194931
6	5.25e6	1.15e5	0.719735	3.560683	7.602284	11.790886	20.173913
7	3.25e7	5.94e5	0.706110	3.555196	7.600736	11.787930	20.168227
8	1.18e8	2.46e6	0.696444	3.552068	7.599579	11.785915	20.164477
9	6.22e8	8.62e6	0.691980	3.551911	–	–	20.162099
10	2.14e9	2.65e7	–	3.551833	–	–	20.160472
11	6.55e9	7.33e7	–	3.551776	–	–	–
Lohne, Hagen, Hjorth-Jensen, Kvaal and Pederiva [20]							
10	CCSD		–	–	7.6241	11.8057	20.1766
20			–	–	7.6252	11.8055	20.1737
10	CCSD(T)		–	–	7.6032	11.7871	20.1623
20			–	–	7.6006	11.7837	20.1570
–	DMC		–	–	7.6001(1)	11.7888(2)	20.1597(2)
Leikanger [45]							
–	DMC		0.6896(1)	3.5539(1)	7.6003(2)	–	20.1596(2)

Table 8.11: E_0 for 6 electrons, with $M = 0$ and $2s = 0$, and with effective interactions. Comparing with results by [20] and [45].

For six electrons we are for the first time getting into some configurations with large dimensionalities. Enough that the use of the super computer was necessary. See 7.3 for the relevant calculations and CPU costs.

Due to comparison with Hirth and other master students, calculations were performed with both bare and effective interactions for the six-electron case. In the effective interactions case, the results agree very closely with DMC results and are for the $\omega = 0.1$, 0.28 and 0.5 cases a little lower.

We will get back to the convergence rate as a function of R in 8.3.5.

8.3.3 12 Electron Results

R	N_Ψ	$N_{\Psi_{0,0}}$	$\omega = 0.01$	$\omega = 0.1$	$\omega = 0.28$	$\omega = 0.5$	$\omega = 1.0$
4	1.26e5	4.26e3	2.904255	13.520243	27.686546	41.786414	68.818532
5	8.65e7	1.63e6	2.764577	13.106935	26.482570	39.922693	66.290115
6	1.11e10	1.49e8	–	12.850344	–	–	66.076116
Lohne, Hagen, Hjorth-Jensen, Kvaal and Pederiva [20]							
10	CCSD		–	–	25.7069	39.2218	65.7552
20			–	–	25.7089	39.2194	65.7409
10	CCSD(T)		–	–	25.6445	39.1659	65.7118
20			–	–	25.6324	39.1516	65.6886
–	DMC		–	–	25.6356(1)	39.159(1)	65.700(1)
Leikanger [45]							
–	DMC		2.4723(5)	12.2694(2)	25.6361(3)	–	65.7011(3)

Table 8.12: E_0 for 12 electrons, with $M = 0$ and $2s = 0$, and with effective interactions. Comparing with results by [20] and [45].

For the three full shell case of 12 electrons, we only have results for effective interactions due to the expensive nature of these calculations. Only ω values of 0.1 and 1.0 have been taken close to the upper limit of `libTardis`' capability, which for the time being is in the area of 4×10^8 and is restricted by the amount of memory available on each node on *Abel*. The current memory usage is somewhat wasteful because it speeds up calculations and we have not yet needed to implement memory saving techniques to run on larger systems.

Not surprisingly DMC performs better for these calculations as now we are getting into a dimensionality where the FCI method starts to blow up in resource usage. For instance, for $R = 7$, the dimensionality $N_\psi \approx 5.6 \times 10^{11}$. With only three free shells for our electrons to move into, we are bound to lose a lot of contributing states even with effective interactions.

8.3.4 20 Electron Results

Lastly, a set of 20-electron runs have been added simply because running these for $R = 5$ is trivial as there is only one shell of 10 holes for the electrons to move into. Not surprisingly the results here are far from good. Again. adding a shell, taking us to $R = 6$, will run us into dimensionality problems as $N_\psi \approx 5.1 \times 10^{11}$.

R	N_Ψ	$N_{\Psi_{0,0}}$	$\omega = 0.01$	$\omega = 0.1$	$\omega = 0.28$	$\omega = 0.5$	$\omega = 1.0$
5	3.00e7	6.10e5	7.864088	34.204867	67.767987	100.93607	164.61280
Lohne, Hagen, Hjorth-Jensen, Kvaal and Pederiva [20]							
10	CCSD		–	–	62.2851	94.0870	156.0128
20			–	–	62.0664	93.9891	155.9601
10	CCSD(T)		–	–	92.1802	93.9864	155.9324
20			–	–	61.9156	93.8558	155.8571
–	DMC		–	–	61.922(2)	93.867(3)	155.868(6)
Leikanger [45]							
–	DMC		6.1418(3)	29.9767(3)	61.9273(6)	–	155.8824(7)

Table 8.13: E_0 for 20 electrons, with $M = 0$ and $2s = 0$, and with effective interactions. Comparing with results by [20] and [45].

8.3.5 Convergence Rates

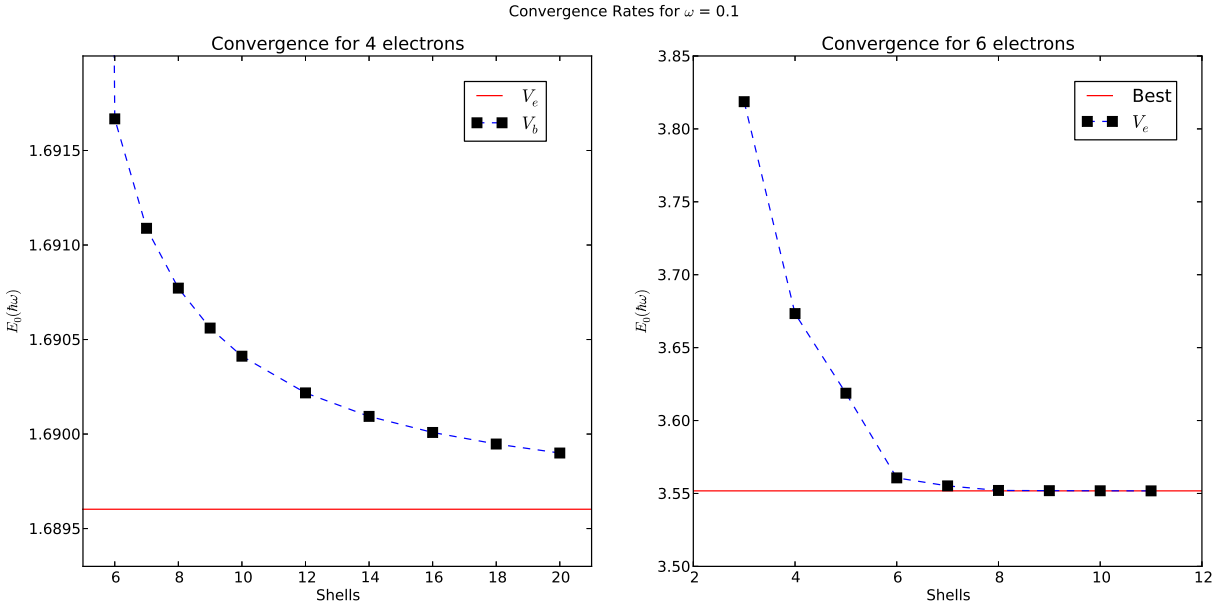

Figure 8.3: The left plot shows how bare interactions converge for increasing R towards a baseline of effective interactions for $R = 20$. The right plot shows how the six-electron results with effective interactions converge to its best value for $R = 11$.

Let us briefly revisit the convergence properties of bare and effective interactions now that we have more results.

The plots in Figure 8.3 illustrate again how much better results effective interactions produce compared to bare interactions as well as how fast effective interactions converge towards its best results as a function of R. The right plot tells us that already for $R = 8$,

the six-electron system has more or less converged. Even for $R = 6$ the result is close. We will exploit that for a more in-depth analysis in Chapter 9.

8.4 Non Full Shell Results

Since there is essentially no difference between full and non-full shell calculations with the FCI method, a set of selected such configurations have been studied as well. As listed in Table 8.14, a configuration of two full shells plus one electron (Table 8.15) and three full shells plus (Table 8.17) and minus (Table 8.16) one electron has been selected.

N_e	R_1	R_2	R_3	R_4	R_5
7	2	4	1	0	0
11	2	4	5	0	0
13	2	4	6	1	0
Max	2	4	6	8	10

Table 8.14: Number of total electrons, N_e, and their distribution across the shells.

Since these are all odd-numbered configurations, there is obviously no total spin 0 state, so a total spin, $2s$, of $+1$ has been chosen. Unfortunately there are no published results that have been found so far that have comparable numbers for the value of ω chosen.

The motivation, then, for running these has been to produce potentially new results and to test the scalability of `libTardis` as the largest configuration $N_e = 13$ and $R = 6$ has a dimensionality of its basis of 3.13×10^8. This configuration requires in the area of 50 gigabytes of memory on the computation nodes on *Abel*, which is approaching the effective memory limit.

8.4.1 7 Electron Results

R	N_Ψ	$N_{\Psi_{0,1}}$	$E_0\Psi_{0,1}$	$N_{\Psi_{2,1}}$	$E_0\Psi_{2,1}$
3	7.91e2	4.80e1	5.1173073	3.60e1	5.1037461
4	7.75e4	2.51e3	4.9709711	2.23e3	4.9545491
5	2.04e6	4.68e4	4.8385200	4.37e4	4.8395010
6	2.70e7	4.88e5	4.7513303	4.66e5	4.7457378
7	2.32e8	3.48e6	4.7228269	3.37e6	4.7181798

Table 8.15: E_0 for 7 electrons, with $\omega = 0.1$, and with effective interactions.

The first interesting thing to notice about the seven-electron system is that the ground state energy is a little harder to pinpoint. For some values of R it is found with $M = 0$ and for some with $M = 2$. The trend is that $M = 2$ is the ground state for $\omega = 0.1$. We will however look a bit further into five and seven-electron configurations again in Chapter 9.

8.4.2 11 and 13 Electron Results

R	N_Ψ	$N_{\Psi_{0,1}}$	$E_0\Psi_{0,1}$	$N_{\Psi_{2,1}}$	$E_0\Psi_{2,1}$
4	1.68e5	5.06e3	11.561326	4.54e3	11.537684
5	5.46e7	9.97e5	11.136607	9.45e5	11.198673
6	4.28e9	5.76e7	10.945194	–	–

Table 8.16: E_0 for 11 electrons, with $\omega = 0.1$, and with effective interactions.

R	N_Ψ	$N_{\Psi_{0,1}}$	$E_0\Psi_{0,1}$	$N_{\Psi_{2,1}}$	$E_0\Psi_{2,1}$
4	7.75e4	2.51e3	15.745094	2.23e3	15.677216
5	1.20e8	2.08e6	15.288787	1.98e6	15.342340
6	2.55e10	3.13e8	14.907751	–	–

Table 8.17: E_0 for 13 electrons, with $\omega = 0.1$, and with effective interactions.

Lastly we have the results for three full shells ±one electron. The last calculation being the largest one tried with `libTardis` to date, and took a little over 92 000 CPU hours to complete.

9
Additional Results for Quantum Dots

"Quantum mechanics: the dreams that stuff is made of."

– Stephen Hawking

The results presented in this chapter have been put in this separate chapter because they are essentially preliminary results. After the code for `libTardis` was more or less complete, and the majority of the ground state energy tables had been calculated, we considered a few options for further study. There were several candidates, including a look at one-body densities, but in the end time restrictions left us with the easiest to perform. Namely producing a large number of data points for small scale systems of six shells. Six shell systems are good configurations for such studies because they require less than 64 bits for the binary representation of Slater determinants, and they are relatively fast to run through as the dimensionality of the basis is manageable for single-node computers. Still, the calculations provided in this chapter took many weeks to complete as there are quite a lot of them.

The plots are presented in this chapter, but the tables of the data points for these plots have been put in Appendix B due to the large number of these. A couple of tables detailing the computation times for these plots have also been added in Appendix B.

9.1 Energy as a Function of Spin, M and ω

First we have a set of simulations of how energy develops as a function of M, spin and ω for six, seven and eight electrons. Each of the 1468 individual calculations needed to generate these plots took anywhere from a few milliseconds to as much as two days to complete, thus it was necessary to restrict these to only six shells. This limits the computer representation of Slater determinants to using 64 bits, which significantly improves performance as we have already mentioned. Also, as we have looked into already in 8.3.5, relatively good results are available at $R = 6$ when effective interactions are used. We do not yet know enough about how these behave when the dimensionality of the selected basis becomes very small. The dimensionality of the basis for each data point in the plots, as well as the numerical value of each of these, are listed in full in Appendix B.

Due to time restrictions, as these plots took about 5-6 weeks to produce, there was not enough time to do a detailed analysis of the error estimates, but we do know that generally, the convergence criterion used produces good results in the cases we have looked at in the previous chapters. With this in mind, let us look at some of the structures that emerge from the plots that follow. The plots for six electrons can be found in figures 9.1 and 9.2; plots for seven electrons in figures 9.3 and 9.4; and plots for eight electrons in

figures 9.5 and 9.6.

The first thing we notice from these plots is the zig-zag pattern that emerges for odd and even values of M in the high-ω plots with ω values of 1.0 and 5.0. There appears to be a lot of degenerate or near degenerate energies in these data points as well. This suspicion is confirmed if we look at the data tables for for instance Figure 9.2. The tables are available in Appendix B, tables B.6 and B.7. If we look at the columns for spin 1/2 and 3/2, we notice that many of these energies are more or less the same. This is a repeating pattern in many of the rows in most of these tables.

