A Schrodinger solver for nano-scale one-particle Quantum-Wells, -Wires and -dots with infinite potential barriers

Sanam Moslemi-Tabrizi

A thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Electrical Engineering) at
Concordia University
Montreal, Quebec, Canada

April, 2007

Your file Votre référence
ISBN: 978-0-494-28921-1
Our file Notre référence
ISBN: 978-0-494-28921-1

Abstract

A Schrodinger solver for nano-scale one-particle Quantum-Wells, -Wires and -dots with infinite potential barriers

Sanam Moslemi-Tabrizi

In this thesis a fast, efficient and simple algorithm and its implementation (in the Java programming language) is presented; the algorithm calculates the energy states and the corresponding wave functions of a one-particle quantum well/wire/dot structure with an arbitrary potential profile by solving the multi-dimensional time-independent Schrodinger equation. The algorithm is based on the Finite Cloud Method (FCM), which is a truly meshless method. The contributions of this thesis are the expansion of FCM to a 3D method and implementing a tool to solve the multi-dimensional Schrodinger equation in an efficient way. To validate the accuracy and efficiency of our implementation we calculated the eigenstates of different rectangular GaAs quantum-wells, -wires and -dots. Comparing the obtained results with the analytical results published in the literature shows our approach to be a successful proof of concept. The results also confirm our implementation of the FCM algorithm to be highly accurate and efficient.

Acknowledgments

The research presented in this thesis could not have been possible if not for the help, understanding, and support of many people.

First and foremost I would like to thank Professor Mojtaba Kahrizi. His valuable guidance, continuous support and great patience helped me with the most difficult aspects of my research.

Thanks also to my thesis defence committee members Dr. Abdel Sebak, Dr. Ali Dolatabadi and Dr. Kash Khorasani for their valuable comments that greatly improved the quality of this thesis.

Last but not least I'd like to thank the loves of my life, Bahman and Roitan, for their care, love and many sacrifices they have made. Their support and presence enriched my mind and made this thesis possible. I would also like to thank my parents for their indescribable love and support.

Sanam Moslemi-Tabrizi

Concordia University

26 February 2007

Contents

List of Tables

List of Figures

List of Symbols

w	angular frequency
$\psi^a(\bar{x})$	approximated of unknown function
d_x, d_y, d_z	cloud size
$\zeta()$	correction Function
α_x, α_y, α_z	dimensionless factor
Δx, Δy, Δz	distance between two neighbor node
Ω	domain of interest
m^*	effective mass of particle
E	energy
f	frequency
H	Hamiltonian
$N_I(\bar{x})$	interpolation function
$\varphi()$	Kernel Function
∇^2	Laplacian Operator
M	moment matrix
p	momentum
h	Planck's constant
$V(r)$	potential Energy Function
$\hbar$	reduced Planck's constant
t	time
P^T	vector of basis function

C^T	vector of coefficients
r	vector of position
ψ	wavefunction
λ	wavelength
k	wavenumber

List of Abbreviations

BEM	boundary element method
CAD	computer aided design
DEM	diffuse element method
EFG	element free Galerkin
FCM	finite cloud method
FDEM	finite difference element method
FDM	finite difference method
FEM	finite element method
LSMFM	least-squares meshfree method
MBE	molecular beam epitaxy
MLPG	meshless local Petrov-Galerkin
MLS	moving least squares method
MOCVD	metalorganic chemical vapor deposition
PDE	partial differential equation
PG DEM	petrov galerkin diffuse element method
PIM	point interpolation method

RKPM	reproducing kernel particle methods
SPH	smooth particle hydrodynamics

1 Introduction

In recent years, due to significant improvements in micro-fabrication techniques like molecular beam epitaxy (MBE) and metalorganic chemical vapor deposition (MOCVD) it is possible to fabricate ultra-small quantum structures. Further miniaturization has resulted in design and fabrication of nano scale devices including pressure sensors [1], resonant tunneling transistors [2], biosensors [3] and photodetectors [4], [5]. Nano-scale devices have a variety of applications in different fields as diverse as nano-electronics, bioelectronics, military and biophysics so their fabrication is of great importance for modern electronics. However, since the cost –both in time and effort– required to fabricate a new nano device is considered extremely high, it is important to perform a realistic computer simulation of the devices in order to predict their characteristics and electrical properties before any further steps in the actual manufacturing process can be taken.

The basic building blocks of the nano-scale devices whose lateral dimensions range anywhere in the range of 1 to 100 nm, are quantum structures –quantum well/wire/dot. The simulation of nano scale devices therefore requires the simulation of the quantum structures. At this point it would be appropriate to briefly go over some definitions. According to quantum theory, the quantum effects are observed by decreasing the size of semiconductor structures to be comparable with de Broglie's wavelength of a particle. A semiconductor structure is called a quantum well, when its size in *one* spatial dimension is comparable with de Broglie's wavelength of a particle. Likewise a quantum wire is a semiconductor structure which confines the motion of a particle in *two* dimensions; in a quantum wire the quantum effects are observed in *two* spatial dimensions. And finally a quantum dot is a nanostructure which shows quantum effects in *all* the three spatial dimensions which consequently confines the motion of a particle in *three* spatial directions.

To simulate and design nano-devices based on quantum structures, it is important to know the eigenenergies and eigenfunctions of the quantum structures which are determined by solving the time-independent Schrodinger equation. The Schrodinger equation is a key equation in quantum mechanics whose solution predicts the behavior of quantum phenomena. As is the case with many partial differential equations, solving the Schrodinger equation poses enormous challenges with both analytical and numerical methods. For sufficiently simple conditions the time-independent Schrodinger equation can be solved analytically however for the more complex conditions, numerical methods often are the only approach possible to obtain a solution. Conventional mesh-based numerical methods have been used now for several decades to solve many problems in various fields of science, however recently their limitations such as the problem of having to deal with large geometrical deformation, crack growth and others have been revealed. The major shortcoming of mesh-based methods originates from these methods' reliance on mesh generation which is a challenging process to say the least.

Recently a new category of numerical methods, called meshless, meshfree or point-based methods, have been shown great promise. These meshless methods overcome many problems of mesh-based methods by eliminating the most problematic part of mesh-based methods namely the mesh generation process. The main advantage of meshless methods is that they use a set of distributed nodes for approximation purposes. The fact that meshless methods do not need mesh generation or connectivity information between the elements attracts many researchers to this promising field of numerical methods which has led to the development of different types of meshless methods in the recent past. Classical smooth particle hydrodynamics (SPH) [6], reproducing kernel particle methods (RKPM) [7], [8], element free Galerkin methods (EFG) [9], [10]and meshless local Petrov-Galerkin methods (MLPG) [11] are examples of point-based methods.

