# FINITE ELEMENT STUDIES OF QUANTUM DOTS

**by**

**Lene Norderhaug Drøsdal**

***THESIS***

*for the degree of*

***MASTER OF SCIENCE***

*(Master in Computational physics)*


*Faculty of Mathematics and Natural Sciences*
*University of Oslo*

*June 2009*

*Det matematisk- naturvitenskapelige fakultet*
*Universitetet i Oslo*

# Contents

# Chapter 1

# Introduction

In this master thesis we study a two-dimensional quantum mechanical system of two electrons in a harmonic oscillator potential and a magnetic field. This is an approximation to the two-electron quantum dot. A quantum dot is a semiconductor capable of confining a small number of free electrons. The topic of quantum dots is very popular. Because of studies of double quantum dots in quantum computing, the two-electron case is especially interesting.

The quantum mechanical system is described by the Schrödinger equation which is a partial differential equation. We solve the time-independent Schrödinger equation numerically using the finite element method and the finite difference method. In this thesis we choose to focus on the finite element method. This method is not commonly used in quantum mechanics, perhaps because it is much more complicated to implement than the finite difference method. However the finite element method gives us more possibilities. For example, by introducing higher order basis functions, we can improve the truncation error in the same implementation. Another advantage of the finite element method which we have not studied here, is the strength of the method on complex geometries.

A great portion of the time spent on this thesis has been dedicated to developing a program which solves the time-independent Schrödinger equation. For comparison we have implemented both the finite difference and the finite element method in a similar structure. In this thesis we have focused on problems with spherical symmetry, where the equations can be transformed into one-dimensional radial equations.

We begin this thesis by giving a short introduction to quantum mechanics in Chapter 2. This chapter gives the required background needed to study the mathematical model for the quantum dot which we derive in Chapter 3. In Chapter3 we first give a short introduction to quantum dots. Then we

introduce the mathematical model for the quantum dot, specialising on the case of one and two electrons. The equations which we will solve numerically are derived and we also study analytic solutions. We begin by deriving the single-electron quantum dot. Then we show that the two-electron equation can be separated into two independent single particle equations by introducing a new set of coordinates.

In Chapter 4 the numerical methods are described in detail. We focus on the finite element method in one dimension. The implementation of the numerical approximations and program structure is given in Chapter 5. For the full source code, see Appendix B. The results of the numerical simulations and a discussion of the methods are given in Chapter 6. Finally, we summarise the thesis in Chapter 7.

I would like to thank my supervisors Morten Hjorth-Jensen and Xing Cai for their helpful advice. I would also like to thank my family and friends for their support, especially my boyfriend Hoa Binh for keeping me calm during this very stressful time.

# Chapter 2

# Introduction to quantum mechanics

In this chapter we give a short introduction to the basic ideas of quantum mechanics. For more details we refer to general texts on quantum mechanics [1, 2]. First we introduce the Schrödinger equation and the fundamental postulates of quantum mechanics. Then we focus on some topics required to derive the quantum dot equations in Chapter 3. The reader who is familiar with quantum mechanics may skip this chapter.

## 2.1 Quantum mechanical postulates

Because of experiments that could not be explained by classical physics, quantum mechanics was developed during the 20th century. We introduce the quantum mechanical formalism using the Dirac *bracket* notation; There is an abstract state vector $|\psi\rangle$, known as the "ket" vector, and its hermitian conjugate the "bra" vector $\langle\psi|$. The inner product is defined as a *bracket* $\langle\psi\,|\psi\rangle$. Observables are associated with operators $\hat{A}$. In a given basis the operators are represented by matrices. The mathematical language is linear algebra. We define the commutator between to operators as

$$\left[\hat{A}, \hat{B}\right] = \hat{A}\hat{B} - \hat{B}\hat{A},$$

which is in general different from zero.

We list the postulates for the mathematical formulation of quantum mechanics [3]:

1. Each physical observable $A$ is represented by a hermitian operator $\hat{A}$.

The measurable values of an observable are the eigenvalues $a_n$ of $\hat{A}$

$$\hat{A}\left|n\right\rangle = a_n\left|n\right\rangle,$$

where $\left|n\right\rangle$ are the corresponding eigenvectors. The energy of the system is associated with the eigenvalues of the Hamiltonian operator $\hat{H}$.

2. A quantum state is described by a state vector $\left|\psi\right\rangle$ in Hilbert space. The state vector holds all observable properties of the quantum state. Any state vector must be normalisable

$$\left\langle\psi\right.\left|\psi\right\rangle = 1.$$

The expectation values of an observable is given by

$$\left\langle\hat{A}\right\rangle = \left\langle\psi\right|\hat{A}\left|\psi\right\rangle.$$

The state vector can be expanded in any complete set of basis vectors $\left|a_i\right\rangle$

$$\left|\psi\right\rangle = \sum_i c_i\left|a_i\right\rangle,$$

where $c_i = \left\langle a_i\right.\left|\psi\right\rangle$ are the components of the vector in that basis. For instance we have the coordinate basis

$$\psi(x) = \left\langle x\right.\left|\psi\right\rangle.$$

3. The time evolution of the state vector is governed by the Schrödinger equation

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

where $\hat{H}$ is the Hamiltonian representing the total energy of the system. In general we have $\hat{H} = \frac{\hat{p}^2}{2m} + \hat{V}$, where $\hat{p}$ is the momentum operator and $\hat{V}$ is the potential energy.

For the rest of this thesis we use the coordinate basis. In the coordinate basis the position and momentum operators are

$$\hat{x} \to x,$$
$$\hat{p} \to -i\hbar\frac{d}{dx}.$$

The normalisation is given by

$$\int_{-\infty}^{\infty} \psi(\boldsymbol{x})^* \psi(\boldsymbol{x}) d\boldsymbol{x}, \tag{2.2}$$

where $\boldsymbol{x}$ is a vector with the same dimensionality as the system. The expectation value is also given as an integral

$$\left\langle \hat{A} \right\rangle = \int \psi(\boldsymbol{x})^* A \psi(\boldsymbol{x}) d\boldsymbol{x}. \tag{2.3}$$

The commutator between $\hat{x}$ and $\hat{p}$ is

$$[\hat{x}, \hat{p}] = i\hbar. \tag{2.4}$$

This is Heisenberg's famous uncertainty principle. In fact this principle is defined as the commutator between any pair of observables. If the commutator is non-zero then the two observables cannot be observed sharply simultaneously.

## 2.2 Scaling to dimensionless units

In quantum mechanics the equations are complicated by many constants. To get rid of them we introduce a dimensionless scaling to so-called atomic units. The scaling is given by

$$\begin{aligned} r &= r_c \overline{r} = \frac{4\pi\epsilon_0\hbar^2}{m_e e^2}\overline{r} = a_0\overline{r}, \\ E &= E_c \overline{E} = \frac{\hbar^2}{m_e a_0^2}\overline{E}, \\ \omega &= \omega_c \overline{\omega} = \frac{\hbar}{m_e a_0^2}\overline{\omega}, \\ B &= B_c \overline{B} = \frac{\hbar}{e a_0^2}\overline{B}. \end{aligned}$$

The constants used here are: Planck's constant $\hbar$, the mass of the electron $m_e$, the elementary charge $e$ (the electron has charge $C = -e$) and the permittivity of space $\epsilon_0$. The values of these constants can be found in a table of physical constants [4]. We define the scaling constant of $r$ as the Bohr radius $a_0 = \frac{4\pi\epsilon_0\hbar^2}{m_e e^2}$. For the scaled equations we must also scale the potentials. Here are some examples,

Harmonic oscillator: $$V = \frac{1}{2} m_e \omega^2 r^2 \rightarrow \overline{V} = \frac{1}{2}\overline{\omega}^2\overline{r}^2, \tag{2.5}$$

Hydrogen atom: $$V = -\frac{e^2}{4\pi\epsilon r} \rightarrow \overline{V} = -\frac{1}{\overline{r}}. \tag{2.6}$$

## 2.3 The time-independent Schrödinger equation

In coordinate basis the Schrödinger equation is given as

$$\begin{aligned} i\hbar\frac{\partial\Psi}{\partial t} =& H\Psi \\ =& \frac{p^2}{2m}\Psi + V\Psi = -\frac{\hbar^2}{2m}\nabla^2\Psi + V\Psi. \end{aligned} \tag{2.7}$$

When we have a time-independent potential $V(\boldsymbol{x})$ we insert the ansatz $\Psi(\boldsymbol{x}, t) = \psi(\boldsymbol{x})U(t)$ in the Schrödinger equation to get a time-dependent equation

$$\frac{\partial U}{\partial t} = -i\frac{E}{\hbar}U, \tag{2.8}$$

and a time-independent equation

$$-\frac{\hbar^2}{2m}\nabla^2\psi + V\psi = E\psi, \tag{2.9}$$

where $E$ is the separation constant. The last equation is known as the time-independent Schrödinger equation and the constant $E$ is identified as the total energy of the system.

To solve the time-independent Schrödinger equation we must specify the potential $V(\boldsymbol{x})$. The time-dependent equation has the solution

$$U(t) = e^{-iEt/\hbar}, \tag{2.10}$$

where we define $U(t)$ as the time evolution operator. The full wave function is

$$\Psi(\boldsymbol{x}, t) = \psi(\boldsymbol{x})e^{-iEt/\hbar}. \tag{2.11}$$

Because $U(t)^*U(t) = 1$, the time-independent wave functions are stationary states and the probability is given by $\int \Psi^*\Psi = \int \psi^*\psi$. in Chapter 4 we discuss methods for solving the time-dependent Schrödinger equation.

## 2.4 The Schrödinger equation in spherical coordinates

For a spherically symmetric potential $V(r)$ we can separate the time-independent Schrödinger equation (2.9) further into a radial equation and an angular equation and solve them separately. Because the angular equation is independent

of $r$ we can solve it in general for any potential $V(r)$. We define spherical coordinates

$$\begin{aligned} r \geq 0 \quad & \text{distance from origin,} \\ 0 \leq \phi \leq 2\pi \quad & \text{angle from x-axis,} \\ 0 \leq \theta \leq \pi \quad & \text{angle from z-axis (3D).} \end{aligned}$$

### 2.4.1 Two dimensions

In two dimensions $\nabla^2$ is given by

$$\nabla^2 = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2}{\partial \phi^2}. \tag{2.12}$$

We insert this in Equation (2.9) and use the ansatz of separation of variables with

$$\psi(r,\phi) = R(r)Y(\phi), \tag{2.13}$$

we also multiply by $\frac{2m}{\hbar^2}r^2$ to obtain

$$Y\left[r\frac{\partial}{\partial r}\left(r\frac{\partial R}{\partial r}\right) - \frac{2m}{\hbar^2}r^2(E - V(r))\right]R + R\frac{\partial^2}{\partial \phi^2}Y = 0.$$

For this equation to hold, each part must be equal to a separation constant. We choose this constant to be $m^2$ and get an angular equation

$$\frac{d^2Y}{d\phi^2} = -m^2 Y, \tag{2.14}$$

and a radial equation

$$-\frac{1}{r}\frac{d}{dr}\left(r\frac{dR}{dr}\right) + \frac{m^2}{r^2}R + \frac{2m}{\hbar^2}\left(V(r) - E\right)R = 0. \tag{2.15}$$

The angular equation (2.14) has the normalised solution

$$Y(\phi) = \frac{1}{\sqrt{2\pi}}e^{im\phi}, \tag{2.16}$$

where the quantum number $m$ can take the values

$$m = 0, \pm 1, \pm 2, \ldots.$$

For the solution $R(r)$ to be normalisable we must require the boundary conditions

$$R(0) = C \text{ and } R(\infty) = 0, \quad \text{where } C \text{ is a constant.} \tag{2.17}$$

In the radial equation (2.15) we introduce the function

$$u(r) = \sqrt{r}R(r) \rightarrow R(r) = \frac{u(r)}{\sqrt{r}}, \tag{2.18}$$

to obtain

$$-\frac{d^2u}{dr^2} + \left[\frac{m^2 - \frac{1}{4}}{r^2} + \frac{2m_e}{\hbar^2}V\right] u = \frac{2m_e}{\hbar^2}Eu.$$

Finally, introducing dimensionless variables (denoted by $\overline{r}$) as defined in Section 2.2, we can rewrite this to

$$-\frac{d^2u}{d\overline{r}^2} + \left[\frac{m^2 - \frac{1}{4}}{\overline{r}^2} + 2\overline{V}\right] u = 2\overline{E}u. \tag{2.19}$$

The boundary conditions (2.17) simplify to

$$u(0) = 0, \quad u(\infty) = 0. \tag{2.20}$$

The total wave function is normalised by

$$\int_0^{2\pi}\int_0^{\infty} \psi^*(r,\theta)\psi(r,\theta)rdrd\theta = 1.$$

Inserting $\psi(r,\theta) = \frac{1}{\sqrt{2\pi}}e^{im\phi}\frac{u(r)}{\sqrt{r}}$ into this expression we have

$$\int_0^{\infty} u^*(r)u(r)dr = 1.$$

### 2.4.2 Three dimensions

Similarly, in three dimensions we have $\nabla^2$ given by

$$\nabla^2 = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\left(\frac{\partial^2}{\partial\phi^2}\right), \tag{2.21}$$

We insert this is Equation (2.9) and use the ansatz

$$\psi(r,\theta,\phi) = R(r)Y(\phi,\theta),$$

to obtain

$$Y\left[\frac{d}{dr}\left(r^2\frac{dR}{dr}\right)-\frac{2mr^2}{\hbar^2}\left(V(r)-E\right)\right]R + R\frac{1}{\sin^2\theta}\left[\sin\theta\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial Y}{\partial\theta}\right)+\frac{\partial^2 Y}{\partial\phi^2}\right]Y=0,$$

We choose the separation constant $l(l+1)$ and get an equation which depends only on $\phi$ and $\theta$

$$\sin\theta\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial Y}{\partial\theta}\right)+\frac{\partial^2 Y}{\partial\phi^2}=-l(l+1)\sin^2\theta Y, \tag{2.22}$$

and a radial equation

$$\frac{d}{dr}\left(r^2\frac{dR}{dr}\right)-\frac{2mr^2}{\hbar^2}\left(V(r)-E\right)R=l(l+1)R. \tag{2.23}$$

The solution of the angular equation (2.22) is more complicated for the three-dimensional case and we refer to texts in quantum mechanics for the derivation, see for example [1]. The normalised angular wave functions are the spherical harmonics

$$Y_l^m(\theta,\phi)=\epsilon\sqrt{\frac{2l+1}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}}e^{im\phi}P_l^m(\cos\theta),, \tag{2.24}$$

$$\epsilon=(-1)^m \text{ for } m\geq 0,\quad \epsilon=1 \text{ for } m\leq 0, \tag{2.25}$$

where $P_l^m$ are the associated Legendre polynomials [5] defined by

$$P_l^m(x)=(1-x^2)^{\frac{1}{2}|m|}\left(\frac{d}{dx}\right)^{|m|}P_l(x),$$

$$P_l(x)=\frac{1}{2^l l!}\left(\frac{d}{dx}\right)^l(x^2-1)^l.$$

The quantum numbers $l$ and $m$ are restricted by

$$l=0,1,2\ldots$$
$$m=-l,-l+1,\ldots,-1,0,1,\ldots,l-1,l.$$

In the three-dimensional case we introduce the function

$$u(r)=rR(r)\rightarrow R(r)=\frac{u(r)}{r}, \tag{2.26}$$

to obtain

$$-\frac{d^2u}{dr^2} + \left[\frac{l(l+1)}{r^2} + \frac{2m_e}{\hbar^2}V\right]u = \frac{2m_e}{\hbar^2}Eu.$$

We introduce the dimensionless scaling of Section 2.2 to get

$$-\frac{d^2u}{d\overline{r}^2} + \left[\frac{l(l+1)}{\overline{r}^2} + 2\overline{V}\right]u = 2\overline{E}u. \tag{2.27}$$

We have the same boundary conditions and normalisation as for the two-dimensional case

$$u(0) = 0, \quad u(\infty) = 0, \tag{2.28}$$

$$\int_0^\infty u^*(r)u(r)dr = 1. \tag{2.29}$$

## 2.5 Angular momentum and Spin

### 2.5.1 Angular momentum

Classically, we define angular momentum as $\boldsymbol{L} = \boldsymbol{r} \times \boldsymbol{p}$ in three dimensions. If we use $\boldsymbol{p} = -i\hbar\nabla$, we have the quantum mechanical expression for $\boldsymbol{L}$

$$L_x = yp_z - zp_y, \quad L_y = zp_x - xp_z, \quad L_z = xp_y - yp_x.$$

We choose to study $L^2$ and $L_z$ (we choose one of the components of $\boldsymbol{L}$) and search for the eigenstates. In spherical coordinates we have

$$L^2 = -\hbar^2\left[\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{\sin^2\theta}\left(\frac{\partial^2}{\partial\phi^2}\right)\right], \tag{2.30}$$

$$L_z = -i\hbar\frac{\partial}{\partial\theta}. \tag{2.31}$$

The spherical harmonics $Y_{lm}$ we defined in the previous section are also eigenfunctions of $L^2$ and $L_z$. The eigenvalues are

$$L^2Y_{lm} = \hbar^2 l(l+1)Y_{lm}, \quad L_zY_{lm} = \hbar mY_{lm}.$$

Because they have the same eigenfunctions $L^2$ and $L_z$, commute with the Hamiltonian $H$

$$[H, L^2] = [H, L_z] = 0.$$

### 2.5.2 Spin

All particles have a spin property, which is derived in relativistic quantum mechanics. We define it in a similar way as the angular momentum

$$S^2\chi_{m_s} = \hbar^2 s(s+1)\chi_{m_s}, \quad S_z\chi_{m_s} = \hbar m_s \chi_{m_s}. \tag{2.32}$$

The eigenfunctions $\chi_{m_s}$ are called the eigenspinors, we define these for electrons soon. The quantum number $s$ can take integer (bosons) and half integer (fermions) values

$$s = 0, \frac{1}{2}, 1, \frac{3}{2} \ldots, \quad m_s = -s, -s+1, \ldots, s-1, s. \tag{2.33}$$

In particular electrons have the property

$$s = \frac{1}{2}, \quad m_s = \pm\frac{1}{2}. \tag{2.34}$$

Because electrons are the focus of this thesis we discuss this case further. There are two eigenstates

$$\text{spin up: } m_s = +\frac{1}{2}, \quad |\uparrow\rangle$$
$$\text{spin down: } m_s = -\frac{1}{2}, \quad |\downarrow\rangle$$

We define the spin state as the spinor

$$\chi = \begin{pmatrix} a \\ b \end{pmatrix} = a\chi_+ + b\chi_-, \tag{2.35}$$

where

$$\chi_+ = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \chi_- = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

represent spin up and spin down states respectively. We define the spin matrix by the Pauli matrix

$$\boldsymbol{S} = \frac{\hbar}{2}\boldsymbol{\sigma}, \tag{2.36}$$

where $\boldsymbol{\sigma}$ has the components

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{2.37}$$

These matrices have the property

$$\sigma_z\chi_+ = +\chi_+ \rightarrow S_z\chi_+ = +\frac{\hbar}{2}, \tag{2.38}$$

$$\sigma_z\chi_- = -\chi_- \rightarrow S_z\chi_- = -\frac{\hbar}{2}. \tag{2.39}$$

### 2.5.3 Two-particle systems

When we have two spin $\frac{1}{2}$ particles we denote the spin states by

$$\uparrow\uparrow, \quad \uparrow\downarrow, \quad \downarrow\uparrow, \quad \downarrow\downarrow,$$

where the first arrow represents the spin of the first particle and the second arrow represents the other particle. For the two-particle states we use capital letters for the quantum numbers $S$, $M$. We wish to group the spin states into symmetric and antisymmetric spin states. We have three symmetric states with $S = 1$ (triplet)

$$\begin{aligned} &\uparrow\uparrow, && M = 1, \\ &\frac{1}{\sqrt{2}}\left(\uparrow\downarrow + \downarrow\uparrow\right), && M = 0, \\ &\downarrow\downarrow, && M = -1, \end{aligned}$$

and one antisymmetric state for $S = 0$ (singlet)

$$\frac{1}{\sqrt{2}}\left(\uparrow\downarrow - \downarrow\uparrow\right), \quad M = 0.$$

Here we have $M = m_1 + m_2$ with values $M = -S, -S+1, \ldots, S-1, S$ as we required for the spin quantum numbers.

Because we are dealing with a system of fermions we require the total wave function to be anti-symmetric under the interchange of two particles

$$\psi(\boldsymbol{r}_1, \boldsymbol{r}_2) = -\psi(\boldsymbol{r}_2, \boldsymbol{r}_1).$$

If we construct a wave function of the form

$$\psi(\boldsymbol{r}_1, \boldsymbol{r}_2) = A\left[\psi_a(\boldsymbol{r}_1)\psi_b(\boldsymbol{r}_2) - \psi_a(\boldsymbol{r}_2)\psi_b(\boldsymbol{r}_1)\right],$$

the Pauli principle follows: *Two identical fermions cannot occupy the same state*, because if $\psi_a = \psi_b$, then the wave function is zero.

For the full two-particle wave function we must require anti-symmetry for fermions. To achieve this we combine a symmetric spatial wave function to an anti-symmetric spin function ($S = 0$) and an anti-symmetric spatial wave function to a symmetric spin function ($S = 1$). The two-particle spin functions $S = 0$ and $S = 1$ were defined in the previous section.

## 2.6 Interaction with the electromagnetic field

In the quantum dot system, which we study in this thesis, we have an electromagnetic field. In this section we derive the equations for interaction with an electromagnetic field in quantum mechanics. This derivation is taken from [3]. We describe the magnetic field $\boldsymbol{B}$, and the electric field $\boldsymbol{E}$ in terms of potentials

$$\boldsymbol{E} = -\nabla\Phi, \tag{2.40}$$
$$\boldsymbol{B} = \nabla \times \boldsymbol{A}, \tag{2.41}$$
$$\nabla \cdot \boldsymbol{B} = 0, \tag{2.42}$$

where $\boldsymbol{A}$ is a vector potential and $\Phi$ is a scalar potential.

In classical theory a charged particle is subject to the Lorentz force $\boldsymbol{F} = e(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B})$. In quantum mechanics we should reproduce this force in the classical limit. To do this we need to incorporate the electromagnetic field into the Schrödinger equation somehow. The electric field gives the contribution $V = e\Phi$, but what about the magnetic field?

For a given magnetic field $\boldsymbol{B}$ we can have many vector potentials $\boldsymbol{A}$ which satisfy Equation (2.41). To show this we add a gradient to the vector potential

$$\boldsymbol{A} \to \boldsymbol{A}' = \boldsymbol{A} + \nabla\chi, \tag{2.43}$$

where $\chi = \chi(\boldsymbol{r})$ is a scalar function. This is called a gauge transformation. The magnetic field stays the same

$$\boldsymbol{B} \to \boldsymbol{B}' = \boldsymbol{B} + \nabla \times \nabla\chi = \boldsymbol{B}, \tag{2.44}$$

because $\nabla \times \nabla = 0$. When we choose a specific $\boldsymbol{A}$ we choose a gauge. We are allowed to do this because our equations are gauge invariant. A common choice, which we will use in this thesis is the Coulomb gauge

$$\nabla \cdot \boldsymbol{A} = 0. \tag{2.45}$$

We further choose to write the vector potential as

$$\boldsymbol{A} = \frac{1}{2}\boldsymbol{B} \times \boldsymbol{r}, \tag{2.46}$$

which automatically fulfills the coulomb gauge (2.45). In fact so will any vector potential $\boldsymbol{A}'$ which transforms as (2.43). Further show we show that by introducing a new Hamiltonian and a phase transformation on the wave function we can always get back expression (2.46).

We introduce a modified Hamiltonian

$$H = \frac{1}{2m}(\boldsymbol{p} - e\boldsymbol{A})^2, \tag{2.47}$$

where we have used the replacement $\boldsymbol{p} \to \boldsymbol{p} - e\boldsymbol{A}$. When we write out the expression

$$(\boldsymbol{p} - e\boldsymbol{A})^2 = \boldsymbol{p}^2 - e\left(\boldsymbol{p}\cdot\boldsymbol{A} + \boldsymbol{A}\cdot\boldsymbol{p}\right) + e^2\boldsymbol{A}^2, \tag{2.48}$$

and study $\boldsymbol{p}\cdot\boldsymbol{A}$ acting on a wave function $\psi$ , we see that when $\nabla\cdot\boldsymbol{A} = 0$ (Coulomb gauge), $\boldsymbol{p}$ and $\boldsymbol{A}$ commute

$$\boldsymbol{p}\cdot\boldsymbol{A}\psi = -i\hbar\nabla\cdot(\boldsymbol{A}\psi) = -i\hbar(\nabla\cdot\boldsymbol{A} + \boldsymbol{A}\cdot\nabla\psi) = \boldsymbol{A}\cdot(-i\hbar\nabla\psi). \tag{2.49}$$

Inserting these expressions back in the Hamiltonian we have

$$H = \frac{1}{2m}\left(\boldsymbol{p}^2 - 2e\boldsymbol{p}\cdot\boldsymbol{A} + e^2\boldsymbol{A}^2\right). \tag{2.50}$$

If we use the choice for $\boldsymbol{A}$ given in expression (2.46) we can write the Hamiltonian as

$$H = \frac{1}{2m}\left(\boldsymbol{p}^2 - e\boldsymbol{B}\cdot\boldsymbol{L} + e^2\boldsymbol{A}^2\right), \tag{2.51}$$

where we have used $(\boldsymbol{B}\times\boldsymbol{r})\cdot\boldsymbol{p} = \boldsymbol{B}\cdot(\boldsymbol{r}\times\boldsymbol{p})$ and recognised the angular momentum as $\boldsymbol{r}\times\boldsymbol{p} = \boldsymbol{L}$ .

We also introduce a phase transformation to the wave function

$$\psi(\boldsymbol{r}) \to \psi'(\boldsymbol{r}) = e^{i\theta(\boldsymbol{r})}\psi(\boldsymbol{r}). \tag{2.52}$$

Acting on this wave function by $\boldsymbol{p} - e\boldsymbol{A}$ we have the expressions

$$\begin{aligned}(\boldsymbol{p} - e\boldsymbol{A})\psi'(\boldsymbol{r}) &= e^{i\theta(\boldsymbol{r})}(\boldsymbol{p} + \hbar\nabla\theta(\boldsymbol{r}) - e\boldsymbol{A})\psi(\boldsymbol{r}),\\ (\boldsymbol{p} - e\boldsymbol{A})^2\psi'(\boldsymbol{r}) &= e^{i\theta(\boldsymbol{r})}(\boldsymbol{p} + \hbar\nabla\theta(\boldsymbol{r}) - e\boldsymbol{A})^2\psi(\boldsymbol{r}).\end{aligned}$$

We now make a gauge transformation on $\boldsymbol{A}$ and choose the phase $\theta(\boldsymbol{r}) = (e/\hbar)\chi(\boldsymbol{r})$

$$\begin{aligned}(\boldsymbol{p} - e\boldsymbol{A}')^2\psi'(\boldsymbol{r}) &= e^{i\theta(\boldsymbol{r})}(\boldsymbol{p} + \hbar\nabla\theta(\boldsymbol{r}) - e\boldsymbol{A} - e\nabla\chi)^2\psi(\boldsymbol{r})\\ &= e^{ie\chi(\boldsymbol{r})/\hbar}(\boldsymbol{p} - e\boldsymbol{A})^2\psi(\boldsymbol{r}).\end{aligned}$$

We have now made the Schrödinger equation gauge invariant because we can compensate for a change in the vector potential $\boldsymbol{A} \to \boldsymbol{A}'$ by a phase

transformation in the wave function $\psi \to \psi'$. We have also introduced the coupling to the magnetic field by $\boldsymbol{p} \to \boldsymbol{p} - e\boldsymbol{A}$.