At lower values of ω, a different pattern emerges. The energies are dominated by a smooth curve with high-spin configurations standing out and "floating" atop the lower spin values. This trend increases with the number of above full-shell electrons, and for eight electrons this pattern is relatively strong for an ω value of 0.01. See the left-most plot of Figure 9.5.

Some of the structures we see in these plots are similar to the structures of spin and M seen by Bårdsen *et al.* [46]. A discussion with him revealed they had some properties in common. The main similarity was the pattern that arises for odd and even values of M where the energy creates this zig-zag pattern. They found that increasing the magnetic field, and thus increasing the effects related to spin, altered this pattern. This information is unavailable in our plots because there is no magnetic field added, but we still see that these patterns depend on spin as well as M.

Adding a magnetic field may indeed be an interesting additional study in itself in order to see how it alters these patterns.

9.1.1 Plots for 6 Electrons

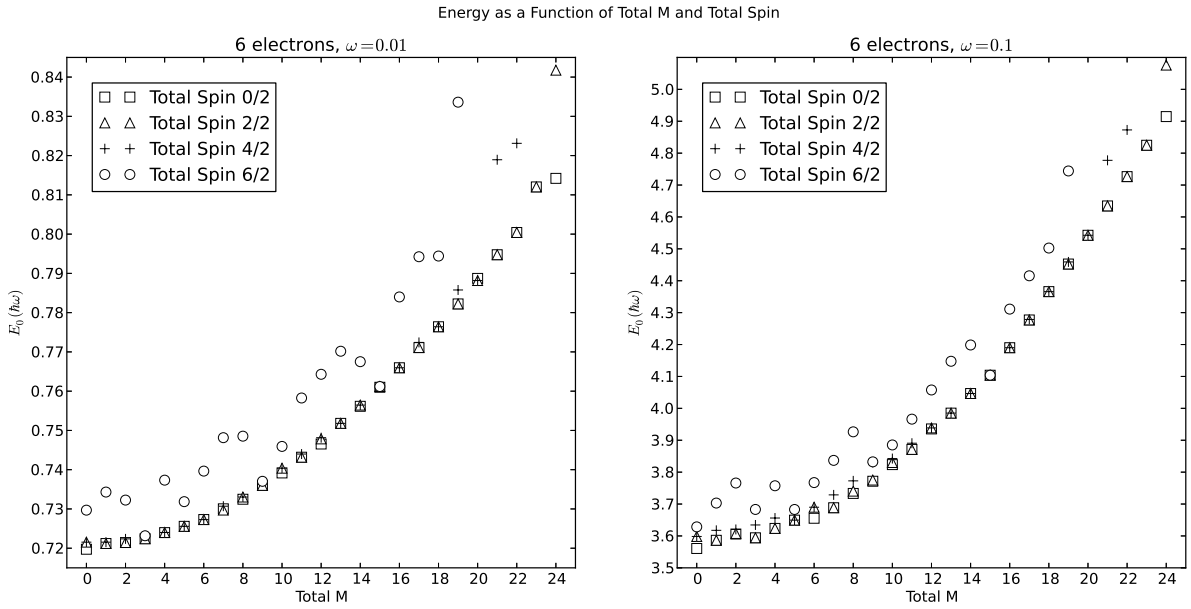

Figure 9.1: Lowest energy eigenvalue for 6 electrons and $R = 6$ as a function of total M and total spin for $\omega = 0.01$ and $\omega = 0.1$

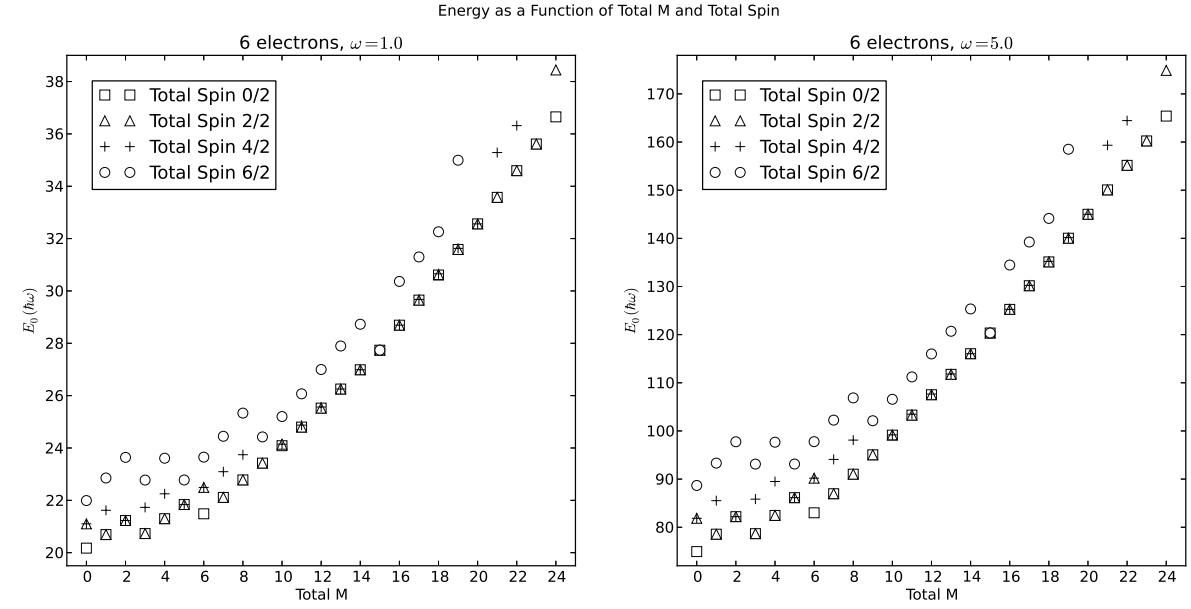

Figure 9.2: Lowest energy eigenvalue for 6 electrons and $R = 6$ as a function of total M and total spin for $\omega = 1.0$ and $\omega = 5.0$

9.1.2 Plots for 7 Electrons

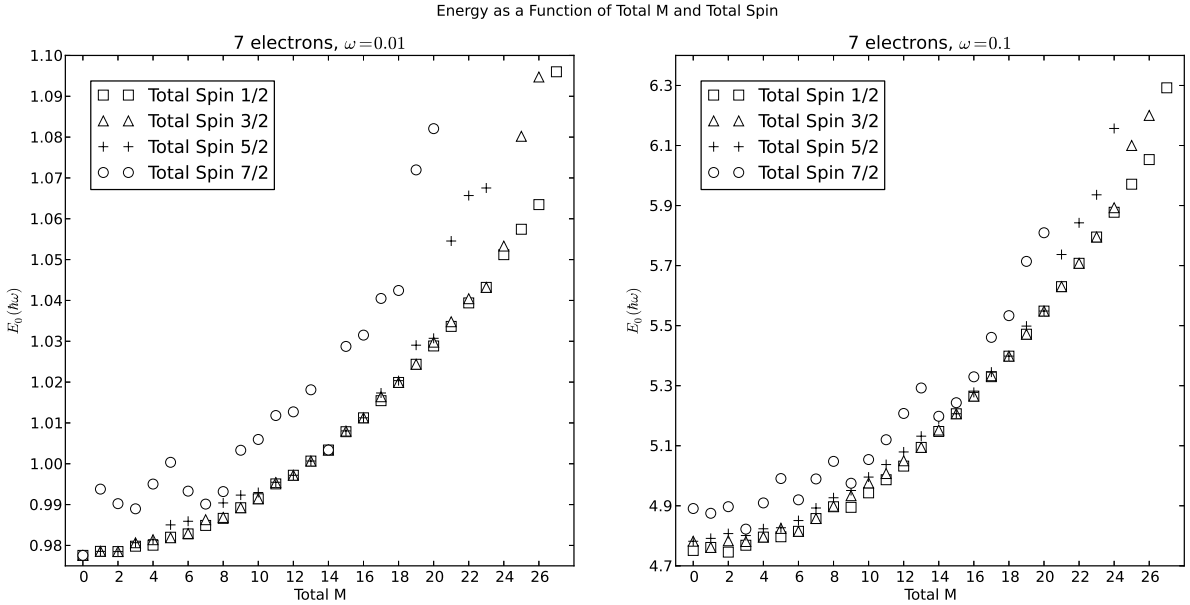

Figure 9.3: Lowest energy eigenvalue for 7 electrons and $R = 6$ as a function of total M and total spin for $\omega = 0.01$ and $\omega = 0.1$

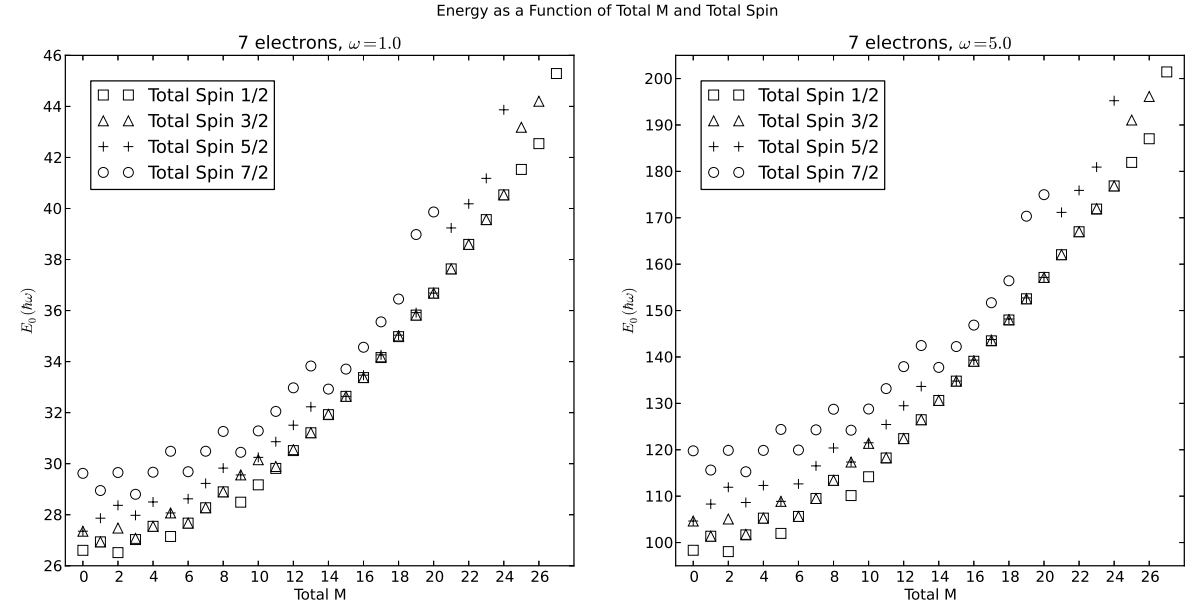

Figure 9.4: Lowest energy eigenvalue for 7 electrons and $R = 6$ as a function of total M and total spin for $\omega = 1.0$ and $\omega = 5.0$

9.1.3 Plots for 8 Electrons

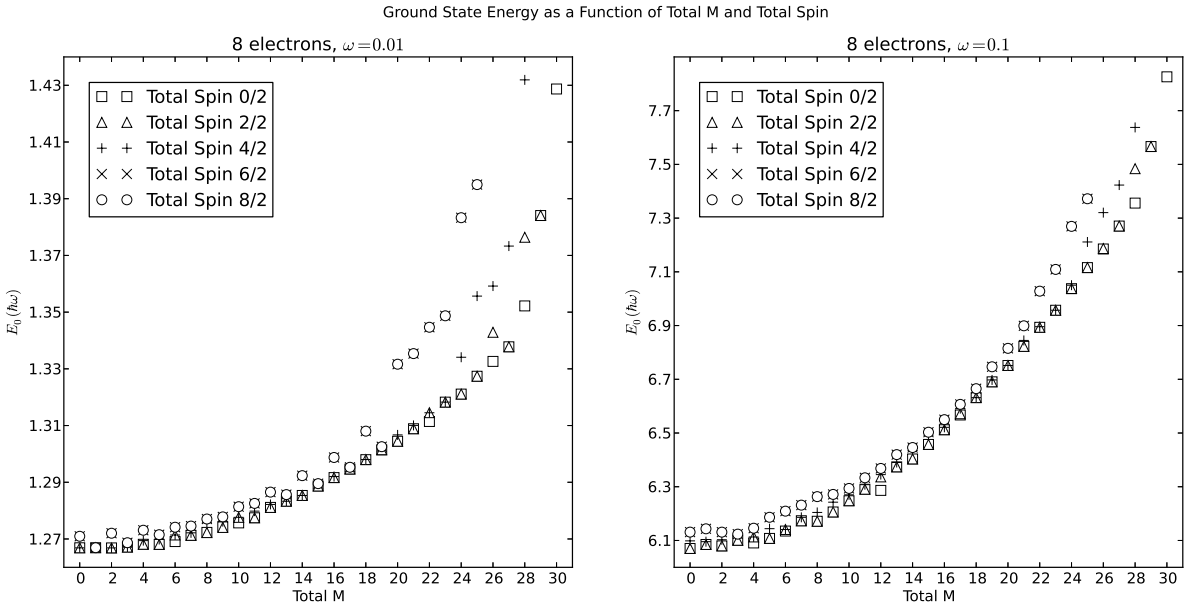

Figure 9.5: Lowest energy eigenvalue for 8 electrons and $R = 6$ as a function of total M and total spin for $\omega = 0.01$ and $\omega = 0.1$