Finite Cloud Method (FCM) [12], [13] is one of the most recent meshless methods. Its major feature compared to other meshless methods is that it is *truly* meshless and doesn't need a background grid at any point. In this thesis we expand and implement FCM to solve the multi-dimensional Schrodinger equation for the one-particle quantum -well, -wire and -box semiconductor structures. The outline of the rest of this thesis is as follows:

- In chapter 2, an overview of quantum mechanics including history of quantum mechanics and the derivation of the Schrodinger equation as well as an overview of numerical methods including mesh-based and meshless methods are given. At the end of this chapter, the state of art in meshless methods is presented.

- In chapter 3, the theory of Finite Cloud Method (FCM) for one- and two- dimensional spaces, its expansion to three dimensions and its implementation is detailed.

- Chapter 4 illustrates some numerical examples to validate our approach. The implementation has been applied to different quantum structures and the results from running the simulation are compared with the expected theoretical results.

- Chapter 5 provides a brief comparison between mesh-based and meshless methods. This chapter also discuses the pros and cons of our approach and finally it suggests some future work directions arising from our work.

- A detailed list of references, arranged according to their appearance, is given following Chapter 5.

- The source code of our tool and the Maple programs used to calculate the theoretical eigenstates, are presented as appendices at the very end of this document.

2 Technical information

2.1 Quantum mechanics

2.1.1 History

In 1687 Isaac Newton (1642-1727) published his famous book The Mathematical Principles of Natural Philosophy or as more commonly known The Principia in which he summarizes the rules of classical mechanics. Classical mechanics is the science that describes the motion of objects in space and had been used for over than 200 years but, by the end of the nineteenth century scientists started to observe phenomena that were not explicable in terms of classical mechanics. In early 1900s scientists were trying to find new rules to describe these observations. On 19 October 1900 Max Planck (1858-1947) announced a new formula for heat radiation in black bodies that fit the experimental evidences well, but at that time he didn't have a theoretical explanation for his formula. Planck was not satisfied with his work and that became a motivation for him to work harder until he came up with a theory that was more satisfactory. Quantum theory as it was later known stated that light consisted of a set of discrete energies known as photon. Planck derived his formula based on this theory, which is now the fundamental idea of quantum mechanics. Planck's formula is given by

$$E = hf = \hbar w \tag{2.1}$$

where h is Planck's constant and its value is $6.626068 \times 10^{-34} \quad J.s$ and f is the frequency of the phonon.

He published his results on 14 December 1900. This date now is the birthday of quantum mechanics [14]. Planck's idea was a revolutionary step in physics and it resulted in more intensive collaborations being formed between researchers in the this field. In 1905 Albert Einstein (1879-1955) proposed the photon theory of light in photoelectric effect. Using Planck's idea of discrete energy, Einstein showed that the discrete energy E released from a matter is proportional to the frequency f of the incident light. His work confirmed that energy released or absorbed by an electron is in discrete form therefore under certain conditions light behaved like individual particles. Einstein won the Noble prize for his work on 1921.

In 1913, Niels Bohr (1885-1962), a Danish scientist who was working with Rutherford developed the idea of atomic structure for hydrogen atom. At that time, Rutherford and other scientists were considering a planetary model for atoms that stated that an atom was made of a cloud of negatively charged electrons which orbited a positive nucleus. Bohr considered the same model for the atom with the difference being that electron-orbits could only have certain discrete quantized energies. He proposed that if an electron orbiting in an outer orbit jumped to an inner orbit it would emit energy and that energy equaled to the difference of the energies of the two orbits. With his theory, the so-called old quantum theory, he described the fundamentals of atomic spectra and for his work Niels Bohr received the 1922 Nobel Prize for Physics. In 1920, Niels Bohr became the head of the Institute for Theoretical Physics in Copenhagen University. The institute's working domain was theoretical and experimental research on atomic physics area, which later led to creation of new quantum mechanics. The institute was a meeting place for many physicists and scientists from all over the world who wanted to join into the research and Werner Heisenberg (1901-1976) was one of them.

Heisenberg worked with three superior scientists: Arnold Sommerfeld, Max Born and Niels Bohr. He was determined to find the laws of quantum mechanics. He thought since he can't see the orbits of electrons, he had better develop a quantum mechanics based on observable entities. In 1925 Heisenberg published his first paper describing the laws of quantum mechanics by using matrices, which was a breakout for quantum mechanics. Two years later on 1927 he announced his uncertainty principle in which he stated: "The more precisely the position is determined, the less precisely the momentum is known in this instant, and vice versa." By his first paper in 1925 Heisenberg introduced a new way to formulate quantum mechanics. He assumed electrons as quantum particles and associated the properties of the particles by arrays of quantities. He stated that the commutative law in arithmetic is not valid for quantum mechanics which means that if A and B are two different observable properties of a particle then the result of AB is not equal to the result of BA. This led to the famous commutative relation, which is the basis of quantum mechanics and uncertainty principle as below:

$$[x,p] = xp - px = i\hbar \tag{2.2}$$

In which x is the position of the particle and p is the momentum of the particle and $\hbar = h/2\pi$ is the reduced Planck's constant. According to Heisenberg's uncertainty principle it is impossible to *precisely* measure the position and momentum as well as the energy and time of a particle at the same time as shown below:

$$\begin{aligned} \Delta x\,\Delta p &\geq \hbar/2 \\ \Delta E\,\Delta t &\geq \hbar/2 \end{aligned} \tag{2.3}$$

In his theory, later called matrix mechanics, Heisenberg used a complicated mathematics for the calculations, which was not familiar to other scientists. Max Born (1882-1970) who was Heisenberg's professor discovered that the strange mathematics that Heisenberg had used was as same as matrix calculations which was invented long time ago by mathematicians like James Joseph Sylvester, Cayley, Hamilton and Grassmann. Born and his student Pascual Jordan (1902-1980) reformulated Heisenberg's quantum theory and published a follow up paper to Heisenberg's paper. This was how matrix mechanics, the first formulation of quantum mechanics was invented.