To get the spin coupling to the magnetic field we must the Pauli Hamiltonian

$$H = \frac{1}{2m} \left( \boldsymbol{\sigma} \cdot \boldsymbol{\Pi} \right)^2 , \tag{2.53}$$

where $\boldsymbol{p} - e\boldsymbol{A} = \boldsymbol{\Pi}$ and $\boldsymbol{\sigma}$ are the Pauli spin matrices defined by Equation (2.37) in Section 2.5.2. It can be shown that the Pauli matrices fulfill the relation

$$(\boldsymbol{\sigma} \cdot \boldsymbol{a})(\boldsymbol{\sigma} \cdot \boldsymbol{b}) = \boldsymbol{a} \cdot \boldsymbol{b} + i\boldsymbol{\sigma} \cdot (\boldsymbol{a} \times \boldsymbol{b}).$$

Inserting this we have

$$(\boldsymbol{\sigma} \cdot \boldsymbol{\Pi})^2 = i\boldsymbol{\sigma} \cdot (\boldsymbol{\Pi} \times \boldsymbol{\Pi}) + \boldsymbol{\Pi}^2. \tag{2.54}$$

Here $\boldsymbol{\Pi}^2$ denotes the standard Hamiltonian. We calculate the cross product using the Einstein's summation convention

$$\begin{aligned} (\boldsymbol{\Pi} \times \boldsymbol{\Pi})_i &= \epsilon_{ijk} \Pi_j \Pi_k = [\Pi_j, \Pi_k] \\ &= -\hbar^2 [\partial_j, \partial_k] + ie\hbar [\partial_j, A_k] - ie\hbar [\partial_k, A_j] + e^2 [A_j, A_k], \end{aligned} \tag{2.55}$$

where $\partial_i \equiv \frac{\partial}{\partial x_i}$. We calculate the commutators:

$$\begin{aligned} [\partial_j, \partial_k] &= 0, \\ [A_j, A_k] &= 0, \\ [\partial_j, A_k] &= (\partial_j A_k), \\ [\partial_k, A_j] &= (\partial_k A_j). \end{aligned}$$

We insert these relations in Equation (2.55) and obtain

$$(\boldsymbol{\Pi} \times \boldsymbol{\Pi})_i = ie\hbar \left( \partial_j A_k - \partial_k A_j \right).$$

Comparing this with $B_i = (\nabla \times \boldsymbol{A})_i = \epsilon_{ijk} \partial_j A_k = \partial_j A_k - \partial_k A_j$ we see that they are the same. This gives

$$(\boldsymbol{\sigma} \cdot \boldsymbol{\Pi})^2 = \boldsymbol{\Pi}^2 - e\hbar \boldsymbol{\sigma} \cdot \boldsymbol{B}. \tag{2.56}$$

Finally, we have the full Hamiltonian for a particle in a magnetic field

$$H = \frac{1}{2m} (\boldsymbol{p} - e\boldsymbol{A})^2 - \frac{e\hbar}{2m} \boldsymbol{\sigma} \cdot \boldsymbol{B} + e\Phi. \tag{2.57}$$

Here we have introduced the coupling of angular momentum to the magnetic field by $\boldsymbol{B} \cdot \boldsymbol{L}$ and the coupling of spin to the magnetic field by $\boldsymbol{B} \cdot \boldsymbol{\sigma}$. This is known as the Zeeman effect and gives a splitting of the energy levels. We also have the purely quantum term $\boldsymbol{A}^2$.

# Chapter 3

# A mathematical model for quantum dots

In this chapter we study the quantum quantum system of a quantum dot. We begin by giving a short introduction to the field of quantum dots. Then we derive the mathematical model used to describe them. We focus on the two dimensional case, but will briefly mention the three dimensional equations as well.

We study the time-independent Schrödinger equation, where we search for the eigenstates of the system. These eigenstates can then be used as a basis for computing the time evolution of the system. First, we derive the single-electron quantum dot, which can be solved analytically. Then we show that the equation for the two-electron quantum dot can be rewritten into two independent single particle equations. This case only has particular analytic solutions, so this is an interesting problem to solve numerically.

## 3.1 Background on quantum dots

The name "quantum dot" refers to a quantum system where electrons are confined in space. Modern techniques allow confinement of only a few electrons. This is achieved in a semiconductor structure. We will only give a short introduction here, for a review article, see Reference [6].

Quantum dots are often called “artificial atoms” because they have many similarities with atoms, this makes them very interesting to study. For these "artificial atoms" we can find degenerate energy levels giving stable states and a shell structure. Experiments have shown that for a quantum dot, electrons can be added in a controlled way by tunneling [7]. The development in

semiconductor technology toward smaller systems also make it important to study such quantum systems.

The properties of a quantum dot can be controlled by changing the geometry, applying electrostatic gates or by applying a magnetic field. There are several techniques for manufacturing quantum dots, but they will not be explained here. In Reference [6] manufacturing techniques are explained. The electrons are confined in a bowl shaped potential, we approximate this potential by a harmonic oscillator.

A very interesting application is the use of quantum dots in quantum computing. In Reference [8] two coupled quantum dots used as a two qubit quantum gate. The quantum bit (qubit) is a two-level system realised by the electron spin. By taking the advantage of the superposition principle in quantum mechanics specialised algorithms can be created for the quantum computer. For the coupled quantum dot the quantum gate operates by tunneling between the two dots. Single qubit operations are performed by applying local magnetic field. The model for this quantum gate is similar to the model which we will derive except that it has a double harmonic oscillator well.

Another application is the use of the optical properties of the quantum dots. In the same way as atoms they can absorb and emit photons. Because we can vary the properties of the quantum dot and therefore the energy levels the wavelength of the emitted light can also be varied.This property may be used in LED lights or lasers, but a much more interesting application is the use of quantum dots in medical imaging. There have been experiments on using quantum dots for cancer targeting and imaging. In Reference [9] such experiments are described. They show promising features.

## 3.2 The mathematical model

We describe the quantum dot as a quantum mechanical system of electrons in a harmonic oscillator well and in an external magnetic field. This quantum system is governed by the Schrödinger equation. In this chapter we focus on the time-independent case

$$H\Psi(\boldsymbol{r}) = E\Psi(\boldsymbol{r}). \tag{3.1}$$

For a system of $N$ electrons we have the Hamiltonian

$$H = \sum_{i=1}^{N} h_i + \sum_{i=1}^{N} \sum_{j \neq i} \frac{e^2}{4\pi\epsilon_0} \frac{1}{|\boldsymbol{r}_i - \boldsymbol{r}_j|}, \tag{3.2}$$

where $h_i$ are the single particle Hamiltonians. The last term is the electrostatic repulsion between two electrons.

In Section 2.6 we derived the single particle Hamiltonian for a particle in an electromagnetic field

$$h_i = \frac{1}{2m}(\boldsymbol{p}_i - e\boldsymbol{A}_i)^2 + e\Phi - \frac{e\hbar}{2m_e}\boldsymbol{\sigma}\cdot\boldsymbol{B} + v(\boldsymbol{r}_i). \tag{3.3}$$

In this thesis we use a constant magnetic field along the z-axis and zero electric field

$$\boldsymbol{B} = (0, 0, B_0), \tag{3.4}$$
$$\boldsymbol{E} = 0. \tag{3.5}$$

When we have no electric field the electric potential is a constant

$$\boldsymbol{E} = -\nabla\Phi \rightarrow \Phi = const.$$

In Section 2.6, we set the electromagnetic vector potential in Equation (2.46):

$$\boldsymbol{A} = \frac{1}{2}\boldsymbol{B}\times\boldsymbol{r} = \frac{B_0}{2}(-y, x, 0). \tag{3.6}$$

The harmonic oscillator potential is given by

$$V(r) = \frac{1}{2}m_e\omega_0^2 r^2, \tag{3.7}$$

where $\omega_0$ is the oscillator frequency describing the shape of the potential. With this potential the single particle Hamiltonian for the quantum dot is

$$h_i = \frac{1}{2m}(\boldsymbol{p}_i - e\boldsymbol{A}_i)^2 + e\Phi - \frac{e\hbar}{2m_e}\boldsymbol{\sigma}\cdot\boldsymbol{B} + \frac{1}{2}m_e\omega_0^2 r_i^2. \tag{3.8}$$

We recall that $\boldsymbol{\sigma}$ is the Pauli spin matrix defined in Section 2.5.2 by Equation (2.37).

If we disregard the electron-electron repulsion in Equation (3.2), we can solve $N$ independent single-particle eigenvalue problems given by

$$h_i\psi_i = E_i\psi_i,$$

where the index $i$ refer to electron $i$. Each particle has its own set of eigenstates $E_\lambda, \psi_\lambda$. The total energy is given as the sum of the single particle energies

$$E = \sum_{i=1}^{N} E_i,$$

and the wave function is the product of the single particle wave functions.

$$\Psi = \prod_{i=1}^{N} \psi_i.$$

In the next section we study the single particle equation with the Hamiltonian (3.8). Then we move on to study the two-particle case. In this case we also have the electrostatic interaction between the two electrons and we will see that this term is the source of complexity.

## 3.3 The single electron quantum dot

For $N = 1$ we have the Hamiltonian

$$H = h = \frac{1}{2m}(\boldsymbol{p} - e\boldsymbol{A})^2 + e\Phi - \frac{e\hbar}{2m_e}\boldsymbol{\sigma} \cdot \boldsymbol{B} + \frac{1}{2}m_e\omega_0^2 r^2. \tag{3.9}$$

To separate out the spin part we use the ansatz

$$\Psi(\boldsymbol{r}) = \psi(\boldsymbol{r})\chi, \tag{3.10}$$

where $\chi$ is the spin function defined by Equation (2.35) in Chapter 2. Inserting this into the time-independent Schrödinger equation we can separate it into a spatial part depending on $\boldsymbol{r}$ and a part which is independent of $\boldsymbol{r}$

$$\begin{aligned} H\Psi =& \chi\left[\frac{1}{2m}(\boldsymbol{p} - e\boldsymbol{A})^2 + \frac{1}{2}m_e\omega_0^2 r^2\right]\psi(\boldsymbol{r}) \\ &+ \psi(\boldsymbol{r}) - \frac{e\hbar}{2m_e}\boldsymbol{\sigma} \cdot \boldsymbol{B}\chi + e\Phi\psi(\boldsymbol{r})\chi \\ &= E\psi(\boldsymbol{r})\chi. \end{aligned} \tag{3.11}$$

To solve this equation for any function $\psi, \chi$ each term must be a constant and the sum of the constants must be equal to the total energy

$$E = E_\Omega + E_s + e\Phi, \tag{3.12}$$

where $\Omega$ denotes the spatial part and $s$ denotes the spin part. The two equations which must be solved are

$$\left[\frac{1}{2m}(\boldsymbol{p} - e\boldsymbol{A})^2 + \frac{1}{2}m_e\omega_0^2 r^2\right]\psi(\boldsymbol{r}) = E_\Omega\psi(\boldsymbol{r}) \text{ and} \tag{3.13}$$

$$-\frac{e\hbar}{2m_e}\boldsymbol{\sigma} \cdot \boldsymbol{B}\chi = E_s\chi. \tag{3.14}$$

We begin by solving the spin equation (3.14). With a constant magnetic field in the z-direction (3.4) we have

$$-B_0\frac{e\hbar}{2m_e}\sigma_z\chi = E_s\chi. \tag{3.15}$$

From Section 2.5.2 we know that the eigenvalues and eigenvectors of $\sigma_z$ are

$$+1, \quad \chi_+ = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \tag{3.16}$$

$$-1, \quad \chi_- = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{3.17}$$

This gives

$$E_s = -B_0\frac{e\hbar}{m_e}m_s, \quad m_s = \pm\frac{1}{2}, \tag{3.18}$$

using atomic units as defined in Section 2.2 we have

$$\overline{E}_s = \overline{B}_0 m_s. \tag{3.19}$$

We focus on the spatial equation (3.13) from now

$$H\psi(\boldsymbol{r}) = \left[\frac{1}{2m_e}\left(\boldsymbol{p} - e\boldsymbol{A}\right)^2 + \frac{1}{2}m_e\omega_0^2 r^2\right]\psi(\boldsymbol{r}) = E\psi(\boldsymbol{r}). \tag{3.20}$$

From Section 2.6 we know that

$$(\boldsymbol{p} - e\boldsymbol{A})^2 = \boldsymbol{p}^2 - e\boldsymbol{B}\cdot\boldsymbol{L} + e^2\boldsymbol{A}^2. \tag{3.21}$$

Using the magnetic field (3.4) we have

$$\boldsymbol{B}\cdot\boldsymbol{L} = B_0 L_z = -i\hbar B_0\frac{\partial}{\partial\theta}. \tag{3.22}$$

Inserting this in Equation (3.20) we obtain the Hamiltonian

$$H = \frac{1}{2m_e}\left(\boldsymbol{p}^2 + ei\hbar B_0\frac{\partial}{\partial\theta} + e^2\boldsymbol{A}^2\right) + \frac{1}{2}m_e\omega_0^2 r^2, \tag{3.23}$$

which we study for the two-dimensional and three-dimensional case.

### 3.3.1 Two dimensions

In two dimensions we have the Hamiltonian

$$H = -\frac{\hbar^2}{2m_e}\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2}{\partial\theta^2} - i\frac{B_0 e}{\hbar}\frac{\partial}{\partial\theta}\right]\psi(\boldsymbol{r}) + \frac{1}{2}m_e\left[\frac{B_0^2 e^2}{4m_e^2} + \omega_0^2\right]r^2\psi(\boldsymbol{r}) = E\psi(\boldsymbol{r}). \tag{3.24}$$

We define

$$\omega^2 = \omega_0^2 + \omega_B^2, \quad \omega_B = \frac{B_0 e}{2m_e}, \tag{3.25}$$

and make a substitution using the angular solution (2.16) from Chapter 2

$$\psi(\boldsymbol{r}) = R(r)\frac{1}{\sqrt{2\pi}}e^{im\theta},$$

to get a one-dimensional radial equation

$$-\frac{\hbar^2}{2m_e}\left[\frac{1}{r}\frac{d}{dr}\left(r\frac{d}{dr}\right) - \frac{m^2}{r^2} + \frac{B_0 e m}{\hbar}\right]R(r) + \frac{1}{2}m_e\omega^2 r^2 R(r) = ER(r).$$

We introduce the dimensionless variables of Section 2.2 and define

$$\overline{\epsilon} = 2\overline{E} + \overline{B}_0 m \tag{3.26}$$

to obtain

$$-\left[\frac{1}{\overline{r}}\frac{d}{d\overline{r}}\left(\overline{r}\frac{d}{d\overline{r}}\right) - \frac{m^2}{\overline{r}^2}\right]R(\overline{r}) + \overline{\omega}^2\overline{r}^2 R(\overline{r}) = \overline{\epsilon}R(\overline{r}), \tag{3.27}$$

$$\overline{\omega}^2 = \overline{\omega}_0^2 + \frac{\overline{B}_0^2}{4}. \tag{3.28}$$

To get rid of the first derivative we make another substitution

$$R(r) = \frac{u(r)}{\sqrt{r}},$$

to get

$$\left[-\frac{d^2}{d\overline{r}^2} + \frac{m^2 - \frac{1}{4}}{\overline{r}^2} + \overline{\omega}^2\overline{r}^2\right]u(\overline{r}) = \overline{\epsilon}u(\overline{r}). \tag{3.29}$$

From Chapter 2 we have the boundary conditions

$$u(0) = 0, \quad u(\infty) = 0,$$

and normalisation of the wave function given by

$$\int_0^\infty u(r)^* u(r) dr = 0.$$

This is the single particle equation for the quantum dot. We observe that this is just a harmonic oscillator potential where $\omega$ and $E$ are shifted because of the magnetic field. This equation has closed form solutions, which we can use to test our algorithm.

We show the derivation of the analytic solutions for the two-dimensional quantum dot in Appendix A.1. The solution is

$$\epsilon = 2\omega(|m| + 1 + 2n).$$

From (3.63) we find the full energy

$$E = (|m| + 1 + 2n)\omega - \frac{1}{2}Bm. \tag{3.30}$$

By studying the analytic solutions we see that the addition of a magnetic field $\boldsymbol{B}$ give a splitting of the energy levels for $\pm m$ in addition to an increased value of $\omega$. The radial wave function is

$$R(r) = \sqrt{\frac{2n!}{(|m| + n)!}} \omega^{(|m|+1)/2} r^{|m|} e^{-\frac{1}{2}\omega r^2} L_n^{|m|}(\omega r^2), \tag{3.31}$$

where $L_n^{|m|}$ are the associated Laguerre polynomials. The first three Laguerre polynomials are given n Table A.1

### 3.3.2 Three dimensions

For the sake of completeness we also quickly set up the equations for the single electron quantum dot in three dimensions. In three dimensions we

have

$$\begin{aligned} H\psi(\boldsymbol{r}) =& \frac{1}{2m_e}\left(\boldsymbol{p} - e\boldsymbol{A}\right)^2 + \frac{1}{2}m_e\omega_0^2 r^2 \\ =& -\frac{\hbar^2}{2m_e}\left[\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right)\right. \\ &\left. + \frac{1}{r^2\sin^2\theta}\left(\frac{\partial^2}{\partial\phi^2}\right) - i\frac{B_0 e}{\hbar}\frac{\partial}{\partial\phi}\right]\psi(\boldsymbol{r}) \\ &+ \frac{1}{2}m_e\omega_0^2 r^2\psi(\boldsymbol{r}) + \frac{1}{2}m_e\frac{B_0^2 e^2}{4m_e^2}\left(x^2+y^2\right)\psi(\boldsymbol{r}) \\ =& E\psi(\boldsymbol{r}). \end{aligned} \tag{3.32}$$

In chapter 2 we gave the angular solution of a spherically symmetric potential, we insert this as $\psi(\boldsymbol{r}) = R(r)Y_{lm_l}$, where $Y_{lm_l}$ is given by (2.24). We recall the definitions of $\boldsymbol{L}^2$ and $L_z$ and its eigenvalues from Section 2.5.1

$$L^2 = -\hbar^2\left[\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{\sin^2\theta}\left(\frac{\partial^2}{\partial\phi^2}\right)\right] \to \hbar^2 l(l+1), \tag{3.33}$$

$$L_z = -i\hbar\frac{\partial}{\partial\theta} \to \hbar m. \tag{3.34}$$

We identify these in the Schrödinger equation (3.32) and insert their eigenvalues. We now have a one dimensional radial equation

$$\begin{aligned} HR(r) =& -\frac{\hbar^2}{2m_e}\left[\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) - \frac{l(l+1)}{r^2} + \frac{B_0 e}{\hbar}m\right]R(r) \\ &+ \frac{1}{2}m_e\omega_0^2 r^2 R(r) + \frac{1}{2}m_e\frac{B_0^2 e^2}{4m_e^2}\left(x^2+y^2\right)R(r) \\ =& ER(r). \end{aligned}$$

In three dimensions we encounter a problem: We get a term $A^2 \to (x^2+y^2)$. Because it has no $z^2$ term we must require

$$\omega_x = \omega_0, \qquad \omega_y = \omega_0, \qquad \omega_z = \sqrt{\omega_0^2 + \frac{B_0^2}{4}},$$

in order to obtain spherical symmetry. Finally, we insert the substitution $R(r) = \frac{u(r)}{r}$ and scale by atomic units

$$\left[-\frac{d^2}{d\overline{r}^2} + \frac{l(l+1)}{\overline{r}^2} + \overline{\omega}^2\overline{r}^2\right]u(\overline{r}) = \overline{\epsilon}u(\overline{r}), \tag{3.35}$$

where $\overline{\omega}$ is defined in (3.28) and $\overline{\epsilon}$ is defined in (3.26).

In the three dimensional case the analytic eigenvalues are

$$\epsilon = 2(2n + l + \frac{3}{2})\omega, \tag{3.36}$$

$$E = (2n + l + \frac{3}{2})\omega - m\omega_B, \tag{3.37}$$

and the radial wave functions are

$$R(r) = \sqrt{\frac{2^{n+l+2}n!}{\sqrt{\pi}(2n+2l+1)!!}}\omega^{(l+3/2)/2}r^l e^{-\omega r^2/2}L_n^{l+\frac{1}{2}}(\omega r^2), \tag{3.38}$$

where $L_n^{l+\frac{1}{2}}(x)$ are the Laguerre polynomials given in Appendix A.1.

## 3.4 The two-electron quantum dot

We begin by separating the spin equation in the same way as we did one the single-particle case. This separation gives $E = E_\Omega + 2e\Phi + E_{s_1} + E_{s_2}$, where $E_{s_i}$ is given in Equation (3.18). We now focus on the spatial Hamiltonian as we did for the two-dimensional case.For the two-electron case we also have an electron-electron interaction term. The Hamiltonian is

$$\begin{aligned} H =& \sum_{i=1}^{2}\left[\frac{1}{2m_e}\left(\boldsymbol{p}_i - e\boldsymbol{A}_i\right)^2 + \frac{1}{2}m_e\omega_0^2 r_i^2\right] \\ &+ \frac{e^2}{4\pi\epsilon_0}\frac{1}{|\boldsymbol{r}_1 - \boldsymbol{r}_2|}. \end{aligned} \tag{3.39}$$

To solve this two-particle problem it is convenient to introduce the coordinates of the centre-of-mass

$$\boldsymbol{R} = \frac{1}{2}\left(\boldsymbol{r}_1 + \boldsymbol{r}_2\right), \qquad \boldsymbol{P} = \boldsymbol{p}_1 + \boldsymbol{p}_2, \tag{3.40}$$

and relative motion

$$\boldsymbol{r} = \boldsymbol{r}_1 - \boldsymbol{r}_2, \qquad \boldsymbol{p} = \frac{1}{2}\left(\boldsymbol{p}_1 - \boldsymbol{p}_2\right). \tag{3.41}$$

When the magnetic field is given as in Equation (3.6), we have

$$\boldsymbol{A}(\boldsymbol{r}) = \boldsymbol{A}(\boldsymbol{r}_1) - \boldsymbol{A}(\boldsymbol{r}_2), \qquad \boldsymbol{A}(\boldsymbol{R}) = \frac{1}{2}\left(\boldsymbol{A}(\boldsymbol{r}_1) + \boldsymbol{A}(\boldsymbol{r}_2)\right).$$

From these definition we calculate some useful relations

$$r_1^2 + r_2^2 = \frac{1}{2}\left(4R^2 + r^2\right), \quad p_1^2 + p_2^2 = \frac{1}{2}\left(4p^2 + P^2\right),$$
$$\boldsymbol{p}_1 \cdot \boldsymbol{A}(\boldsymbol{r}_1) + \boldsymbol{p}_2 \cdot \boldsymbol{A}(\boldsymbol{r}_2) = \boldsymbol{p} \cdot \boldsymbol{A}(\boldsymbol{r}) + \boldsymbol{P} \cdot \boldsymbol{A}(\boldsymbol{R}) \text{ and}$$
$$A(\boldsymbol{r}_1)^2 + A(\boldsymbol{r}_2)^2 = \frac{1}{2}A(\boldsymbol{r})^2 + 2A(\boldsymbol{R})^2.$$

First we write out the Hamiltonian in original set of coordinates $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$

$$\begin{aligned} H =& \frac{1}{2m_e}\left(p_1^2 + p_2^2 - 2e\left[\boldsymbol{p}_1 \cdot \boldsymbol{A}(\boldsymbol{r}_1) + \boldsymbol{p}_2 \cdot \boldsymbol{A}(\boldsymbol{r}_2)\right] + e^2\left[A(\boldsymbol{r}_1)^2 + A(\boldsymbol{r}_2)^2\right]\right) \\ &+ \frac{1}{2}m_e\omega_0^2(r_1^2 + r_2^2) + \frac{e^2}{4\pi\epsilon_0}\frac{1}{|\boldsymbol{r}_1 - \boldsymbol{r}_2|}. \end{aligned}$$

Inserting the new set of coordinates defined in Equations (3.40) and (3.41) we have

$$\begin{aligned} H =& \frac{1}{2m_e}\left(2p^2 + \frac{1}{2}P^2 - 2e\left[\boldsymbol{p} \cdot \boldsymbol{A}(\boldsymbol{r}) + \boldsymbol{P} \cdot \boldsymbol{A}(\boldsymbol{R})\right] + e^2\left[\frac{1}{2}A(\boldsymbol{r})^2 + 2A(\boldsymbol{R})^2\right]\right) \\ &+ \frac{1}{2}m_e\omega_0^2(\frac{1}{2}r^2 + 2R^2) + \frac{e^2}{4\pi\epsilon_0}\frac{1}{r}. \end{aligned} \tag{3.42}$$

In this equation we identify two independent parts

$$H = 2H_r + \frac{1}{2}H_R, \tag{3.43}$$

where $H_r$ depends only on the relative coordinate

$$\begin{aligned} H_r =& \frac{1}{2m_e}\left(p^2 - e\boldsymbol{p} \cdot \boldsymbol{A}(\boldsymbol{r}) + e^2\frac{1}{4}A(\boldsymbol{r})^2\right) \\ &+ \frac{1}{2}m_e\frac{1}{4}\omega_0^2 r^2 + \frac{1}{2}\frac{e^2}{4\pi\epsilon_0}\frac{1}{r}, \end{aligned} \tag{3.44}$$

and $H_R$ depends only on the centre-of-mass coordinate

$$\begin{aligned} H_R =& \frac{1}{2m_e}\left(P^2 - 2e\boldsymbol{P} \cdot 2\boldsymbol{A}(\boldsymbol{R}) + e^2 4A(\boldsymbol{R})^2\right) \\ &+ \frac{1}{2}m_e\omega_0^2 4R^2. \end{aligned} \tag{3.45}$$

We now introduce the ansatz

$$\psi(\boldsymbol{r}_1, \boldsymbol{r}_2) = \psi_r(\boldsymbol{r})\psi_R(\boldsymbol{R}). \tag{3.46}$$

Using this we define two independent single particle equations

$$H_r\psi_r = E_r\psi_r, \tag{3.47}$$
$$H_R\psi_R = E_R\psi_R. \tag{3.48}$$

The energy is given by

$$E = 2E_r + \frac{1}{2}E_R. \tag{3.49}$$

To rewrite our equations in a form similar to the single particle equation (3.20) we define

$$\omega_r = \frac{1}{2}\omega_0, \qquad \omega_R = 2\omega_0,$$

and

$$\boldsymbol{A}_r = \frac{1}{2}\boldsymbol{A}(\boldsymbol{r}) \rightarrow B_r = \frac{1}{2}B_0,$$
$$\boldsymbol{A}_R = 2\boldsymbol{A}(\boldsymbol{R}) \rightarrow B_R = 2B_0.$$

Using these expressions we obtain

$$H_r = \frac{1}{2m_e}\left(\boldsymbol{p} - e\boldsymbol{A}_r\right)^2 + \frac{1}{2}m_e\omega_r^2 r^2 + \frac{1}{2}\frac{e^2}{4\pi\epsilon_0}\frac{1}{r}, \tag{3.50}$$
$$H_R = \frac{1}{2m_e}\left(\boldsymbol{P} - e\boldsymbol{A}_R\right)^2 + \frac{1}{2}m_e\omega_R^2 R^2. \tag{3.51}$$