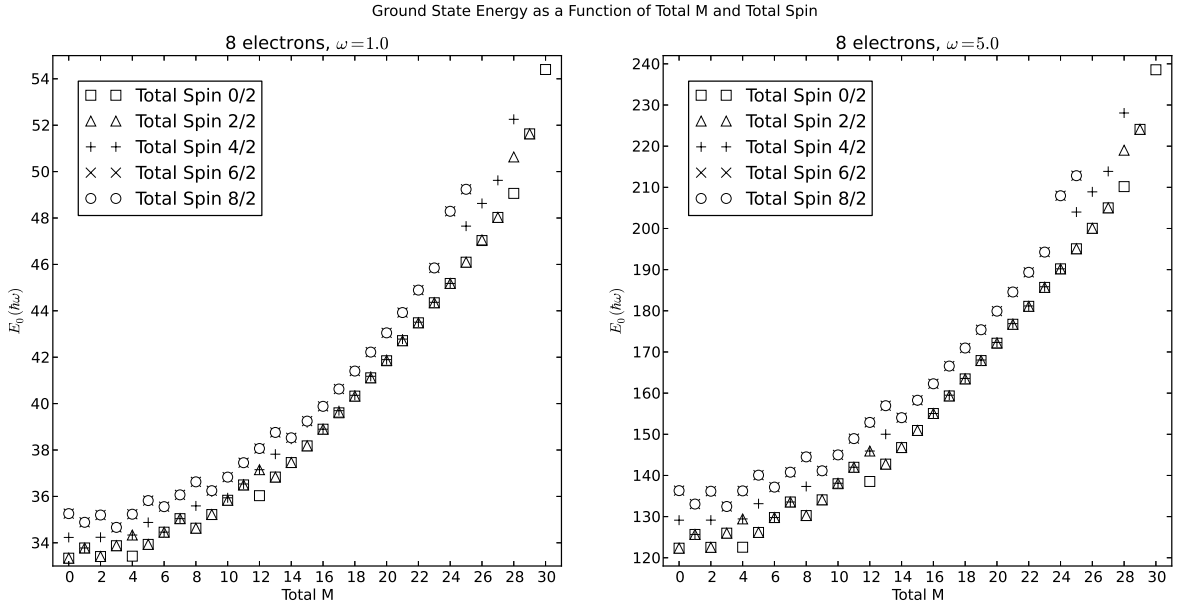

Figure 9.6: Lowest energy eigenvalue for 8 electrons and $R = 6$ as a function of total M and total spin for $\omega = 1.0$ and $\omega = 5.0$

9.2 Coefficients

A set of plots that were a little easier to produce, were the plots contained in this section. They take the Ritz vectors, the approximate eigenvectors of our full Hamiltonian matrix, which contains approximate coefficients for our wave function, and look at the dominating configurations of single-particle states. This has been done for four, five, six and seven electrons. Eight electrons was planned, but the computation time blows up at around seven electrons and time did not allow for further calculations.

The analysis of the composition of the ground state wave function yields several interesting results. We show in figures 9.2.1, 9.2.2, 9.2.3 and 9.2.4, the most important contributions to the ground states of the four, five, six and seven electron systems, respectively. We list only the largest coefficients that arise from the linear expansion in terms of the various basis Slater determinants. The labelling of the legend follows the labelling of the various single-particle states, displayed again in Figure 9.7. As an example, for four electrons, total spin 0 and $M = 0$, we assume that the ground state is dominated by two electrons in the first shell and two electrons in the second shell, but with opposite spins and different m values. This results in the labelling $|11\ 1001\ 000000\rangle$, where the first position to the left refers to single-particle state 1 in Figure 9.2.1. If a single-particle state is occupied we assign a value of one, else we set it to zero.

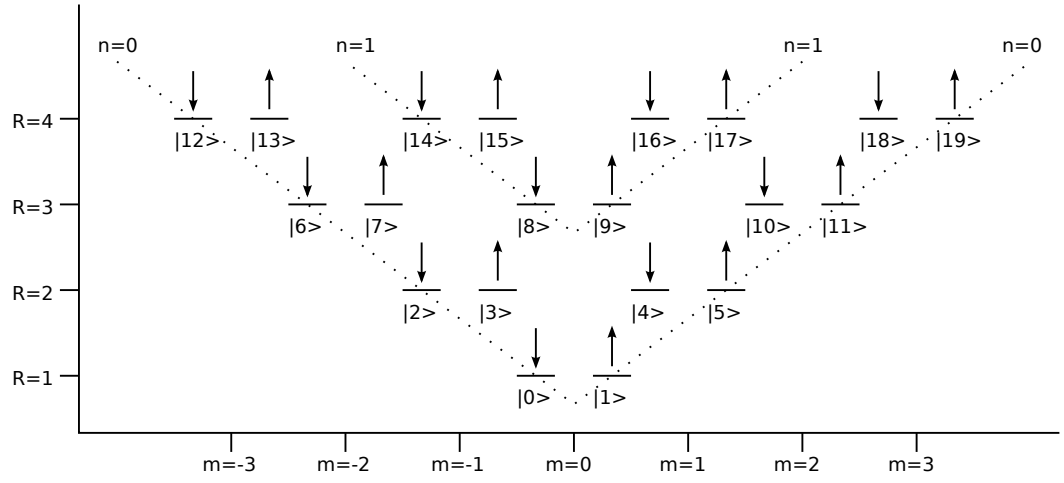

Figure 9.7: The structure of single-particle states for a harmonic oscillator potential in two dimensions. Here illustrated as a system of four shells, R, with single-particle states from $|0\rangle$ to $|19\rangle$. Each state represents a bit of 0 or 1 in our binary Slater determinants.

There are several interesting observations which can be extracted from these figures. First we note that for larger values of the harmonic oscillator frequency ω, there is essentially one configuration which dominates, namely the assumed ground state configuration from the independent particle model. This configuration dominates for ω values from approximately 0.5 and higher, a result which has important consequences for many-body methods like Coupled Cluster theory 3.3 and diffusion Monte Carlo 3.4.2, where a reference Slater determinant $|\Phi_0\rangle$ is used as an ansatz for the calculational procedures. With

smaller and smaller values of the frequency ω this ansatz breaks down and we start to see strong correlations arising from other Slater determinants. These correlations are due to the mixing of essentially one-particle one-hole and two-particle two-hole contributions. For $\omega \leq 0.1$, methods like Coupled Cluster theory or diffusion Monte Carlo, would require a multi-reference ansatz for the wave functions [6]. For smaller values of ω, this is also a region where a phase transition from a standard fermionic system to a Wigner-Seitz type of crystallization takes place [47]. In such situations, a single Slater determinant ansatz for the wave function clearly breaks down.

From the above mentioned figures, we note also that there are some interesting differences between the even and odd number of electron systems. For the four and six electron systems (the latter is a closed shell system) we see that one single Slater determinant (which is the ansatz for the ground state) dominates rather strongly even for small values of ω. It costs energy to break a pair while for the odd electron systems, we see that the absence of an electron from the closed shell system or the addition of one, allows for stronger admixtures of states that represent typically a one-particle one-hole excitation. This is obviously due to the fact that the attached or removed electrons are easier to excite than a system where the electrons are paired. However, the five electron system, that is, one hole with respect to the closed shell system of six electrons, has much stronger admixtures with other states than the particle added system with seven electrons. The seven electron system can easily be interpreted as one particle on top of the six electron ground state mixed with some dominating, but weakly coupling, low-excitation energy one-particle one-hole states. The whole system, on the other hand, shows a stronger admixture of one-particle one-hole excitations, a feature which is partly expected since, when ω is lowered, there are several Slater determinants, either of the one-hole type or one-particle one-hole type, which are close in the unpertured energy, yielding thereby a stronger mixing ratio.

All the these observations have strong consequences for many-body methods like coupled cluster for small values of ω. Most likely, it is not clear whether coupled cluster theory with a single reference Slater determinant, see the calculations of Lohne *et al.* [20], may converge, if at all, for values of $\omega \leq 0.1$.

9.2.1 Plots for 4 Electrons

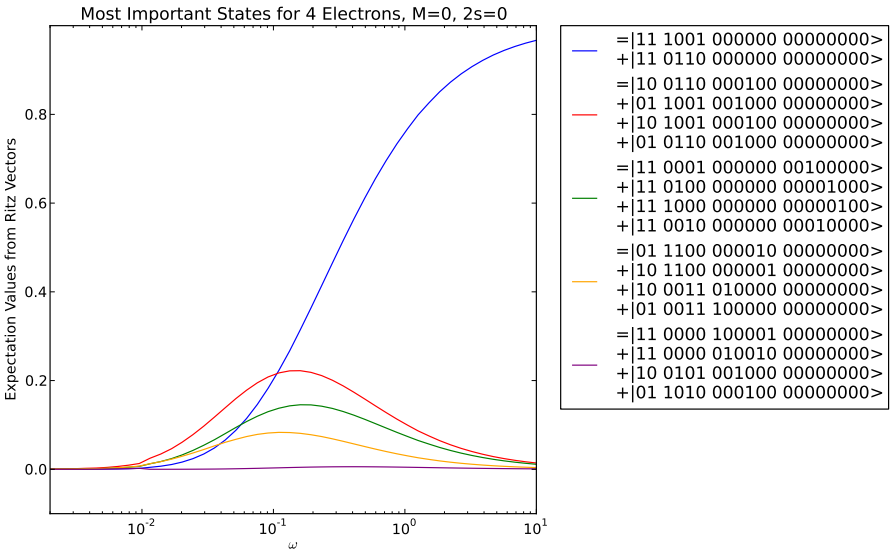

Figure 9.8: The dominating single-particle configurations for a system of 4 electrons in 6 shells with $M = 0$ and $2s = 0$, and with effective interactions.

9.2.2 Plots for 5 Electrons

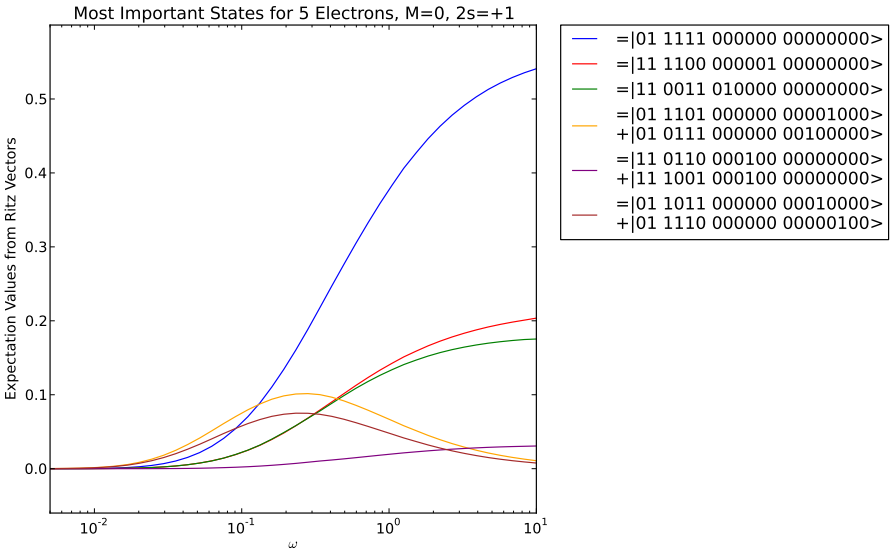

Figure 9.9: The dominating single-particle configurations for a system of 5 electrons in 6 shells with $M = 0$ and $2s = 1$, and with effective interactions.

9.2.3 Plots for 6 Electrons

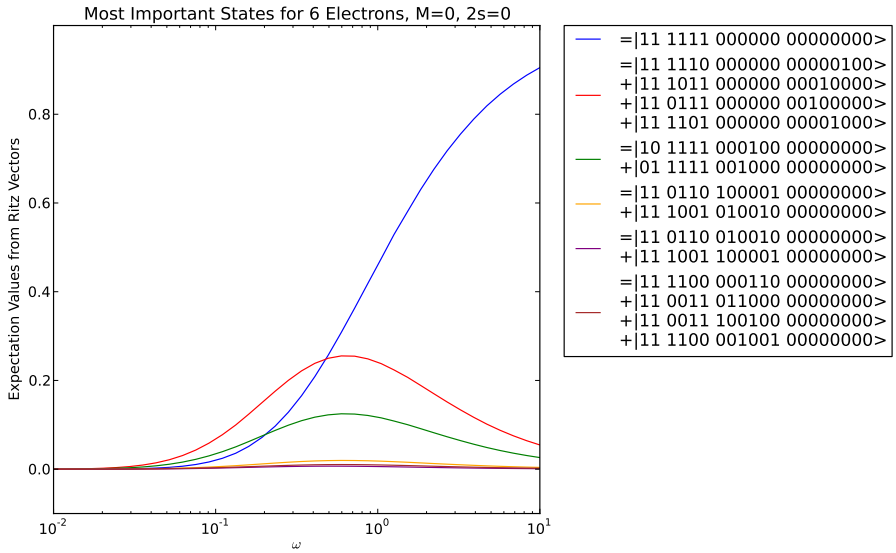

Figure 9.10: The dominating single-particle configurations for a system of 6 electrons in 6 shells with $M = 0$ and $2s = 0$, and with effective interactions.