Meanwhile Louis de Broglie (1892- 1987) was working on the duality theory of the matter. In 1924 in his doctorial thesis, Broglie proposed that wave-particle duality did not belong to light only and other matter could exhibit wave-particle duality in certain circumstances. He associated each particle with a wavelength (λ). The famous de Broglie's wavelength relation is as below:

$$\lambda = h/p = h/mv \tag{2.4}$$

Where h is Planck's constant, p is the momentum, m is the mass and v is the velocity of the particle. In his work de Broglie was inspired by Einstein's theory in photoelectric effect i.e.

$$E = hf \tag{2.5}$$

where f is the frequency of the light. Considering the relation between frequency and wavelength we see that a particle with greater energy has shorter wavelength. Later in 1927 Clinton Davisson and Lester Germer did an experiment that confirmed de Broglie's hypothesis. They built a device to measure the intensity of the fired electrons. The electrons were directed at a nickel target. Davisson and Germer monitored the dependency between the reflected electron's intensity and the angle at which the electrons were observed. The experiment determined that the reflected electrons had the same diffraction patterns that Bragg's law predicts for X-Rays. Bragg's law, which was formulated by William Lawrence Bragg (1890-1971), allows us to calculate the wavelength and therefore the energy of the incident x-ray beams as below:

$$n\lambda = 2d\sin\theta \tag{2.6}$$

where λ is the wavelength of the incident X-ray beam, d is the distance between the atomic layers, θ is the incident angle at which the intensity peaks and n is an integer. The diffraction described by Bragg's law used to be a property of waves, so the Davisson-Gremer experiment demonstrated the wave-like nature of electrons and thereby de Broglie's hypothesis was experimentally confirmed. De Broglie's idea was a significant step in quantum mechanics, which followed other important developments.

Erwin Schrodinger (1887-1961) was an Austrian physicist. He entered the university of Vienna in 1906 and received his doctorate on 1910. He studied analytical mechanics, applications of partial differential equations to dynamics, eigenvalue problems, statistical mechanics, algebra and Maxwell's equations. Schrodinger who didn't know about Heisenberg's work was inspired by de Broglie and Einstein's ideas and was trying to find a wave equation for matter that would demonstrate the wave behavior of particles. In 1926, Schrodinger published six papers explaining a new formulation for quantum mechanics, namely wave mechanics for which he won the Noble prize on 1933. In his first paper on January 1926

Schrodinger proposed the wave equation for matter now called the Schrodinger equation, which is the second formulation for quantum mechanics. In his 3rd paper on May 1926 he proved that Heisenberg's matrix mechanics is equivalent to wave mechanics. In that paper he wrote, "To each function of the position- and momentum- coordinates in wave mechanics there may be related a matrix in such a way that these matrices, in every case satisfy the formal calculation rules of Born and Heisenberg."

2.1.2 Schrodinger equation

Schrodinger's equation is the fundamental equation of quantum mechanics, which is equivalent to Newton's motion equation in classical mechanics. It predicts the wave behavior of matter and governs the microscopic world. The solution of the Schrodinger equation is the wave function $(\Phi(r,t))$ which describes the quantum states of the system [15], [16], [17]. Schrodinger equation has two basic forms, time-dependent and time-independent Schrodinger equation. The solution of the time-dependent form demonstrates how the system changes with time, and the time-independent Schrodinger equation gives us the energy eigenstates of the system.

Hamiltonian mechanics

Hamiltonian mechanics, which was introduced by the Irish mathematician Sir William Rowan Hamilton (1805-1865), is a re-derivation of Newton's laws and is more useful in understanding the concepts of conservation laws (conservation of momentum and energy). According to Hamiltonian mechanics, "the quantum Hamiltonian H is the observable corresponding to the total energy of the system." [18] Hamiltonian H is a function that relates the energy of a system to its coordinates and momentum. If r is the position vector and p is the momentum vector of the particle then the Hamiltonian H is given by:

$$H(r,p) = P^2/2m + V(r) \tag{2.7}$$

Description of a Wave

A wave is a movement through space with a repeating pattern and propagates in a given direction usually transferring energy. It can be described by its parameters such as Amplitude, Wavelength, Wavenumber, Frequency, Period and Angular frequency. Below is a brief explanation of these parameters.

The *Amplitude* (usually shown by A) of the wave is a nonnegative value, which represents the maximum vertical distance of the particle from the rest position i.e. the maximum value of wave's magnitude in one cycle. The *Wavelength* (λ) of the wave is the horizontal length of a repeating unit of the wave pattern and it can simply be measured as the distance between two sequential crests of a wave. *Wavenumber* (k) is a parameter related to wavelength, which is more commonly used in quantum mechanics than the wavelength. Wavenumber is the number of complete wave cycles per unit distance and is given by

$$k = 2\pi/\lambda \tag{2.8}$$

As a periodic event, the parameter *Period* (T) is defined for a wave. Period refers to the time needed for a particle to make one complete cycle and *frequency* (f) is the number of periods in one unit time given by

$$f = 1/T \tag{2.9}$$

The last parameter is the *Angular frequency* (w), which represents the angular speed of the particle in terms of radian per second and is given by

$$w = 2\pi f \tag{2.10}$$

Mathematically, description of a wave using the above parameters is given by

$$\Phi(r,t) = A\exp(\,i\,(\,k.r - wt\,)\,) \tag{2.11}$$

r : position vector

t : time

A : Amplitude

k : wavenumber

w : angular frequency

Derivation of the Schrodinger equation

As mentioned earlier, de Broglie associated a wave with a particle, according to his theory the relation between the wavelength and momentum of a particle is given by equation 2.4.

From Hamiltonian mechanics we know that the total energy of a system is given by

$$E = H(r,p) = P^2/2m + V(r) \tag{2.12}$$

Substituting Planck's formula in equation 2.12 we will have

$$\hbar w = P^2/2m + V(r) \tag{2.13}$$

The calculation below will finally result in the Schrodinger equation;

$$k = 2\pi/\lambda = 2\pi/(h/p) = p/\hbar \tag{2.14}$$

$$\begin{aligned} \Phi(r,t) &= A\exp(\,i\,(\,k.r - wt\,)\,) \\ &= A\exp(\,i\,(\,r.p/\hbar - wt\,)\,) \\ &= A\exp(\,i\,(\,(p_x x + p_y y + p_z z)/\hbar - wt\,)\,) \end{aligned} \tag{2.15}$$

$$\begin{aligned} \frac{\partial\Phi(r,t)}{\partial t} &= -iAw\exp(\,i\,(\,k.r - wt\,)\,) \\ &= -iw\Phi \end{aligned} \tag{2.16}$$

$$w = i\frac{\partial\Phi(r,t)}{\partial t}\frac{1}{\Phi(r,t)} \tag{2.17}$$

$$\begin{aligned}\frac{\partial \Phi}{\partial x} &= i\frac{p_x}{\hbar}A\exp(i\,(k.r-wt\,)) \\ &= i\frac{p_x}{\hbar}\Phi \end{aligned} \tag{2.18}$$

$$\frac{\partial^2 \Phi}{\partial x^2} = -\frac{p_x^2}{\hbar^2}\Phi \Longrightarrow p_x^2 = -\hbar^2\frac{\partial^2 \Phi}{\partial x^2}\frac{1}{\Phi} \tag{2.19}$$