We observe that the centre-of-mass equation (3.47) behaves just like the single-electron equation (3.20). The relative motion equation (3.48) also has a $1/r$-term. Introducing $\Omega$ and $K$ we set up a general Hamiltonian

$$H = \frac{1}{2m_e}\left(\boldsymbol{p} - e\boldsymbol{A}\right)^2 + \frac{1}{2}m_e\Omega_0^2 r^2 + K\frac{e^2}{4\pi\epsilon_0}\frac{1}{2r}, \tag{3.52}$$

which represents the single-particle Hamiltonian (3.8),the centre-of-mass Hamiltonian (3.51) and the relative Hamiltonian (3.50), with constants given by

$$\text{Single particle: } \Omega_0 = \omega_0, \quad K = 0, \quad B = B_0, \tag{3.53}$$
$$\text{Centre-of-mass: } \Omega_0 = 2\omega_0, \quad K = 0, \quad B = 2B_0 \text{ and} \tag{3.54}$$
$$\text{Relative motion: } \Omega_0 = \frac{1}{2}\omega_0, \quad K = 1 \quad B = \frac{1}{2}B_0. \tag{3.55}$$

We want to solve the time-independent Schrödinger equation $H\psi = E\psi$ for the general Hamiltonian (3.52). For the magnetic potential defined in Equation (2.46) it is

$$\begin{aligned} H = &- \frac{\hbar^2}{2m_e}\left[\nabla^2 - i\frac{eB_0}{\hbar}\frac{\partial}{\partial\theta}\right] \\ &+ \frac{1}{2}m_e\frac{{B_0}^2e^2}{4{m_e}^2}(x^2+y^2) + \frac{1}{2}m_e{\Omega_0}^2r^2 + K\frac{e}{4\pi\epsilon_0}\frac{1}{2r}. \end{aligned} \tag{3.56}$$

### 3.4.1 Two dimensions

For the two-dimensional case we use $\psi(\boldsymbol{r}) = \frac{1}{\sqrt{2\pi}}\frac{u(r)}{\sqrt{r}}e^{im\phi}$ as we did for the single-electron quantum dot. We also introduce the dimensionless variables defined in Section 2.2 to obtain

$$\left[-\frac{d^2}{d\overline{r}^2} + \frac{m^2-\frac{1}{4}}{\overline{r}^2} + \overline{\Omega}^2\overline{r}^2 + \frac{K}{\overline{r}}\right]u(\overline{r}) = \overline{\epsilon}u(\overline{r}), \tag{3.57}$$

where we have defined

$$\overline{\epsilon} = 2\overline{E} + \overline{B}m, \tag{3.58}$$

$$\overline{\Omega}^2 = \overline{\Omega}_0^2 + \frac{\overline{B}^2}{4}. \tag{3.59}$$

From Reference [10] we have a set of analytic solutions. More details about the derivation is given in Appendix A.2. Here we give a list of eigenvalues for $m = 0$ and $m = 1$ in Tables 3.1 and 3.2.

| n | $1/\omega_r$ | $\epsilon_r/2$ |
|---|---|---|
| 2 | 0.200000e+1 | 0.100000e+1 |
| 3 | 0.120000e+2 | 0.250000e+0 |
| 4 | 0.370880e+2 | 0.107852e+0 |
| 4 | 0.291199e+1 | 0.137363e+1 |
| 5 | 0.844674e+2 | 0.591944e-1 |
| 5 | 0.155326e+2 | 0.321903e+0 |
| 6 | 0.161253e+3 | 0.372085e-1 |
| 6 | 0.450281e+2 | 0.133250e+0 |
| 6 | 0.371853e+1 | 0.161354e+1 |

Table 3.1: Some corresponding values of $\omega_r$ and $\epsilon_r$ for m=0. See Ref. [10] for an extensive list.

| n | $1/\omega_r$ | $\epsilon_r/2$ |
|---|---|---|
| 2 | 0.600000e+1 | 0.500000e+0 |
| 3 | 0.280000e+2 | 0.142857e+0 |
| 4 | 0.725576e+2 | 0.689107e-1 |
| 4 | 0.744236e+1 | 0.671830e+0 |
| 5 | 0.146604e+3 | 0.409266e-1 |
| 5 | 0.333961e+2 | 0.179662e+0 |
| 6 | 0.257194e+3 | 0.272168e-1 |
| 6 | 0.840644e+2 | 0.832695e-1 |
| 6 | 0.874155e+1 | 0.800773e+0 |

Table 3.2: Corresponding values of $\omega_r$ and $\epsilon_r$ for m=1. See Ref. [10] for more values.

### 3.4.2 Three dimensions

Similarly, in three dimension we use $\psi(\boldsymbol{r}) = \frac{u(r)}{r}Y_{lm_l}$ to get

$$\left[-\frac{d^2}{d\overline{r}^2} + \frac{l(l+1)}{\overline{r}^2} + \overline{\Omega}^2\overline{r}^2 + \frac{K}{\overline{r}}\right]u(\overline{r}) = \overline{\epsilon}u(\overline{r}), \tag{3.60}$$

with the same definitions for $\overline{\Omega}$ and $\overline{\epsilon}$ as for the two-dimensional case and the requirement $\omega_z = \sqrt{{\omega_o}^2 + \frac{{B_0}^2}{4}}$. This requirement is necessary for spherical symmetry, see Section 3.3.2. As for the two dimensional case only particular analytic solutions may be found. See Reference [11] for a derivation. In table 3.3 we list some of these eigenvalues.

| n | $1/\omega_r$ | $\epsilon_r/2$ |
|---|---|---|
| 2 | 4 | 0.6250 |
| 3 | 20 | 0.1750 |
| 4 | 54.7386 | 0.0822 |
| | 5.26137 | 0.8553 |
| 5 | 115.299 | 0.0477 |
| | 24.7010 | 0.2227 |
| 6 | 208.803 | 0.0311 |
| | 64.8131 | 0.1003 |
| | 6.38432 | 1.0181 |

Table 3.3: Particular analytic solutions for the relative equation in three dimensions for $l = 0$. Table taken from [11].

### 3.4.3 Anti-symmetric wave functions for two particles

When we are working with fermions we require that the total wave function is anti-symmetric under the interchange of two particles $\boldsymbol{r}_1 \leftrightarrow \boldsymbol{r}_2$. In our new set of coordinates an interchange means that

$$\boldsymbol{R} \rightarrow \boldsymbol{R}, \quad \boldsymbol{r} \rightarrow -\boldsymbol{r}.$$

We observe that the centre-of-mass wave function is always symmetric. For the relative coordinates in two dimensions, an interchange of particles gives

$$r \rightarrow r, \quad \phi \rightarrow \phi + \pi.$$

The only part of the wave function that changes is $e^{im\phi}$. From this we have

$$e^{im\phi} \rightarrow e^{im\phi}e^{im\pi} = (-1)^m e^{im\phi}.$$

For even $m$ the spatial wave function is symmetric, and we must use an anti-symmetric spin function. When $m$ is odd the spatial wave function is anti-symmetric, and we must use a symmetric spin-function. In Section 2.5.2 we ordered the four spin states into symmetric (triplet) and anti-symmetric (singlet) states.

## 3.5 Summary

We now summarise the equations we have derived in this chapter. From now on we only use the dimensionless variables defined in Section 2.2. We remove all the overline symbols for the variables and let $\overline{r} \rightarrow r$ etc. From the general equation defined by the Hamiltonian (3.52) we get a radial equation. In two dimensions it is

$$\left[-\frac{d^2}{dr^2} + \frac{m^2 - \frac{1}{4}}{r^2} + \Omega^2 r^2 + \frac{K}{r}\right] u(r) = \epsilon u(r), \tag{3.61}$$

and in three dimensions it has a similar form

$$\left[-\frac{d^2}{dr^2} + \frac{l(l+1)}{r^2} + \Omega^2 r^2 + \frac{K}{r}\right] u(r) = \epsilon u(r). \tag{3.62}$$

In both equations we have

$$\epsilon = 2E + Bm, \tag{3.63}$$

$$\Omega^2 = \Omega_0^2 + \frac{B^2}{4}. \tag{3.64}$$

We can use these equations to solve the single-electron quantum dot, and the centre-of-mass equation and the relative motion equation for the two-electron quantum dot, where the constants must be defined for each problem by

$$\text{Single particle: } \Omega_0 = \omega_0, \quad K = 0, \quad B = B_0, \tag{3.65}$$

$$\text{Centre-of-mass: } \Omega_0 = 2\omega_0, \quad K = 0, \quad B = 2B_0 \text{ and} \tag{3.66}$$

$$\text{Relative motion: } \Omega_0 = \frac{1}{2}\omega_0, \quad K = 1 \quad B = \frac{1}{2}B_0. \tag{3.67}$$

The total energy of the two-particle problem is

$$E = 2E_r + \frac{1}{2}E_R + 2\Phi - B_o M_s, \tag{3.68}$$

where $M_s = m_{s_1} + m_{s_2}$. The energy depend on the quantum numbers

$$n_R = 0, 1, 2, \ldots, \quad n_r = 0, 1, 2, \ldots,$$
$$m_{s_1} = \pm\frac{1}{2}, \quad m_{s_2} = \pm\frac{1}{2},$$

in two dimensions we also have

$$m_R = 0, \pm 1, \pm 2, \ldots, \quad m_r = 0, \pm 1, \pm 2, \ldots,$$

and in three dimensions

$$l_R = 0, 1, 2, \ldots, \quad l_r = 0, 1, 2, \ldots,$$
$$m_R = 0, \pm 1, \pm 2, \ldots \pm l_R, \quad m_r = 0, \pm 1, \pm 2, \ldots \pm l_R.$$

# Chapter 4

# Numerical methods

In this chapter we give the numerical methods used in this thesis. The main focus is on solving partial differential equations. We first have a short section about the finite difference method and the finite element method, where we focus on the finite element method. We also discuss the of parallelisation of differential equations. In this thesis we will solve a one-dimensional equation, therefore we focus on the one-dimensional cases in these methods. In Section 4.4 we give a special numerical method for solving the time-dependent Schrödinger equation, the Blanes-Moan method. The last part of this chapter give a short introduction to eigenvalue problems and the ARPACK software.

## 4.1 Finite difference method (FDM)

In the finite difference method we partition the domain into a grid with $N$ nodes $x_0, x_1, \ldots x_{N-1}$. On this grid we search for an approximation $u_i$ to the exact solution $u(x_i)$ on each node. If the number of nodes is $N$, then the distance between the nodes is

$$h = \frac{x_{max} - x_{min}}{N - 1}.$$

The first derivative is approximated by

$$\frac{du_i}{dx} \approx \frac{u_{i+1} - u_{i-1}}{2h}. \tag{4.1}$$

Using this approximation we calculate the second derivative

$$\begin{aligned}\frac{d^2u_i}{dx^2} \approx & \frac{u'_{i+1/2} - u'_{i-1/2}}{h} \\ &= \frac{u_{i+1} - u_i}{h^2} - \frac{u_i - u_{i-1}}{h^2} \\ &= \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}.\end{aligned} \tag{4.2}$$

The truncation error can be calculated by a Taylor expansion of $u$ around $x_{i\pm1}$

$$u_{i\pm1} = u_i \pm hu' + \frac{h^2u''}{2} \pm \frac{h^3u'''}{6} + \dots.$$

Using this in Equation 4.2 we have

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = u'' + \frac{h^2u^{(4)}}{12} + \frac{h^4u^{(6)}}{360} + O(h^6), \tag{4.3}$$

which gives

$$\frac{d^2u_i}{dx^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + O(h^2). \tag{4.4}$$

The finite difference method approximates the differential equation at the nodes. For a differential operator $D$ we get a matrix: $D\boldsymbol{u} \to A\boldsymbol{u}$, where $\boldsymbol{u}$ is a vector consisting of all the $u_i$ in the domain. For simple boundary conditions of the form $u(x_k) = f$, we write $u_k = f$. Another type of boundary condition involving the derivative $u'(x_k) = g$, can be implemented by the discretisation of the first derivative (4.1) as $u_{k+1} - u_{k-1} = g$, or by another first derivative discertisation. The boundary conditions are implemented directly on the linear system, we see show this in the example below.

**Example: The Poisson equation**

For the Poisson equation

$$\begin{aligned}-\frac{d^2}{dx^2}u(x) = f(x), \quad & x \in (0,1), \\ u(0) = 0, \quad & u(1) = 0,\end{aligned} \tag{4.5}$$

we discretise the interior points by

$$-u_{i-1} + 2u_i - i_{i+1} = h^2 f(x_i), \quad i = 1, \dots i = N-2, \tag{4.6}$$

and set the boundary conditions as

$$u_0 = 0, \quad u_{N-1} = 0.$$

We get a tridiagonal linear system

$$A\boldsymbol{u} = \boldsymbol{b},$$

where

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & \dots & 2 & -1 \\ 0 & \dots & \dots & \dots & \dots & 0 & 1 \end{pmatrix},$$

and

$$\boldsymbol{b} = h^2 \begin{pmatrix} 0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-2} \\ 0 \end{pmatrix}.$$

### 4.1.1 Richardson extrapolation

The approximation to the second derivative given above is on the form

$$T(h) = T(0) + a_1 h^2 + a_2 h^4 ...,$$

where $T(0)$ is the solution when $h \to 0$, and $a_1, a_2 \dots$ are constants independent of $h$. For an approximation like this, we can use Richardsons extrapolation [12]. The Richardson extrapolations consists of doing approximations for different step lengths $h$, and using these approximations to eliminate error terms. For example in we calculate $T(h)$ and $T(\frac{h}{2})$ we can eliminate $a_1$ by

$$\frac{4T(\frac{h}{2}) - T(h)}{3} = T(0) - \frac{1}{4} a_2 h^4 + \dots.$$

We can obtain the formula

$$\begin{aligned} T_m^{(k)} &= T_{m-1}^{(k+1)} + \frac{T_{m-1}^{(k+1)} - T_{m-1}^{(k)}}{4^m - 1}, \quad m > 0 \\ &= \frac{4^m T_{m-1}^{(k+1)} - T_{m-1}^{(k)}}{4^m - 1}, \end{aligned} \tag{4.7}$$

where $T_0^{(k)} = T(\frac{h}{2^k})$, which results in an improved error

$$T_m^{(k)} = T(0) + a_{m+1}^{k,m} h^{2(m+1)} + a_{m+2}^{k,m} h^{2(m+2)} + \dots \tag{4.8}$$

$$= T(0) + O(h^{2(m+1)}). \tag{4.9}$$

## 4.2 Finite element method (FEM)

The finite element method (FEM) is more complicated to implement than the finite difference method, but it provides a more flexible method to approximate differential equations. For example, the finite element method can easily be extended to higher order approximations and can be used for complex geometries. We provide a short introduction here, focusing on the one-dimensional case. For a good introductory text to the finite element method see for example *Computational Partial Differential Equations* [13], or see Reference [14].

### 4.2.1 One dimensional finite element method

We introduce the finite element method by the following steps:

1. Divide the domain $\Omega$ into $M$ non-overlapping elements,

   $$\Omega_e, \quad e = 1, \dots, M.$$

   Each element has $n_e$ nodes, where the global nodes are denoted by

   $$x^{[i]}, \quad i = 1, \dots, N.$$

   The total number of nodes is $N = M(n_e - 1) + 1$ because two neighbouring elements share one node. The size of each element $h_e$ is the distance between the two boundary nodes of the element.

2. We write our differential equation as

   $$\mathcal{L}(u(x)) = 0,$$

   where $\mathcal{L}$ is a differential operator specific to our problem.

3. We approximate the function $u(x)$ by:

   $$u(x) \approx \hat{u} = \sum_{j=1}^{N} u_j N_j(x), \tag{4.10}$$

   where $u_j$ are the unknowns and $N_j(x)$ are basis functions.

4. We minimise the residual $\mathcal{L}(\hat{u})$ by

$$\int_\Omega \mathcal{L}(\hat{u}) N_i d\Omega = 0, \quad i = 1, \ldots, N. \tag{4.11}$$

5. The basis functions $N_i$ are simple piecewise polynomials which are non-zero only for a few elements that contain the node $x^{[i]}$. For linear basis functions $\hat{u}$ is a piecewise linear function.

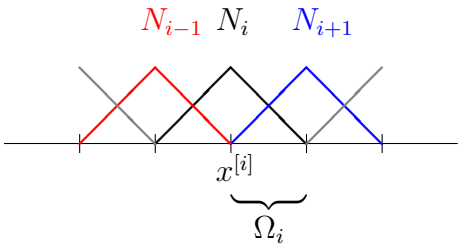


Figure 4.1: Linear basis functions over four elements.

For the basis functions$N_i$ we require:

1. $N_i$ is a polynomial over each element uniquely defined by its values at the nodes in the element

2. $N_i(x^{[j]}) = \delta_{ij} \rightarrow \hat{u}(x^{[j]}) = u_j$

The simplest case is the one dimensional element with two nodes placed at the boundaries and $N_i$ given as piecewise linear polynomials. Then we have element $e$ given by

$$\Omega_e = \left[x^{[e]}, x^{[e+1]}\right], \quad h_e = x^{[e+1]} - x^{[e]}.$$

The basis functions can be calculated using the property $N_i(x^j) = \delta_{ij}$, see Figure 4.1. We will not set up the basis functions here, but note that they only give contribution to the two closest neighbours. For quadratic polynomials we have three nodes per element, two at the boundaries and one in the centre of the element. The basis functions are piecewise quadratic functions. We discuss the basis functions further for local elements in the next section. For linear elements we get a piecewise linear approximation $\hat{u}(x)$ like we have shown in Figure 4.2. For higher order basis function, we have the same order approximation.

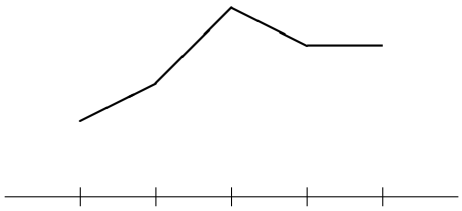

Figure 4.2: Piecewise linear basis functions give a piecewise linear approximation $\hat{u}(x)$.

### Boundary conditions

For boundary conditions of the form $u(x_k) = f$, we introduce $u_k = f$ directly into the linear system as we did for the finite difference method. The other type involving the derivative $u'(x_k) = g$ is implemented in an elegant way using partial integration on $\hat{u}''$,

$$\int_\Omega N_i N_j'' d\Omega u_j = -\int_\Omega N_i'(x) N_j'(x) d\Omega u_j + N_i(x_{max})\hat{u}'(x_{max}) - N_i(x_{min})\hat{u}'(x_{min}).$$

We can now set $N_k(x_k)\hat{u}'(x_k) = g$.

### Example: Poisson equation

For a simple example we study the Poisson equation as we did for the finite difference method

$$-\frac{d^2}{dx^2}u(x) = f(x), \quad x \in (0,1), \tag{4.12}$$

$$u(0) = 0, \quad u(1) = 0. \tag{4.13}$$

This expression gives

$$\mathcal{L}(u(\boldsymbol{x})) = -\frac{d^2}{dx^2}u(x) - f(x).$$

Inserting this into Equation (4.11) we get

$$-\sum_{j=1}^{N}\int_\Omega N_i(x) N_j''(x) d\Omega u_j - \int_\Omega f(x) N_i(x) = 0, \quad i = 1, \ldots, N.$$

For expressions of the form $N_i(x)N_j''(x)$ we use integration by parts

$$\sum_{j=1}^{N} \int_\Omega N_i'(x)N_j'(x)d\Omega u_j - N_i(x_{max})\hat{u}'(x_{max}) + N_i(x_{min})\hat{u}'(x_{min}) - \int_\Omega f(x)N_i(x) = 0, \quad i = 1, \ldots, N.$$

The boundary integral terms always give zero contribution for $i = 2, \ldots, N-1$. For our set of boundary conditions (4.13) we also get zero contribution for $i = 0$ and $i = N$. We use linear elements. Because of the properties of the basis functions we only get contributions to the sums for $j = (i-1, i, i+1)$

$$\sum_{j=i-1}^{i+1} \int_\Omega N_i'(x)N_j'(x)d\Omega u_j - \int_\Omega f(x)N_i(x) = 0, \quad i = 2, \ldots, N-1.$$

The result is a tridiagonal linear system

$$\sum_{j=i-1}^{i+1} A_{ij}u_j = b_i,$$

where

$$A_{ij} = \int_\Omega N_i'(x)N_j'(x)d\Omega, \quad b_i = \int_\Omega f(x)N_i(x).$$

In addition we must impose the boundary conditions $u_1 = 0$ and $u_N = 0$ directly in this linear system. Here we have set up the linear system for this specific problem, but a similar construction can be set up for any problem. In general an operator $\mathcal{L}$ working on a function $u$ gives a matrix, while a function independent of $u$ (like $f(x)$) gives a vector. In this example we end up with a linear system of equations, but we will see in Chapter 5 that for eigenvalue problems we end up with a generalised eigenvalue problem. For a one dimensional problem the matrix will have a bandwidth of $2n_e - 1$, where $n_e$ is the number of nodes per element. For higher dimensions this may not be the case.

### 4.2.2 Element-by-element formulation

In this section we introduce the element-by-element formulation. By introducing local coordinates we can generalise the elements regardless of the size

and shape of the elements to standard boundaries. Following the example from the previous section we write a sum over each element $e$

$$A_{ij} = \sum_{e=1}^{M} A_{ij}^{(e)}, \quad A_{ij}^{(e)} = \int_{\Omega_e} N_i' N_j' d\Omega \text{ and} \tag{4.14}$$

$$b_i = \sum_{e=1}^{M} b_i^{(e)}, \quad b_i^{(e)} = \int_{\Omega_e} N_i f d\Omega. \tag{4.15}$$

We define the matrix $A_{ij}^{(e)}$ as an element matrix, and $b_i^{(e)}$ as an element vector, note that the integrals above depend on the problem we are solving. In element $e$, $A_{ij}^{(e)}$ is different from zero only for the nodes that belong to this element. Therefore the element matrix/vector has the size $n_e$.

We now define local coordinates and numbering of nodes. We choose the local coordinate

$$\xi \in [-1, 1],$$

and define local node numbers

$$r, s = 1, \ldots, n_e,$$

that map to the global node numbers $i, j$ by $i = q(e, r)$ and $j = q(e, s)$. For the one-dimensional case we have the simple expression

$$\begin{aligned} q(e, r) =& (n_e - 1)(e - 1) + r, \\ e = 1, \ldots, M, \quad & r = 1, \ldots, n_e, \quad q = 1, \ldots, N, \end{aligned} \tag{4.16}$$

but for higher dimensions we generally have a more complex geometry and must set up a table for $q(e, r)$. Now all basis functions will be equal in local coordinates. One element is drawn in Figure 4.3, where the basis functions $N_i$ are linear and are defined by the property $N_i(\xi_j) = \delta_{ij}$. Calculations and expressions for linear and quadratic basis functions are given further down.

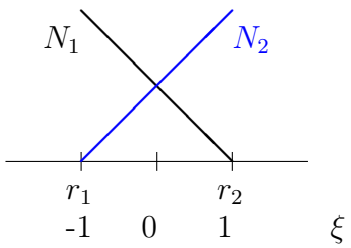


Figure 4.3: One element in local coordinates with linear basis functions.

We map from global coordinates $x$ to local coordinates $\xi$ by

$$x^{(e)}(\xi) = \sum_{r=1}^{n_e} \tilde{N}_r(\xi) x^{q(e,r)}, \tag{4.17}$$

where $\tilde{N}_r(\xi)$ are the basis functions defined for local coordinates (expressions are given on the next page ). The derivatives transform as

$$\frac{dN_i}{dx} = \frac{d\tilde{N}_r}{d\xi}\frac{d\xi}{dx} = J^{-1}\frac{d\tilde{N}_r}{d\xi}, \tag{4.18}$$

$$\frac{dN_j}{dx} = \frac{d\tilde{N}_s}{d\xi}\frac{d\xi}{dx} = J^{-1}\frac{d\tilde{N}_s}{d\xi}, \tag{4.19}$$

where $r, s$ are local node numbers corresponding to $i = q(e, r)$ and $j = q(e, s)$. The integral is given by

$$\int_{x^{[e]}}^{x^{[e+1]}} dx = \int_{-1}^{1} detJ d\xi, \tag{4.20}$$

where $J$ is defined as

$$J_{i,j} = \frac{\partial x_j}{\partial \xi_j},$$

in the case of one-dimensional piecewise linear basis functions $J$ has the simple form $J = \frac{h_e}{2}$.

After calculating the integrals each element will have it's own $n_e \times n_e$ element matrix and element vector of size $n_e$ given by $A^{(e)}$ with matrix elements $A^{(e)}_{rs}$, and element vector $b^{(e)}$ with $b^{(e)}_r$. We assemble the element matrices/vectors by adding them to the global system using $i = q(e, r)$ to find the correct global nodes

$$A_{q(e,r),q(e,s)} += A^{(e)}_{r,s},$$
$$b_{q(e,r)} += b^{(e)}_r.$$

For the matrix case this is shown in Figure 4.4. For the one dimensional case we see that we get an overlap of matrix element $A^{(e+1)}_{11}$ and $A^{(e)}_{n_e n_e}$. The same goes for the element vectors, we get an overlap of $b^{(e+1)}_1$ and $b^{(e)}_{n_e}$.

### 4.2.3 Local basis functions

#### Basis functions for linear polynomials

In Figure 4.3 we drew the linear basis functions in local coordinates. We now give the mathematical expressions. For linear polynomials the local

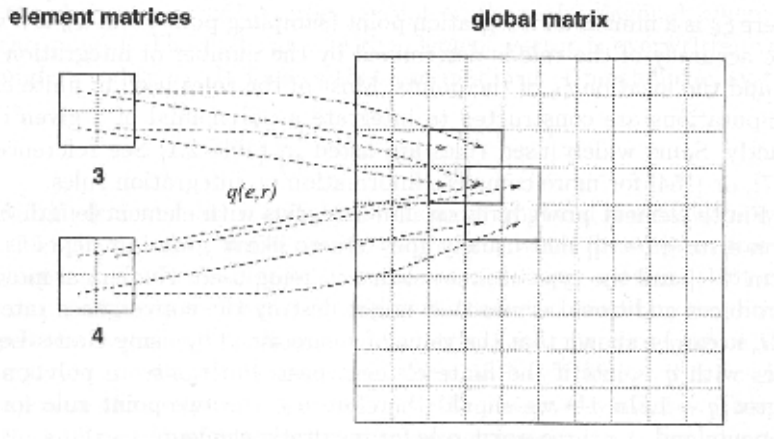


Figure 4.4: The global matrix is assembled for the element matrices. In a one-dimensional problem they are added along the diagonal with overlap of matrix element $A_{11}^{(e+1)}$ and $A_{n_e n_e}^{(e)}$. Figure taken from [13].

basis functions are

$$\tilde{N}_1(\xi) = \frac{1}{2}(1-\xi), \tag{4.21}$$

$$\tilde{N}_2(\xi) = \frac{1}{2}(1+\xi), \tag{4.22}$$

and the mapping to global coordinates are given by

$$q(e,r) = e - 1 + r.$$

When we have equally spaced nodes expression (4.17) results in

$$\begin{aligned} x^{(e)}(\xi) &= \frac{1}{2}\left(x^{[e]} + x^{[e+1]}\right) + \xi\frac{1}{2}\left(x^{[e+1]} - x^{[e]}\right) \\ &= \frac{1}{2}\left(x^{[e]} + x^{[e+1]}\right) + \xi\frac{1}{2}h \end{aligned} \tag{4.23}$$

### Basis functions for quadratic polynomials

For quadratic basis functions we have three nodes per element, $n_e = 3$. Two nodes are placed at the boundary of the element

$$\Omega_e = [x^{[2e-1]}, x^{[2e+1]}],$$

and one is placed at the centre $x^{[2e]}$.