9.2.4 Plots for 7 Electrons

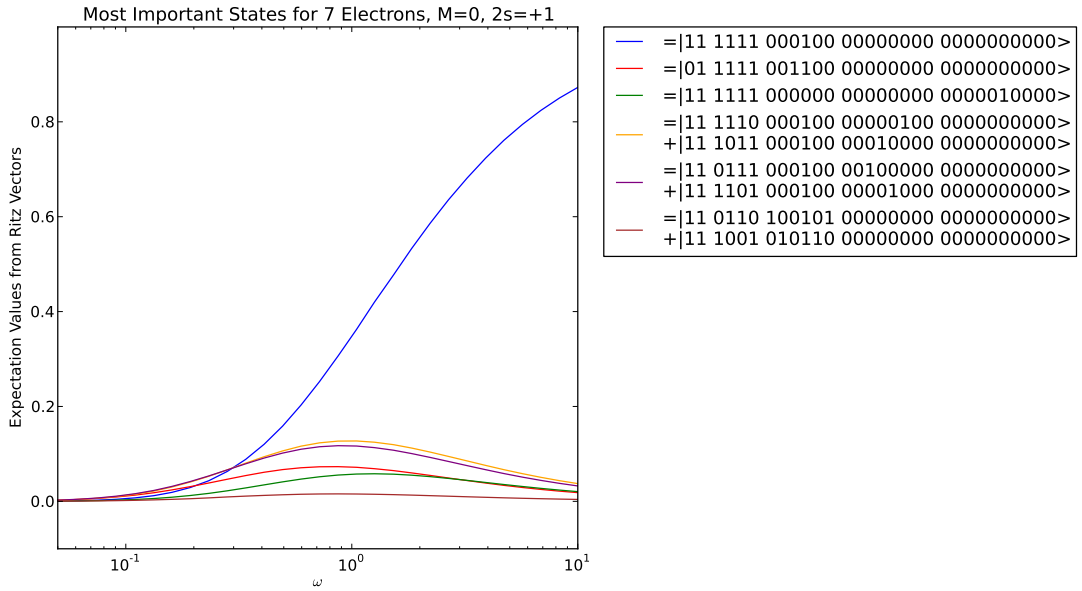

Figure 9.11: The dominating single-particle configurations for a system of 7 electrons in 6 shells with $M = 0$ and $2s = 1$, and with effective interactions.

Conclusion and Prospects

One of the main goals of this master's project was to develop a flexible and efficient code for solving many-body problems in quantum mechanics by use of object oriented programming techniques. The resulting code, contained in a library we named `libTardis`, is a pure object oriented library. The code is divided into three parts. One part, the `System` object, handles the configuration and data of the system we wish to study, the quantum mechanical potential we place our system in is contained in the `Potential` class, and the combination of these can be passed on to various solver classes. Included is a simple diagonalisation solver and a FCI solver based on the Lanczos algorithm.

We have performed a number of tests of `libTardis`' performance and scalability. Through several stages of refining and optimising the code, it has been rewritten using more efficient algorithms. A lot was gained by implementing binary manipulation of Slater determinants, and the important search function that searches through our basis was optimised using a binary comparison algorithm and a binary search algorithm.

The initialisation part of the code, that is building the interaction elements of the potential, was improved significantly by incorporating parts of a code written by Simen Kvaal [35]. This code has been used to generate the two-particle interaction elements the FCI implementation of the Lanczos algorithm requires at its core.

Lastly, the implementation of both on-node shared memory parallelisation and multi-node parallelisation has greatly increased the dimensionality of systems we are able to study. The capacity of the code increased from a few thousand basis elements in the early days, to over three hundred million in the largest calculations we eventually performed.

The Quantum Dot

The target of the research in this project was quantum dots. We selected to work with single-potential two-dimensional quantum dots that can be approximated by a simple two-dimensional harmonic oscillator. By taking advantage of the optimised bare interaction as well as the effective interaction code by Kvaal, we were able to produce very good ground state calculations for several systems of quantum dots. The largest being two electrons in thirty shells and thirteen electrons in six shells. The latter exceeding ninety thousand hours of CPU time on the Univeristy of Oslo's new super computer *Abel*.

The implementation of OpenMP ran into a few problems when we needed to write to a single Lanczos vector from multiple processes. A preliminary solution was to provide one such vector for each process, but this is an inherently wasteful approach. This did not pose a problem for the systems we have been performing calculations on though, but it will prevent the code from running on larger systems. There are solutions to this that will undoubtedly decrease performance somewhat, but at the same time it will increase the capacity of `libTardis`.

Comparison to Published Results

There exist a lot of published and unpublished data on the ground state energies of various configurations of quantum dots that are suitable for comparison with the FCI method. Some are calculated by other implementations of FCI, as well as coupled cluster and Monte Carlo methods. It was always a goal to produce comparable results to these. That goal was achieved, and not only served as a benchmark for early development of the code, but also gave us confidence in the ability of the code to calculate reliable results where no comparison results were found. Due to the computationally expensive nature of these calculations, not much data is available for the largest configurations explored.

The FCI Method

The principle of the FCI method is relatively simple compared to other more efficient methods. However, the FCI method is still worth the effort. For a given configuration, it provides exact results and serves as a benchmark for other methods that can then be scaled beyond the reach of FCI. During this project it became clear, as we already knew, that the FCI method runs into problems with dimensionality relatively quickly. By implementing effective interactions, we were able to produce good results while reducing the dimensionality of the system we perform calculations on.

That said, even with further optimisations of the code with regards to scalability, not a lot more can be gained due to the exponential nature of the dimensionality as a function of its parameters. However, an efficient FCI code can be used on small scale systems where we wish to study other properties than necessarily just the ground state and first few excited energy states of a system.

Preliminary Results

Other interesting and preliminary studies done with `libTardis` was to look at how the lowest energy state evolved with increasing values of total angular momentum and spin. We have shown that these have distinct structures that change with the value of the harmonic oscillator frequency.

We also did a study of which electron configurations dominate in quantum dots of different number of electrons. These results serve as a basis for checking the assumptions that are made about the nature of such systems when studies using coupled cluster and diffusion Monte Carlo methods are performed. The results produced go a long way to confirm these assumptions.

Prospects and Further Studies

We have far from explored all possibilities available to us with an FCI code like `libTardis`. A couple of studies on the table early on, if time would allow, was to look at one and two-body densities for quantum dots as well as time evolution. Time did unfortunately *not* allow for this, although code development of the one and two-body density calculations were started.

Further studies are also required to say something more about the additional results we provided. They were finished calculating only shortly before the deadline of this

thesis. On such possible study is looking more into the contribution of single-particle configurations to the wave function. More calculations are needed that expand these to the full wave function as we only looked at one block of the Hamiltonian, but already at this stage it looks promising.

Also an error study of the relations between frequency, angular momentum and spin is required in order to conclude anything certain about the finer structure of these results. It would also be interesting to add a magnetic field to the potential in order to study further its effect on different spin states.

However, the main prospects of `libTardis` is the extension of the library to include a wider variety of potentials such as three-dimensional harmonic and also atomic nuclei. The latter both for studying atomic electrons as well as neutron and protons in the nucleus. This will also require extending the Lanczos algorithm to three-body interactions. An extension that in itself is simple, but computationally expensive.

It is the personal goal of the author of this code and this thesis to do so should such an opportunity arise. In fact this has been kept in mind all along as the code has been developed.

Part IV

APPENDICES

A
Additional Source Code

A.1 Single-Node Usage Example of libTardis

```
//# Threads : 1

#include <cstdlib>
#include <iostream>
#include <fstream>
#include "../libTardis/libTardis.hpp"

using namespace std;
using namespace tardis;

int main(int argc, char* argv[]) {

    stringstream ssOut;

    int    iShells    = 6;
    int    iParticles = 8;
    int    iM         = 4;
    int    iMs        = 2;
    bool   bEnergyCut = false;

    double dOmega     = 0.1;
    double dLambda    = 0.0;

    System *oSystem = new System();
    if(argc > 1) oSystem->SetLogFile(argv[1]);

    ofstream oOutput;
    oOutput.open("tempQueue/output.txt");

    ssOut << endl;
    ssOut << "System Config:" << endl;
    ssOut << endl;
    ssOut << "Shells:     " << iShells << endl;
    ssOut << "Particles:  " << iParticles << endl;
    ssOut << "Total M:    " << iM << endl;
    ssOut << "Total Spin: " << iMs << endl;
    ssOut << "Omega:      " << dOmega << endl;
    ssOut << "Lambda:     " << dLambda << endl;
    ssOut << endl;
    oSystem->GetLog()->Output(&ssOut);

    oSystem->SetPotential(iShells, QDOT2D, Q2D_EFFECTIVE);
    oSystem->SetParticles(iParticles);
    oSystem->SetQNumber(QN_M, iM);
    oSystem->SetQNumber(QN_MS, iMs);
    oSystem->SetVariable(VAR_LAMBDA, dLambda);
    oSystem->SetVariable(VAR_OMEGA, dOmega);
    oSystem->EnableEnergyCut(bEnergyCut);
    oSystem->BuildPotential();
    oSystem->BuildBasis();

    Lanczos oLanczos(oSystem);
    double dEnergy = oLanczos.Run();

```

```
55      cout << "Energy: " << setprecision(10) << setw(11) << dEnergy << endl;
56      oOutput << "P 8, ";
57      oOutput << "Sh 6, ";
58      oOutput << "M 4, ";
59      oOutput << "Ms 2, ";
60      oOutput << "Om 0.1, ";
61      oOutput << "Lm 0.0, ";
62      oOutput << "NoECut, ";
63      oOutput << "Veff, ";
64      oOutput << "Lz";
65      oOutput << "\t | Energy: " << setprecision(10) << setw(11) << dEnergy << endl;
66
67      oOutput.close();
68
69      return 0;
70  }
```

Listing A.1: Single-Node Usage Example of `libTardis`.