By substituting equation 2.17 and equation 2.19 in equation 2.13 we have

$$\hbar i\frac{\partial \Phi}{\partial t}\frac{1}{\Phi} = \frac{1}{2m}(-\hbar^2\frac{\partial^2 \Phi}{\partial x^2} - \hbar^2\frac{\partial^2 \Phi}{\partial y^2} - \hbar^2\frac{\partial^2 \Phi}{\partial z^2})\frac{1}{\Phi} + V(x,y,z) \tag{2.20}$$

$$i\hbar\frac{\partial \Phi}{\partial t} = \frac{-\hbar^2}{2m}(\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} + \frac{\partial^2 \Phi}{\partial z^2}) + V(x,y,z)\Phi(x,y,z,t) \tag{2.21}$$

We rewrite equation 2.21 as

$$\frac{-\hbar^2}{2m}\nabla^2\Phi + V\Phi = i\hbar\frac{\partial \Phi}{\partial t} \tag{2.22}$$

where ∇^2 is the Laplacian operator given by

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{2.23}$$

Equation 2.22 is the time-dependent Schrodinger equation. The solution of equation 2.22, $\Phi(r,t)$, is the wavefunction of the system which gives us all the information about the system such as momentum, position and energy of the particles as well as probability density of finding the particles. The latter which is the most useful information is given by $\Phi^*\,\Phi$. Since the probability of finding a particle in the entire domain is unity, the wavefunction must satisfy the normalization condition as below

$$\int_{-\infty}^{+\infty} \Phi(r,t)\, \Phi^*(r,t)\, dr = 1 \qquad (2.24)$$

Recalling Bohr's theory of atomic model, we know that a stationary state with an energy E has the frequency $f = E/h$. So the wavefunction $\Phi(r,t)$ can be written as;

$$\Phi(r,t) = \psi(r) \exp(-i \frac{E}{\hbar} t) \qquad (2.25)$$

Using the new form of wavefunction (equation 2.25) will reduce the normalization condition to the form below:

$$\int_{-\infty}^{+\infty} \psi(r)\, \psi^*(r)\, dr = 1 \qquad (2.26)$$

Also by substituting the resulting wavefunction from equation 2.25, in time-dependent Schrodinger equation (equation 2.22) as well as using the concept of effective mass of particles (m^*), we will get the time-independent Schrodinger equation as shown in equation 2.27.

$$-\frac{\hbar^2}{2m^*} \nabla^2 \psi(r) + V(r)\psi(r) = E\psi(r) \qquad (2.27)$$

The time-independent Schrodinger equation is a linear second-order partial differential equation. It can be written in short notation as:

$$H\psi = E\psi \qquad (2.28)$$

Where H represents the Hamilton operator.

$$H \equiv -\frac{\hbar^2}{2m^*}\nabla^2 + V(r) \tag{2.29}$$

Equation 2.28 is an eigenvalue equation in which E is the eigenvalue and ψ is the eigenfunction of the equation. Schrodinger equation has solutions only for certain discrete values of E where these discrete values are the eigenenergies of the system. Likewise there is a corresponding eigenfunction (ψ) for every eigenvalue which describe the eigenstates of the system.

2.1.3 Solving the Schrodinger equation and the problems

Schrodinger equation is a linear partial differential equation (PDE). "A partial differential equation (PDE) is a relation invoking an unknown function of several independent variables and its partial derivatives with respect to those variables." [18] It is well known that an analytic solution to the Schrodinger equation is intractable except for very simple potential functions. Here we show some examples of simple cases.

1. A particle in an infinite well

Suppose we have an infinite well with width a, the potential function for this well is defined by

$$V(x) = \begin{cases} 0 & 0 < x < a \\ \infty & elsewhere \end{cases} \tag{2.30}$$

The Dirichlet boundary conditions is given by

$$\psi(0) = \psi(a) = 0 \tag{2.31}$$

In region $0 < x < a$ the Schrodinger equation is written as

$$-\frac{\hbar^2}{2m^*}\frac{\partial^2\psi(x)}{\partial x^2} = E\psi(x) \tag{2.32}$$

The general solution for equation 2.32 is given by

$$\psi(x) = A\sin(kx) + B\cos(kx) \tag{2.33}$$

in which

$$k = \sqrt{\frac{2m^*E}{\hbar^2}} \tag{2.34}$$

Using the boundary conditions in equation 2.31 we obtain the energy levels of a particle in the infinite well as

$$E = \frac{\hbar^2\pi^2}{2m^*a^2}n^2 \qquad n = 1, 2, \cdots \tag{2.35}$$

Fulfilling the normalization condition equation 2.26 we obtain the wavefunction as

$$\psi(x) = \sqrt{\frac{2}{a}}\,\sin(\frac{n\pi}{a}x) \qquad for\,n = 1, 2, \cdots \tag{2.36}$$

2. A particle in a square well

To obtain the energy levels of a particle in a square well with infinite barriers we consider a well with width a and the potential V_0 inside the well, the potential is given by equation 2.37 and the boundary conditions is same as equation 2.31.

$$V(x) = \begin{cases} V_0 & 0 < x < a \\ \infty & elsewhere \end{cases} \tag{2.37}$$

The Schrodinger equation here is given as

$$-\frac{\hbar^2}{2m^*}\frac{\partial^2\psi(x)}{\partial x^2} + (V_0 - E)\psi(x) = 0 \tag{2.38}$$

If we assume

$$k = \sqrt{\frac{2m^*(E - V_0)}{\hbar^2}} \tag{2.39}$$

the general form of wavefunction can be written as

$$\psi(x) = A\,\exp(ikx) + B\,\exp(-ikx) \tag{2.40}$$

By enforcing the boundary conditions we will obtain the energy levels as blew

$$E = V_0 + \frac{\hbar^2\pi^2}{2m^*a^2}n^2 \qquad n = 1,2,\cdots \tag{2.41}$$

For potential functions, which are not very simple even for the one-dimensional case, we can't solve the equation exactly and we need to use the approximation techniques like WKB, Perturbation and Variational methods. The WKB approximation also known as the semiclassical calculation is the method to solve the Schrodinger equation only for potential functions that vary slowly with the dimension; it can be used for the systems with more than one dimension only if the potential is symmetric and the equation can be transferred to a radial equation. The WKB method is used to estimate the eigenenergies of the system but as we mentioned above it has very limited applications. The next approximation is the perturbation theory. In this method a complex system is approximated by a simpler system for which we can find the exact eigenstates. But as the dimensions get smaller and/or the potential functions get more complicated these methods can't be applied to solve the equation; hence we resort to numerical methods to solve the equation.