The local basis functions are

$$\tilde{N}_1(\xi) = \frac{1}{2}\xi(\xi - 1), \tag{4.24}$$
$$\tilde{N}_2(\xi) = (1 + \xi)(1 - \xi), \tag{4.25}$$
$$\tilde{N}_3(\xi) = \frac{1}{2}\xi(1 + \xi), \tag{4.26}$$

and the mapping to global coordinates are given by

$$q(e, r) = 2(e - 1) + r.$$

When we have equally spaced nodes expression (4.17) results in

$$\begin{aligned} x^{(e)}(\xi) &= x^{[2e]} + \xi\frac{1}{2}\left(x^{[2e+1]} - x^{[2-+1]}\right) \\ &= x^{[2e]} + \xi\frac{1}{2}h. \end{aligned} \tag{4.27}$$

The quadratic basis functions are given in Figure 4.5.

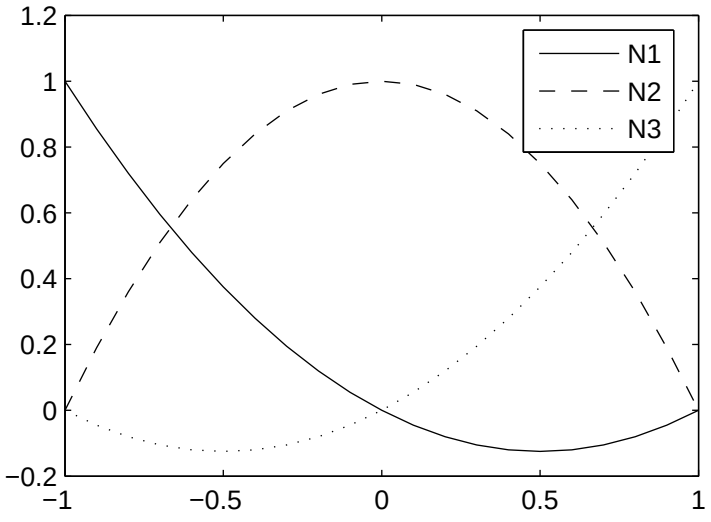


Figure 4.5: Quadratic basis functions.

**Calculation of local basis functions**

We require that the basis functions have the property $N_i(\xi_j) = \delta_{ij}$. For the first linear basis function we have

$$\begin{aligned} N_i(\xi) &= a\xi + b, \\ N_1(-1) &= 1 = -a + b, \\ N_1(1) &= 0 = a + b, \end{aligned}$$

which gives the linear system for $a$ and $b$

$$\begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Compactly, we write the two linear systems for $N_1$ and $N_2$ as

$$\begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \delta_{i1} \\ \delta_{i2} \end{bmatrix}.$$

The solution for the basis functions are

$$\begin{aligned} N_1(\xi) &= \frac{1}{2} - \frac{1}{2}\xi, \\ N_2(\xi) &= \frac{1}{2} + \frac{1}{2}\xi. \end{aligned}$$

Similarly, for quadratic elements we have the linear systems

$$\begin{bmatrix} 1 & -1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} \delta_{i1} \\ \delta_{i2} \\ \delta_{i3} \end{bmatrix}$$

We now have three linear systems to solve, resulting in (4.24) - (4.26). The same procedure can be done for higher order polynomials.

### 4.2.4 Algorithm

To get an overview of a finite element algorithm we set it up for the Poisson example (4.12) here. However this algorithm can be generalised using other element vectors/matrices than $b_r$ and $A_r s$. For example in the quantum dot equation we get two element matrices and no element vector.

FINITE ELEMENT ALGORITHM
Initialise grid
set global and element matrices/vectors = 0
**for** e=1, ...,m LOOP OVER ALL ELEMENTS
**for** r,s=1, ..., $n_e$ LOOP OVER LOCAL NODES
Calculate the integrals for the local element matrices/vectors ($A_{rs}$ and $b_r$)
Set essential boundary conditons
Add the local element matrix $A^{(e)}$ and vector $b^{(e)}$ to the global system
Solve the resulting linear system

### 4.2.5 Higher dimensions

In this thesis we are dealing with one-dimensional equations, and will not focus on higher dimensions, we just give an idea of how the finite element method works for higher dimensions. For more information, we refer to [13]. In higher dimensions the strength of the method is the flexibility in the grid. The grid can be a complex structure, but by the element formalism the local elements have simple shapes. An example of this is given in Figure 4.6.

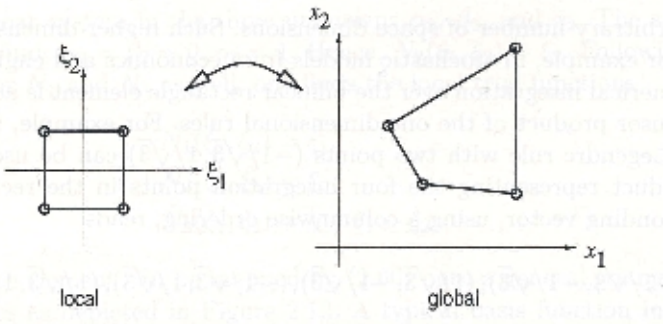


Figure 4.6: Here we have a mapping of an element with complex shape, in local coordinates it is mapped to the $[-1,1] \times [-1,1]$ square. Figure taken from [13].

In higher dimensions we can calculate new basis functions using the property $N_i(x^{[j]}) = \delta_{ij}$ or we generalise the one-dimensional basis functions. For example in two dimensions we have bilinear basis functions defined over the

rectangle $[-1,1]\times[-1,1]$ by

$$\begin{aligned}\tilde{N}_1(\xi_1,\xi_2) &= \tilde{N}_1(\xi_1)\tilde{N}_1(\xi_2),\\ \tilde{N}_2(\xi_1,\xi_2) &= \tilde{N}_2(\xi_1)\tilde{N}_1(\xi_2),\\ \tilde{N}_3(\xi_1,\xi_2) &= \tilde{N}_1(\xi_1)\tilde{N}_2(\xi_2),\\ \tilde{N}_4(\xi_1,\xi_2) &= \tilde{N}_2(\xi_1)\tilde{N}_2(\xi_2).\end{aligned}$$

The assembly of the global linear system for higher dimensions is more complicated and we have to make sure to add the contribution to a node from all the elements containing that node.

### 4.2.6 Time-dependent problems

For time-dependent problems it is most common to use the finite element method for discretisation in space and the finite difference method for discretisation in time. If we denote the time step as $u^l = u(\boldsymbol{x}, t_l)$ we discretise the time derivative by a finite difference scheme and then apply the finite element discretisation

$$u^l(\boldsymbol{x}) \approx \hat{u}^l = \sum_{j=1}^{N} u_j^l N_j(\boldsymbol{x}). \tag{4.28}$$

We show this on a simple example

$$\frac{du}{dt} = Lu.$$

We discertise this using the Euler method in time $\frac{du}{dt} \approx \frac{u^{l+1}-u^l}{\Delta t}$ and get

$$u^{l+1} = u^l + \Delta t L u^l.$$

We now introduce the discretisation (4.28) and obtain

$$\int_\Omega N_i N_j d\Omega u_j^{l+1} = \int_\Omega N_i N_j d\Omega u_j^l + \Delta t \int_\Omega N_i L N_j d\Omega u_j^l, \tag{4.29}$$

$$\rightarrow A u^{l+1} = A u^l + \Delta t B u^l, \tag{4.30}$$

when we write out the matrices. This gives a linear system to be solved for each time step. We discuss some special numerical methods for the evolution of the time-dependent Schrödinger equation in 4.4.

## 4.3 Solving partial differential equations in parallel

The capacity of single processors can no longer keep up with the demand for larger and faster simulations in scientific computing. To meet these increasing demands we need parallel computing. In parallel computing we divide the work among a number of processors to run larger and faster simulations. The memory on a multiple processor computer can be organised in several ways. The most common way is the distributed memory system, where each processor has its own memory. For such systems data must be exchanged explicitly by message passing. MPI - "message passing interface" is a useful library which provides functions for communication between processors.This must be provided by the programmer. Because of communication cost due to message passing a parallel program will not acquire full speed-up

$$S(P) = \frac{T(1)}{T(P)} \leq P. \tag{4.31}$$

Here $T(i)$ is the time spent using $i$ processors and $P$ is the number of processors in parallel. The partitioning of the work is dependent on the algorithm. For an efficient parallel algorithm the work load should be evenly distributed to avoid idle time, and the communication cost should be minimised. In this thesis we focus on parallelisation of partial differential equations (PDE).

When solving a PDE there are two time consuming parts: building the matrices/vectors in the system and solving this system (linear system or eigenvalue problem). The first step is to partition the data among the processors. In most cases the spatial grid is divided into $P$ sub grids. Each processor only has access to its own sub grid data. On this sub grid the processor applies operations in the same way as a sequential solver. In addition we provide communication between neighbouring sub grids. After reviewing the linear system and linear algebra operations we give more details of the partitioning of finite difference grid and finite element grids.

To solve a linear system or an eigenvalue problem in parallel we must use iterative solvers. Iterative solvers generally rely on the linear algebra operations: matrix-vector product, vector addition and the inner product. The task of parallelising an iterative solver thus consists of parallelising these operations according to the partitioning of the matrices and vectors.

### 4.3.1 Parallel linear algebra operations

In a matrix the rows are partitioned

$$A = \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & \cdots & A_{1P} \\ A_{21} & A_{22} & \cdots & A_{2P} \\ \vdots & \vdots & \ddots & \vdots \\ A_{P1} & A_{P2} & \cdots & A_{PP} \end{array}\right], \tag{4.32}$$

where $A_{ij}$ are block matrices. Processor $p$ holds the blocks $A_{ip}$. The lines divide the matrix/vector among the processors. For a vector we have

$$\boldsymbol{b} = \left[\begin{array}{c} \boldsymbol{b}_1 \\ \hline \boldsymbol{b}_2 \\ \hline \vdots \\ \hline \boldsymbol{b}_P \end{array}\right]. \tag{4.33}$$

From this partitioning we set up the linear algebra operations:

#### Vector addition

The vector addition $\boldsymbol{w} = \boldsymbol{x} + \boldsymbol{y}$ is given by local vector additions

$$\boldsymbol{w}_p = \boldsymbol{x}_p + \boldsymbol{y}_p, \quad p = 1, \ldots, P. \tag{4.34}$$

This operation requires no communication.

#### Inner product

The inner product $s = \boldsymbol{x} \cdot \boldsymbol{y}$ is calculated locally on each processor $p$ by

$$s_p = \boldsymbol{x}_p \cdot \boldsymbol{y}_p. \tag{4.35}$$

All the local inner products $s_p$ are sent to a master node where the full inner product is calculated $s = \sum_1^P s_p$ (or distributed to all processors using MPI_ALLREDUCE with MPI_SUM).

#### Matrix-vector product

The matrix-vector product $\boldsymbol{w} = A\boldsymbol{b}$ requires communication between all processors for a full matrix $A$. However when we discretise a PDE, the matrix is sparse. By using a smart partitioning, many of the block matrices are zero and we only need communication between a few processors.

Using the definitions above the result should be

$$\boldsymbol{w} = \left[\begin{array}{c} \boldsymbol{w}_1 = A_{11}\boldsymbol{b}_1 + A_{12}\boldsymbol{b}_2 + \ldots A_{1P}\boldsymbol{b}_P \\ \hline \boldsymbol{w}_2 = A_{21}\boldsymbol{b}_1 + A_{22}\boldsymbol{b}_2 + \ldots A_{2P}\boldsymbol{b}_P \\ \hline \vdots \\ \hline \boldsymbol{w}_P = A_{P1}\boldsymbol{b}_1 + A_{P2}\boldsymbol{b}_2 + \ldots A_{PP}\boldsymbol{b}_P \end{array}\right].$$

From these expressions we see that we can calculate local matrix-vector products. The local result $A_{pp}\boldsymbol{b}_p$ is stored in $\boldsymbol{w}_p$, while the array $A_{pq}\boldsymbol{b}_q$ is sent to processor $q$ and added to the vector $\boldsymbol{w}_q$.

### 4.3.2 Grid partitioning

#### Partitioning of finite difference grids

On a finite difference grid we divide the grid by planes that are perpendicular to the coordinate axes. Special attention must be paid to the internal boundaries along these planes. For example if we study the second derivative operator $(u_{i-1} - 2u_i + u_{i+1})$, we see that at the boundaries we need information from a grid point in the next sub grid. To handle this problem we set up ghost points along the internal boundaries. See Figure 4.7 for a one-dimensional partitioned grid with ghost points. The values of the ghost points are not set by the finite difference scheme, but are received from the neighbouring grid after it has been updated there.

Figure 4.7: A one dimensional grid, with sub grid computational points (∘) and ghost points (•).

#### Partitioning of finite element grids

For the finite element method we must divide the elements among the processors. When we are working with a one-dimensional problem, the task of dividing the elements is simple. However for complex grids this is a difficult challenge and we must use partitioning algorithms, see [15]. Each processor works on its own group of elements and sets up its linear system as if it was a standard sequential solver. We refer to the "global" matrix for processor $p$ as $A_p$ and the vector as $\boldsymbol{b}_p$. For a finite element grid the interior boundary

points are shared between two (or more in higher dimensions) sub grids, because of this $A_p$ and $\boldsymbol{b}_p$ will not be equal to $A$ and $\boldsymbol{b}$ at the interior boundary nodes. For a vector $\boldsymbol{b}_i$ we sum the values at the shared boundary nodes, so that all the boundary nodes have the correct (and same) value. This special construction affects the linear algebra operations.

The vector addition is the same. For the finite element method the boundary nodes are duplicated and local inner product must be adjusted by

$$c_i = c_i - \sum_k \frac{o_k - 1}{o_k} u_k v_k, \tag{4.36}$$

where $k$ runs over all internal boundary nodes and $o_k$ is the number of sub grids sharing this node. In the one-dimensional case each internal boundary node belong to two processors and have $o_k = 2$. The matrix-vector product is simplified. First we calculate the local matrix vector products $\boldsymbol{w}_i = A_i \boldsymbol{u}_i$. To get the correct values and the internal boundaries we add the contribution from all the neighbouring sub grids for $\boldsymbol{w}_i$.

## 4.4 Time evolution of the Schrödinger equation

The time-dependent Schrödinger equation is

$$i\hbar \frac{\partial}{\partial t} \psi(t) = H(t)\psi(t), \tag{4.37}$$

where $\psi$ is also a function of the spatial coordinates, but for simplicity we just write the time coordinate here. We define the time evolution operator $U$ by

$$\psi(t_2) = U(t_2, t_1)\psi(t_1), \quad \psi(t) = U(t, 0)\psi(0), \tag{4.38}$$

and rewrite the Schrödinger equation into an equation for $U$

$$i\hbar \frac{\partial}{\partial t} U(t) = H(t)U(t). \tag{4.39}$$

For a time-independent Hamiltonian the time evolution operator is $U(t) = e^{-iEt/\hbar}$, where $E$ is a diagonal matrix with the eigenvalues along the diagonal. To avoid working with imaginary numbers we can use imaginary time $t = i\tau$, we also use atomic units as defined in Section 2.2. The equation for $U$ is then

$$\frac{\partial}{\partial \tau} U(\tau) = H(\tau)U(\tau) \tag{4.40}$$

Using a simple Euler scheme we have

$$\frac{\partial}{\partial\tau} \approx \frac{u^{l+1} - u^l}{\Delta\tau}, \tag{4.41}$$

$$\rightarrow u^{l+1} = \left[1 + \Delta\tau H^l\right] u^l, \tag{4.42}$$

with $u^l = U(\tau)$ and $H^l = H(\tau)$. A more accurate finite difference scheme is the Runge-Kutta method. It has an accuracy of $O(h^5)$, where $h$ is the step length. The next time step is calculated by

$$k_1 = hH^l u^l, \tag{4.43}$$

$$k_2 = hH^l(u^l + \frac{1}{2}k_1), \tag{4.44}$$

$$k_3 = hH^l(u^l + \frac{1}{2}k2), \tag{4.45}$$

$$k_4 = hH^l(u^l + k3), \tag{4.46}$$

$$u^{l+1} = u^l + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{4.47}$$

Here $k_i$ are also vectors of the same size as $u$.

### 4.4.1 Splitting of the Hamiltonian

We write the Hamiltonian on the form

$$H = H_0 + H_1(t), \tag{4.48}$$

where $H_0$ is time-independent. If we solve the time-independent Schrödinger equation for $H_0$, as we have done for the quantum dot in Chapter 3, we can do a change of basis

$$H_0 = PEP^\dagger, \tag{4.49}$$

where $E$ is a diagonal matrix with the eigenvalues of $H_0$ on the diagonal, and $P$ is a unitary matrix where the columns are the eigenvectors. We also define

$$d(t) = P^\dagger u(t) \text{ and } W(t) = P^\dagger H_1(t) P. \tag{4.50}$$

If we apply $P^\dagger$ on $H^l u^l$ we have

$$P^\dagger H^l u^l = P^\dagger H_0 u^l + P^\dagger H_1^l u^l \tag{4.51}$$

$$= EP^\dagger u^l + W^l P^\dagger u^l \tag{4.52}$$

$$= \left[E + W^l\right] d^l. \tag{4.53}$$

### 4.4.2 Blanes-Moan method

When we have a separable Hamiltonian we can use the Blanes-Moan method described in [16]. In this article another splitting is defined, but we use the splitting defined in Equation (4.48). We approximate the time evolution operator by discretising the total time interval $(0, t)$ into $N_t$ steps

$$U(t,0) = U(t_1,t_0)U(t_2,t_1)\cdots U(t_{N_t},t_{N_t-1}) \tag{4.54}$$

$$= \prod_{n=0}^{N_t-1} U(t_n+\Delta t, t_n), \tag{4.55}$$

where

$$\Delta t = \frac{t}{N_t}, \quad t_n = n\Delta\tau, \quad n = 0\ldots, N_t - 1. \tag{4.56}$$

For a separable Hamiltonian $H = A + B$ the solution can be approximated by

$$U = e^{A_1\Delta t/2} e^{B_{\frac{1}{2}}\Delta t} e^{A_0\Delta t/2} + O(\Delta t^3), \tag{4.57}$$

$$A_k = -iA(t_n + k\Delta t), \quad B_{\frac{1}{2}} = -iB(t_n + \frac{1}{2}\Delta t), \quad k = 1, 2. \tag{4.58}$$

If we choose to use the basis we defined above and set $A = E$ and $B = W$ we have

$$U = e^{-iE\Delta t/2} e^{-i\Delta t W(t_n+\frac{1}{2}\Delta t)} e^{-iE\Delta t/2} + O(\Delta t^3), \tag{4.59}$$

where the expressions are simplified because $E$ is independent of time. The exponential $e^{-iE\Delta t/2}$ is diagonal because $E$ is a diagonal matrix.

A higher precision formula is also defined in [16], where $U$ as approximated by

$$U = e^{H^{(1)}} e^{H^{(0)}} e^{-H^{(1)}} + O(\Delta t^5), \tag{4.60}$$

$$H^{(k)} = -\frac{i}{\Delta t^k} \int_{t_n}^{t_n+\Delta t} \left[ s - \left( t_n + \frac{1}{2}\Delta t \right) \right]^k H(s) ds, \quad k = 1, 2. \tag{4.61}$$

Writing out the integrals we have

$$H^{(0)} = -i \int_{t_n}^{t_n+\Delta t} H(s) ds, \tag{4.62}$$

$$H^{(1)} = -\frac{i}{\Delta t} \int_{t_n}^{t_n+\Delta t} \left[ s - t_n - \frac{1}{2}\Delta t \right] H(s) ds$$

$$= \frac{i}{\Delta t} \left[ t_n + \frac{1}{2}\Delta t \right] \int_{t_n}^{t_n+\Delta t} H(s) ds - \frac{i}{\Delta t} \int_{t_n}^{t_n+\Delta t} s H(s) ds. \tag{4.63}$$

Here we also use the splitting of the Hamiltonian $H = E + W(t)$ for $H^{(k)}$. This simplify the integrals because $E$ is time-independent and go outside the integrals.

$$\int_{t_n}^{t_n+\Delta t} H(s)ds = E\Delta t + \int_{t_n}^{t_n+\Delta t} W(s)ds, \tag{4.64}$$

$$\int_{t_n}^{t_n+\Delta t} sH(s)ds = E\Delta t\left(t_n + \frac{1}{2}\Delta_t\right) + \int_{t_n}^{t_n+\Delta t} sW(s)ds. \tag{4.65}$$

## 4.5 Eigenvalue problems

The time-dependent Schrödinger equation is a differential equation which must be solved as an eigenvalue problem. We are searching for the eigenstates of the Hamiltonian which is a differential operator

$$H\boldsymbol{\psi} = E\boldsymbol{\psi}. \tag{4.66}$$

In the previous sections we have discussed the discretisation of differential equations. When discretising (4.66) we get a matrix eigenvalue problem. For the finite difference method we have a standard eigenvalue problem

$$A\boldsymbol{u} = \lambda\boldsymbol{u}, \tag{4.67}$$

and for the finite element method we have a generalised eigenvalue problem

$$A\boldsymbol{u} = \lambda B\boldsymbol{u}. \tag{4.68}$$

The reason that we get $B\boldsymbol{u}$ on the right side is that for the finite element method a vector give $\boldsymbol{u} \rightarrow B\boldsymbol{u}$.

In the special case where $B$ is a symmetric positive definite matrix we can reduce the generalised eigenvalue problem to a standard one. We do a Cholesky decomposition of $B$

$$B = LL^T,$$

and rewrite our problem to

$$\begin{aligned}&\left(L^{-1}AL^{-T}\right)\left(L^T u\right) = \lambda\left(L^T u\right)\\ &\rightarrow Cy = \lambda y,\end{aligned}$$

which is a standard eigenvalue problem.

The standard way to solve eigenvalue problems is by solving the equation

$$det(A - \lambda I) = 0, \tag{4.69}$$

for $\lambda$ and solving the linear systems

$$(A - \lambda_i I)x_i, \tag{4.70}$$

for the eigenvectors $x_i$. However this method is not suitable for large problems.

### 4.5.1 The ARPACK eigenvalue solver

There are several numerical methods for solving a standard eigenvalue problem, such as the Jacobi method, see for example Reference [17]. We will however use the ARPACK software [18]. The ARPACK library provides solvers for both the standard eigenvalue problem and the generalised eigenvalue problem. There is also a parallel extension: P_ARPACK. The ARPACK software uses the implicitly restarted Arnoldi method. It is an iterative solver which requires the computation of matrix-vector products. For more details about the software package, see the ARPACK user guide [18], and for numerical methods, see Reference [19]. We will discuss how this package is implemented in Chapter 5.

# Chapter 5

# Implementation of the numerical methods

In this chapter we discuss the implementation of the numerical methods. First we focus on the time-independent Schrödinger equation

$$H_0\psi = E\psi. \tag{5.1}$$

In Section 5.1 we give the finite element and finite difference discretisations of the quantum dot equations. We have built a general solver using these methods. Details about the program are given in Section 5.2. Then, in Section 5.3, we move on to the time-dependent Schrödinger equation where the Hamiltonian is given by

$$H = H_0 + H_1(t), \tag{5.2}$$

where $H_0$ is the time-independent Hamiltonian which has already been solved and $H_1(t)$ is a time-dependent perturbation. We use the Blanes-Moan method, which was described in Section 4.4. Because we already have a set of eigenvalues and eigenvectors for $H_0$ we use them as a basis to simplify the equations.

## 5.1 Implementation of the radial equation

In Chapter 3 we discussed the quantum system for the one-electron and the two-electron quantum dot. There we derived a general one-dimensional equation which describes this system. We choose to write it in a more general form as

$$-\frac{d^2u(r)}{dr^2} + Y(r)u(r) = \epsilon u(r), \tag{5.3}$$

where $Y(r) = \frac{D}{r^2} + 2\overline{V}$ includes both the potential and the constants from the angular equation

$$\text{2D:} \quad Y(r) = \frac{m^2 - \frac{1}{4}}{r^2} + 2\overline{V}, \tag{5.4}$$

$$\text{3D:} \quad Y(r) = \frac{l(l+1)}{r^2} + 2\overline{V}. \tag{5.5}$$

For the quantum dot the potential is

$$2\overline{V} = \Omega^2 r^2 + \frac{K}{r}, \tag{5.6}$$

where the constants depend on the problem to be solved: single-electron quantum dot (3.65); two-electron centre-of-mass (3.66); two-electron relative-coordinates (3.67).

The boundary conditions are

$$u(0) = 0, \quad u(\infty) = 0. \tag{5.7}$$

We must approximate $\infty$ by a finite value $r_{max}$. This value must be large enough to not influence the results. This is further described in Section 5.2.

After we have discretised the Schrödinger equation using either the finite element method or the finite difference method, we get an eigenvalue problem

$$A\boldsymbol{u} = \epsilon B\boldsymbol{u}, \tag{5.8}$$

where A and B (B=I for FDM) are matrices. We are searching for the eigenvalues $\epsilon$ and eigenvectors $u(r)$. Implementation of boundary conditions for eigenvalue problem are more complicated than for a standard linear system, this is discussed in Section 5.1.3. The eigenvalue problems are solved using ARPACK++[18].

### 5.1.1 Finite difference equations

We discretise the grid by

$$r_i = ih \qquad u_i = \psi(r_i) \qquad h = \frac{1}{N-1} \qquad i = 0, 1, ...N-1,$$

where $N$ is the number of points. For the second derivative we have

$$\frac{\partial^2}{\partial r^2} u_i \simeq \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + O(h^2).$$

Using this approximation in Equation (5.3) we get a tridiagonal matrix eigenvalue problem for the interior points and simple boundary conditions

$$-u_{i-1} + u_i\left(2 + Y(ih)h^2\right) - u_{i+1} = \epsilon h^2 u_i, \quad i = 1..N-2, \tag{5.9}$$

$$u_0 = 0, \tag{5.10}$$

$$u_{N-1} = 0. \tag{5.11}$$

For the two-dimensional quantum dot we have for interior points

$$\begin{aligned} -u_{i-1} + u_i\left(2 + \frac{m^2 - \frac{1}{4}}{i^2} + \frac{Kh}{i} + \omega^2 i^2 h^4\right) \\ -u_{i+1} = \epsilon h^2 u_i, \quad i = 1, \ldots, i = N-2. \end{aligned} \tag{5.12}$$

The finite difference method is simple to implement. We build a tridiagonal matrix

$$A = \begin{pmatrix} 2+Y_1 & -1 & 0 & 0 & \ldots & 0 & 0 \\ -1 & 2+Y_2 & -1 & 0 & \ldots & 0 & 0 \\ 0 & -1 & 2+Y_3 & -1 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & 2+Y_{N-3} & -1 \\ 0 & \ldots & \ldots & \ldots & \ldots & -1 & 2+Y_{N-2} \end{pmatrix},$$

where $Y_i = Y(ih)h^2 = \frac{m^2-\frac{1}{4}}{i^2} + \frac{Kh}{i} + \omega^2 i^2 h^4$. The implementation of boundary conditions are described in Section 5.1.3. The eigenvalue problem is

$$A\boldsymbol{u} = \epsilon h^2 \boldsymbol{u}. \tag{5.13}$$

### 5.1.2 Finite element equations

We use the flexible element-by-element formulation of the finite element method as described in Section 4.2.2. Here we give the mathematical calculations, in Section 5.2 we give more details about how the algorithm is implemented in the program.