A.2 Multi-Node Usage Example of libTardis

```
1   //# Nodes : 2
2
3   #include <cstdlib>
4   #include <iostream>
5   #include <fstream>
6   #include "libTardis/libTardis.hpp"
7   #include "mpi.h"
8
9   using namespace std;
10  using namespace tardis;
11  using namespace arma;
12
13  int main(int argc, char* argv[]) {
14
15      //
16      // Job Configurastion
17      //
18
19      int     iShells      = 8;
20      int     iParticles   = 4;
21      int     iM           = 0;
22      int     iMs          = 0;
23      bool    bEnergyCut   = false;
24
25      double dOmega        = 1.0;
26      double dLambda       = 0.0;
27
28      int     iSystem      = QDOT2D;
29      int     iInteraction = Q2D_EFFECTIVE;
30
31      //
32      // OpenMPI Code
33      //
34
35      stringstream ssOut;
36      ofstream     oOutput;
37      time_t       tTime;
38      double       dEnergy = 0.0;
39      int          iProc, iRank;
40
41      MPI_Init(&argc, &argv);
42      MPI_Comm_size(MPI_COMM_WORLD, &iProc);
43      MPI_Comm_rank(MPI_COMM_WORLD, &iRank);
44
45      System *oSystem = new System();
46
```

```
47      if(iRank == 0) {
48          ssOut << endl;
49          ssOut << "System Config:" << endl;
50          ssOut << endl;
51          ssOut << "Shells:      " << iShells << endl;
52          ssOut << "Particles:   " << iParticles << endl;
53          ssOut << "Total M:     " << iM << endl;
54          ssOut << "Total Spin:  " << iMs << endl;
55          ssOut << "Omega:       " << dOmega << endl;
56          ssOut << "Lambda:      " << dLambda << endl;
57          ssOut << endl;
58          oSystem->GetLog()->Output(&ssOut);
59      } else {
60          oSystem->GetLog()->SetSilent(true);
61      }
62
63      // Build System
64      oSystem->SetPotential(iShells, iSystem, iInteraction);
65      oSystem->SetParticles(iParticles);
66      oSystem->SetQNumber(QN_M, iM);
67      oSystem->SetQNumber(QN_MS, iMs);
68      oSystem->SetVariable(VAR_LAMBDA, dLambda);
69      oSystem->SetVariable(VAR_OMEGA, dOmega);
70      oSystem->EnableEnergyCut(bEnergyCut);
71      oSystem->BuildPotential();
72
73      if(iRank == 0)  {
74          time(&tTime);
75          cout << "Starting building basis: " << ctime(&tTime);
76      }
77      oSystem->BuildBasis();
78
79      if(iRank == 0)  {
80          time(&tTime);
81          cout << "Starting Lanczos: " << ctime(&tTime);
82      }
83      Lanczos oLanczos(oSystem);
84
85      int     iReady=0;
86      int     iNodes=0;
87      int     iBasisDim = oSystem->GetBasis()->GetSize();
88      int     iDone = 0;
89      double dTStart, dTStop;
90
91      vector<int>     vChunk(iProc+1);
92      vector<double>  vTime(iProc);
93      vector<double>  vReturn(iBasisDim);
94      vector<double>  vSend(iBasisDim);
95
96      for(int i=0; i<iProc; i++)  vTime[i]  = 1;
97
98      // Master Node
99      if(iRank == 0)  {
100
101         iReady = 1;
102         MPI_Reduce(&iReady, &iNodes, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
103
104         if(iNodes == iProc) {
105
106             ssOut << endl;
107             ssOut << iProc << " nodes are ready ..." << endl;
108             oSystem->GetLog()->Output(&ssOut);
109
110             double dTAvg;
111
112             Col<double>    *mLzV;
113             Col<double>    *mLzW;
114             Row<double>    *mLzA;
115             Row<double>    *mLzB;
```

```
116             Row<double>     *mLzC;
117             Row<double>     *mLzE;
118             Col<double>     *mEnergy;
119             Col<int>         mLzI;
120             Col<int>         mLzN;
121             vector<double>   vLzW;
122             vector<int>      vPrev(iProc);
123
124             for(int i=0; i<=iProc; i++) vChunk[i] = ceil((iBasisDim/(double)iProc)*i);
125             vChunk[iProc] = iBasisDim;
126
127             mLzV    = oLanczos.GetLanczosVectorV();
128             mLzW    = oLanczos.GetLanczosVectorW();
129             mLzA    = oLanczos.GetLanczosVectorA();
130             mLzB    = oLanczos.GetLanczosVectorB();
131             mLzC    = oLanczos.GetLanczosVectorC();
132             mLzE    = oLanczos.GetLanczosVectorE();
133             mEnergy = oLanczos.GetEnergies();
134
135             mLzI.quiet_load("LanczosI.arma");
136             if(mLzI.n_rows == 0) {
137                 oLanczos.RunInit();
138                 ssOut << "Master node initialised ..." << endl;
139                 ssOut << endl;
140                 oSystem->GetLog()->Output(&ssOut);
141                 mLzI.zeros(1);
142             } else {
143                 mLzV->quiet_load("LanczosV.arma");
144                 mLzW->quiet_load("LanczosW.arma");
145                 mLzA->quiet_load("LanczosA.arma");
146                 mLzB->quiet_load("LanczosB.arma");
147                 mLzC->quiet_load("LanczosC.arma");
148                 mLzE->quiet_load("LanczosE.arma");
149
150                 mLzN.quiet_load("LanczosN.arma");
151                 oLanczos.SetLanczosIt(mLzI(0));
152
153                 if(mLzN.n_elem == iProc+1) {
154                     vChunk = conv_to< vector<int> >::from(mLzN);
155                 }
156             }
157
158             while(iDone == 0) {
159
160                 dTAvg = 0.0;
161                 for(int i=0; i<iProc; i++) {
162                     dTAvg += vTime[i];
163                     vPrev[i] = vChunk[i+1]-vChunk[i];
164                 }
165                 dTAvg /= iProc;
166                 vChunk[0] = 0;
167                 for(int i=0; i<iProc; i++) {
168                     vChunk[i+1] = vChunk[i] + ceil(vPrev[i]*dTAvg/vTime[i]);
169                     if(vChunk[i+1] > iBasisDim) vChunk[i+1] = iBasisDim;
170                 }
171                 vChunk[iProc] = iBasisDim;
172
173                 ssOut << "Loads: 0:" << vChunk[1]-vChunk[0];
174                 for(int i=1; i<iProc; i++) ssOut << ", " << i << ":" << vChunk[i+1]-
                        vChunk[i];
175                 ssOut << endl;
176
177                 oSystem->GetLog()->Output(&ssOut);
178
179                 MPI_Bcast(&vChunk[0], iProc+1, MPI_INT, 0, MPI_COMM_WORLD);
180
181                 time(&tTime);
182                 cout << "Starting new iterations : " << ctime(&tTime);
183
```

```
184                 vLzW = conv_to< vector<double> >::from(*mLzW);
185                 MPI_Bcast(&vLzW[0], iBasisDim, MPI_DOUBLE, 0, MPI_COMM_WORLD);
186
187                 time(&tTime);
188                 cout << "Done broadcasting       : " << ctime(&tTime);
189
190                 dTStart = MPI_Wtime();
191                 oLanczos.RunSlave(*mLzW, vSend, vChunk[iRank], vChunk[iRank+1]);
192                 dTStop  = MPI_Wtime()-dTStart;
193                 MPI_Gather(&dTStop, 1, MPI_DOUBLE, &vTime[0], 1, MPI_DOUBLE, 0, MPI_
                        COMM_WORLD);
194
195                 ssOut << "Times: 0:" << ceil(vTime[0]);
196                 for(int i=1; i<iProc; i++) ssOut << ", " << i << ":" << ceil(vTime[i])
                        ;
197                 ssOut << endl;
198
199                 time(&tTime);
200                 cout << "Done calculating        : " << ctime(&tTime);
201
202                 MPI_Reduce(&vSend[0], &vReturn[0], iBasisDim, MPI_DOUBLE, MPI_SUM, 0,
                        MPI_COMM_WORLD);
203                 for(int j=0; j<iBasisDim; j++) mLzV->at(j) += vReturn[j];
204
205                 time(&tTime);
206                 cout << "Done receiving          : " << ctime(&tTime) << endl;
207
208                 iDone = oLanczos.RunMaster();
209
210                 mLzV->save("LanczosV.arma");
211                 mLzW->save("LanczosW.arma");
212                 mLzA->save("LanczosA.arma");
213                 mLzB->save("LanczosB.arma");
214                 mLzC->save("LanczosC.arma");
215                 mLzE->save("LanczosE.arma");
216
217                 mLzI(0) = oLanczos.GetLanczosIt();
218                 mLzI.save("LanczosI.arma");
219
220                 mLzN = conv_to< Col<int> >::from(vChunk);
221                 mLzN.save("LanczosN.arma");
222
223                 MPI_Bcast(&iDone, 1, MPI_INT, 0, MPI_COMM_WORLD);
224
225                 cout << endl;
226             }
227
228             dEnergy = mEnergy->at(0);
229
230             ssOut << endl;
231             ssOut << "Eigenvalues:" << endl;
232             ssOut << *mEnergy << endl;
233             ssOut << endl;
234             oSystem->GetLog()->Output(&ssOut);
235             oSystem->GetBasis()->Save("Coeff.arma", SAVE_COEFF_ARMA);
236
237         } else {
238
239             ssOut << endl;
240             ssOut << "Not all nodes are answering, exiting ..." << endl;
241             ssOut << endl;
242             oSystem->GetLog()->Output(&ssOut);
243
244             MPI_Finalize();
245             return 0;
246
247         }
248
249     // Slave Nodes
```

```
250     } else {
251
252         iReady = 1;
253         MPI_Reduce(&iReady, &iNodes, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
254
255         Col<double>    mLzW;
256         vector<double> vLzW(iBasisDim);
257         vector<double> vLzV(iBasisDim);
258
259         while(iDone == 0) {
260             MPI_Bcast(&vChunk[0], iProc+1, MPI_INT, 0, MPI_COMM_WORLD);
261             MPI_Bcast(&vLzW[0], iBasisDim, MPI_DOUBLE, 0, MPI_COMM_WORLD);
262             mLzW = conv_to< Col<double> >::from(vLzW);
263             dTStart = MPI_Wtime();
264             oLanczos.RunSlave(mLzW, vSend, vChunk[iRank], vChunk[iRank+1]);
265             dTStop  = MPI_Wtime()-dTStart;
266             MPI_Gather(&dTStop, 1, MPI_DOUBLE, &vTime[0], 1, MPI_DOUBLE, 0, MPI_COMM_
                WORLD);
267             MPI_Reduce(&vSend[0], &vReturn[0], iBasisDim, MPI_DOUBLE, MPI_SUM, 0, MPI_
                COMM_WORLD);
268             MPI_Bcast(&iDone, 1, MPI_INT, 0, MPI_COMM_WORLD);
269         }
270
271     }
272
273     // Data Output
274     if(iRank == 0) {
275         cout << endl;
276         cout << "Energy: " << setprecision(10) << setw(11) << dEnergy << endl;
277     }
278
279     MPI_Finalize();
280
281     return 0;
282 }
```

Listing A.2: Multi-Node Usage Example of `libTardis`.

B
Data Points and CPU Time for Plots

B.1 CPU Time

This appendix lists all the data points calculated for the plots in section 9.1, table B.1 lists the total run-times for the data of each of those plots, and table B.2 lists the run-times for the plots in section 9.2.

Job	Computer	Cores	Run Time	CPU Time
6 Electrons, $\omega = 0.01$	TheBeast	6	08:55:42	54
6 Electrons, $\omega = 0.1$	TheBeast	6	07:55:18	48
6 Electrons, $\omega = 1.0$	TheBeast	6	05:49:14	36
6 Electrons, $\omega = 5.0$	TheBeast	6	07:08:30	42
7 Electrons, $\omega = 0.01$	TheBeast	6	2:21:01:15	414
7 Electrons, $\omega = 0.1$	Gizmo	8	2:01:54:12	400
7 Electrons, $\omega = 1.0$	Sigma	4	4:06:16:09	408
7 Electrons, $\omega = 5.0$	TheBeast	6	1:14:37:46	234
8 Electrons, $\omega = 0.01$	Abel	16	8:00:30:42	3088
8 Electrons, $\omega = 0.1$	TheBeast	6	21:01:17:49	3030
8 Electrons, $\omega = 1.0$	Gizmo	8	14:09:10:00	2760
8 Electrons, $\omega = 5.0$	Abel	16	4:16:40:36	1808

Table B.1: Each row represents the data calculated for each of the sub plots presented in 9.1. The run time listed for *Abel* here is the total run time across the nodes, i.e. as if the job was run on one node only.

Job	Computer	Data Points	Cores	Run Time	CPU Time
4 Electrons	Sigma	50	4	00:14:23	1
5 Electrons	Sigma	50	4	03:20:48	13
6 Electrons	Sigma	50	4	1:12:09:37	145
7 Electrons	Sigma/Gizmo	50	4/8	12:06:00:27	1451

Table B.2: Each row represents the data calculated for each of the sub plots presented in 9.2.

B.2 6 Electron Data

B.2.1 Dimension of Basis

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2
0	115 148	82 474	28 787	3 826
1	113 624	81 419	28 373	3 766
2	109 411	78 310	27 269	3 607
3	102 574	73 387	25 451	3 351
4	93 792	66 972	23 151	3 020
5	83 434	59 508	20 429	2 638
6	72 344	51 439	17 551	2 236
7	60 932	43 237	14 597	1 826
8	49 961	35 297	11 801	1 443
9	39 716	27 966	9 206	1 098
10	30 672	21 467	6 965	802
11	22 892	15 946	5 058	558
12	16 563	11 434	3 550	372
13	11 520	7 901	2 373	232
14	7 740	5 242	1 526	137
15	4 966	3 330	922	75
16	3 060	2 014	532	36
17	1 778	1 155	281	15
18	988	622	141	6
19	504	312	61	1
20	244	143	25	–
21	104	59	7	–
22	41	21	2	–
23	12	6	–	–
24	4	1	–	–

Table B.3: Dimension of basis after selection of M and Spin for 6 electrons in 6 shells. Dimension of the full basis is $5.25\text{e}6$

B.2.2 For Omega = 0.01

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2
0	0.719735	0.721544	0.721541	0.729692
1	0.721193	0.721204	0.721538	0.734301
2	0.721442	0.721446	0.722376	0.732256
3	0.722440	0.722400	0.723170	0.723170
4	0.723971	0.723999	0.724006	0.737334
5	0.725572	0.725567	0.725573	0.731857
6	0.727304	0.727284	0.727284	0.739640
7	0.730035	0.729713	0.730701	0.748152
8	0.732471	0.733032	0.733032	0.748538
9	0.736017	0.736001	0.737040	0.737040
10	0.739169	0.740377	0.740377	0.745927
11	0.743180	0.743180	0.743985	0.758261
12	0.746540	0.747842	0.748149	0.764296
13	0.751799	0.751799	0.751799	0.770171
14	0.756145	0.756450	0.756449	0.767508
15	0.760993	0.761179	0.761179	0.761179
16	0.765967	0.765960	0.765959	0.783999
17	0.771099	0.771099	0.772416	0.794241
18	0.776409	0.776405	0.776404	0.794415
19	0.782224	0.782224	0.785775	0.833608
20	0.788722	0.788176	0.788176	–
21	0.794754	0.794752	0.818943	–
22	0.800414	0.800414	0.823125	–
23	0.812027	0.812027	–	–
24	0.814201	0.841758	–	–

Table B.4: E_0 for 6 electrons in 6 shells, with $\omega = 0.01$, with effective interactions.

B.2.3 For Omega = 0.1

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2
0	3.560683	3.598380	3.598379	3.628416
1	3.586357	3.586369	3.617586	3.703067
2	3.605932	3.607738	3.620340	3.765878
3	3.594215	3.594218	3.634423	3.683067
4	3.623603	3.623599	3.656179	3.757303
5	3.649546	3.649542	3.649536	3.682965
6	3.655389	3.690058	3.690009	3.767247
7	3.688451	3.688449	3.728720	3.836893
8	3.733282	3.740447	3.772667	3.926373
9	3.771480	3.775132	3.829922	3.832262
10	3.823604	3.830177	3.842584	3.885257
11	3.871254	3.871261	3.890677	3.966136
12	3.935461	3.938641	3.938637	4.057584
13	3.985052	3.985051	3.985050	4.147563
14	4.046442	4.046442	4.046440	4.198596
15	4.103862	4.103866	4.103865	4.103860
16	4.190112	4.190107	4.190107	4.311066
17	4.277101	4.277099	4.278684	4.415521
18	4.365927	4.365868	4.365855	4.502482
19	4.452413	4.452430	4.459001	4.744131
20	4.542561	4.542547	4.542547	–
21	4.634539	4.634538	4.777484	–
22	4.726422	4.726422	4.872790	–
23	4.824756	4.824756	–	–
24	4.914604	5.075989	–	–

Table B.5: E_0 for 6 electrons in 6 shells, with $\omega = 0.1$, with effective interactions.