2.2 Numerical solutions

In order to understand physical phenomena, scientists and engineers simulate and model the phenomena with mathematical models. These models can be simple algebraic formulas, ordinary differential equations, integral equations or partial differential equations. Since the mathematical models represent the physical phenomena, certain properties and characteristics of the physical phenomena can be explained and predicted by observing the model. In most cases there are no analytical solutions for the equations these models represent so they have to be solved numerically. Traditionally in pre-computer era scientists had used numerical analysis with pencil and paper. After the invention of computers, numerical analysis were rapidly and widely developed and continue developing. Nowadays CAD (computer aided design) tools use numerical methods to model and simulate a diverse range of phenomena. The main idea in numerical analysis is to numerically approximate unknown functions and solve equations such that there are upper bounds on errors.

Although numerical methods do not provide an exact solution for a given problem – only an approximation–, a good approximation is still very important since a lot of real world phenomena cannot be expressed exactly anyway. In computational electronics and mechanics, which are the fields of modeling and simulating novel semiconductor devices, two broad categories of numerical analysis are used for obtaining the solutions, namely mesh-based methods and meshless or point-base methods. In the rest of this chapter we will explain the two methods and compare them with each other.

2.2.1 Mesh methods

Various mesh methods, such as the finite element method (FEM) [19], the boundary element method (BEM) [20] or the finite difference element method (FDEM) [21], [22] are widely used in today's CAD tools to simulate advanced engineering systems. Among these techniques FEM is the most practical technique and there are lots of commercial packages such as ANSYS, ALGOR and LUSAS, which use FEM in their computations. In the mesh-based methods, the domain of the governing differential equation, which describes the system, is discretized into a mesh of discrete elements. For each element there is an approximated form of the principle equation. By using the connectivity information within the mesh and solving the simplified equations for each element, the solution to the governing partial differential equation is found from which the solution to the actual physical problem is obtained. Mesh-based methods as the name imply, concentrate on mesh generation. They require the generation of an elaborate mesh to perform the required numerical analysis. The quality of the mesh has the most influence in the quality of the solution. Mesh generation is expressed by dividing the physical domain into finite and smaller elements and connecting these elements to each other in order to discretize the whole domain. The generated mesh should be fine enough to get a more detailed solution; however generating too fine a mesh increases the computation time. In addition to the number of elements in a mesh, the shape of the elements also influences the accuracy of the calculation. Since the exact solutions are unavailable, the best way to find out if a generated mesh is accurate enough is to estimate the discretization error for the mesh, then re-mesh the domain and repeat the procedure until the desired accuracy is met. The mesh generating or re-meshing time is usually several times more than the processing time for solving the equations, specially in three dimensions and/or other complicated domains and this leads to the main weakness of the mesh-based methods.

In nano and micro engineering we usually deal with complex 3D structures. Generating the mesh, re-meshing the domain and obtaining the connectivity information between the elements for these structures are a demanding process. This and other similar problems have led to some alternative methods based on meshless analysis.

2.2.2 Meshless methods

Meshless methods sometimes referred to as point-based methods offer simple and efficient solutions for some of the most complex physical phenomena. Given the known problems with the mesh methods cited earlier, over the last few years, research into meshless methods has seen explosive growth and has moved the focus of research in modeling nano and micro devices from the more traditional mesh methods. In this thesis we use a meshless method to solve the Schrodinger equation and we focus on the application of meshless methods for partial differential equations. Meshless methods are performed as the three step process set out below [23], [24]:

Step1: Representation of the domain

For a given problem, first the domain of the problem (Ω) is represented by a set of scattered points(nodes), the density of which is controlled by the desired accuracy. Unlike the mesh methods, no connectivity information for the points is needed therefore the distribution of these points is easy and compared to the mesh generation in mesh methods, computation time is not excessive.

Step2: Interpolation

The second step is to interpolate the unknown function u at any point $\bar{x} = (x,y,z)$ in the domain (Ω). For this purpose a support domain (Ω_s) is defined for each node. A support domain is a small domain around any point $\bar{x}$ of the main domain whose nodes are used to interpolate the unknown function at the point $\bar{x}$. The size of the support domain (d_s), which determines the number of nodes in the support domain, is very important for the sake of accuracy of the interpolation. The dimension of the support domain (d_s) is defined by [24]

$$d_s = \alpha_s d_c \tag{2.42}$$

where d_c is the characteristic length which is the distance or the average distance of two neighbor nodes and α_s is the dimensionless size of the domain which is usually determined experimentally –it is usually between 2 to 3–. Now that we have the support domain around any node in the domain we interpolate the unknown function u as below

$$u(\bar{x}) = \sum_{i=1}^{n} \varphi_i(\bar{x}) u_i \tag{2.43}$$

Where n is the number of nodes in the support domain (Ω_s) ,$\varphi(\bar{x})$ is the interpolation(shape) function of the i_{th} node in the support domain and u_i is the value of unknown function at i_{th} node in the support domain. In meshless methods interpolation is a central step and lots of researches and developments are currently going on in this area. In the literature many improved and expanded techniques such as Reproducing Kernel approximation, Moving Least Squares method (MLS) and Point Interpolation Method(PIM) are used for the interpolating step. However generally speaking we can classify interpolation techniques to three broad categories as:

1. Finite integral representation techniques: such as Smoothed Particle Hydrodynamics (SPH) and Reproducing Kernel approximation

2. Finite series representation techniques: such as Moving Least Squares method (MLS) and Point Interpolation Method(PIM)

3. Finite differential representation techniques: such as Finite Difference Method (FDM)

In this thesis, to construct the shape functions, the fixed reproducing kernel technique is used, which is a finite integral representation method; we explain these methods in more detail below.

In these methods a function is represented in the form of

$$u^a(x) = \int_s W(x-s)u(s)ds \tag{2.44}$$

where W is the kernel weighting function. Choosing the proper kernel functions is an important key in the accuracy and coverage of the numerical results and there are some essential conditions that a proper kernel function must satisfy [24] (Note : x_k is the node that the kernel function is centered).