For the general Equation (5.3), we set up local elements. The second derivative term gives

$$-\frac{d^2u(r)}{dr^2} \to -\sum_{j=1}^{N} \int_0^{r_{max}} N_i(r)\frac{d^2}{dr^2}N_j(r)dru_j = \sum_{j=1}^{N} \int_0^{r_{max}} \frac{dN_i}{dr}\frac{dN_j}{dr}dru_j$$

The element matrix $T^{(e)}$ arising from this term is

$$T_{rs}^{(e)} = \frac{2}{h_e}\int_{-1}^{1} \frac{dN_r}{d\xi}\frac{dN_s}{d\xi}d\xi. \tag{5.14}$$

Here we have omitted the boundary terms arising from the partial integration because of the boundary conditions $u(0) = 0$ and $u(r_{max}) = 0$. For this expression and the following $r, s = 1, \dots n_e$ and $T^{(e)}_{rs}$ are the elements of the $n_e \times n_e$ matrix $T^{(e)}$. The potential term $Y(r)$ gives

$$Y(r)u(r) \to \sum_{j=1}^{N} \int_0^{r_{max}} N_i Y(r) N_j dr, \tag{5.15}$$

which gives another element matrix contribution $Y^{(e)}$

$$Y^{(e)}_{rs} = \frac{h_e}{2} \int_{-1}^{1} N_r Y(r(\xi)) N_s d\xi, \tag{5.16}$$

where we must use $r(\xi)$ as defined in (4.17) for $Y(r(\xi))$. For the global mapping we have $q(e, s) = (n_e - 1)(e - 1) + s$, which gives the positions of the global nodes as

$$\begin{aligned} r^q &= r_{min} + (q-1)\frac{h}{n_e - 1} \\ &= r_{min} + (e-1)h + \frac{s-1}{n_e - 1}h, \\ q &= 1, \dots, N, \quad s = 1, \dots, n_e, \quad e = 1, \dots M. \end{aligned} \tag{5.17}$$

Knowing this expression we calculate $r(\xi)$ as in (4.17). On the right side we have

$$\epsilon u(r) \to \epsilon \sum_{j=1}^{N} \int_0^{r_{max}} N_i N_j dr, \tag{5.18}$$

which also gives an element matrix $M^{(e)}$

$$\epsilon M^{(e)}_{rs} = \epsilon \frac{h_e}{2} \int_{-1}^{1} N_r N_s d\xi. \tag{5.19}$$

The global matrices are built by adding up the element matrices as described in Section 4.2.2. Building the global system we get a generalised eigenvalue problem

$$\begin{aligned} &\sum_{e=1}^{N} \left( T^{(e)} + V^{(e)} \right) \boldsymbol{u} = \epsilon \sum_{e=1}^{N} M^{(e)} \boldsymbol{u} \\ &\left( A\boldsymbol{u} = \epsilon B \boldsymbol{u} \right), \end{aligned} \tag{5.20}$$

where $A$ and $B$ are the global matrices defined by

$$A = \sum_{e=1}^{N} \left( T^{(e)} + V^{(e)} \right), \quad B = \sum_{e=1}^{N} M^{(e)}. \tag{5.21}$$

Boundary conditions and eigenvalue solvers are discussed in the next section. In the one-dimensional case the matrices are banded with bandwidth $2n_e - 1$.

### 5.1.3 Boundary conditions for eigenvalue problems

To impose boundary conditions on an eigenvalue problem is more complicated that for a simple linear system of equations, which arise from a standard discretisation of a differential equation. The generalised eigenvalue problem is

$$Au = \lambda Bu. \tag{5.22}$$

For the finite difference method we have $B = I$, which reduces the problem to a standard eigenvalue problem. We define a projection matrix $P$

$$P : \mathbb{C}^n \to \mathbb{C}^{n-c},$$

where $c$ is the number of boundary conditions and $P$ is a $(n-c) \times n$ matrix which is identical to the identity matrix $I_n$ with row number $j$ removed for boundary nodes $x_j$. The transformation due to $P$ on a vector and a matrix is given by

$$\begin{aligned} v &= Pu, \\ \tilde{X} &= PXP^T. \end{aligned}$$

The effect of this transformation is that the boundary nodes are removed from the vector $u$ to form $v$, and the boundary rows/columns are removed from $X$. The reduced eigenvalue problem is

$$\tilde{A}v = \lambda \tilde{B}v. \tag{5.23}$$

This derivation is taken from [20].

For the boundary conditions in our problem we remove the first and last columns/row from the matrices, giving a $(N-2)\times(N-2)$ eigenvalue problem. If we want to use the full vector we simply add a zero entry before and after the calculated eigenvector, the missing eigenvalues have the value $\lambda = 0$.

For the finite difference method this is simple to implement when we set up the matrix because we can omit the boundary nodes from the loop and directly set up a $(N-2) \times (N-2)$ matrix. For the finite element method it is more complicated. We first build the full global matrices and then remove the rows/columns from the matrices.

## 5.2 Program

The program for solving the quantum dot equations is built to solve the general time-independent radial equation (5.3). When we use this general equation we have generalised the method to any spherically symmetric potential $V(r)$, we can also use the same equation to solve the equations for the two-electron quantum dot as described in Chapter 3.

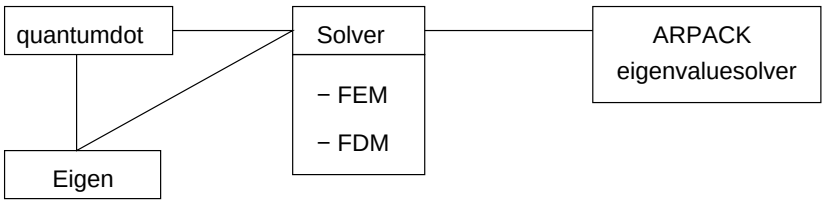


Figure 5.1: Class diagram for the quantum dot solver.

The program is implemented in C++ using object-oriented programming. Here we give some details about the program, see Appendix B for the full source code. A class diagram is drawn in Figure 5.1.

The main part of the code is the `Solver` class and its subclasses `FDM` and `FEM`. Here the equations from Sections 5.1.1 and 5.1.2 are implemented, and an ARPACK++ eigenvalue solver is called to solve the resulting eigenvalue problem. By using subclasses we get a similar interface to the two methods, the only difference is the constructors. The most simple solver may be called as:

```
FEM quantumdot(M, n_e, h);
//or FDM quantumdot(N, h);
quantumdot.set_potential(dim, qn, omega2, K);
quantumdot.solve(num);
quantumdot.solultion.show_eigenvalues();
```

We give more details about the solvers in Section 5.2.1. The arguments for these functions are given in Table 5.1. For the finite element method basis functions for linear and quadratic elements are implemented; n_e can take the values (2,3). Higher order basis functions may also be implemented, see 5.2.3. The `Solver` returns the eigenstates $(\epsilon_\lambda, \boldsymbol{u}_\lambda)$ in an `Eigenstates` object.

The class `Eigenstates`, stores all the eigenstates of a system, it is also responsible manipulating the eigenstates, and provide output. Because we solve for $\epsilon$, we have a function which calculates $E = \frac{1}{2}\epsilon - \frac{1}{2}Bm$; `Eigen::scale_shift(...)`. We also provide functions for normalising and orthogonalising the eigenvec-

| | |
|---|---|
| **constructor** | |
| M | number of elements |
| n_e | number of nodes per element |
| N | number of nodes |
| h | step length |
| **set_potential** | |
| dim | dimensions (2,3) |
| qn | quantum number (m in 2D, l in 3D) |
| $\omega^2$ | scaled oscillator frequency |
| K | strength of electron-electron interaction term |
| **solve** | |
| num | number of eigenvalues to be found |

Table 5.1: Input variable for the `solver` class.

tors. Some of the available functions are given in Table 5.2. A simple `matrix` class has also been implemented.

| | |
|---|---|
| `show_eigenvalues()` | prints the eigenvalues to screen |
| `print(filename, m, omega)` | print to file for the matlab script on file.m |
| `scale(scale)`, | scales the eigenvalues ($E = \frac{1}{2}\epsilon$) |
| `shift(shift)`, | shifts the eigenvalues ($E = \epsilon - \frac{1}{2}Bm$) |
| `scale_shift(scale, shift)` | scales/shifts the eigenvalues ($E = \frac{1}{2}\epsilon - \frac{1}{2}Bm$) |
| `normalise()` | normalise all the eigenvectors |
| `orthogonalise_set()` | orthonormalise the eigenvectors |

Table 5.2: Available functions in the `Eigenstates` class. The functions manipulate the set of eigenstates and print information. These functions are called in the `quantumdot` class.

The class `quantumdot` is a code where we call the `Solver`. As explained earlier the solver can be called using only the small code bit above, but here we have implemented some more features: input from file, variation over $r_{max}$, etc. More information about this class is given in Section 5.2.2.

### 5.2.1 The `Solver` class

The general structure of the `Solver` classes `FDM` and `FEM` is:

- Constructors: Parameters are set from the arguments, matrices and pointers are initialised, for `FEM` the integration points and weights are also set.

- `set_potential`: The function $Y(r)$ is set.
- `make_system`: We build the matrices $A$ ( and $B$) with boundary conditions.
- `solve_system`: We call the ARPACK++ solvers.

For simplicity we have set up a function `Solver::solve(){make_system(); solve_system();}`. We focus on the `make_system()` function. For the `FEM` class we have:

```
void FEM::make system(){
  for(int e=1; e<=M; e++){ //loop over all elements
    //build the element matrices
    calc_element(e, element_left, element_right);
     for(int i=0; i<n_e; i++){
      for(int j=0; j<n_e; j++){
        //fill the global matrix
        left[(n_e-1)*(e-1)+i][(n_e-1)*(e-1)+j]+=
            element_left[i][j];
        right[(n_e-1)*(e-1)+i][(n_e-1)*(e-1)+j]+=
            element_right[i][j];
      }
    }
  }
bc();
}
```

This functions loops over all elements, calculates the element matrices and adds them to the global system. The calculation of the element matrices is done by:

```
void FEM::calc_element(int e, matrix &eleft, matrix &
    eright){
  for(int i=0; i<n_e; i++){
    for(int j=0; j<n_e; j++){
      for(int k=0; k<int_N; k++){
        M_ij=h/2.0*N_i(i,xi[k])*N_i(j,xi[k]);
        int1[k]=M_ij*potential[e-1][k]+ 2.0/h*dN_i(i,xi
            [k])*dN_i(j,xi[k]);
        int2[k]=M_ij;
      }
      eleft[i][j]=integrate(int1);
```

```
        eright[i][j]=integrate(int2);
      }
    }
}
```

The integration is done by Gaussian quadrature, using Legendre polynomials, see Appendix A.3. The `integrate` function takes in an array of function values $f(x_i)$ and multiplies them by a predefined set of weights. In the `FDM` class building the matrix is very simple:

```
void FDM::make_system(){
  for(int i=2; i<N; i++){
    bm[i-1][i-1]=2 +potential[i-1];
    bm[i-1][i]=bm[i-1][i-2]=-1;
  }
  bm[0][0]=2 +potential[0];
  bm[N-1][N-1]=2 +potential[N-1];
  bm[0][1]= bm[N-1][N-2]=-1;
```

### ARPACK++ eigenvalue solver

In `solve_system` the eigenvalue problems are solved using the library ARPACK++, which is a C++ extension of ARPACK [18]. Because ARPACK++ comes with a suitable banded matrix class we pass our matrices to this class and call the eigenvalue solver. Below we show an example for a standard eigenvalue solver

```
//tridiagonal matrix -> arpack structure
double* a_array=bm.get_array_symb(N, 2);
ARbdSymMatrix<double> amat(N, n_e-1, a_array);

//eigenvalue solver object: nev=number of eigenvalues,
    ''SM'' = search for smallest eigenvalues
ARluSymStdEig<double > eigensolver1(nev, amat, "SM",
   0, 0.0, 0, resid , true);

//solve by for example
eigensolver.FindEigenvectors();
```

The variable `nev` is important, if this value is to low, we may not find any eigenstates. In our program this is checked and `nev` is increased automatically.

| Name | Description | Default | Allowed values |
|---|---|---|---|
| type | Which solver is used | 2 | 1=FDM, 2=FEM |
| problem | change variables to fit problem | 0 | 1=single, 2=relative, 3=com, 4=hydrogen, 0=no change |
| dim | dimension of the problem | 2 | 2, 3 |
| omega | harmonic oscillator frequency | 1 | omega $> 0$ |
| B | strength of magnetic field | 0 | $B \geq 0$ |
| m | quantum number m | 1 | (0, 1, 2, ...) |
| l | quantum number l | 1 | (0, 1, 2, ...) |
| rmax | initial value of $r_{max}$ | 0 | rmax>0 |
| h | step size/element size | 0.1 | h>0 |
| n_e | number of nodes per element | 2 | 2,3 |
| num_eigen | number of eigenstates | 10 | |
| all | calculate a full set eigenstates | false | true, false |
| rich | Richardson extrapolation order | 0 | 0, 1, ... |
| maxit | max. iterations over $r_{max}$ | 10 | 0, 1, ... |

Table 5.3: Input read from the `quantumdot` class that may be given in qd.inp. The default value is chosen if no value is specified. Note that input parameters should be given in atomic units.

### 5.2.2 The `quantumdot` class

The `quantumdot` class is not necessary to call the `Solver`, but adds some new features which we will discuss in this section. From this class we call the functions of the `Eigenstates` class given in Table 5.2. After a set of eigenstates is obtained using the `Solver`, the function `orthogonalise_set()` is called to make the eigenvectors othogonal and normalised. To calculate $E$ from Equation (3.63) we call `solution.scale_shift(0.5, 0.5*B*m)`. We also print information to screen and to file. In particular we have set up a function for printing table to latex which is used in Chapter 6.

In the `initialise()` function we read the variables from the file *qd.inp* using the configuration file manager [21]. In Table 5.3 we provide a list of variables and their default values.

The main function in this class is the loop over $r_{max}$: `rm_loop(...)`. In this function we increase $r_{max}$ and calculate a new set of eigenvalues for each value of $r_{max}$. We start with an initial value and increase $r_{max}$ until the difference a number of the lowest eigenvalues is lower than a set limit. This is to ensure that the eigenvalues are not influenced by the value of $r_{max}$, which should be $\infty$, but is cut off in the computations. We refer to Appendix B for

the code.

We have also implemented the Richardson extrapolation as described in Section 4.1.1. As the number of eigenvalues increases when $h$ decreases we extrapolate on a fixed number of eigenvalues. Equation 4.7 is implemented as:

```
Eigenstates Rich(int j, Eigenstates a, Eigenstates b)
  {
  Eigenstates out(num, 0 ,1);
  for(int i=0; i<num; i++){
    out.eigenvalue[i]=(pow(4.0, j)*a.eigenvalue[i]-b.
        eigenvalue[i])/(pow(4.0,j)-1);
  }
  return out;
}
```

This functions calculates $T_m^k$ in Equation (4.7) with $T_{m-1}^{(k+1)}$ (identified as `Eigenstates a`) and $T_{m-1}^{(k)}$ (identified as `Eigenstates b`). The order of extrapolation $m$ is given as $j$.

### 5.2.3 Improvements to the program

Here give som guidelines in expanding the program. We focus on the implementation of the finite element solver as the finite difference case is very simple.

#### Higher order basis functions

We have only implemented linear and quadratic basis functions in the program. To implement higher order basis functions the expressions for $N_i$ need to be calculated. This can be done using the property $N_i(x_j) = \delta_{ij}$, see Section 4.2.3. In the program only small changes need to be made to the functions `N_i(...)` and `dN_i(...)`, where the basis functions and its derivatives are calculated, see Appendix B for the code. A small change in the constructor should also be made allowing higher values of $n_e$. As the order of the basis functions increase there may be a need to implement better numerical integration rules.

#### Potentials

The program can be used for any spherically symmetric potential $V(r)$ or a two-particle potential which may be transformed as described in Section

3.4: $V(r_1, r_2) \rightarrow V(r) + V(R)$. The program is set up to handle potentials of the form $V(r) = Ar^2 + \frac{B}{r}$, but by changing the `set_potential` function this potential may have another form. Note that this functionality must also be added to the `FDM` class in the same way.

### Uneven step length

We may choose to vary the size of the elements $h \rightarrow h_e$. To implement this we need to calculate the node positions $x^q$. An array of element sizes $h_e$ must also be implemented. This improvement can be used to improve the accuracy close to $r = 0$ where the wave functions change more rapidly. To implement this the uneven step length must also be implemented in class `Eigenstates` for correct normalisation.

### General time optimisation

The code could be more optimised in order to run faster, however the most time consuming part is the eigenvalue solver, which was not made as a part of this thesis.

### Parallelisation

The parallelisation of a program like this is a complicated task, we must first partition the data and then use a parallel eigenvalue solver. For the parallel solver we tried to use P_ARPACK [18], which seemed to be a good solver, however we were not able to install the library. For the first part we could use the `FEM` and `FDM` class as they are implemented with a few changes. To use the sequential code to build local matrices, we must first divide the elements (or nodes for the finite difference method), and set a local $r_{min}$ value. For $P$ processors we can define the local number of elements `m_i`. If we have $M$ total element to divide among $P$ processors we can divide the elements by the code:

```
int m_i=M/P;
int rest=M%P;
if(my_rank>0 && my_rank<=rest) m_i++;
```

The value $r_{min}$ must also be calculated, this can be dine by calculating $r_{max} = h * m_i$ on processor $i$ and passing it to $r_{min}$ on processor $i + 1$. Then we call the solvers as before to build local matrices. Changes must also be made to the structure of class `eigenstates`, and we need to implement changes to the linear algebra operations as explained in Section 4.3.

## 5.3 Implementation of time evolution

The Hamiltonian is split into a time-dependent part and a time-independent part

$$H = H_0 + H_1(t). \tag{5.24}$$

For $H_0$ we calculate a full set of orthogonal eigenvectors $\boldsymbol{u}$ and corresponding eigenvalues $E$ using the radial solver. Using these eigenfunctions we define an eigenvector basis as we explained in Section 4.4. We use the Blanes-Moan method, see Equations (4.60)-(4.65) in Section 4.4.

We have set up the time evolution in a matlab script. The script reads the solutions for $H_0$ from file and sets up the eigenvector matrix $P$ and a diagonal matrix $E$ of the eigenvalues. The step length $\Delta t$ and total time $T$ must be given.

We must also set an initial state $\boldsymbol{d}(0) = P^T\boldsymbol{c}(0)$ given in the eigenvector basis. Here $\boldsymbol{c}$ is the state vector in coordinate basis and $\boldsymbol{d}$ is the state vector in the eigenvector basis. The time-independent Hamiltonian $H_0$ gives the diagonal matrix $E$ in the eigenvector basis.

We use the function $H_1(t) = W\sin(ft)$ as the time-dependent perturbation and calculate the integrals analytically for now. The constants $f$ and $W$ must also be given, if $W = 0$ there is no time perturbation and we can compare to analytic results. The analytic expressions for the integrals are

$$\int \sin(ft)dt = -\frac{1}{f}\cos(ft), \tag{5.25}$$

$$\int t\sin(ft)dt = \frac{1}{f^2}\left[\sin(ft) - tf\cos(ft)\right]. \tag{5.26}$$

For a spatially independent $H_1(t)$, $H^{(k)}$ is just the identity matrix times a scalar which is calculated from the integrals above. If $H_1(t)$ also depends on $r$ we must transform it in the same basis, and we get a matrix.

For each time step we calculate $c(t+\Delta t) = U(t+\Delta t, t)c(t)$ using Equation (4.60). To calculate the expression $\boldsymbol{w} = e^M\boldsymbol{u}$, where M is a matrix, we use the function `expv` from Expokit [22] in matlab.

# Chapter 6

# Results of numerical simulations

We begin by studying the single-electron quantum dot in two and three dimensions. We compare the results to known analytic results, in order to verify our solver. Then we move to the relative coordinates equation of the two-electron case and study the special analytic solutions from References [10] and [11].

## 6.1 Single electron quantum dot

In Chapter 3 we simplified the single particle equation to

$$\text{2D:} \quad -\frac{d^2u}{dr^2} + \left[\frac{m^2 - \frac{1}{4}}{r^2} + \Omega^2 r^2 + \right] u = \epsilon u, \tag{6.1}$$

$$\text{3D:} \quad -\frac{d^2u}{dr^2} + \left[\frac{l(l+1)}{r^2} + \Omega^2 r^2 + \right] u = \epsilon u. \tag{6.2}$$

This is the equation for both the single particle equation and the centre-of-mass equation for the two-particle problem (when the constants are scaled as in Equation (3.66)). However the scaling is not important for the numerical algorithm. The analytic solutions are given in Section 3.3. In two dimemensions they are

$$\frac{\epsilon}{2\Omega} = (|m| + 1 + 2n), \tag{6.3}$$

$$u(r) = \sqrt{\frac{n!}{\pi(m+n)!}}\omega^{(|m|+1)/2} r^{|m|+\frac{1}{2}} e^{-\frac{1}{2}\Omega r^2} L_n^{|m|}(\Omega r^2). \tag{6.4}$$

In three dimensions the solutions are

$$\frac{\epsilon}{2\Omega} = (2n + l + \frac{3}{2}), \tag{6.5}$$

$$u(r) = \sqrt{\frac{2^{n+l+2}n!}{\sqrt{\pi}(2n+2l+1)!!}}\omega^{(l+3/2)/2}r^{l+1}e^{-\omega r^2/2}L_n^{l+\frac{1}{2}}(\omega r^2). \tag{6.6}$$

For the simulations we only use positive values for $m$ because Equation (6.1) is only dependent on $|m|$. First we study the dependency on the variable $r_{max}$.

### 6.1.1 Dependence on $r_{max}$

The solution domain for Equation (6.1) is $r \in (0, \infty)$. In the numerically simulation we have to cut off this domain to $r \in (0, r_{max})$. We wish to set this cut-off to a value which will not influence our results. For a chosen number of wanted eigenvalues we increase the value of $r_{max}$ until the difference in this set of eigenvalues is small.

We set the variables: ($m = 1$, $\Omega = 1$, $h = 0.05$) and run simulations for both the finite difference method and for the finite element method using linear elements. We vary the number of eigenvalues we request.. The value of $r_{max}$ which was found is given in Table 6.1. From this table we observe that when we increase the number of required eigenvalues, we must also increase $r_{max}$. In Figure 6.1 a plot of the wave functions are given, from this we see that the higher energy wave functions stretch further out towards $r_{max}$. The two methods give approximately the same results for $r_{max}$.

We also do a comparison between two values of $r_{max}$ where we search for all the eigenvalues in the set. The results are given in Table 6.3 and 6.4. We observe that the lowest eigenvalues are the same for both sets, while for higher eigenvalues the results depend on the value of $r_{max}$. When we increase $r_{max}$ we also increase the size of the system $N = \frac{r_{max}}{h} + 1$. Because of this we will never be able to find accurate values for all $N$ eigenvalues, but we can choose the number of accurate eigenvalues we need.

For the oscillator frequency $\Omega$ we do a similar examination of $r_{max}$. The results are given in Table 6.2. From the analytic wave functions in Equation (6.4) we see that the term $e^{-\frac{1}{2}\Omega r^2}$ will make the wave functions stretch further out for lower values of $\Omega$. Another way to explain this is to look directly Equation (6.1). For the single-electron quantum dot we can introduce a new scaling $r' = \sqrt{\Omega}r$. Using a scaling like this we should be able to solve one equation and scale the result to fit for and $\Omega$. From this formula $r_{max}$ should behave as we see in Table 6.2; When $\Omega$ decreases, we must increase $r_{max}$ and then $r'_{max} = \sqrt{\Omega}r_{max}$ stays the same. The variable $h$, $n_e$ do not affect $r_{max}$.

| Number of eigenvalues | $r_{max}$ | |
|---|---|---|
| | FDM | FEM |
| 5 | 8 | 8 |
| 10 | 9 | 9 |
| 20 | 12 | 12 |
| 50 | 17 | 17 |
| 100 | 22 | 23 |
| 200 | 30 | 32 |

Table 6.1: For a given number a wanted eigenvalues, we increase $r_{max}$ until the difference in the set is small and we say that the eigenvalues are independent of $r_{max}$. The final value for $r_{max}$ is given. These simulations are done for the single-electron case in two-dimensions, using both the finite element method (linear basis functions) and the finite difference method. We have set the other variables to $m = 1$, $h = 0.05$, $\Omega = 1$.

| $\Omega$ | $r_{max}$ | |
|---|---|---|
| | FDM | FEM |
| 5 | 5 | 5 |
| 2 | 7 | 7 |
| 1 | 9 | 9 |
| 0.5 | 12 | 12 |
| 0.25 | 17 | 17 |
| 0.1 | 25 | 25 |

Table 6.2: For different values of the oscillator frequency $\Omega$ we calculate the value of $r_{max}$ required for the 10 lowest eigenvalues to be independent of $r_{max}$. Simulations are run using the finite element method, with linear elements for the single-electron problem intwo-dimensions. The other variables are $m = 1$ and $h = 0.05$.

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 2.0010167 | 2 | 5.08363586e-04 |
| 4.0030369 | 4 | 7.59216691e-04 |
| 6.0062741 | 6 | 1.04568301e-03 |
| 8.010737 | 8 | 1.34212308e-03 |
| 10.016429 | 10 | 1.64292139e-03 |
| | ⋮ | |
| 64.13828 | 60 | 6.89713358e-02 |
| 67.186421 | 62 | 8.36519481e-02 |
| 70.357684 | 64 | 9.93388138e-02 |
| | ⋮ | |
| 2413.2751 | 394 | 5.12506380e+00 |
| 2421.611 | 396 | 5.11517936e+00 |
| 2433.0098 | 398 | 5.11309004e+00 |

Table 6.3: Selected eigenvalues of the single-electron quantum dot in two dimensions. Calculated using the finite element method with linear elements and variables $m = 1$, $h = 0.05$, $r_{max} = 10$ and $\Omega = 1$.

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 2.0010167 | 2 | 5.08363587e-04 |
| 4.0030369 | 4 | 7.59216691e-04 |
| 6.0062741 | 6 | 1.04568301e-03 |
| 8.010737 | 8 | 1.34212308e-03 |
| 10.016429 | 10 | 1.64292139e-03 |
| | ⋮ | |
| 60.557721 | 60 | 9.29534841e-03 |
| 62.595228 | 62 | 9.60045048e-03 |
| 64.633945 | 64 | 9.90538985e-03 |
| | ⋮ | |
| 461.0907 | 394 | 1.70280971e-01 |
| 464.95678 | 396 | 1.74133274e-01 |
| 468.85346 | 398 | 1.78023771e-01 |
| | ⋮ | |

Table 6.4: Selected eigenvalues of the single-electron quantum dot in two dimensions. Calculated using the finite element method with linear elements and variables $m = 1$, $h = 0.05$, $r_{max} = 25$ and $\Omega = 1$.