B.2.4 For Omega = 1.0

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2
0	20.173913	21.101493	21.101414	21.991643
1	20.691501	20.691510	21.619869	22.853564
2	21.228374	21.228295	21.228771	23.641041
3	20.730298	20.730305	21.731427	22.774287
4	21.299027	21.298971	22.249363	23.608688
5	21.838858	21.838785	21.838777	22.777288
6	21.485339	22.493366	22.492053	23.650713
7	22.112052	22.112050	23.094058	24.447031
8	22.780531	22.780809	23.742951	25.332559
9	23.422283	23.422872	24.422802	24.422759
10	24.087444	24.143708	24.144056	25.201357
11	24.798895	24.798888	24.872285	26.069178
12	25.523667	25.523794	25.564302	26.997742
13	26.252388	26.252327	26.252948	27.896044
14	26.988677	26.988666	26.988662	28.727919
15	27.736636	27.736650	27.736654	27.736623
16	28.692343	28.692340	28.692326	30.363500
17	29.650360	29.650358	29.674410	31.298158
18	30.607781	30.625048	30.653963	32.260079
19	31.583247	31.583260	31.626293	34.994362
20	32.562567	32.562574	32.562556	–
21	33.575330	33.575309	35.286720	–
22	34.595110	34.596263	36.316103	–
23	35.613609	35.613609	–	–
24	36.650695	38.442673	–	–

Table B.6: E_0 for 6 electrons in 6 shells, with $\omega = 1.0$, with effective interactions.

B.2.5 For Omega = 5.0

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2
0	74.943501	81.855477	81.855203	88.680330
1	78.560378	78.560381	85.523173	93.321190
2	82.197887	82.205019	82.197845	97.752100
3	78.668548	78.668560	85.837192	93.114864
4	82.453774	82.453744	89.493792	97.662082
5	86.141985	86.141814	86.141634	93.126994
6	83.000793	90.188718	90.188495	97.774531
7	86.963236	86.963239	94.088725	102.24463
8	91.024291	91.024817	98.099414	106.86297
9	95.042784	95.042783	102.12056	102.12022
10	99.130301	99.184991	99.184927	106.57599
11	103.29263	103.29284	103.47220	111.22365
12	107.52340	107.52668	107.65694	115.99433
13	111.75807	111.75794	111.81287	120.70147
14	116.04191	116.04183	116.04198	125.33254
15	120.34131	120.34133	120.34148	120.34118
16	125.24881	125.24877	125.24871	134.45567
17	130.15970	130.15971	130.23365	139.23533
18	135.11035	135.11335	135.21463	144.13780
19	140.02854	140.02686	140.15047	158.50624
20	144.99685	144.99684	144.99676	–
21	150.06149	150.06125	159.33470	–
22	155.16068	155.16703	164.45102	–
23	160.23335	160.23335	–	–
24	165.39080	174.88035	–	–

Table B.7: E_0 for 6 electrons in 6 shells, with $\omega = 5.0$, with effective interactions.

B.3 7 Electron Data

B.3.1 Dimension of Basis

Total M	Spin 1/2	Spin 3/2	Spin 5/2	Spin 7/2
0	488 090	266 062	73 222	7 872
1	482 561	262 926	72 330	7 747
2	466 406	253 922	69 692	7 459
3	440 564	239 432	65 508	6 951
4	406 699	220 560	60 030	6 334
5	366 737	198 257	53 628	5 579
6	322 973	173 971	46 657	4 801
7	277 594	148 806	39 519	3 980
8	232 769	124 109	32 544	3 219
9	190 237	100 736	26 040	2 496
10	151 447	79 592	20 206	1 885
11	117 289	61 056	15 189	1 353
12	88 287	45 483	11 030	943
13	64 476	32 786	7 727	616
14	45 628	22 874	5 201	391
15	31 207	15 366	3 355	225
16	20 592	9 943	2 060	125
17	13 061	6 149	1 201	60
18	7 941	3 638	657	28
19	4 601	2 032	335	9
20	2 531	1 074	156	3
21	1 308	524	65	–
22	631	238	23	–
23	279	95	7	–
24	111	34	1	–
25	38	9	–	–
26	11	2	–	–
27	2	–	–	–

Table B.8: Dimension of basis after selection of M and Spin for 7 electrons in 6 shells. Dimension of the full basis is $2.70\text{e}7$

B.3.2 For Omega = 0.01

Total M	Spin 1/2	Spin 3/2	Spin 5/2	Spin 7/2
0	0.977562	0.977560	0.977560	0.977559
1	0.978548	0.978562	0.978542	0.993780
2	0.978495	0.978493	0.978493	0.990229
3	0.979808	0.980650	0.980668	0.988968
4	0.980058	0.981394	0.981397	0.995027
5	0.982012	0.981875	0.985031	1.000354
6	0.982824	0.982925	0.985916	0.993294
7	0.984898	0.986295	0.990122	0.990119
8	0.986628	0.986849	0.990407	0.993184
9	0.989234	0.989233	0.992323	1.003307
10	0.991428	0.991426	0.992956	1.005950
11	0.995069	0.995381	0.995467	1.011814
12	0.997187	0.997186	0.997181	1.012715
13	1.000662	1.000661	1.000659	1.018116
14	1.003367	1.003367	1.003363	1.003362
15	1.007852	1.007997	1.008050	1.028739
16	1.011254	1.011253	1.011252	1.031505
17	1.015451	1.016413	1.017332	1.040484
18	1.019915	1.019915	1.020357	1.042438
19	1.024369	1.024367	1.029040	1.071970
20	1.028851	1.029777	1.030724	1.082072
21	1.033642	1.034779	1.054552	–
22	1.039396	1.040419	1.065687	–
23	1.043196	1.043194	1.067539	–
24	1.051210	1.053312	1.112898	–
25	1.057430	1.080199	–	–
26	1.063469	1.094723	–	–
27	1.096000	–	–	–

Table B.9: E_0 for 7 electrons in 6 shells, with $\omega = 0.01$, with effective interactions.

B.3.3 For Omega = 0.1

Total M	Spin 1/2	Spin 3/2	Spin 5/2	Spin 7/2
0	4.751330	4.781961	4.781855	4.890906
1	4.761248	4.761243	4.791450	4.875031
2	4.745738	4.783457	4.807446	4.897027
3	4.768611	4.781018	4.800923	4.822281
4	4.795780	4.795782	4.823427	4.909554
5	4.796852	4.824611	4.826895	4.990755
6	4.814953	4.814951	4.850829	4.919887
7	4.857507	4.857534	4.892629	4.989453
8	4.896787	4.898414	4.926770	5.047941
9	4.894828	4.933620	4.950911	4.975090
10	4.943128	4.976134	4.995624	5.054037
11	4.986799	5.007417	5.037380	5.120044
12	5.032202	5.049849	5.079337	5.207532
13	5.094475	5.094471	5.131904	5.292391
14	5.147570	5.152160	5.189435	5.198160
15	5.206577	5.206550	5.206555	5.243330
16	5.265345	5.265317	5.277676	5.329605
17	5.330051	5.333134	5.345206	5.460965
18	5.398454	5.398451	5.398450	5.533320
19	5.471497	5.471495	5.498521	5.714021
20	5.548597	5.548593	5.548590	5.809436
21	5.629940	5.629910	5.736913	–
22	5.707700	5.707684	5.842489	–
23	5.795232	5.795219	5.935784	–
24	5.877680	5.892767	6.156783	–
25	5.971106	6.100082	–	–
26	6.053125	6.200210	–	–
27	6.292192	–	–	–

Table B.10: E_0 for 7 electrons in 6 shells, with $\omega = 0.1$, with effective interactions.

B.3.4 For Omega = 1.0

Total M	Spin 1/2	Spin 3/2	Spin 5/2	Spin 7/2
0	26.605256	27.353256	27.353215	29.621739
1	26.937369	26.937350	27.863675	28.947910
2	26.515117	27.476536	28.368163	29.653679
3	27.026167	27.073381	27.976905	28.800605
4	27.546190	27.546201	28.499789	29.665214
5	27.147364	28.069723	28.069693	30.486284
6	27.671859	27.671824	28.623577	29.685183
7	28.273485	28.273442	29.227302	30.487949
8	28.897004	28.896661	29.826716	31.263612
9	28.491864	29.559827	29.559824	30.446094
10	29.170841	30.154758	30.248043	31.285347
11	29.814872	29.889875	30.861171	32.048331
12	30.513208	30.549129	31.508315	32.972337
13	31.212784	31.212764	32.228863	33.827503
14	31.921849	31.934717	32.922890	32.922748
15	32.638703	32.638616	32.638605	33.706803
16	33.373708	33.373926	33.467743	34.562290
17	34.158097	34.186610	34.264309	35.557323
18	34.977874	35.005642	35.035977	36.451347
19	35.823612	35.823586	35.918674	38.980111
20	36.684641	36.684560	36.684549	39.863712
21	37.633317	37.633294	39.237672	–
22	38.592790	38.592917	40.183195	–
23	39.563715	39.564895	41.180816	–
24	40.534931	40.535110	43.866651	–
25	41.530798	43.182282	–	–
26	42.541726	44.202017	–	–
27	45.288237	–	–	–

Table B.11: E_0 for 7 electrons in 6 shells, with $\omega = 1.0$, with effective interactions.

B.3.5 For Omega = 5.0

Total M	Spin 1/2	Spin 3/2	Spin 5/2	Spin 7/2
0	98.340522	104.63863	104.63858	119.77502
1	101.36948	101.36947	108.29714	115.62715
2	98.052561	105.07385	111.92210	119.87284
3	101.65760	101.76274	108.63687	115.26248
4	105.26948	105.26914	112.31569	119.86572
5	101.97547	108.87682	108.87630	124.38362
6	105.63386	105.63387	112.64526	119.94278
7	109.49637	109.49370	116.52308	124.29612
8	113.41042	113.40988	120.38688	128.73268
9	110.12836	117.35016	117.34952	124.21732
10	114.18367	121.37788	121.50475	128.78111
11	118.23216	118.29150	125.44762	133.18051
12	122.39965	122.40326	129.48761	137.91646
13	126.47673	126.46965	133.65150	142.46362
14	130.63104	130.62972	137.74920	137.74868
15	134.79927	134.79857	134.79851	142.24527
16	139.07772	139.07760	139.31327	146.86591
17	143.47927	143.55638	143.76118	151.68858
18	147.97448	148.04114	148.17355	156.42304
19	152.53860	152.53840	152.72312	170.33938
20	157.14716	157.14695	157.14686	174.97783
21	162.04119	162.04096	171.16151	–
22	166.99122	166.99548	175.90050	–
23	171.87407	171.95618	180.92001	–
24	176.86509	176.86509	195.22737	–
25	181.93102	191.06409	–	–
26	187.02798	196.13804	–	–
27	201.43463	–	–	–

Table B.12: E_0 for 7 electrons in 6 shells, with $\omega = 5.0$, with effective interactions.

B.4 8 Electron Data

B.4.1 Dimension of Basis

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2	Spin 8/2
0	2 083 055	1 584 648	681 564	151 564	13 362
1	2 061 832	1 568 509	674 270	149 881	13 204
2	1 999 419	1 520 225	652 781	144 726	12 697
3	1 899 358	1 443 387	618 424	136 635	11 919
4	1 767 260	1 341 546	573 160	125 900	10 880
5	1 610 212	1 221 009	519 576	113 360	9 687
6	1 436 268	1 087 261	460 481	99 524	8 373
7	1 253 700	947 445	398 840	85 294	7 049
8	1 070 465	807 023	337 387	71 165	5 746
9	893 580	672 047	278 580	57 874	4 551
10	728 807	546 335	224 319	45 714	3 475
11	580 338	433 621	176 000	35 118	2 570
12	450 789	335 363	134 382	26 114	1 820
13	341 224	252 760	99 736	18 829	1 241
14	251 410	185 167	71 827	13 076	802
15	180 048	131 862	50 110	8 767	495
16	125 133	90 948	33 782	5 617	283
17	84 232	60 773	21 958	3 454	153
18	54 795	39 131	13 710	2 004	73
19	34 352	24 296	8 194	1 106	32
20	20 687	14 424	4 661	562	11
21	11 916	8 203	2 510	268	3
22	6 533	4 404	1 267	112	–
23	3 386	2 244	594	43	–
24	1 645	1 054	253	12	–
25	740	463	96	3	–
26	303	178	31	–	–
27	110	63	8	–	–
28	34	16	1	–	–
29	8	4	–	–	–
30	1	–	–	–	–

Table B.13: Dimension of basis after selection of M and Spin for 8 electrons in 6 shells. Dimension of the full basis is $1.18\text{e}8$

B.4.2 For Omega = 0.01

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2	Spin 8/2
0	1.2669000	1.2668800	1.2668700	1.2709500	1.2795100
1	1.2669200	1.2669300	1.2669400	1.2669100	1.2811600
2	1.2668600	1.2668600	1.2668500	1.2720300	1.2810600
3	1.2671600	1.2671400	1.2671400	1.2686700	1.2823300
4	1.2681300	1.2681300	1.2694200	1.2730800	1.2743400
5	1.2680900	1.2680900	1.2698200	1.2715100	1.2733500
6	1.2690800	1.2714000	1.2714000	1.2741400	1.2879400
7	1.2712800	1.2712800	1.2721700	1.2745500	1.2868700
8	1.2723200	1.2722400	1.2757700	1.2770400	1.2952900
9	1.2742800	1.2740600	1.2761300	1.2777900	1.2880600
10	1.2756700	1.2776200	1.2776400	1.2813800	1.2978700
11	1.2774800	1.2774800	1.2798200	1.2825500	1.2857200
12	1.2811400	1.2811200	1.2821800	1.2865300	1.2878600
13	1.2833200	1.2833200	1.2833100	1.2856500	1.3129200
14	1.2853500	1.2853500	1.2853400	1.2923300	1.3111000
15	1.2886700	1.2886700	1.2895200	1.2895100	1.3121300
16	1.2916500	1.2916500	1.2920100	1.2987400	1.3250400
17	1.2946600	1.2946700	1.2952400	1.2952800	1.3192800
18	1.2979700	1.2980000	1.2980100	1.3080300	1.3548300
19	1.3014200	1.3014700	1.3025300	1.3025400	1.3570200
20	1.3044600	1.3044700	1.3067100	1.3316300	1.3731200
21	1.3088800	1.3088800	1.3100500	1.3353700	1.4051300
22	1.3114400	1.3145400	1.3145100	1.3446700	–
23	1.3182600	1.3182700	1.3182600	1.3487100	–
24	1.3211000	1.3211000	1.3340900	1.3833000	–
25	1.3273900	1.3273900	1.3556500	1.3949800	–
26	1.3326200	1.3428900	1.3591900	–	–
27	1.3377800	1.3377800	1.3733000	–	–
28	1.3521900	1.3764000	1.4319100	–	–
29	1.3840700	1.3840700	–	–	–
30	1.4286900	–	–	–	–

Table B.14: E_0 for 8 electrons in 6 shells, with $\omega = 0.01$, with effective interactions.