1. The kernel function should be zero outside the support domains.

$$W(x_k - x) = 0 \qquad \textit{outside the support domain} \tag{2.45}$$

2. For convenience, it is desired that the integral of the kernel function equals unity.

$$\int_\Omega W(x_k - s)ds = 1 \tag{2.46}$$

3. To satisfy the consistency condition, which is the ability of the approximation technique to reproduce the unknown function at each node in the domain, it is required that

the kernel function has the delta function behavior.

$$W(x) \longrightarrow \delta(x) \quad as \quad d_s \longrightarrow 0 \quad (d_s \ is \ the \ support \ domain \ size) \tag{2.47}$$

Considering the conditions above, different kinds of functions can be used as the weight function which we will explain three of them here [25]:

***Cubic spline function*:** In case of using cubic spline as a kernel function, we divide the support domain to n intervals by choosing $n+1$ points (x_i, y_i) such that the spline has to go through them, then for each interval i we define the cubic spline function as

$$S_i(x) = a_{i0} + a_{i1}x + a_{i2}x^2 + a_{i3}x^3 \qquad for \qquad i = 1 \cdots n \tag{2.48}$$

We need $4n$ equations to find the unknown coefficients of the cubic spline function. Due to the condition that the spline function is continuous, the two end points of each interval must satisfy the condition

$$\begin{aligned} S_i(x_i) &= y_i \\ S_i(x_{i+1}) &= y_{i+1} \qquad for \qquad i = 1 \cdots n \end{aligned} \tag{2.49}$$

It is also required that the first and second derivatives of the cubic spline function be continuous as well, so the interior points must satisfy the following conditions

$$\begin{aligned} S_i'(x_{i+1}) &= S_{i+1}'(x_{i+1}) \\ S_i''(x_{i+1}) &= S_{i+1}''(x_{i+1}) \qquad for \qquad i = 1 \cdots n-1 \end{aligned} \tag{2.50}$$

By satisfying conditions equation 2.49 and equation 2.50 we get $2n$ and $2(n-1)$ equations respectively. Now we need two more equations to be able to obtain the weighting function's coefficients. There are some standard ways in the literature to obtain these equations one of which is to set the second derivative of the cubic spline function to be zero at the endpoints of the support domain as follows [26].

$$S_0''(x_0) = S_{n-1}''(x_n) = 0 \tag{2.51}$$

With this calculation we built a cubic spline function that can be used as the kernel function for interpolating.

Quartic spline function **:** The quartic spline function can be built similar to the cubic spline function with the difference being that the polynomials are in order of four and also the third derivative of the function must be smooth and continuous as well as first and second derivatives.

Gaussian function **:** Gaussian function is the probability density function of normal (gaussian) distribution. Due to the fact that normal distribution is an extremely important class of distributions in statistics, and many analysts use this distribution in modeling and approximating the physical phenomena, the gaussian weight function plays a very important role in simulations. In general a gaussian function is in the form of

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp(\frac{-(x-\mu)^2}{2\sigma^2}) \qquad (2.52)$$

In the next chapter we will explain the gaussian function and how to choose its parameters in order to get a proper kernel approximation.

Step3: Discretization

Having the shape functions and their derivatives in hand, now we have to convert the complex partial differential equation to a linear system of algebraic equations. Employing a point-collocation on the PDE will discretize the PDE and convert the problem of solving the PDE to a matrix problem. There are different ways to approximate a function. When we approximate the unknown function with a discrete form, we lose a bit of accuracy but instead since we are working with a set of points and their interpolation functions, there is no need for the connectivity information between the points, in contrast to what we have in mesh-based methods, therefore calculations in meshless methods are less complicated.

2.2.3 state of art in meshless methods

The origin of meshless methods go back to 1977 when L. B. Lucy [27] first introduced SPH and then J.J. Monaghan [28] modified that. The Smoothed Particle Hydrodynamics (SPH) [6] method is an example of using a finite integral representation method to build shape functions. Lucy [27] (1977) and R. A. Gingold & J. J. Monaghan [29] (1977) initially and independently introduced SPH, which is a meshless, Lagrangian, particle method. The most significant advantage of SPH is that as a meshless method it doesn't need the mesh generation as well as the connectivity information, which are the major problems with FEM and other mesh-based methods. Another characteristic of SPH is the Lagrangian nature of the approximation. Since SPH is a Lagrangian method, the particles in SPH have physical meaning which distinguishes this method from other meshless methods in which particles are just points used to construct the interpolation functions. SPH was first used in modeling astrophysics phenomena however due to its properties, many scientists and researchers were interested in SPH and there was an ongoing active development on this method leading to more applications of this method. Although SPH can handle solving complex problems where traditional mesh-based methods have difficulties mentioned earlier, there are some shortcomings of this method such as inconsistency, instability and inaccuracy. Monaghan (1982)[28] explained SPH as a kernel approximation and improved the stability of SPH, however more developments were needed to improve the accuracy and consistency of the method. In SPH the approximated form of an unknown function $u(x)$ is given by

$$u^a(x) = \int_\Omega W(x-s,h)u(s)ds \tag{2.53}$$

Where $W(x-s,h)$ is the kernel function and h is the smoothing length. By adding a correction function to the kernel function in SPH and employing a Galerkin method for the discretization step, W. K. Liu and his colleges (1995) introduced a new method named reproducing kernel particle method (RKPM) [30], [7], [8], which improved the accuracy and consistency of SPH specially near the boundaries. The approximation in RKPM is the discretized form of the reproducing kernel approximation defined by

$$u^a(x) = \int_\Omega \overline{W}(x-s)u(s)ds \tag{2.54}$$

Where $\overline{W}(x-s)$ is the modified kernel function and is given by

$$\overline{W}(x-s) \; = \; C(x,s)W(x-s) \tag{2.55}$$

Where $C(x,s)$ is the correction function and $W(x-s)$ is the kernel function. The correction function is typically expressed in form of a linear combination of polynomial basis functions given by

$$C(x,s) = C_0(x) + C_1(x)(x-s) + C_2(x)(x-s)^2 + \cdots + C_n(x)(x-s)^n \tag{2.56}$$

in which $C_i(x)$'s are the correction coefficients and their number depends on the highest order of the derivative terms and also the number of variables in the governing partial differential equation. The correction coefficients are functions of x and they are found by satisfying the consistency conditions. By substituting the Taylor series expansion of function $u(s)$ around the point x in equation 2.54 and writing the consistency conditions for all the derivative terms in the governing partial differential equation we can determine the correction coefficients. The procedure is shown in detail in [7], [31].