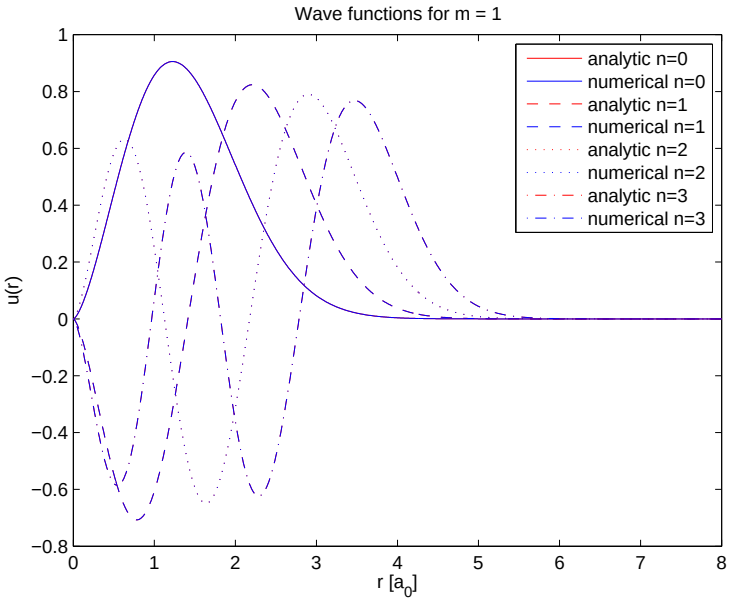


Figure 6.1: Radial wave functions $u(r) = \sqrt{r}R(r)$ for the single-electron quantum dot with $m = 1$ in two dimensions. Calculated using the finite element method with linear basis functions and $h = 0.0125$, $r_{max} = 8$, $\Omega = 1$.

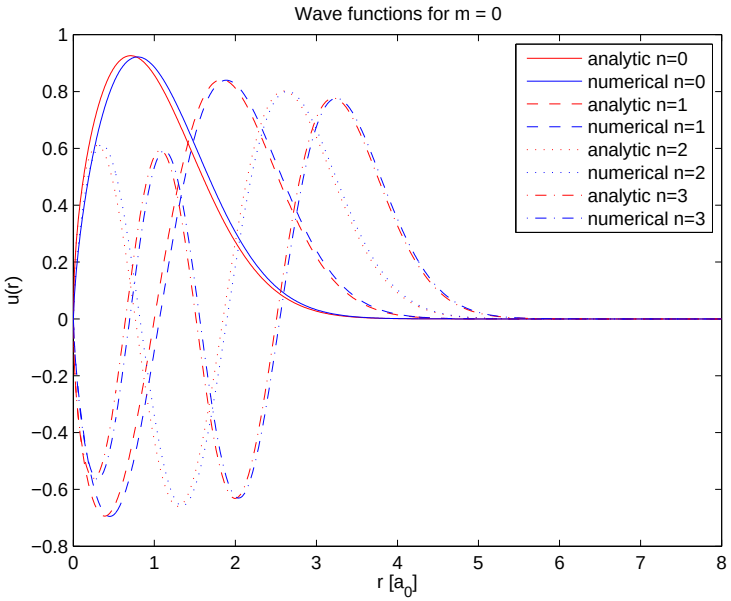


Figure 6.2: Radial wave functions $u(r) = \sqrt{r}R(r)$ for the single-electron quantum dot with $m = 0$ in two dimensions. Calculated using the finite element method with linear basis functions and $h = 0.0125$, $r_{max} = 8$, $\Omega = 1$.

### 6.1.2 Analysis of results and methods

We begin by studying the two-dimensional case. For $m = 0$ and $m = 1$ we provide a list of eigenvalues in Tables 6.6 and 6.5. The four lowest eigenfunctions are given in Figures 6.2 and 6.1. For $m = 1$ the results in Table 6.5 show that all methods give good results, we also see that the wave functions fit perfectly in Figure 6.1. In fact for $|m| > 0$ we always have good results. We also observe that the quadratic elements give the best results as we would expect. However for the special value $m = 0$ we encounter problems. From Table 6.6 and Figure 6.2 we see that the results are poor. We have the same problem for both the finite difference method and the finite element method. In particular we have the strange result that the results are (almost) not improved when moving from linear basis functions to quadratic basis functions. Increasing the step length $h$ does not help this problem much either. We will discuss this problem soon, first we study the three-dimensional equation.

The results for the single-electron case in three dimensions is given for $l = 0$ in Table 6.7 and Figure 6.3. For $l = 5$ they are given in Table 6.8 and Figure 6.4. For all values of $l$ we have good results, actually the relative error is improved for the quadratic elements compared to the two-dimensional case.

We now focus on the special case when $\mathbf{m = 0}$. Because we know the analytic results for this problem we can study the analytic wave function, see Equation (6.4). From this expression we have $u(r) \sim r^{|m|+1/2}$ and $u'(r) \sim r^{|m|-1/2}$. When $m = 0$, we see that $u'(r) \sim r^{-1/2} \to \infty$ when $r \to 0$. Because of this property in the wave function we cannot expect our methods to converge. For the three dimensional case we have $u'(r) \sim r^l$, which does not cause any problems.

We now focus on the other cases where we do not have this problem. For the finite difference method and the finite element method with linear elements the error in the eigenvalues are of the same order, while for quadratic elements we have a better approximations as we would expect.We observe that the relative error increases for higher energy eigenvalues, as can be from various Tables.This also contributes to the fact that we will not find a set of $N$ accurate eigenvalues. However for a quantum system the lower energy eigenstates are most important, and in the time evolution they have the largest contribution ($e^{-iE_k t}$).

For the eigenvectors we sometimes encounter the problem they they have the opposite sign of the analytic eigenvectors. However for an eigenvalue problem, if $\boldsymbol{u}$ is an eigenvector then so is $-\boldsymbol{u}$.

For the finite element method $h$ is the element size, so for quadratic elements the distance between to nodes is actually $h/2$. You may wonder if this is the source of the improved accuracy. To study this we run two

**Finite difference method**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 1.9999304 | 2 | 3.47758683e-05 |
| 3.999798 | 4 | 5.04927300e-05 |
| 5.9995893 | 6 | 6.84482795e-05 |
| 7.9993037 | 8 | 8.70403792e-05 |
| 9.9989408 | 10 | 1.05918082e-04 |
| 11.998501 | 12 | 1.24954233e-04 |
| 13.997983 | 14 | 1.44089955e-04 |
| 15.997387 | 16 | 1.63292947e-04 |
| 17.996715 | 18 | 1.82527496e-04 |
| 19.99597 | 20 | 2.01487059e-04 |
| 21.995237 | 22 | 2.16520650e-04 |
| 23.995294 | 24 | 1.96085047e-04 |

**Finite element method, linear basis functions**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 2.0000738 | 2 | 3.68779263e-05 |
| 4.0002104 | 4 | 5.25921498e-05 |
| 6.0004233 | 6 | 7.05436101e-05 |
| 8.000713 | 8 | 8.91300995e-05 |
| 10.00108 | 10 | 1.08000658e-04 |
| 12.001524 | 12 | 1.27028126e-04 |
| 14.002046 | 14 | 1.46153668e-04 |
| 16.002646 | 16 | 1.65346374e-04 |
| 18.003323 | 18 | 1.84604560e-04 |
| 20.004084 | 20 | 2.04213976e-04 |
| 22.005013 | 22 | 2.27854615e-04 |
| 24.006899 | 24 | 2.87477627e-04 |

**Finite element method, quadratic basis functions**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 2.0000014 | 2 | 7.01871725e-07 |
| 4.0000028 | 4 | 7.02921632e-07 |
| 6.0000042 | 6 | 7.04262228e-07 |
| 8.0000056 | 8 | 7.05923273e-07 |
| 10.000007 | 10 | 7.07912093e-07 |
| 12.000009 | 12 | 7.10229793e-07 |
| 14.00001 | 14 | 7.12900898e-07 |
| 16.000011 | 16 | 7.16620426e-07 |
| 18.000013 | 18 | 7.38405358e-07 |
| 20.000021 | 20 | 1.07439392e-06 |
| 22.000119 | 22 | 5.38998160e-06 |
| 24.00109 | 24 | 4.54314891e-05 |

Table 6.5: We list the lowest eigenvalues of the single-electron quantum dot in two dimensions. Calculated using both the finite difference method and the finite element method. The variable in these simulations are $\mathbf{m} = \mathbf{1}$, $\Omega = 1$, $h = 0.0125$ and $r_{max} = 10$.

**Finite difference method**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 1.1450401 | 1 | 1.45040144e-01 |
| 3.1552631 | 3 | 5.17543540e-02 |
| 5.1609823 | 5 | 3.21964607e-02 |
| 7.1649492 | 7 | 2.35641675e-02 |
| 9.1679422 | 9 | 1.86602497e-02 |
| 11.170294 | 11 | 1.54812904e-02 |
| 13.172179 | 13 | 1.32445427e-02 |
| 15.1737 | 15 | 1.15799859e-02 |
| 17.174923 | 17 | 1.02895684e-02 |
| 19.175895 | 19 | 9.25761816e-03 |
| 21.176678 | 21 | 8.41324150e-03 |
| 23.177626 | 23 | 7.72285566e-03 |

**Finite element method, linear basis functions**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 1.1235395 | 1 | 1.23539516e-01 |
| 3.1312205 | 3 | 4.37401785e-02 |
| 5.1356745 | 5 | 2.71349045e-02 |
| 7.1389755 | 7 | 1.98536484e-02 |
| 9.1417016 | 9 | 1.57446263e-02 |
| 11.144098 | 11 | 1.30997947e-02 |
| 13.146291 | 13 | 1.12531734e-02 |
| 15.148357 | 15 | 9.89048040e-03 |
| 17.150344 | 17 | 8.84374927e-03 |
| 19.152285 | 19 | 8.01501981e-03 |
| 21.154234 | 21 | 7.34449168e-03 |
| 23.156529 | 23 | 6.80563029e-03 |

**Finite element method, quadratic basis functions**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 1.1055834 | 1 | 1.05583420e-01 |
| 3.1111093 | 3 | 3.70364246e-02 |
| 5.1141641 | 5 | 2.28328200e-02 |
| 7.1163118 | 7 | 1.66159786e-02 |
| 9.1179824 | 9 | 1.31091550e-02 |
| 11.119356 | 11 | 1.08505884e-02 |
| 13.120528 | 13 | 9.27135242e-03 |
| 15.121551 | 15 | 8.10337047e-03 |
| 17.12246 | 17 | 7.20355747e-03 |
| 19.123283 | 19 | 6.48857029e-03 |
| 21.124062 | 21 | 5.90772829e-03 |
| 23.125122 | 23 | 5.44007452e-03 |

Table 6.6: We list the lowest eigenvalues of the single-electron quantum dot in two dimensions. Calculated using both the finite difference method and the finite element method. The variable in these simulations are $\mathbf{m} = \mathbf{0}$, $\Omega = 1$, $h = 0.0125$ and $r_{max} = 10$.

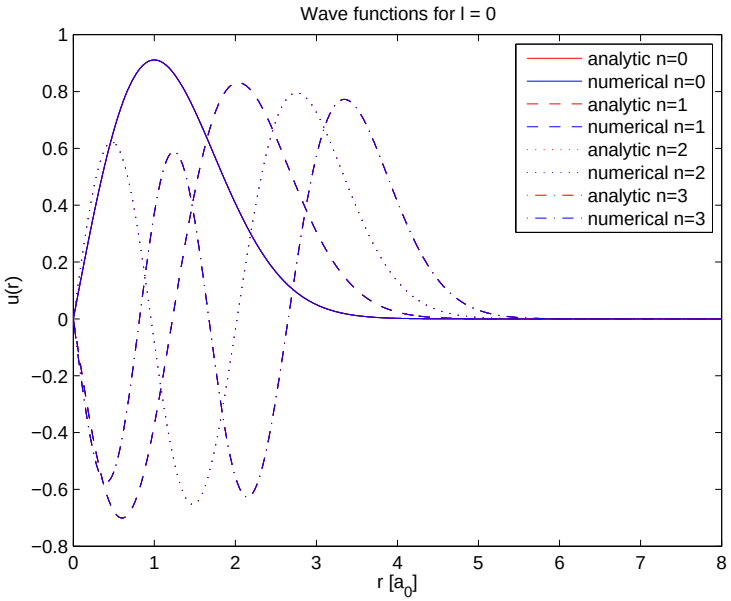


Figure 6.3: Radial wave functions $u(r) = rR(r)$ for the single electron quantum dot with $l = 0$ in three dimensions. Calculated using the finite element method with linear basis functions and $h = 0.0125$, $r_{max} = 8$, $\Omega = 1$.

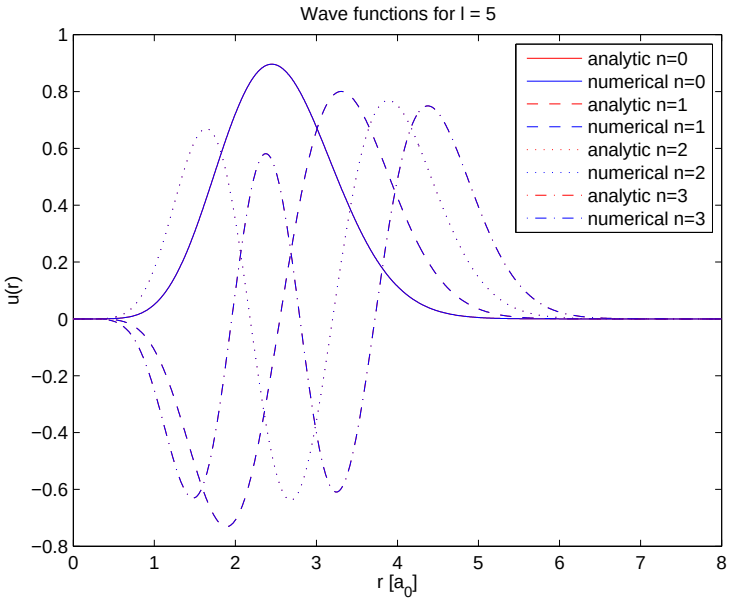


Figure 6.4: Radial wave functions $u(r) = rR(r)$ for the single electron quantum dot with $l = 5$ in three dimensions. Calculated using the finite element method with linear basis functions and $h = 0.0125$, $r_{max} = 8$, $\Omega = 1$.

**Finite difference method**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 1.4999756 | 1.5 | 1.62763277e-05 |
| 3.4998779 | 3.5 | 3.48784722e-05 |
| 5.4997021 | 5.5 | 5.41577864e-05 |
| 7.4994482 | 7.5 | 7.35731453e-05 |
| 9.4991161 | 9.5 | 9.30391081e-05 |
| 11.498706 | 11.5 | 1.12529671e-04 |
| 13.498218 | 13.5 | 1.32034232e-04 |
| 15.497651 | 15.5 | 1.51547364e-04 |
| 17.497006 | 17.5 | 1.71058302e-04 |
| 19.496287 | 19.5 | 1.90414747e-04 |
| 21.495537 | 21.5 | 2.07589502e-04 |
| 23.495220 | 23.5 | 2.03391544e-04 |

**Finite element method, linear basis functions**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 1.5000244 | 1.5 | 1.62758245e-05 |
| 3.5001221 | 3.5 | 3.48760605e-05 |
| 5.5002978 | 5.5 | 5.41519410e-05 |
| 7.5005517 | 7.5 | 7.35623400e-05 |
| 9.5008837 | 9.5 | 9.30218168e-05 |
| 11.501294 | 11.5 | 1.12504367e-04 |
| 13.501782 | 13.5 | 1.31999406e-04 |
| 15.502348 | 15.5 | 1.51502107e-04 |
| 17.502993 | 17.5 | 1.71017734e-04 |
| 19.503719 | 19.5 | 1.90696789e-04 |
| 21.504571 | 21.5 | 2.12588001e-04 |
| 23.506019 | 23.5 | 2.56126019e-04 |

**Finite element method, quadratic basis functions**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 1.5000000 | 1.5 | 1.47114321e-10 |
| 3.5000000 | 3.5 | 5.72095768e-10 |
| 5.5000000 | 5.5 | 1.33481654e-09 |
| 7.5000000 | 7.5 | 2.43728143e-09 |
| 9.5000000 | 9.5 | 3.87768133e-09 |
| 11.500000 | 11.5 | 5.65745479e-09 |
| 13.500000 | 13.5 | 7.78466582e-09 |
| 15.500000 | 15.5 | 1.05605527e-08 |
| 17.500000 | 17.5 | 2.19973765e-08 |
| 19.500004 | 19.5 | 1.93495943e-07 |
| 21.500055 | 21.5 | 2.56298974e-06 |
| 23.500621 | 23.5 | 2.64429919e-05 |

Table 6.7: We list the lowest eigenvalues of the single-electron quantum dot in three dimensions calculated using both the finite difference method and the finite element method. The variable in these simulations are $l = 0$, $\Omega = 1$, $h = 0.0125$ and $r_{max} = 8$.

**Finite difference method**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 6.4999785 | 6.5 | 3.30079248e-06 |
| 8.4998968 | 8.5 | 1.21388049e-05 |
| 10.499737 | 10.5 | 2.50510850e-05 |
| 12.499499 | 12.5 | 4.00823503e-05 |
| 14.499183 | 14.5 | 5.63560713e-05 |
| 16.498789 | 16.5 | 7.34198565e-05 |
| 18.498317 | 18.5 | 9.09959390e-05 |
| 20.497775 | 20.5 | 1.08551731e-04 |
| 22.497269 | 22.5 | 1.21364539e-04 |
| 24.497791 | 24.5 | 9.01533094e-05 |
| 26.505445 | 26.5 | 2.05464543e-04 |
| 28.543458 | 28.5 | 1.52484023e-03 |

**Finite element method, linear basis functions**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 6.5000215 | 6.5 | 3.30074116e-06 |
| 8.5001032 | 8.5 | 1.21381462e-05 |
| 10.500263 | 10.5 | 2.50487599e-05 |
| 12.500501 | 12.5 | 4.00770768e-05 |
| 14.500817 | 14.5 | 5.63465136e-05 |
| 16.501211 | 16.5 | 7.34063445e-05 |
| 18.501684 | 18.5 | 9.10218285e-05 |
| 20.502243 | 20.5 | 1.09415055e-04 |
| 22.502996 | 22.5 | 1.33140178e-04 |
| 24.504941 | 24.5 | 2.01666020e-04 |
| 26.514235 | 26.5 | 5.37184126e-04 |
| 28.554294 | 28.5 | 1.90506711e-03 |

**Finite element method, quadratic basis functions**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 6.5000000 | 6.5 | 4.62243577e-11 |
| 8.5000000 | 8.5 | 2.31825435e-10 |
| 10.500000 | 10.5 | 6.51294130e-10 |
| 12.500000 | 12.5 | 1.35557300e-09 |
| 14.500000 | 14.5 | 2.39091677e-09 |
| 16.500000 | 16.5 | 4.62762725e-09 |
| 18.500001 | 18.5 | 2.95981328e-08 |
| 20.500009 | 20.5 | 4.54655861e-07 |
| 22.500133 | 22.5 | 5.91820972e-06 |
| 24.501367 | 24.5 | 5.57949383e-05 |
| 26.509841 | 26.5 | 3.71370387e-04 |
| 28.548878 | 28.5 | 1.71500277e-03 |

Table 6.8: We list the lowest eigenvalues of the single-electron quantum dot in three dimensions calculated using both the finite difference method and the finite element method. The variable in these simulations are $l = 5$, $\Omega = 1$, $h = 0.0125$ and $r_{max} = 8$.

simulations. First linear basis functions for $h/$ then quadratic basis functions with $h$. The results are given in Table 6.9. From this we observe that the quadratic elements give a better approximation. In this spesific test the improvement is of the order $h$.

## 6.2 Relative coordinates equation

The relative motion equation for the two-electron quantum dot is

$$\text{2D:} \quad -\frac{d^2u}{dr^2} + \left[\frac{m^2 - \frac{1}{4}}{r^2} + \Omega^2 r^2 + \frac{1}{r}\right] u = \epsilon u, \tag{6.7}$$

$$\text{3D:} \quad -\frac{d^2u}{dr^2} + \left[\frac{l(l+1)}{r^2} + \Omega^2 r^2 +\right] u = 2Eu. \tag{6.8}$$

For this problem we only have limited analytic solutions to compare with, see Section A.2 and References [10] and [11]. We use these to test our method.

For $m = 0$ and $m = 1$ we calculate the eigenvalues for special values of $\Omega$. The results are given in Tables 6.11 and 6.10 along with the analytic solutions from Section A.2. Comparing these results to the analytic ones, we observe the same behaviour as for the single-particle problem; the results are good for $|m| > 0$, but for $m = 0$ we have the same problem. We repeat this analysis for the analytic eigenvalues in three dimensions. The result for $l = 0$ is given in Table 6.12. The results are good also for the three-dimensional case.

**Finite element method, linear basis functions, element size $h$**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 2.0000738 | 2 | 3.68779258e-05 |
| 4.0002104 | 4 | 5.25921499e-05 |
| 6.0004233 | 6 | 7.05436101e-05 |
| 8.0007130 | 8 | 8.91300999e-05 |
| 10.001080 | 10 | 1.08000658e-04 |
| 12.001524 | 12 | 1.27028126e-04 |
| 14.002046 | 14 | 1.46153647e-04 |
| 16.002646 | 16 | 1.65345613e-04 |
| 18.003323 | 18 | 1.84585253e-04 |
| 20.004077 | 20 | 2.03860578e-04 |
| 22.004910 | 22 | 2.23163496e-04 |
| 24.005820 | 24 | 2.42488304e-04 |
| 26.006808 | 26 | 2.61830841e-04 |
| 28.007873 | 28 | 2.81187987e-04 |
| 30.009017 | 30 | 3.00557420e-04 |

**Finite element method, quadratic basis functions, element size $h/2$**

| Numerical | Analytic | Relative error = \|a-n\|/a |
|---|---|---|
| 2.0000056 | 2 | 2.81644890e-06 |
| 4.0000113 | 4 | 2.83224441e-06 |
| 6.0000171 | 6 | 2.85279791e-06 |
| 8.0000230 | 8 | 2.87846745e-06 |
| 10.000029 | 10 | 2.90935622e-06 |
| 12.000035 | 12 | 2.94551250e-06 |
| 14.000042 | 14 | 2.98696317e-06 |
| 16.000049 | 16 | 3.03372603e-06 |
| 18.000056 | 18 | 3.08581283e-06 |
| 20.000063 | 20 | 3.14323227e-06 |
| 22.000071 | 22 | 3.20599036e-06 |
| 24.000079 | 24 | 3.27409180e-06 |
| 26.000087 | 26 | 3.34754030e-06 |
| 28.000096 | 28 | 3.42633824e-06 |
| 30.000105 | 30 | 3.51048782e-06 |

Table 6.9: Comparison of finite element with linear basis functions with $h$ to quadratic basis functions with $h/2$ for the single electron case in two dimensions. We have used $h = 0.0125$. The other variables are kept constant, $m = 1$, $h = 0.0125$, $r_{max}$=10.

| $1/\Omega$ | Analytic $\epsilon/2$ | Numerical $\epsilon/2$ | Relative error | $n'$ |
|---|---|---|---|---|
| 0.600000e+1 | 0.500000e+0 | 0.50000097e+0 | 1.9482e-6 | 0 |
| 0.280000e+2 | 0.142857e+0 | 0.14285717e+0 | 1.9987e-7 | 0 |
| 0.725576e+2 | 0.689107e-1 | 0.68910763e-1 | 9.0844e-7 | 0 |
| 0.744236e+1 | 0.671830e+0 | 0.67183231e+0 | 3.4361e-6 | 1 |
| 0.146604e+3 | 0.409266e-1 | 0.40926587e-1 | 3.1662e-7 | 0 |
| 0.333961e+2 | 0.179662e+0 | 0.17965263e+0 | 5.2167e-5 | 1 |
| 0.257194e+3 | 0.272168e-1 | 0.27216809e-1 | 3.4746e-7 | 0 |
| 0.840644e+2 | 0.832695e-1 | 0.83269499e-1 | 1.2104e-8 | 1 |
| 0.874155e+1 | 0.800773e+0 | 0.80077731e+0 | 5.3849e-6 | 2 |

Table 6.10: Results for the relative motion equation for the two-electron quantum dot in two dimensions with $m = 1$. The numerical simulation is run for specific values of $\Omega$ as given in Table 3.2 in order to find the analytic eigenvalues. For the numerical eigenvalues $n'$ is the order of the eigenvalue. Simulations are run with $h = 0.0125$.

| $1/\Omega$ | Analytic $\epsilon/2$ | Numerical $\epsilon/2$ | Relative error | $n'$ |
|---|---|---|---|---|
| 0.200000e+1 | 0.100000e+1 | 0.10123751e+1 | 0.0124 | 0 |
| 0.120000e+2 | 0.250000e+0 | 0.25023028e+0 | 9.2111e-4 | 0 |
| 0.370880e+2 | 0.107852e+0 | 0.10804700e+1 | 0.0018 | 0 |
| 0.291199e+1 | 0.137363e+1 | 0.13851445e+1 | 0.0084 | 1 |

Table 6.11: Results for the relative motion equation for the two-electron quantum dot in two dimensions with $m = 0$. The numerical simulation is run for specific values of $\Omega$ as given in Table 3.2 in order to find the analytic eigenvalues. For the numerical eigenvalues $n'$ is the order of the eigenvalue. Simulations are run with $h = 0.0125$.

| $1/\omega_r$ | Analytic $\epsilon_r/2$ | Numerical $\epsilon/2$ | Relative error | $n'$ |
|---|---|---|---|---|
| 4.00000 | 0.6250 | 0.62500112e+0 | 1.7973e-6 | 0 |
| 20.0000 | 0.1750 | 0.17500005e+0 | 2.6018e-7 | 0 |
| 54.7386 | 0.0822 | 0.82208891e-1 | 1.0816e-4 | 0 |
| 5.26137 | 0.8553 | 0.85529407e+0 | 6.9324e-6 | 1 |
| 115.299 | 0.0477 | 0.47702062e-1 | 4.3232e-5 | 0 |
| 24.7010 | 0.2227 | 0.22266321e+0 | 1.6521e-4 | 1 |
| 208.803 | 0.0311 | 0.31129841e-1 | 9.5953e-4 | 0 |
| 64.8131 | 0.1003 | 0.10028838e+0 | 1.1590e-4 | 1 |
| 6.38432 | 1.0181 | 0.10181255e+1 | 2.5004e-5 | 2 |

Table 6.12: Results for the relative motion in three dimensions with $l = 0$ for selected values of $\Omega$ as given in Table 3.3. For the numerical eigenvalues $n'$ is the order of the eigenvalue. Simulations are run with $h = 0.0125$.

# Chapter 7

# Concluding remarks

The study in this thesis has been two-fold; it has been a study of the interesting quantum system of two-electrons in a quantum dot, but also a study of the finite element method for solving the time-independent Schrödinger equation.

After giving a background to quantum mechanics we gave an overview of quantum dots, focusing on the mathematical model. Note that this model is just an approximation to a quantum dot. We showed that the one-electron quantum dot can be described by a modified harmonic oscillator, which is a known quantum mechanical problem with analytic solutions. For the two-electron quantum dot we introduced centre-of-mass and relative motion coordinates and showed that the two-particle equation can be rewritten as two independent single particle equations. The centre-of-mass equation is also a modified harmonic oscillator equation, while the relative coordinates equation is more complicated.

The other main topic of this thesis was the use of numerical methods for solving differential equations. We gave a thorough introduction to the finite element method in one dimension and also mentioned the finite difference method. Both methods were implemented in a program for solving the time-independent Schrödinger equation for a spherically symmetric potential. This program were then used to study the one-electron and two-electron quantum dot. We would have liked to parallelise this program as well, however we had big problems making the P_ARPACK [18] library work.