B.4.3 For Omega = 0.1

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2	Spin 8/2
0	6.070217	6.070215	6.098501	6.131079	6.147338
1	6.085101	6.085099	6.095533	6.143404	6.242197
2	6.079783	6.079771	6.102718	6.130976	6.197399
3	6.100743	6.100718	6.123599	6.123592	6.260962
4	6.090732	6.113073	6.113080	6.145577	6.195881
5	6.107664	6.107652	6.143896	6.186346	6.298900
6	6.135045	6.140805	6.140769	6.209126	6.326956
7	6.173498	6.173044	6.186099	6.231543	6.255437
8	6.171510	6.171507	6.204337	6.263169	6.331207
9	6.206086	6.206087	6.242525	6.271437	6.410030
10	6.247899	6.247889	6.269899	6.293690	6.348678
11	6.290761	6.290771	6.307734	6.333268	6.428769
12	6.286382	6.336576	6.345166	6.368131	6.489935
13	6.374008	6.374009	6.390260	6.419566	6.545443
14	6.403407	6.403413	6.434955	6.445914	6.463567
15	6.457529	6.457532	6.491726	6.502951	6.561314
16	6.512229	6.512230	6.521211	6.549710	6.608859
17	6.567085	6.571668	6.590895	6.606641	6.755592
18	6.631737	6.631766	6.631652	6.665019	6.798962
19	6.690620	6.690624	6.697643	6.746559	6.897927
20	6.751430	6.751415	6.751403	6.815160	7.012793
21	6.822497	6.822490	6.844658	6.899119	7.148755
22	6.893578	6.893627	6.895862	7.028038	–
23	6.957055	6.957062	6.957042	7.109029	–
24	7.037319	7.037860	7.050277	7.269300	–
25	7.115777	7.115763	7.211196	7.372056	–
26	7.185125	7.186658	7.320449	–	–
27	7.270494	7.270491	7.423023	–	–
28	7.355896	7.484008	7.637603	–	–
29	7.566397	7.566397	–	–	–
30	7.825989	–	–	–	–

Table B.15: E_0 for 8 electrons in 6 shells, with $\omega = 0.1$, with effective interactions.

B.4.4 For Omega = 1.0

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2	Spin 8/2
0	33.338165	33.338151	34.231886	35.257813	36.078329
1	33.778572	33.778546	33.778490	34.883712	36.954582
2	33.408119	33.408114	34.238580	35.197337	36.195276
3	33.875012	33.875042	34.665319	34.665264	37.007802
4	33.423910	34.334462	34.334400	35.232398	36.172508
5	33.931678	33.931612	34.877534	35.819956	37.041044
6	34.458951	34.458951	34.458927	35.557870	37.807926
7	35.041591	35.041565	35.041559	36.067354	36.961272
8	34.630421	34.630425	35.590488	36.628013	37.769514
9	35.218220	35.218183	36.197008	36.246419	38.573766
10	35.834211	35.834084	35.939924	36.831991	37.784973
11	36.495397	36.495382	36.536638	37.452610	38.690620
12	36.031970	37.151295	37.151426	38.065380	39.460056
13	36.834410	36.834398	37.819465	38.757578	40.257925
14	37.459040	37.459013	38.466314	38.525696	39.385193
15	38.178957	38.178949	39.179144	39.245124	40.252610
16	38.897980	38.897975	38.896209	39.882204	40.957904
17	39.603411	39.630614	39.695647	40.629397	42.015158
18	40.323776	40.323675	40.357766	41.402672	42.737580
19	41.112571	41.112412	41.176651	42.220069	43.722822
20	41.847684	41.863454	41.895083	43.050189	44.645188
21	42.715332	42.708799	42.780148	43.924630	46.980717
22	43.484409	43.484294	43.508958	44.892555	–
23	44.347473	44.347458	44.366248	45.847932	–
24	45.179577	45.179623	45.179531	48.289631	–
25	46.096465	46.096450	47.650104	49.238125	–
26	47.027722	47.054261	48.628113	–	–
27	48.025236	48.025236	49.623346	–	–
28	49.061404	50.632028	52.254056	–	–
29	51.627788	51.627788	–	–	–
30	54.400776	–	–	–	–

Table B.16: E_0 for 8 electrons in 6 shells, with $\omega = 1.0$, with effective interactions.

B.4.5 For Omega = 5.0

Total M	Spin 0/2	Spin 2/2	Spin 4/2	Spin 6/2	Spin 8/2
0	122.33315	122.33312	129.13414	136.31442	142.96444
1	125.66795	125.66644	125.66639	133.05028	147.57179
2	122.53491	122.53339	129.14660	136.16477	143.25661
3	125.98313	125.98333	132.45849	132.45805	147.75631
4	122.55992	129.43873	129.43767	136.25319	143.19271
5	126.15443	126.15407	133.13937	140.06888	147.80281
6	129.76931	129.76953	129.76925	137.16719	152.07818
7	133.56001	133.56056	133.55970	140.79098	147.49172
8	130.25671	130.25669	137.32890	144.51320	151.97289
9	134.08098	134.08100	141.13214	141.12936	156.45527
10	137.98778	138.10013	138.10365	144.99491	152.01303
11	141.98957	141.99005	141.98775	148.94765	156.70721
12	138.53747	145.92943	145.92904	152.88697	160.99667
13	142.75439	142.75439	150.01403	156.94546	165.52171
14	146.77349	146.77348	154.03222	154.03091	160.85804
15	150.92945	150.92935	158.16057	158.27214	165.49017
16	155.04833	155.04780	155.04774	162.27246	169.75956
17	159.32508	159.32318	159.49729	166.57326	174.67852
18	163.46591	163.46603	163.56536	170.95849	178.99964
19	167.91216	167.91152	168.03877	175.39884	183.95531
20	172.13197	172.20421	172.31402	179.91800	188.56006
21	176.73640	176.72173	176.88636	184.57091	202.09191
22	181.09045	181.09117	181.18372	189.35073	–
23	185.68707	185.68754	185.84620	194.26509	–
24	190.19839	190.19839	190.19787	207.95583	–
25	195.07270	195.07243	203.99455	212.82575	–
26	200.04391	200.04563	208.90066	–	–
27	205.01865	205.01866	213.86985	–	–
28	210.17324	219.01470	228.05953	–	–
29	224.07944	224.07943	–	–	–
30	238.56507	–	–	–	–

Table B.17: E_0 for 8 electrons in 6 shells, with $\omega = 5.0$, with effective interactions.

Bibliography

[1] J. J. Brehm and W. J. Mullin, *Introduction to the Structure of Matter*. John Wiley and Sons, Inc, 1989.

[2] D. J. Griffiths, *Introduction to Quantum Mechanics*. Pearson Prentice Hall, 2005.

[3] R. L. Liboff, *Introductory Quantum Mechanics*. Addison Wesley, 4th ed., 2003.

[4] F. Ravndal, "Notes on quantum mechanics," 2006. Lecture Notes.

[5] Y. M. Wang, "Coupled-cluster studies of double qunatum dots," Master's thesis, University of Oslo, 2011.

[6] I. Shavitt and R. J. Bartlett, *Many-Body Methods in Chemistry and Physics*. Cambridge University Press, 2009.

[7] J. C. Slater, "The theory of complex spectra," *Physical Review*, vol. 34, 1929.

[8] G.-C. Wick, "The evaluation of the collision matrix," *Physical Review*, vol. 80, 1950.

[9] M. P. Lohne, "Coupled-cluster studies of qunatum dots," Master's thesis, University of Oslo, 2010.

[10] S. Kvaal, *Analysis of many-body methods for quantrum dots*. PhD thesis, University of Osle, 2009.

[11] L. Fedichkin, M. Yanchenko, and K. A. Valiev, "Novel coherent quantum bit using spatial quantization levels in semiconductor quantum dot," *arXiv: Quantum Physics*, 2000.

[12] K. Hoshino, A. Gopal, M. S. Glaz, D. A. Vanden Bout, and X. Zhang, "Nanoscale fluorescence imaging with quantum dot near-field electroluminescence," *Applied Physics Letters*, vol. 101, 2012.

[13] M. A. Walling, J. A. Novak, and J. R. E. Shepard, "Quantum dots for live cell and in vivo imaging," *International Journal of Molecular Science*, vol. 10, 2009.

[14] A. Luque, A. Martí, and A. J. Nozik, "Solar cells based on quantum dots: Multiple exciton generation and intermediate bands," *MRS Bulletin*, vol. 32, 2007.

[15] M. Ciorga, A. S. Sachrajda, P. Hawrylak, C. Gould, P. Zawadzki, S. Jullian, Y. Feng, and Z. Wasilewski, "Addition spectrum of a lateral dot from coulomb and spin-blockade spectroscopy," *Physical Review B*, vol. 61, 2000.

[16] C. Hirth, "Studies of quantum dots – ab initio coupled-cluster analysis using opencl and gpu programming," Master's thesis, University of Oslo, 2012.

[17] M. Hjorth-Jensen, "Quantum mechanics for many-particle systems," 2012. Lecture Notes.

[18] C. Kittel, *Introduction to Solid State Physics*. John Wiley and Sons, Inc, 8th ed., 2005.

[19] R. R. Whitehead, A. Watt, B. J. Cole, and I. Morrison, "Computational methods for shell-model calculations," *Advances in Nuclear Physics*, vol. 9, 1977.

[20] M. P. Lohne, G. Hagen, M. Hjorth-Jensen, S. Kvaal, and F. Pederiva, "Ab initio computation of the energies of circular quantum dots," *Physical Review B*, vol. 84, 2010.

[21] S. Kvaal, "Harmonic oscillator eigenfunction expansions, quantum dots, and effective interactions," *Physical Review B*, vol. 80, 2009.

[22] K. Varga, P. Navratil, J. Usukura, and Y. Suzuki, "Stochastic variational approach to few-electron artificial atoms," *Physical Review B*, vol. 63, 2001.

[23] S. Kvaal, M. Hjorth-Jensen, and H. M. Nilsen, "Effective interactions, large-scale diagonalization, and one-dimensional quantum dots," *Physical Review B*, vol. 76, 2007.

[24] S. Kvaal, "Geometry of effective hamiltonians," *Physical Review C*, vol. 78, 2008.

[25] F. E. Harris, H. J. Monkhorst, and D. L. Freeman, *Algebraic and Diagrammatic Metjods in Many-Fermion Theory*. Oxford University Press, 1992.

[26] J. M. Thijssen, *Computational Physics*. Cambridge University Press, 2nd ed., 2007.

[27] C. D. Sherrill and H. F. Schaefer, "The configuration interaction method: A.dvances in highly correlated approaches," *Advances in Quantum Chemistry*, vol. 34, 1999.

[28] F. Coester, "Bound states of a many-particle system," *Nuclear Physics*, vol. 7, 1958.

[29] F. Coester and H. Kümmel, "Short-range correlations in nuclear wave functions," *Nuclear Physics*, vol. 17, 1960.

[30] J. Čížek and J. Paldus, "Correlation problems in atomic and molecular systems iii. rederivation of the coupled-pair many-electron theory using the traditional quantum chemical methodst," *International Journal of Quantum Chemistry*, vol. 5, 1971.

[31] T. D. Crawford and H. F. Schaefer, "An introduction to coupled cluster theory for computational chemists." University of Georgia.

[32] M. Hjorth-Jensen, "Computational physics," 2010. Lecture Notes.

[33] F. Pederiva, C. J. Umrigar, and E. Lipparini, "Diffusion monte carlo study of circular quantum dots," *arXiv: Condensed Matter*, 2000.

[34] J. J. Brehm and W. J. Mullin, *Monte Carlo Methods in ab Initio Quantum Chemistry*. World Scientific, 1994.