Due to meshfree methods significant features, there has been an ongoing development in this area of numerical methods. As one of the most recent meshfree techniques we can name the diffuse element method (DEM), element free Galerkin (EFG) [9], [10], meshless local Petrov-Galerkin (MLPG) [11], least-squares meshfree method (LSMFM) and so forth. Nayroles, Touzot and Villon in 1992 introduced DEM where they used MLS approximation method for interpolation and combined it with Galerkin method to discretize and solve partial differential equations. In 1994 Belytschko and his colleagues modified DEM, specially in computing derivatives of shape functions and called it the EFG method. Since DEM, EFG and RKPM are all based on Galerkin method, they are not truly meshless and they need a background grid for the integrations. To overcome this problem Atluri and Zhu developed MLPG in 1998 which is similar to EFG in that both construct the shape functions using MLS approximation. The major difference between MLPG and EFG is in the integration step. In Galerkin based methods integration is done over the entire domain which calls for background grid, however, in MLPG a Petrov-Galerkin method is used and that reduces the integration to be done over a small local subdomain of each node. The latest improvement in meshless methods is replacing Galerkin method with a point collocation method to omit the integration step completely. In point collocation method only nodes are used to discretize the partial differential equation and we do not need any kind of background grid. G. Li and N.R. Aluru developed a new meshless method called finite cloud method (FCM) [12], [13] in which they combined fixed reproducing kernel technique with point collocation to solve partial differential equations. Due to the fact that FCM is a point collocation based method, it doesn't need any background mesh at any point, so it is a truly meshless method. This feature makes FCM an efficient and easy approach to solve many problems in computational mechanics. In original paper [12] FCM was introduced for one- and two- dimensions. In this thesis we expand FCM to three dimensional space (to our knowledge hasn't been done before) in which the formulations are more complex and the resulting matrices are very large. We then implement FCM –for one-/ two- /three- dimensions– and the needed eigenvalue solver in the Java programming language to solve the Schrodinger equation for nano structures (well/ wire/ box) in infinite barriers with arbitrary potential functions.

3 Theory of FCM and its implementation

3.1 Finite Cloud Method

As mentioned earlier the Finite Cloud Method (FCM) is a truly meshless method, which is originally developed by G. Li and N.R. Aluru [12], [13]. As a meshless method, the first step in FCM is to represent the physical domain Ω with a set of N_p distributed points. These scattered nodes can be distributed uniformly or non-uniformly over the entire domain. For simplicity it is preferred to have a uniform set of nodes but sometimes depending on the geometry of the system, stability of the solution and accuracy of the results we need to add /remove nodes to/from some areas. Due to the fact that FCM is a meshfree method, the case of non-distributed nodes can be easily dealt with FCM. For any node x in the domain Ω we define a local small domain called *a cloud* around it, which contains a number of nodes. The size of the cloud for the node x, which determines how many nodes are in the vicinity of x, is very important for the sake of accuracy and stability of the approximations. For more stability in our solution it is important to increase the number of points in the clouds; however this increase well make the results to be less accurate. On the other hand decreasing the number of points in the cloud will cause the singular matrix problem in our computation. We have to therefore optimize the cloud size in a balancing act to achieve an all around "reasonable" solution [32]. We use rectangular clouds by dividing each coordinate into a number of nodes. This way, we have an opportunity to increase or decrease the cloud size for any node if needed, which is a plus for our implementation specially if we improve our method using an adaptive algorithm. Similar to the support domain, the cloud size d_x, given by equation 3.1 is a factor of the distance between the neighbor nodes in the domain and α_x, the dimensionless factor, is also defined and optimized by experiments.

$$d_x = \alpha_x \Delta x \tag{3.1}$$

Now, after representing the domain Ω and defining the clouds around each node, we are ready to construct the interpolation functions for each node in Ω. For simplicity we first explain FCM in one dimensional space and then expand it to more dimensions. Having said that the difference between multi- and one- dimensional cases is mostly in constructing the shape functions, we start with explaining the interpolation for one- two- and three- dimensions in detail.

3.1.1 The fixed reproducing kernel approximation in one dimension

Fixed reproducing kernel technique approximates any unknown function $u(x)$ by

$$u^a(x) = \int_\Omega C(x,s)\varphi(x_k - s)u(s)ds \tag{3.2}$$

where $\varphi(x_k - s)$ is the kernel function centered at x_k, we redefine $\varphi(x_k - s)$ as

$$\varphi(x_k - s) = \frac{1}{d_x} W(\frac{x_k - s}{dx}) \tag{3.3}$$

Where d_x is the cloud size and W is the new kernel function. For the best approximation it is ideal to use Dirac delta function as the kernel function but due the problem that Dirac delta function is not suitable for the numerical analysis we use an approximated form of Dirac delta function for the weighting purpose such as spline, exponential, gaussian functions and etc. If we compare the fixed reproducing kernel approximation equation 3.2 with the reproducing kernel approximation equations 2.54 and 2.55, explained earlier, we see that the difference is in the kernel function, which is centered at node x_k for fixed reproducing kernel. This treatment has some advantages over other interpolation techniques, which we will explain later. $C(x,s)$ is the correction function given by

$$C(x,s) = P^T(s)\,C(x) \tag{3.4}$$

in which $P^T(s)$ is the vector of basis functions and $C^T(x)$ is the vector of correction function coefficients. As mentioned earlier the number of the basis functions and correction function coefficients depend on the highest order and also the number of the variables in the partial differential equation. In this thesis we will use FCM to solve the Schrodinger equation which is a second order partial differential equation, so in the rest of this matter we assume that the highest order of the partial differential equation is two hence in the 1D case the quadratic basis function vector is given by

$$P^T(s) = [1,\, s,\, s^2] \tag{3.5}$$

And the correction function coefficients vector is given by

$$C^T(x) = [c_0,\, c_1,\, c_2] \tag{3.6}$$

The correction function coefficients can be obtained by fulfilling the consistency conditions below

$$\int_{\Omega} P^T(s)C(x)\varphi(x_k - s)\ p_i(s)\,ds = p_i(x)\ \ for\ \ i = 1 \cdots 3 \tag{3.7}$$

The consistency condition in equation 3.7 can be rewritten in matrix form as

$$MC(x) = P(x) \qquad or \qquad C(x) = M^{-1}P(x) \tag{3.8}$$

in which M is the moment matrix and its elements are given by

$$M_{ij} = \int_{\Omega} p_i(s)\varphi(x_k - s)p_j(s)\,ds \qquad i,j = 1 \cdots 3 \tag{3.9}$$

As we notice in equation 3.9, due to the fact that the kernel function is centered at the point x_k, the elements of the moment matrix are not functions of x; hence the moment matrix in this approach is a constant matrix which is a plus for this method. As we will see, to discretize the Schrodinger equation we have to compute the partial derivatives of the interpolation functions, which requires the computation of partial derivatives of the moment matrix. Having a constant moment matrix makes these computations easier and faster.