After studying the results in Chapter 6 we see that our program give good results except for the special case when $m = 0$ in two dimensions where we discovered that we have a divergence $u'(r) \sim r^{-1/2} \to \infty$. Compared to the finite difference method the finite element method give results with approximately the same relative error for linear elements, however for quadratic

elements we get a better approximation. It is possible to extend the program with higher order basis functions, see Section 5.2.3. A disadvantage of the finite element method is that we end up with a generalised eigenvalue problem instead of a standard eigenvalue problem. Fortunately the arpack++ library has implemented this. However this may be more time consuming to solve, but this has not been studied in this thesis.

A very interesting problem is the double quantum dot, consisting of two coupled quantum dots. This system is similar to the potential we have studied. Unfortunately for us this problem breaks the spherical symmetry and can not be studied by our model. To study this a two-dimensional (or three-dimensional) solver must be constructed.

# Appendix A

# Mathematical details

## A.1 Analytic solutions of the single-electron harmonic oscillator

Here we derive the analytic solutions for the two-dimensional harmonic oscillator. We begin with Equation (3.27) from Chapter 3

$$-\left[\frac{1}{r}\frac{d}{dr}\left(r\frac{d}{dr}\right)-\frac{m^2}{r^2}\right]R(r)+\left(\omega^2r^2-\epsilon\right)R(r)=0. \tag{A.1}$$

First we study the limits $r\rightarrow 0$ and $r\rightarrow\infty$. For $r\rightarrow 0$ the $m^2/r^2$ term dominates and we have

$$-\left[\frac{1}{r}\frac{d}{dr}\left(r\frac{d}{dr}\right)-\frac{m^2}{r^2}\right]R(r)=0,$$

which has the solution

$$R(r)=r^{|m|}.$$

For $r\rightarrow\infty$ the $r^2$ term dominates

$$-\left[\frac{1}{r}\frac{d}{dr}\left(r\frac{d}{dr}\right)\right]R(r)+\omega^2r^2R(r)=0,$$

and we have the solution

$$R(r)=e^{-\frac{1}{2}\omega r^2/2}.$$

Now we insert a solution of the form

$$R(r)=r^{|m|}e^{-\frac{1}{2}\omega r^2}g(r), \tag{A.2}$$

into (A.1) to get

$$-\frac{d^2g}{dr^2} - \left(2|m| - 2\omega r^2 + 1\right)\frac{1}{r}\frac{dg}{dr} + 2\omega\left(|m| + 1\right)g - \epsilon g = 0.$$

We can simplify further by changing variables to

$$y = \omega r^2,$$

which gives us

$$y\frac{d^2g}{dy^2} + \left(|m| + 1 - y\right)\frac{dg}{dy} - \frac{1}{2}\left(|m| + 1 - \frac{\epsilon}{2\omega}\right)g = 0. \tag{A.3}$$

We can rewrite this as

$$yg'' + \left(|m| + 1 - y\right)g' + \lambda g = 0, \tag{A.4}$$

$$\lambda = -\frac{1}{2}\left(|m| + 1 - \frac{\epsilon}{2\omega}\right). \tag{A.5}$$

This equation is known as the associated Laguerre differential equation. We try a solution of the form

$$g(y) = \sum_{n=0}^{\infty} a_n y^n, \tag{A.6}$$

with the derivatives given by

$$g(y) = \sum_{n=0}^{\infty} n a_n y^{n-1},$$

$$g(y) = \sum_{n=0}^{\infty} n(n-1) a_n y^{n-2}.$$

Inserting this in (A.4) we get

$$y\sum_{n=0}^{\infty} n(n-1)a_n y^{n-2} + \left(|m| + 1 - y\right)\sum_{n=0}^{\infty} n a_n y^{n-1} + \lambda\sum_{n=0}^{\infty} a_n y^n = 0,$$

multiplying by $y$ and shifting terms we have

$$\rightarrow \sum_{n=0}^{\infty} n(n-1)a_n y^n + \left(|m| + 1\right)\sum_{n=0}^{\infty} n a_n y^n - \sum_{n=0}^{\infty} n a_n y^{n+1} + \lambda\sum_{n=0}^{\infty} a_n y^{n+1} = 0.$$

In the first two sums we shift the index $n \to n+1$ (the lower limit can still be set to 0 because this term equals 0) and divide by the entire expression by $y$ to get

$$\sum_{n=0}^{\infty} \left[(n+1)(|m|+1+n)a_{n+1} - (n-\lambda)a_n\right] y^n = 0.$$

From this we can find a recursive relation for $a_{n+1}$ from $a_n$

$$a_{n+1} = \frac{(n-\lambda)}{(n+1)(|m|+1+n)} a_n.$$

We can now write Equation (A.6) as

$$g(y) = \left[1 - \frac{\lambda}{(|m|+1)} y - \frac{\lambda(1-\lambda)}{(|m|+1)(|m|+2)2} y^2 - \ldots\right] a_0. \tag{A.7}$$

For convergence we must requite $\lambda \geq 0$. This expansion is cut off when $n = \lambda$, because all terms after that will be zero. This property is important for the function to be normalisable.We set $\lambda$ in (A.5) to $n$ giving

$$\epsilon = 2\omega(|m|+1+2n).$$

The solution for $g(y)$ is the Laguerre polynomials $L_n^{|m|}$ and radial wave function $R(r)$ is

$$R(r) = r^{|m|} e^{-\frac{1}{2}\omega r^2} L_n^{|m|}(\omega r^2), \tag{A.8}$$

and with normalisation we have

$$R(r) = \sqrt{\frac{2n!}{(|m|+n)!}} \omega^{(|m|+1)/2} r^{|m|} e^{-\frac{1}{2}\omega r^2} L_n^{|m|}(\omega r^2). \tag{A.9}$$

The Rodrigues representation for the associated Laguerre polynomials [23] is

$$L_n^{|m|}(y) = \frac{e^y y^{-|m|}}{n!} \frac{d^n}{dy^n} \left(y^{n+|m|} e^{-y}\right), \tag{A.10}$$

$$= \sum_{j=0}^{n} (-1)^m \frac{(n+|m|)!}{(n-j)!(|m|+j)!j!} x^j, \tag{A.11}$$

and the first four Laguerre polynomials are given in Table A.1.

| n | $L_n^{\lvert m\rvert}(y)$ |
|---|---|
| 0 | $1$ |
| 1 | $-y+\lvert m\rvert+1$ |
| 2 | $\frac{1}{2}\left[y^2-2(\lvert m\rvert+2)y+(\lvert m\rvert+1)(\lvert m\rvert+2)\right]$ |
| 3 | $\frac{1}{6}[-y^3+3(\lvert m\rvert+3)y^2-3(\lvert m\rvert+2)(\lvert m\rvert+3)y$ $+(\lvert m\rvert+1)(\lvert m\rvert+2)(\lvert m\rvert+3)]$ |

Table A.1: The first four associated Laguerre polynomials $L_n^{|m|}$.

## A.2 Particular solutions for the relative motion

For the relative equation in the two-electron quantum dot, there is no general closed form solution. However we can find particular solutions. In Ref. [10] such solutions are derived. It is shown that for particular values of $n$ and $m$ we can find a value for $\omega$ which gives a closed form solution. The criteria for finding such solutions is fulfilled when

$$F(|m|,n,\epsilon'',\omega_r)=0 \text{ and} \tag{A.12}$$
$$\epsilon''=2(|m|+n), \tag{A.13}$$

where $F$ is defined by a recursion relation

$$a_\nu = F(|m|,\nu,\epsilon'',\omega_r)a_0, \tag{A.14}$$
$$a_\nu = \frac{1}{\nu(\nu+2|m|)}\left\{\frac{1}{\sqrt{\omega_r}}a_{\nu-1}+[2(\nu+|m|-1)-\epsilon'']a_{\nu-2}\right\}, \tag{A.15}$$
$$a_1 = \frac{1}{2(|m|+\frac{1}{2})}\frac{1}{\sqrt{\omega}_r}a_0, \quad a_0\neq 0 \tag{A.16}$$

and

$$\epsilon''=\frac{\epsilon_r}{\omega_r}\rightarrow \epsilon_r=2\omega_r(|m|+n). \tag{A.17}$$

Note that the definitions of symbols in this thesis are different from Reference [10]. For $n=2$ we calculate

$$\omega_r=\frac{1}{2(2|m|+1)}. \tag{A.18}$$

Similarly, for $n=3$ we have

$$\omega_r=\frac{1}{4(4|m|+3)}. \tag{A.19}$$

| m | $\omega_r$ | $\epsilon_r$ |
|---|---|---|
| 0 | 1/2 | 2 |
| 1 | 1/6 | 1 |
| 2 | 1/10 | 4/5 |
| 3 | 1/14 | 5/7 |

Table A.2: Calculations of $\omega_r$ and $\epsilon_r$ for n=2. Calculated from Equations (A.17) and (A.18).

| m | $\omega_r$ | $\epsilon_r$ |
|---|---|---|
| 0 | 1/12 | 1/2 |
| 1 | 1/28 | 2/7 |
| 2 | 1/44 | 5/22 |
| 3 | 1/60 | 1/5 |

Table A.3: Calculations of $\omega_r$ and $\epsilon_r$ for n=3. Calculated from Equations (A.17) and (A.18).

In Tables A.2 and A.2 we list some values for $\omega_r$ and $\epsilon_r$ calculated by these equations.

For $m = 0$ and $m = 1$ we list some of the eigenvalues in Table 3.1 and Table 3.2, for an extensive list see Reference[10].

## A.3 Numerical integration

In this section we summarise numerical integration methods [24]. The focus is on Gaussian quadrature methods. Numerical integration formulas approximate the integral by

$$I = \int_a^b f(x)dx \approx \sum_{i=1}^{N} \omega_i f(x_i), \tag{A.20}$$

where N is the number of integration points and $\omega$ the weights. For the trapezoidal rule the weights are

$$\omega : \left\{ \frac{h}{2}, h, h, \ldots, h, \frac{h}{2} \right\},$$

and for the Simpson's rule we have

$$\omega : \left\{ \frac{h}{3}, \frac{4h}{3}, \frac{2h}{3}, \frac{4h}{3}, \ldots, \frac{4h}{3}, \frac{h}{3}, \right\}.$$

These methods are equal step methods, based on Taylor expansions. We may obtain a higher precision if we use Gaussian quadrature where the integration points are also determined. A Gaussian quadrature formula is given by

$$I = \int_a^b f(x) = \int_a^b W(x)g(x) \approx \sum_{i=1}^{N} \omega_i g(x_i),$$

where $W(x)$ is the weight function and $g(x)$ is a smooth function. The weight function are given by an orthogonal polynomial. Gaussian quadrature is known to integrate a polynomial $p$ of degree $2N-1$ exactly. The error is given by[24]

$$\int_a^b W(x)f(x) - \sum_{i=1}^{N} \omega_i f(x_i) = \frac{f^{2N}(\xi)}{(2N)!} \int W(x)[qN(x)]^2 dx,$$

where $qN(x)$ is the orthogonal polynomial and $\xi$ is a number in the interval.

The mesh points $x_i$ are the zeros of the chosen orthogonal polynomial. The weights are determined from the inverse matrix, see for example Ref. [24].

We focus on the Legendre polynomials, which are defined in the interval $[-1, 1]$.The Legendre polynomials have the weighting function $W(x) = 1$. The weights and mesh points for the Legendre polynomial is given in table A.4.

| $n$ | $x_i$ | $\omega_i$ |
|---|---|---|
| 2 | $\pm\frac{1}{3}\sqrt{3}$ | 1 |
| 3 | 0 | $\frac{8}{9}$ |
| | $\pm\frac{1}{5}\sqrt{15}$ | $\frac{5}{9}$ |
| 4 | $\pm\frac{1}{35}\sqrt{525-70\sqrt{30}}$ | $\frac{1}{36}(18+\sqrt{30})$ |
| | $\pm\frac{1}{35}\sqrt{525+70\sqrt{30}}$ | $\frac{1}{36}(18-\sqrt{30})$ |
| 5 | 0 | $\frac{128}{225}$ |
| | $\pm\frac{1}{21}\sqrt{245-14\sqrt{70}}$ | $\frac{1}{900}(322+13\sqrt{70})$ |
| | $\pm\frac{1}{21}\sqrt{245+14\sqrt{70}}$ | $\frac{1}{900}(322-13\sqrt{70})$ |

Table A.4: The weights and grid points for the Gauss-Legendre quadrature. Taken from [25].

# Appendix B

# Source code

In this appendix we give the full source code. For more information about the implementation see Chapter 5. The program can be compiled using a variation of this Makefile.

**Makefile**

```
#MAKEFILE
HEADERDEST := /fys/compphys/include
LAPACKLIB := -llapack

qd: Solver.h FEM.o FDM.o ConfigFile.o quantumdot.cpp
    Eigen.h matrix.h
#       mpicxx
        icc  -o qd quantumdot.cpp FEM.o FDM.o
           ConfigFile.o -O3 lib/libarpack.a lib/
           libarpack++.a -lgfortran lib/libblas.a -I$(
           HEADERDEST) $(LAPACKLIB) -ansi -Wall -wd1572
            -wd981 -wd383 -pg

FDM.o: FDM.cpp Solver.h Eigen.h matrix.h
        g++ -c FDM.cpp -I/site/compphys/include

FEM.o: FEM.cpp matrix.h Eigen.h Solver.h
        g++ -c FEM.cpp -I/site/compphys/include

ConfigFile.o: ConfigFile.cpp ConfigFile.h
        g++ -c ConfigFile.cpp -pg
```

```
clean:
        rm -f qd.x *.o *~ *#
```

## B.1 Main program: class quantumdot

```
#include "FEM.h"
#include "ConfigFile.h"
#include <string>

class quantumdot{

private:
  int M; //Number of elements
  int N; //Number of nodes
  double h; //FEM: size of element, FDM: distance
     between nodes
  bool full_set; //are we searching for a full set
     eigenstates
  int n_eigenvalues; //number of eigenvalues to be
     found
  int n_compare;

  int m, l; //quantum numbers
  double omega_0, omega2, omega; //oscillator frequency
  double B; //magnetic field
  double K; //interaction term: K=1 for relative
     equation

  int dim; //specify 2D or 3D problem
  int type; //specify FDM=1 or FEM=2
  int problem; //specify which general problem to be
     solved !!!

  double r_min;
  double r_max;

  Eigenstates solution; //final solution is stored
```

```
  int n_richardson; //order of the Richardson
     extrapolation
  int max_iterations; //maximum number of iterations in
      rmax

  int n_e, n_int; //number of nodes per element and
     integration points for FEM

  char* path;
  char file[80];
  char run[5];

public:
  quantumdot(){
    initialise(); //read input from file and set
       variables

    //loop over increasing r_max values
    rm_loop(0.0001, max_iterations);
    solution.scale_shift(0.5, -0.5*B*m); //calculates E
       = 0.5eps-0.5Bm

    output_filename();
    single_latex(file);

    if(K==0.0) {
      eigenvalue_report();
    }
    else {
      solution.show_eigenvalues();
    }

    solution.orthogonalise_set(); //orthogonalise
       eigenvectors
    solution.print(file, m, l, omega); //print
       eigenstates to matlab file file.m
  }

private:
  //simple FEM solver
  Eigenstates FEMsolve(bool vec){
```

```
    cout << "FEM: " <<  "r_max = " << r_max << "  , M =
       " << M <<", h = " << h <<endl;
    FEM x(M,n_e,n_int,h);

    if(dim==2) x.set_potential(2,m,omega2,K, r_min);
    else if(dim==3) x.set_potential(3,l,omega2,K, r_min
       );
    x.solve(n_eigenvalues, vec );
    return x.solution;
}

//simple FDM solver
Eigenstates FDMsolve(bool vec){
    cout << "FDM: " <<  "r_max = " << r_max << "  , N =
       " << M+1 << ", h = " << h <<endl;
    FDM x(M+1,h);

    if(dim==2) x.set_potential(2,m,omega2,K);
    else if(dim==3) x.set_potential(3,l,omega2,K);
    x.solve(n_eigenvalues, vec);
    return x.solution;
}

//print a set of eigenvalues, comparing the to
    analytic solutions
void eigenvalue_report(){ //assumes K=0!
    double analytic;
    double numeric;
    double mag = 0.5*B*m;
    double num;
    if(dim==2) num=abs(m)+1;
    else if(dim==3) num=l+1.5;

    cout << "Numerical solution  " << "Analytic
       solution  " << "Relative error |a-n|/a" <<endl;
    if(omega==0) omega=1;
    for(int i=0; i<n_eigenvalues; i++){
      numeric=solution.eigenvalue[i];
      analytic=num*omega-mag;
      cout << fixed << setw(15) << setprecision(8) <<
         numeric;
```

```
      cout << setw(15) << analytic;
      cout << scientific << setw(30) << setprecision(8)
          << abs(analytic-numeric)/analytic;
      cout << endl;
      num+=2;
    }
    cout << "-----------------" <<endl;
  }

  //print a set of eigenvalues, comparing the to
     analytic solutions to file.tex
  void single_latex(char* filein){
    char* ext = ".tex";
    char file[80];
    strcpy(file, filein);
    strcat(file, ext);

    double analytic;
    double numeric;
    double mag = 0.5*B*m;
    double num;
    if(dim==2) num=abs(m)+1;
    else if(dim==3) num=l+1.5;
    //if(omega==0) omega=1; //fjern denne senere

    ofstream ofile;
    ofile.open (file);
    ofile << "\\begin{table}[hbp]" <<endl;
    ofile << "\\centering" << endl;
    ofile << "\\begin{tabular}{ccc}" << endl;
    ofile << "Numerical & Analytic & Relative error = |
       a-n|/a\\\\" <<endl;
    ofile << "\\hline" << endl;
    for(int i=0; i<n_eigenvalues; i++){
      numeric=solution.eigenvalue[i];
      analytic=num*omega-mag;
      ofile << setw(10) << setprecision(8) << numeric
         << " & ";
      ofile << setw(3)  << analytic << " & ";
      ofile << scientific << setw(16) << setprecision
         (8) << abs(analytic-numeric)/analytic;
```

```
      ofile << " \\\\ " <<endl;
      ofile.unsetf ( ios_base::scientific );
      num+=2;
    }
    ofile << "\\end{tabular}" <<endl;
    ofile << "\\caption{ " << "m=" << m << ", h=" << h
       << ", $r_{max}$=" << r_max;
    if(type==1) ofile << " ,FDM, rich = " <<
       n_richardson;
    else ofile <<" ,FEM, $n_e$=" << n_e << ", $n_{int}$
       = " << n_int;
    ofile <<"}" <<endl;
    ofile << "\\label{}" <<endl;
    ofile << "\\end{table}" <<endl;
    ofile.close();
}

//read input from file using configuration manager:
    http://www-personal.umich.edu/~wagnerr/ConfigFile.
    html
void initialise(){
  double temp;
    ConfigFile config( "qd.inp");
    config.readInto( temp, "omegainv", 0.0);
    if(temp!=0) omega_0=1.0/temp;
    else config.readInto( omega_0, "omega", 1.0);
    config.readInto( B, "B", 0.0);
    config.readInto( m, "m", 1);
    config.readInto( l, "l", 1);
    config.readInto( K, "K", 0.0);
    config.readInto( h, "h", .1);
    config.readInto( max_iterations, "maxit", 10);

    //read r_max from file or calculate, set M
    config.readInto( r_min, "r_min", 0.0);
    config.readInto( r_max, "r_max", 0.0);
    if(r_max==0.0) r_max=init_rmax(omega);
   M=(int)ceil(r_max/h);

    config.readInto( dim, "dim", 2);
    if(dim!=3) dim =2;
```

```
    cout << dim <<endl;
    config.readInto( type, "type", 2);
    if(type==2){
      config.readInto( n_e, "n_e", 2);
      if(n_e>3) n_e=3;
      config.readInto( n_int, "n_int", 5);
     N=M*(n_e-1)+1;
    }
    else {
      type=1;
      config.readInto(n_richardson, "rich", 0);
     N=M+1;
    }

    //change constants for problem type
    config.readInto( problem, "problem", 0);
    if(problem==2) relative();
    else if(problem==3) com();
    else if(problem==1) single();
    else problem=0;

    //include the magnetic field in omega
    omega2=omega_0*omega_0+0.25*B*B;
    omega=sqrt(omega2);

    config.readInto(full_set, "all" , false);
    if(!full_set) config.readInto( n_eigenvalues, "
       num_eigenvalues", 10);
    if(full_set) n_eigenvalues=N-2;
    config.readInto( n_compare, "num_eigenvalues", 10);


    //Output info to screen
    cout << "Solving: -u'' + ";
    if(dim==3) cout << "l(l+1)";
    else cout << "(m^2-0.25)";
    cout << "u/r^2 + w^2r^2u";
    if(K!=0.0) cout << "+" << K << " u/r = eps u";
    cout << " = eps u";
    cout <<endl << "w = " << omega_0 << ", B = " << B;
    if(dim==3) cout << " ,l = " << l;
```

```
    cout << "  ,m = " << m <<endl << "Using ";
    if(type==1) cout << "FDM, with step length h = " <<
        h << " Richardson extrapolation order = " <<
       n_richardson <<endl;
    else cout <<"FEM, with element size h = " << h << "
       , using " << n_e << " nodes per element and " <<
        n_int << " integration points"<<endl;

  }

  //set up output file
  void output_filename(){
    path= "../Tekst/Double/";//3d/";
    strcpy(file , path);

//     strcat(file , "r");
//     sprintf(run, "%04.0f" ,r_max);
//     strcat(file , run);
//     strcat(file , "w");
//     sprintf(run, "%g" ,omega);
//     strcat(file , run);

//     strcat(file , "h");
//     sprintf(run, "%g" ,h);
//     strcat(file , run);
    strcat(file , "B");
    sprintf(run, "%g" ,B);
    strcat(file , run);
//     strcat(file , "m");
//     sprintf(run, "%i" ,m);
//     strcat(file , run);
//     strcat(file , "l");
//     sprintf(run, "%i" ,l);
//     strcat(file , run);

    if(type==1) strcat(file , "FDM");
    else {
      strcat(file , "FEM");
      sprintf(run, "%i" ,n_e);
      strcat(file , run);
    }
```

```
  cout << "Output to: " << file << endl;
}

//initialise r_max if not set
int init_rmax(double omega){
  double epsilon=0.001;
  if(omega==0) return (int)-log(epsilon)+1;
  else return (int)ceil(sqrt(-2* log(epsilon)/omega))
     ;
}

//change variables according to problem
void relative(){ K=1; omega_0*=0.5; }
void com(){ K=0; omega_0*=2; }
void single(){ K=0;}

//solve for increasing r_max until the a stopping
   criterea is reached
//the Eigenvalues and Eigenvector of the final r_max
   is stored
void rm_loop(double limit, int max_iterations){

  Eigenstates temp;
  int count=0;
  double diff=1;
  if(max_iterations>0){
    cout <<endl<< "Starting loop over rmax" << endl;
    if(type==1) {
      if(n_richardson==0) solution=FDMsolve(false);
      else solution=Richardson(n_richardson, false);
    }
    if(type==2) solution=FEMsolve(false);
  }
  while(diff>limit && count<max_iterations){

    r_max++;
    M=(int)ceil(r_max/h);
    if(full_set) n_eigenvalues=N-2; //pass på for ne
       !!!
```

```
    if(type==1) {
      if(n_richardson==0) temp=FDMsolve(false);
      else temp=Richardson(n_richardson, false);
    }
    if(type==2) temp=FEMsolve(false);

    diff=solution.compare_set(temp, n_compare);
    cout << "Difference in set =  " << diff << endl;
    solution=temp;
    count++;
  }
  cout << "Done looping over rmax -> rmax = " <<
     r_max << endl << endl;
  if(max_iterations>0 && count == max_iterations)
     cout << "Reached the maximum number of
     iterations in r_max." << endl;

  if(type==1) {
    if(n_richardson==0) solution=FDMsolve(true);
    else solution=Richardson(n_richardson, true);
  }
  if(type==2) solution=FEMsolve(true);

}

//Richardson extrapolations
Eigenstates Richardson(int R_it, bool vec){

  cout << "Richardsom Iteration " << 0 << ": ";
    //cout << solution.step_length;
  Eigenstates* saved=new Eigenstates[R_it+1];
  saved[0]=FDMsolve(vec);

  Eigenstates temp;
  Eigenstates next;

  for(int i=0; i<R_it; i++){
    h*=0.5;
    M=(int)ceil(r_max/h);
    cout << "R Iteration " << i+1 << ": ";
```

```
      temp=FDMsolve(vec);
      next=Rich(1, temp, saved[0]);
      saved[0]=temp;

      for(int j=1; j<=i; j++){
        temp=next;
        next=Rich(j+1, temp, saved[j]);
        saved[j]=temp;

      }

      saved[i+1]=next;

    }
    h*=pow(2.0,R_it);
    M=(int)ceil(r_max/h);
    //the last calculated eigenvectors are stored in
       saved[0], the most accurate eigenvalues are
       stored in saved[0], so we copy and return this
       as a set

    Eigenstates final;
    final=saved[0];

    for(int i=0; i<n_eigenvalues; i++) final.eigenvalue
       [i]=saved[R_it].eigenvalue[i];

    delete[] saved;
    return final;
  }

  Eigenstates Rich(int j, Eigenstates a, Eigenstates b)
     {
    Eigenstates out(n_eigenvalues, 0 ,1);
    for(int i=0; i<n_eigenvalues; i++){
      out.eigenvalue[i]=(pow(4.0, j)*a.eigenvalue[i]-b.
         eigenvalue[i])/(pow(4.0,j)-1);
    }
    return out;
  }
```

```
};

int main(int argc, char** args){
  quantumdot qd;
}
```

### B.1.1 Input file "qd.inp"

```
type = 2; #1=FDM, 2=FEM
problem = 0; #1=single, 2=relative, 3=com, 0=input
dim=3 # 2, 3
num_eigenvalues = 20
all = false #calculate full set of eigenstates -> set
   r_max first, maxit=0

m=1
l=0

omega=1
#omegainv=3
B=0
K=1

n_e=3
maxit=0

h=0.0125
r_min=0
rich=0
r_max=9
```

## B.2 Solver.h

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <fstream>
#include "Eigen.h"
```

```
#include "matrix.h"
#include "arpack++/arbsmat.h"
using namespace std;

class Solver{

 public:
  int N; //size of global matrix
  double h; //step_length/element size
  Eigenstates solution; //solution

 public:

  void virtual solve(int, bool){};
  void virtual set_potential(int, int, double, double)
     {};
  virtual ~Solver(){}
};

class FDM : public Solver {

 public:

  matrix bm;
  FDM(double h, double min, double max);
  FDM(int N, double h);
  ~FDM();

  void solve(int nev, bool vectors){make_system();
     solve_system(nev, vectors);};
  void make_system();
  void solve_system(int, bool);

  void set_potential(int, int, double, double);
  void matvec_product(double *, double*);

 private:
  double* potential;

 };
```

```
class FEM : public Solver {

 public:
  int M; //number of elements
  int n_e; //number of nodes per element
  int int_N; //number of integration points

  matrix left;
  matrix right;

  matrix potential;
  double x_min;

  double* xi;
  double* weights;