[35] S. Kvaal, "Open source fci code for quantum dots and effective interactions," *Physical Review E*, 2008.

[36] C. Sanderson, "Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments," *NICTA Technical Report*, 2010.

[37] C. Sanderson, *API Reference for Armadillo 3.4 (Ku De Ta)*. NICTA, Accessed Des. 2012. Available from: http://arma.sourceforge.net/docs.html.

[38] P. Arbenz, "Lecture notes on solving large scale eigenvalue problems," 2012.

[39] G. H. Golub and C. F. Van Loan, *Matrix Computations*. Johns Hopkins, 3rd ed., 1996.

[40] C. C. Paige, "Accuracy and effectiveness of the lanczos algorithm for the symmetric eigenproblem," *Linear Algebra and its Applications*, vol. 34, 1980.

[41] Silicon Graphics Computer Systems, Inc., *Standard Template Library Programmer's Guide*, 1994. Available from: http://www.sgi.com/tech/stl.

[42] P. E. Black, "Dictionary of algorithms and data structures," 2011. Available from: http://www.nist.gov/dads/HTML/binarySearch.html.

[43] E. Anisimovas and A. Matulis, "Energy spectra of few-electron quantum dots," *Journal of Physics: Condensed Matter*, vol. 10, 1998.

[44] M. Rontani, C. Cavazzoni, D. Bellucci, and G. Goldoni, "Full configuration interaction approach to the few-electron problem in artificial atoms," *The Journal of Chemical Physics*, 2006.

[45] K. Leikanger, "Project paper for computational physics ii: Quantum mechanical systems." University of Oslo, 2012.

[46] G. Bårdsen, E. Tölö, and A. Harju, "Magnetism in tunable quantum rings," *Physical Review B*, vol. 80, 2009.

[47] S. M. Reimann, M. Koskinen, and M. Manninen, "On the formation of wigner molecules in small quantum dots," *arXiv: Condensed Matter*, 2000.

List of Tables

4.1 The hardware specifications for the four main computers used for development and testing of `libTardis`. These are also the main computers used for the majority of the calculations where the jobs have been small enough to not require the use of a super computer. 41

6.1 A map of all configurations for a three-shell system. The configurations are a set of two single-particle states, $|p, q\rangle$, running from 0 to 11. See Figure 2.2 for reference for the positions of these states in our shell model. 63

7.1 The lowest eigenvalue for a test run with 4 electrons in 6 shells with $\lambda = 6$, $M = 0$ and $2s = 0$ with bare interactions. 70

7.2 The 8 lowest eigenstates of a system of 4 electrons in 6 shells with $\lambda = 6$, $M = 0$ and $2s = 0$. Presented with error estimates. 71

7.3 Run-times for some of the major jobs. The job column describes jobs by number of particles (P), number of shells (Sh), total M (M), total spin times two (2s) and omega (ω). 'E' indicates effective interactions, while 'B' indicates bare interactions. The column labelled 'It.' lists the number of Lanczos iterations needed for convergence as this number highly affects computation time. All runs have $< 1 \times 10^{-6}$ as convergence criterion.
* Ran on 1280 cores for part of the calculation. 78

7.4 An overview of the efficiency of the OpenMPI implementation. The numbers are based on the output files. One early version of the code did not output each node's calculation time (Calc), so wait time (Wait) and merge time (Merge) is unknown. This unknown time is baked into between calculation time and communication time (Comm). For the other cases, the calculation time is accurate, so in case of negative numbers elsewhere, this is due to round-off errors when calculating the remaining quantities when these numbers are small. This is caused by the clock only counting in integer increments. 79

8.1 E_0 for 2 electrons, with $\omega = 1.0$, $M = 0$ and $2s = 0$, and with effective interactions with and without energy cut and with bare interactions. 85

8.2 E_0 for 4 electrons, with $\omega = 1.0$, $M = 0$ and $2s = 0$, and with effective interactions with and without energy cut and with bare interactions. 86

8.3 Comparison of 6-shell results with both bare and effective interaction elements with Kvaal [35] and Rontani *et al.* [44]. 86

8.4 Comparison of 7-shell results with both bare and effective interaction elements with Kvaal [35] and Rontani *et al.* [44]. 87

8.5 Comparison of 8-shell results with both bare and effective interaction elements with Kvaal [35] and Rontani *et al.* [44]. 87
8.6 9 and 10-shell results with both bare and effective interaction elements. . . 88
8.7 Number of total electrons, N_e, and their distribution across the shells. . . . 88
8.8 E_0 for 2 electrons, with $M = 0$ and $2s = 0$, and with bare interactions. Comparing with results by [20]. 89
8.9 E_0 for 2 electrons, with $M = 0$ and $2s = 0$, and with effective interactions and energy cut. Comparing with results by [20] and [45]. 89
8.10 E_0 for 6 electrons, with $M = 0$ and $2s = 0$, and with bare interactions. Comparing with results by [16]. 90
8.11 E_0 for 6 electrons, with $M = 0$ and $2s = 0$, and with effective interactions. Comparing with results by [20] and [45]. 90
8.12 E_0 for 12 electrons, with $M = 0$ and $2s = 0$, and with effective interactions. Comparing with results by [20] and [45]. 91
8.13 E_0 for 20 electrons, with $M = 0$ and $2s = 0$, and with effective interactions. Comparing with results by [20] and [45]. 92
8.14 Number of total electrons, N_e, and their distribution across the shells. . . . 93
8.15 E_0 for 7 electrons, with $\omega = 0.1$, and with effective interactions. 93
8.16 E_0 for 11 electrons, with $\omega = 0.1$, and with effective interactions. 94
8.17 E_0 for 13 electrons, with $\omega = 0.1$, and with effective interactions. 94

B.1 Each row represents the data calculated for each of the sub plots presented in 9.1. The run time listed for *Abel* here is the total run time across the nodes, i.e. as if the job was run on one node only. 117
B.2 Each row represents the data calculated for each of the sub plots presented in 9.2. 117
B.3 Dimension of basis after selection of M and Spin for 6 electrons in 6 shells. Dimension of the full basis is 5.25e6 118
B.4 E_0 for 6 electrons in 6 shells, with $\omega = 0.01$, with effective interactions. . . 119
B.5 E_0 for 6 electrons in 6 shells, with $\omega = 0.1$, with effective interactions. . . . 120
B.6 E_0 for 6 electrons in 6 shells, with $\omega = 1.0$, with effective interactions. . . . 121
B.7 E_0 for 6 electrons in 6 shells, with $\omega = 5.0$, with effective interactions. . . . 122
B.8 Dimension of basis after selection of M and Spin for 7 electrons in 6 shells. Dimension of the full basis is 2.70e7 123
B.9 E_0 for 7 electrons in 6 shells, with $\omega = 0.01$, with effective interactions. . . 124
B.10 E_0 for 7 electrons in 6 shells, with $\omega = 0.1$, with effective interactions. . . . 125
B.11 E_0 for 7 electrons in 6 shells, with $\omega = 1.0$, with effective interactions. . . . 126
B.12 E_0 for 7 electrons in 6 shells, with $\omega = 5.0$, with effective interactions. . . . 127
B.13 Dimension of basis after selection of M and Spin for 8 electrons in 6 shells. Dimension of the full basis is 1.18e8 128
B.14 E_0 for 8 electrons in 6 shells, with $\omega = 0.01$, with effective interactions. . . 129
B.15 E_0 for 8 electrons in 6 shells, with $\omega = 0.1$, with effective interactions. . . . 130
B.16 E_0 for 8 electrons in 6 shells, with $\omega = 1.0$, with effective interactions. . . . 131
B.17 E_0 for 8 electrons in 6 shells, with $\omega = 5.0$, with effective interactions. . . . 132

List of Figures

2.1 A double lateral quantum dot system. Electrons or holes are confined in the middle layer using the electrostatic gates (structures on the surface). The illustration is from a public domain source. 25

2.2 The structure of single-particle states for a harmonic oscillator potential in two dimensions. Here illustrated as a system of four shells, R, with single-particle states from $|0\rangle$ to $|19\rangle$. Each state represents one unique set of quantum numbers n and m as well as spins $\uparrow$ and $\downarrow$. 28

5.1 Two plots showing computation time of 10^8 creation operations and 10^8 annihilation operations on a `Slater` object using a binary represenation of single-particle states via the `bitset` class. The `bitset` class will use an array of long variables (64-bit in our case) to store these states in the event more than one long variable is needed. The left plot shows the increase in computation time as a function of number of shells, R. As can be seen from this plot, the number of shells only affect the speed whenever the `bitset` object needs to add another long variable of storage to its internal array. A step-like structure can be seen, but as the number of states blow up rapidly, the steps smooth out. The right plot shows the same situation, but now as a function of states (bits). The step-structure for each new 64-but variable is clearly visible. 50

7.1 The convergence properties of a system of 4 electrons in 6 shells with $\lambda = 6$, $M = 0$ and $2s = 0$. The plot illustrates how there may be more than one "shelf" where the eigenvalues seem to converge. The higher shelf converges at $\delta_k < 1 \times 10^{-5}$, while the lower shelf converges at around $\delta_k < 1 \times 10^{-8}$ with a value of 23.638717. 71

7.2 The convergence properties of the Lanczos algorithm using bare and effective interactions on a system of 4 electrons in 8 shells with a set of ω-values. Convergence is defined in the code as $\left(E_0^{(k-1)}/E_0^k\right) - 1$. The plot's y-axis displays the 10-logarithm of the convergence for each iteration of the Lanczos process. 72

7.3 The convergence properties of the Lanczos algorithm using bare and effective interactions on a system of 8 electrons in 6 shells with a set of ω-values. Convergence is defined in the code as $\left(E_0^{(k-1)}/E_0^k\right) - 1$. The plot's y-axis displays the 10-logarithm of the convergence for each iteration of the Lanczos process. 73

7.4 Multi-core scalability on the computer "Lincoln" with the optimised-for-speed version of the matrix-vector function of the Lanczos algorithm. The reason the curve flattens out from 17 cores is due to the hyper-threading kicking in. All 16 physical cores are running at maximum capacity, and hyper-threading is not able to squeeze any more power out of these cores. The line in the plots represents a theoretical linear scaling of the code's performance as a function of cores. The squares are the inverse of the actual computation times divided by the time for only one core, thus showing how many times faster n cores is than 1 core. 74
7.5 Multi-core scalability of `libTardis` on the computer *Gizmo* before and after optimisation of queueing of the main Lanczos vector. The diagonal line in the plots represents a theoretical linear scaling of the code's performance as a function of cores. The horizontal line represents a scaling of one. The squares are the inverse of the actual computation times divided by the time for only one core, thus showing how many times faster n cores is than one core. 76
7.6 The performance of `libTardis` when distributed across multiple nodes on a super computer. The inter-node communication is very fast, and for the relatively small system of 8 electrons in 6 shells does not generate enough traffic to be a significant factor. The line in the plot indicates a linear performance scale, and the squares are the actual computation times. Each node has 16 cores, hence the step-length of 16 along the axes. 77

8.1 Accuracy of the different interaction elements compared to the best result, and comparison between the Lancsoz algorithm and straight forward diagonalisation of the two-electron Hamiltonian. Here for $\omega = 0.1$ and $\omega = 0.28$. 84
8.2 Accuracy of the different interaction elements compared to the best result, and comparison between the Lancsoz algorithm and straight forward diagonalisation of the two-electron Hamiltonian. Here for $\omega = 0.5$ and $\omega = 1.0$. 84
8.3 The left plot shows how bare interactions converge for increasing R towards a baseline of effective interactions for $R = 20$. The right plot shows how the six-electron results with effective interactions converge to its best value for $R = 11$. 92

9.1 Lowest energy eigenvalue for 6 electrons and $R = 6$ as a function of total M and total spin for $\omega = 0.01$ and $\omega = 0.1$ 97
9.2 Lowest energy eigenvalue for 6 electrons and $R = 6$ as a function of total M and total spin for $\omega = 1.0$ and $\omega = 5.0$ 97
9.3 Lowest energy eigenvalue for 7 electrons and $R = 6$ as a function of total M and total spin for $\omega = 0.01$ and $\omega = 0.1$ 98
9.4 Lowest energy eigenvalue for 7 electrons and $R = 6$ as a function of total M and total spin for $\omega = 1.0$ and $\omega = 5.0$ 98
9.5 Lowest energy eigenvalue for 8 electrons and $R = 6$ as a function of total M and total spin for $\omega = 0.01$ and $\omega = 0.1$ 99
9.6 Lowest energy eigenvalue for 8 electrons and $R = 6$ as a function of total M and total spin for $\omega = 1.0$ and $\omega = 5.0$ 99

9.7 The structure of single-particle states for a harmonic oscillator potential in two dimensions. Here illustrated as a system of four shells, R, with single-particle states from $|0\rangle$ to $|19\rangle$. Each state represents a bit of 0 or 1 in our binary Slater determinants. 100
9.8 The dominating single-particle configurations for a system of 4 electrons in 6 shells with $M = 0$ and $2s = 0$, and with effective interactions. 102
9.9 The dominating single-particle configurations for a system of 5 electrons in 6 shells with $M = 0$ and $2s = 1$, and with effective interactions. 103
9.10 The dominating single-particle configurations for a system of 6 electrons in 6 shells with $M = 0$ and $2s = 0$, and with effective interactions. 103
9.11 The dominating single-particle configurations for a system of 7 electrons in 6 shells with $M = 0$ and $2s = 1$, and with effective interactions. 104