  FEM(){};
  ~FEM(){delete[] xi; delete[] weights; };
  FEM(int elements, int local, int int_N, double h);


  void solve(int nev, bool vectors){make_system();
     solve_system(nev, vectors);};
  double N_i(int, double);
  double dN_i(int, double);

  double integrate(double*);
  void calc_element(int e, matrix &left, matrix &right)
     ;
  void set_integration_points();

  void set_potential(int dim, int qn, double omega2,
     double K, double x_min);
  void test1D(double);
  void make_system();
  void solve_system(int, bool);
  void bc();

};
```

## B.3 Finite element Solver

```
#include "Solver.h"
#include "arpack++/argsym.h"
#include "arpack++/arbgsym.h"

FEM::FEM(int elements, int local, int int_N_in, double
    step_length){

   n_e=local;
   if(n_e>3) n_e=3;
   else if(n_e<2) n_e=2;
   int_N=5;
   //int_N=int_N_in;
   //if(int_N!=2 || int_N!=4 || int_N!=5) int_N=5;

   M=elements;
   N=M*(n_e-1)+1;
   h=step_length;

   left=matrix(N,N,0);
   right=matrix(N,N,0);

   weights=new double[int_N];
   xi=new double[int_N];

   set_integration_points();
   potential=matrix(N,N,0);
   }


void FEM::set_potential(int dim, int qn, double omega2,
     double K, double x_min){
   double x1, x2, nabla;
   this->x_min=x_min;
   if(dim==2) nabla=qn*qn-0.25;
   else if(dim==3) nabla=qn*(qn+1);
   else{ nabla=0; cout << "error in set_potential!"<<
      endl;}
```

```
  double div=n_e-1;
  for(int i=0; i<M; i++){
    for(int j=0; j<int_N; j++){
      x1=0;
      for(int s=0; s<n_e; s++)
        x1+=N_i(s, xi[j])*(x_min+(s/div+i)*h);
      x2=x1*x1;
      potential[i][j]=nabla/x2 + K/x1+omega2*x2;
    }
  }
}


void FEM::bc(){ //parallelisation: only boundary
    elements
  left.remove(N-1);
  right.remove(N-1);
  left.remove(0);
  right.remove(0);
  N-=2;
}



double FEM::N_i(int i, double x){ //basis functions
  if(n_e == 2){
    if(i==0) return 0.5-0.5*x;
    else if(i==1) return 0.5+0.5*x;
  }else if(n_e == 3){
    if(i==0) return 0.5*x*(x-1);
    else if(i==1) return 1-x*x;
    else if(i==2) return 0.5*x*(1+x);
  }
}
double FEM::dN_i(int i, double x){ //basis functions
   if(n_e == 2){
    if(i==0) return -0.5;
    else if(i==1) return 0.5;
  }else if(n_e == 3){
    if(i==0) return x-0.5;
    else if(i==1) return -2*x;
```

```
    else if(i==2) return x+0.5;
  }
}


void FEM::make_system(){
  matrix element_left=matrix(n_e,n_e);
  matrix element_right=matrix(n_e,n_e);

  for(int e=1; e<=M; e++){
    calc_element(e, element_left, element_right);
     for(int i=0; i<n_e; i++){
      for(int j=0; j<n_e; j++){
        left[(n_e-1)*(e-1)+i][(n_e-1)*(e-1)+j]+=
           element_left[i][j];
        right[(n_e-1)*(e-1)+i][(n_e-1)*(e-1)+j]+=
           element_right[i][j];
      }
    }
  }
  bc();
}

void FEM::solve_system(int nev,  bool vectors){
  double* temp=new double[N];
  int nconv;

  double* a_array=left.get_array_symb(N, n_e);
  double* b_array=right.get_array_symb(N, n_e);

  ARbdSymMatrix<double> amat(N, n_e-1, a_array);
  ARbdSymMatrix<double> bmat(N, n_e-1, b_array);

  double* resid=new double[N];
  for(int i=0; i<N; i++) resid[i]=1;

  if(nev>N) nev=N;
  if(nev>N-1){ //number of eigenvalues are higher than
     arpack can handle -> calculate in two calls
    int nev1=nev/2;
    int nconv1;
```

```
    ARluSymGenEig<double > eigensolver1(nev1, amat,
       bmat, "SM", 0, 0.0, 0, resid , true);
    ARluSymGenEig<double > eigensolver2(nev-nev1, amat,
       bmat, "LM", 0, 0.0, 0, resid , true);

    if(vectors){
      nconv1 = eigensolver1.FindEigenvectors();
      for(int i=0; i<N; i++) resid[i]=1;
      nconv = eigensolver2.FindEigenvectors();
      solution=Eigenstates(nev,N,h/(n_e-1));

      for(int i=0; i<nconv1; i++){
        for(int j=0; j<N; j++) {
          temp[j]=eigensolver1.Eigenvector(i,j);
        }
        solution.add_pair(i, eigensolver1.Eigenvalue(i)
           , temp);
      }
      for(int i=0; i<nconv; i++){
        for(int j=0; j<N; j++) {

          temp[j]=eigensolver2.Eigenvector(i,j);
        }
        solution.add_pair(i+nconv1, eigensolver2.
           Eigenvalue(i), temp);
      }
    }else{
      nconv1 = eigensolver1.FindEigenvalues();
      for(int i=0; i<N; i++) resid[i]=1;
      nconv = eigensolver2.FindEigenvalues();
      solution=Eigenstates(nev,0,h/(n_e-1));
      for(int i=0; i<nconv1; i++) solution.eigenvalue[i
         ]=eigensolver1.Eigenvalue(nconv1-1-i);
      for(int i=0; i<nconv; i++) solution.eigenvalue[i+
         nconv1]=eigensolver2.Eigenvalue(i);
    }
  }else{
    ARluSymGenEig<double > eigensolver(nev, amat, bmat,
        "SM", 0, 0.0, 0, resid , true);

    if(vectors){
```

```
      nconv = eigensolver.FindEigenvectors();
      solution=Eigenstates(nconv,N,h/(n_e-1));
      for(int i=0; i<nconv; i++){
        for(int j=0; j<N; j++) temp[j]=eigensolver.
            Eigenvector(i,j);
        solution.add_pair(i, eigensolver.Eigenvalue(i),
            temp);
      }

    }else{
      nconv = eigensolver.FindEigenvalues();
      solution=Eigenstates(nconv,0,h/(n_e-1));
      for(int i=0; i<nconv; i++) solution.eigenvalue[i
          ]=eigensolver.Eigenvalue(nconv-1-i);
    }
    if(nconv<nev) {
      cout << "nconv<num"<<endl;
      solve_system(nev+10, vectors);
    }
  }

  delete[] resid;
  delete[] temp;
  delete[] a_array;
  delete[] b_array;
}

double FEM::integrate(double* func){
  double sum=0;
  for(int i=0; i<int_N; i++){
    sum+=weights[i]*func[i];
  }
  return sum;
}

void FEM::calc_element(int e, matrix &eleft, matrix &
    eright){
  double* int1=new double[int_N];
  double* int2=new double[int_N];
  double M_ij;
  double xp;
```

```
  for(int i=0; i<n_e; i++){
    for(int j=0; j<n_e; j++){
      for(int k=0; k<int_N; k++){
        M_ij=h/2.0*N_i(i,xi[k])*N_i(j,xi[k]);
        int1[k]=M_ij*potential[e-1][k]+ 2.0/h*dN_i(i,xi
            [k])*dN_i(j,xi[k]);;
        int2[k]=M_ij;
      }
      eleft[i][j]=integrate(int1);
      eright[i][j]=integrate(int2);
    }
  }

  delete[] int1;
  delete[] int2;
}


void FEM::set_integration_points(){
  //http://mathworld.wolfram.com/Legendre-
      GaussQuadrature.html
  if(int_N==2){
    xi[0] = -sqrt(1.0/3.0);
    weights[0]= 1;
    xi[1]=-xi[0];
    weights[1]=1;
  }
  else if(int_N==4){
    //gauss-legendre for polynomial of 4th order
    xi[0]= -sqrt(525+70*sqrt(30))/35.0;
    weights[0]= (18-sqrt(30))/36.0;
    xi[1]=-sqrt(525-70*sqrt(30))/35.0;
    weights[1]=(18+sqrt(30))/36.0;
    xi[2]=-xi[1];
    weights[2]=weights[1];
    xi[3]=-xi[0];
    weights[3]=weights[0];
  }else{ //if(int_N==5){
    //gauss-legendre for polynomial of 5th order
    xi[0]=-sqrt(245+14*sqrt(70))/21.0;
```

```
    weights[0]=(322.0-13*sqrt(70))/900.0;
    xi[1]=-sqrt(245-14*sqrt(70))/21.0;
    weights[1]=(322.0+13*sqrt(70))/900.0;
    xi[2]=0;
    weights[2]=128.0/225.0;
    xi[3]=-xi[1];
    weights[3]=weights[1];
    xi[4]=-xi[0];
    weights[4]=weights[0];
  }
}
```

## B.4 Finite difference Solver

```
#include "Solver.h"
#include "arpack++/arssym.h"
#include "arpack++/arbssym.h"

FDM::FDM(int num_points, double step){
  h=step;
  N=num_points-2;
  potential=new double[N];
  bm = matrix(N,N,0);
}

FDM::~FDM(){delete[] potential;}

void FDM::set_potential(int dim, int qn, double omega2,
    double K){
  double h4=h*h*h*h;
  double nabla;
  double h2=h*h;

  if(dim==2) nabla=qn*qn-0.25;
  else if(dim==3) nabla=qn*(qn+1);
  else{ nabla=0; cout << "error in set_potential!"<<
     endl;}

  for(int i=1; i<=N; i++)
```

```
    potential[i-1]=nabla/(i*i)+omega2*i*i*h4+ K*h/i;
}


void FDM::make_system(){
  for(int i=2; i<N; i++){
    bm[i-1][i-1]=2 +potential[i-1];
    bm[i-1][i]=bm[i-1][i-2]=-1;
  }
  bm[0][0]=2 +potential[0];
  bm[N-1][N-1]=2 +potential[N-1];
  bm[0][1]= bm[N-1][N-2]=-1;
}
void FDM::solve_system(int nev, bool vectors){
  double h2=h*h;
  double* temp=new double[N];
  int nconv;

  double* a_array=bm.get_array_symb(N, 2);
  ARbdSymMatrix<double> amat(N, 1, a_array);

  double* resid=new double[N];
  for(int i=0; i<N; i++) resid[i]=1;

  if(nev>N) nev=N;
  if(nev>N-1){
    int nev1=nev/2;
    int nconv1;
    ARluSymStdEig<double > eigensolver1(nev1, amat, "
       SM", 0, 0.0, 0, resid , true);
    ARluSymStdEig<double > eigensolver2(nev-nev1, amat,
         "LM", 0, 0.0, 0, resid , true);

    if(vectors){
      nconv1 = eigensolver1.FindEigenvectors();
      for(int i=0; i<N; i++) resid[i]=1;
      nconv = eigensolver2.FindEigenvectors();
      solution=Eigenstates(nev,N,h);

      for(int i=0; i<nconv1; i++){
        for(int j=0; j<N; j++)  {
```

```
        temp[j]=eigensolver1.Eigenvector(i,j);
      }
      solution.add_pair(i, eigensolver1.Eigenvalue(i)
        /h2, temp);
    }
    for(int i=0; i<nconv; i++){
      for(int j=0; j<N; j++) {

        temp[j]=eigensolver2.Eigenvector(i,j);
      }
      solution.add_pair(i+nconv1, eigensolver2.
        Eigenvalue(i)/h2, temp);
    }
  }else{
    nconv1 = eigensolver1.FindEigenvalues();
    for(int i=0; i<N; i++) resid[i]=1;
    nconv = eigensolver2.FindEigenvalues();
    solution=Eigenstates(nev,0,h);
    for(int i=0; i<nconv1; i++) solution.eigenvalue[i
      ]=eigensolver1.Eigenvalue(nconv1-1-i)/h2;
    for(int i=0; i<nconv; i++) solution.eigenvalue[i+
      nconv1]=eigensolver2.Eigenvalue(i)/h2;
  }
}else{
  ARluSymStdEig<double> eigensolver(nev, amat, "SM",
    0, 0.0, 0, resid , true);
  if(vectors){

    nconv = eigensolver.FindEigenvectors();
    solution=Eigenstates(nconv,N,h);
    for(int i=0; i<nconv; i++){
      for(int j=0; j<N; j++)  temp[j]=eigensolver.
        Eigenvector(i,j);
      solution.add_pair(i, eigensolver.Eigenvalue(i)/
        h2, temp);
    }

  }else{
    nconv = eigensolver.FindEigenvalues();
    solution=Eigenstates(nconv,0,h);
```

```
      for(int i=0; i<nconv; i++) solution.eigenvalue[i
         ]=eigensolver.Eigenvalue(nconv-1-i)/h2;
    }
    if(nconv<nev) {
      cout << "Increase number of eigenvalues in search
          for arpack..."<<endl;
      solve_system(nev+10, vectors);
    }
  }

  delete[] resid;
  delete[] temp;
  delete[] a_array;
}
```

## B.5 Eigenstates class

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <string>
using namespace std;

class Eigenstates{

 public:
  int number; //number of eigenstates
  int length; //length of eigenvectors
  double step_length; //even step_length

  //quantum numbers?
  int* n;

  double* eigenvalue;
  double** eigenvector;

  void allocate(int N, int l, double h){
    number=N;
    length=l;
```

```
    step_length=h;

    n=new int[number];
    eigenvalue=new double[number];
    eigenvector=new double*[number];
    for(int i=0; i<number; i++) {
      eigenvector[i]=new double[length];
    }
  }

  void deallocate(){
    for(int i=0; i<number; i++) {
      delete[] eigenvector[i];
    }
    delete[] eigenvalue;
    delete[] eigenvector;
    delete[] n;
   }

  Eigenstates(){ allocate(1,1,1.0); }
  Eigenstates(int N, int l, double h){ allocate(N,l,h);
      }
  Eigenstates(const Eigenstates &in){
    allocate(in.number, in.length, in.step_length);
    for(int i=0; i<number; i++){
      eigenvalue[i]=in.eigenvalue[i];
      for(int j=0; j<length; j++) eigenvector[i][j]=in.
         eigenvector[i][j];
    }
  }

  ~Eigenstates(){deallocate();}

  Eigenstates& operator=(const Eigenstates& in){
    if (this != &in){
      if(number!=in.number || length!=in.length){
        deallocate();
        allocate(in.number, in.length, in.step_length);
      }
      for(int i=0; i<number; i++){
        eigenvalue[i]=in.eigenvalue[i];
```

```
        for(int j=0; j<length; j++) eigenvector[i][j]=
            in.eigenvector[i][j];
      }
    }
    return *this;
  }

  void find_zeros(){
    int zeros;
    for(int i=0; i<number; i++){
      zeros=0;
      for(int j=1; j<length; j++){
        if((abs(eigenvector[i][j]-eigenvector[i][j-1])
            >10e-8) && ((eigenvector[i][j-1] >0 &&
            eigenvector[i][j]<0) || (eigenvector[i][j-1]
             <0 && eigenvector[i][j]>0))  ){
          zeros++;
        }
      }
      n[i]=zeros;
    }
  }

  //add an eigenstate to the set
  void add_pair(int k, double val, double* vec){
    //assumes k<number, vec[l]
    for(int i=0; i<length; i++){
      eigenvector[k][i]=vec[i];
    }
    eigenvalue[k]=val;
  }

  //print eigenvalues to screen
  void show_eigenvalues(){
    show_eigenvalues(1);
  }

  //print eigenvalues scaled by omega to screen
  void show_eigenvalues(double omega){
    if (omega==0) omega=1;
    cout << "Eigenvalues: " <<endl;
```

```
    for(int i=0; i<number; i++){
      cout << eigenvalue[i]/omega << endl;
  }
    cout << "-----------------" <<endl;
  }

  //prints eigenvectors to screen, only useful for a
     small set!
  void print_eigenvectors(){
    cout << "Eigenvectors" <<endl;
    for(int i=0; i<number; i++){
      for(int j=0; j<length; j++){
        cout <<eigenvector[i][j] << "      ";
      }
      cout << endl <<endl;
    }
    cout << "-----------------" <<endl;
  }

  //prints the set to a matlab script
  void print(char * filein, int m, int l, double omega)
     {
    char* ext = ".m";
    char file[80];
    strcpy(file, filein);
    strcat(file, ext);

    ofstream ofile;
    ofile.open (file);
    ofile << "m = " << m << ";" << endl;
    ofile << "l = " << l << ";" << endl;
    ofile << "omega = " << omega << ";" << endl;
    ofile << "h = " << step_length << ";" << endl;
    ofile << "E = [ " ;
    for(int i=0; i<number; i++) ofile << scientific <<
       setprecision(21) << setw(28)  << eigenvalue[i];
    ofile << "];" << endl;

    ofile << "V = [ " <<endl;
    for(int j=0; j<length; j++){
```

```
      for(int i=0; i<number; i++) ofile << scientific
         <<setprecision(21) << setw(30) <<  eigenvector
         [i][j];
      ofile << endl;
    }
    ofile << "];" << endl;
    ofile.close();
  }

  //shift all eigenvalues: val=val+shift
  void shift(double shift){
    for(int i=0; i<number; i++){
      eigenvalue[i] += shift;
    }
  }

  //scale all eigenvalues: val=val*scale
  void scale(double scale){
    for(int i=0; i<number; i++){
      eigenvalue[i] *= scale;
    }
  }

  //shift and scale all eigenvalues val=val*scale+shift
  void scale_shift(double scale, double shift){
    for(int i=0; i<number; i++){
      eigenvalue[i] = scale*eigenvalue[i]+shift;
    }
  }

  //dot product eigenvector(i)*eigenvector(j)
  double dot(int i, int j){ // legge til dim + r senere
   double prod=0;

   for(int k=0; k<length; k++){ //u_0, u_N = 0
     prod+=eigenvector[i][k]*eigenvector[j][k];
   }
   prod*=step_length;
   return prod;
  }
```

```
//normalise eigenvector(i)
void normalise_vec(int i){
 double norm=sqrt(dot(i,i));
 for(int j=0; j<length; j++){
   eigenvector[i][j]=eigenvector[i][j]/norm;
 }
}

//normalise all eigenvectors
void normalise(){
  for(int i=0; i<number; i++){
    normalise_vec(i);
  }
}

//use the basis of eigenvectors to create an
   orthogonal set
void orthogonalise_set(){
  double d;
  normalise_vec(0);
  for(int i=1; i<number; i++){
    for(int j=0; j<i; j++){
      d=dot(i,j);
      for(int k=0; k<length; k++){
        eigenvector[i][k]-=d*eigenvector[j][k];
      }
      normalise_vec(i);
    }
  }
}

//compare the n_compare first eigenvalues to set b,
   returning sum/a_i-b_i/, used i rm_loop i
   quantumdot.cpp, compare algo may be improved to
   check the convergence of eigenvalue(n_compare)
double compare_set(Eigenstates b, int n_compare){
  double error=0;
  n_compare = min(min(n_compare, number), b.number);
  for(int i=0; i<n_compare; i++){
    error+=abs(eigenvalue[i]-b.eigenvalue[i]);
  }
```

```
    return error;
  }

};
```

## B.6 Simple matrix class

```
#include <iostream>
#include <iomanip>
using namespace std;

class matrix{

public:
  int rows;
  int cols;
  double** mat;

  matrix(int N, int M){allocate(N,M);}
  matrix(int N, int M, double val){
    allocate(N,M);
    for(int i=0; i<rows; i++){
      for(int j=0; j<cols; j++){
        mat[i][j]=val;
      }
    }
  }
  matrix(int N){allocate(N,N);}
  matrix(){ rows=0; cols=0;}
  virtual ~matrix(){deallocate(); }

  double* operator[](int i) const{return mat[i];}

  void allocate(int N, int M){
    rows=N;
    cols=M;
    mat=new double*[rows];
    for(int i=0; i<rows; i++){
      mat[i]=new double[cols];
```

```
      for(int j=0; j<cols; j++) mat[i][j]=0;
    }
  }

  void deallocate(){
    for(int i=0; i<rows; i++){
       delete[] mat[i];
    }
    if(rows>0)
      delete[] mat;
  }

  matrix(const matrix &in){
    allocate(in.rows, in.cols);
    for(int i=0; i<rows; i++){
      for(int j=0; j<cols; j++){
        mat[i][j]=in[i][j];
      }
    }
  }

  matrix& operator=(const matrix &in){
    deallocate();
    allocate(in.rows, in.cols);
    for(int i=0; i<rows; i++){
      for(int j=0; j<cols; j++){
        mat[i][j]=in[i][j];
      }
    }
    return *this;
  }

  void print(){ //cout
    for(int i=0; i<rows; i++){
      for(int j=0; j<cols; j++){
        cout << setw(8) << setprecision(3) << mat[i][j
            ];
      }
      cout <<endl;
    }
  }
```

```
void matvec(double* in, double* out){
      for(int i=0; i<rows; i++){
        out[i]=0;
        for(int j=0; j<cols; j++) out[i]+=mat[i][j]*
            in[j];
      }
}


double* get_array_symb(int size, int bands){ //som
    symmetrisk, bands= diag + upper
    double* array= new double[size*bands];
    for(int i=0; i<size; i++){
     for(int j=i; j<(i+bands); j++){
       array[i*(bands-1)+j] = mat[i][j]; //fyller litt
           utenfor, men det er ikke minneproblemer
     }
   }
    return array;
}

void remove(int pos){ //removes column/row pos
  double** copy=new double*[rows-1];
  for(int i=0; i<rows-1; i++) copy[i]=new double[cols
      -1];
  for(int i=0; i<pos; i++){
    for(int j=0; j<pos; j++) copy[i][j]=mat[i][j];
    for(int j=pos+1; j<cols; j++) copy[i][j-1]=mat[i
        ][j];
  }
  for(int i=pos+1; i<rows; i++){
    for(int j=0; j<pos; j++) copy[i-1][j]=mat[i][j];
    for(int j=pos+1; j<cols; j++) copy[i-1][j-1]=mat[
        i][j];
  }
  mat=copy;
  rows--;
  cols--;
}
```

```
};
```

```
clear
%fortegnsproblemer - ved superposisjon

%read
eigenset;
[l n]=size(V);
D=diag(E);
I=eye(l);
T=20;
t=0;
dt=0.25;
steps=T/dt;
W=1;
%wf=2;
f=1*2*pi/T;

r = linspace(h, l*h , l);

d=zeros(l,1);
%d(wf)=1; %velge hvilken tilstand vi starter i
d(1)=1;
d(6)=1;
d(4)=1;
d=d/norm(d);

d0=d;
c=V*d;
p=conj(c).*c;
a=d0;

%plot analytic for W=0 --
figure(2)
plot(r,real(c),'b') %real part
hold on
plot(r,imag(c),'r') %imaginary part

plot(r,real(a),'b--')
plot(r,imag(a),'r--')
```

```
plot(r, p, 'g') % c*c
ylim([-1,1]);
hold off
Movie(1)=getframe;

for j=0:steps-1 %evolve from tj to tj+1

    %H(s)
    int1=D*dt+I*-W*(cos(f*(t+dt))-cos(f*t))/f;
    %sH(s)
    int2=D*dt*(t+0.5*dt)+I*W*(sin(f*(t+dt))-t*f*cos(f*(t
        +dt))-sin(f*dt)+t*f*cos(f*dt))/f/f;

    H0=-i*int1;
    H1=i/dt*((t+0.5*dt)*int1-int2);

    d=expv(-1, H1, d);
    d=expv(1, H0, d);
    d=expv(1, H1, d);

    t=t+dt;
    c=V*d;
    p=conj(c).*c;

    a=V*diag(exp(-i*E*t))*d0;
    diff=c-a;
    sum(diff); %sammenlikne med analytisk for W=0

    plot(r,real(c), 'b')
    hold on
    plot(r,imag(c),'r')
    plot(r,real(a),'b--')
    %hold on
    plot(r,imag(a),'r--')
    plot(r, p, 'g')
    text(10*h, 0.8, ['t=', num2str(t)])
    hold off
    ylim([-1,1]);
    Movie(j+2)=getframe;
    %sin(f*t)
```

```
end


%movie(Movie,1,1)
```

# Bibliography


[1] David j. Griffiths. *Introduction to Quantum Mechanics*. Pearson, 2005.

[2] R. Shankar. *Principles of Quantum Mechanics*. Plenum Press, 1994.

[3] Finn Ravndal. Notes on quantum mechanics. Lecture notes - FYS 3110.

[4] NIST (National Institute og Standards and Technology). Codata internationally recommended values of the fundamental physical constants.

[5] Eric W. Weisstein. Legendre polynomial. From MathWorld–A Wolfram web resource http://mathworld.wolfram.com/LegendrePolynomial.html.

[6] Stephanie M. Reimann and Matti Manninen. Electronic structure of quantum dots. *Rev. Mod. Phys*, 74:1283, 2002.

[7] R. C. Ashoori. Electrons in artificial atoms. *Nature*, 379:413, 1996.

[8] Daniel Loss and David P. DiVincenzo. Quantum computation with quantum dots. *Phys. Rev. A*, 57(1):120–126, Jan 1998.

[9] Xiaohu Gao, Yuanyuan Cui, Richard M. Levenson, Leland W K . Chung, and Shuming Nie. In vivo cancer targeting and imaging with semiconductor quantum dots. *nature biotechnology*, 22(8):969, August 2004.

[10] M. Taut. Two electrons in a homogeneous magnetic field: particular analytical solutions. *J. Phys. A: Math. Gen.*, 27:1045, 1994.

[11] M. Taut. Two electrons in an external oscillator potential: Particular analytic solutions of a coulomb correlation problem. *Phys. Rev. A*, 48(5):3561–3566, Nov 1993.

[12] Andreassen et al. *Numeriske metoder*. Tapir, 1986.

[13] H.P. Langtangen. *Computational Partial Differential Equations: Numerical Methods and Diffpack Programming*. Springer, 2003.

[14] L. Ramdas Ram-Mohan. *Finite Element and Boundary Element Applications in Quantm Mechanics*. Oxford University Press, 2002.

[15] H.P. Langtangen and A. Tveito. *Advanced topics in computational partial differential equations: numerical methods and diffpack programming*. Springer, 2003.

[16] S Blanes and P. C. Moan. Splitting methods for the time-dependent Schrödinger equation. *Physics Letters A*, 265:35–42, 2000.

[17] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C++, The art of scientific Computing*. Cambridge University Press, 1999.

[18] Arpack software. http://www.caam.rice.edu/software/ARPACK/.

[19] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins University Press, 1996.

[20] Simen. Kvaal. A critical study of the finite difference and finite element methods for the time dependent Schrödinger equation. Master's thesis, University of Oslo, 2004.

[21] Rick Wagner. Configuration file reader for c++. http://www-personal.umich.edu/~wagnerr/ConfigFile.html.

[22] Expokit - matrix exponential software package for dense and sparse matrices. http://www.maths.uq.edu.au/expokit/.

[23] Eric W. Weisstein. Laguerre polynomial. From MathWorld–A Wolfram web resource http://mathworld.wolfram.com/LaguerrePolynomial.html.

[24] Morten. Hjorth-Jensen. Lecture notes on computational physics. http://www.uio.no/studier/emner/matnat/fys/FYS3150/h08/undervisningsmateriale/Lecture%20Notes/lecture2008.pdf, 2006. Lecture notes in FYS-3150 and FYS 4410.

[25] Eric W. Weisstein. Legendre-gauss quadrature. From MathWorld–A Wolfram web resource http://mathworld.wolfram.com/Legendre-GaussQuadrature.html.