If we define a $3 \times N_p$ matrix F as

$$F = \begin{pmatrix} p_1(x_1) & p_1(x_2) & \cdots & p_1(x_{N_p}) \\ p_2(x_1) & p_2(x_2) & \cdots & p_2(x_{N_p}) \\ p_3(x_1) & p_3(x_2) & \cdots & p_3(x_{N_p}) \end{pmatrix} \tag{3.10}$$

And a $N_p \times N_p$ diagonal matrix W as

$$W = \begin{pmatrix} \varphi(x_k - x_1)\Delta V_1 & 0 & \cdots & 0 \\ 0 & \varphi(x_k - x_2)\Delta V_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varphi(x_k - x_{N_p})\Delta V_{N_p} \end{pmatrix} \tag{3.11}$$

then we can write the moment matrix M as $M = F\,W\,F^T$. By substituting M in equation 3.8 and then using the auxiliary result to replace $C(x)$ in equation 3.4 we can rewrite the fixed reproducing kernel approximation (equation 3.2) as

$$\psi^a(x) = \int_\Omega P^T(x)M^{-1}P(s)\varphi(x_k - s)\psi(s)\,ds \tag{3.12}$$

The discrete form of equation 3.12 can be written as

$$\psi^a(x) = \sum_{I=1}^{N_p} N_I(x)U_I \tag{3.13}$$

where U_I is the nodal unknown and $N_I(x)$ is the fixed reproducing kernel interpolation (shape) function given by

$$N_I(x) = P^T(x)M^{-1}P(x_I)\varphi(x_k - x_I)\Delta V_I \tag{3.14}$$

in which V_I is the nodal volume associated with node I. Li and Aluru in [12] have shown that choosing the nodal volumes to unity will cause the shape functions to be identical with the ones constructed by least square methods. In addition to this feature, they showed that using either exact or unit nodal volumes will build almost identical interpolation functions, which is a great advantage of this method. Since it is difficult to find the exact nodal volume for complex geometries or when the distribution of points in the domain is not uniform, by having the explained feature we can simply use unity nodal volumes in difficult cases.

An important point that should be carefully considered here, is the issue of constructing multi-valued shape functions. As can be seen in equation 3.14 the interpolation functions constructed by fixed reproducing kernel technique are multi-valued. To clarify this issue in detail, recall the representation of the domain with the scattered nodes and the clouds around them. As mentioned earlier we center the kernel function at each point in the domain. Consider two points x_a and x_b and the clouds around each of them, respectively cloud a and cloud b. Assume that points a and b are in a position that their clouds overlap each other and so we have at least one node (node c) that belongs to both of the clouds. According to equation 3.14 we will get different shape functions at point c given by

$$N_{c-a}(x) = P^T(x)M^{-1}P(x_c)\varphi(x_a - x_c)\Delta V_c \tag{3.15}$$

$$N_{c-b}(x) = P^T(x)M^{-1}P(x_c)\varphi(x_b - x_c)\Delta V_c \tag{3.16}$$

However Li and Aluru have a solution for this problem [12]. By using a point collocation method they determined single-valued shape functions for the nodes. If we compute the interpolations at the node where the kernel function is centered, the shape functions will be limited to a single value. (for more detail see [12])

The derivatives of the interpolation functions can be computed from equation 3.14, due the fact that the moment matrix is a constant matrix, obtaining the derivatives of interpolation functions is a straightforward calculation. To solve the Schrodinger equation in 1D we need the second derivative of $N_I(x)$ i.e. $\frac{d^2N_I(x)}{dx^2}$, which is calculated as follows

$$P^T(s) = [1, x, x^2] \tag{3.17}$$

$$\frac{d\,P^T(x)}{dx} = [0,\ 1,\ 2x] \tag{3.18}$$

$$\frac{d^2\,P^T(x)}{dx^2} = [0,\ 0,\ 2] \tag{3.19}$$

$$\frac{d^2N_I(x)}{dx^2} = N_{I,xx}(x) = [0,0,2]M^{-1}P(x_I)\varphi(x_k - x_I)\Delta V_I \tag{3.20}$$

With the above procedure we are able to obtain the shape functions and their derivatives for all the nodes representing the domain.

3.1.2 The fixed reproducing kernel approximation in two dimensions

In the two dimensional space, which is explained in detail in [12], the domain Ω is represented by $N_p = N_x \times N_y$ points, where N_x and N_y are the number of nodes in each dimension. Assuming $\bar{x} = (x,y)$ and $\bar{r} = (r,s)$ as the space position vectors and $d\bar{r} = dr\,ds$, the fixed reproducing approximation is stated as

$$\psi^a(\bar{x}) = \int_\Omega \zeta(x,y,,r,s)\varphi(\bar{x}_k - \bar{x})\psi(\bar{r})d\bar{r} \tag{3.21}$$

$\varphi(\bar{x}_k - \bar{x})$ which is the multi-dimensional kernel is defined by the products of one-dimensional kernel functions as below:

$$\varphi(\bar{x}_k - \bar{x}) = \frac{1}{d_x} W(\frac{x_k - x}{dx}) \frac{1}{d_y} W(\frac{y_k - y}{dy}) \tag{3.22}$$

$\zeta(x,y,r,s)$, the correction function is given by

$$\zeta(x,y,r,s) = P^T(\bar{r})C(\bar{x}) \tag{3.23}$$

For $2nd$ order partial differential equations with two variables the quadratic basis function $P^T(\bar{r}) = P^T(r,s)$ is defined by

$$P^T(\bar{r}) = [1,\ r,\ s,\ rs,\ r^2,\ s^2] \tag{3.24}$$

The vector of correction function coefficients, $C(\bar{x})$ is determined by satisfying the reproducing condition given by

$$\int_\Omega P^T(\bar{r})C(\bar{x})\varphi(\bar{x}_k - \bar{r})\ p_i(\bar{r})\,d\bar{r} = p_i(\bar{x})\ \ for\ \ i = 1\cdots 6 \tag{3.25}$$

We rewrite equation 3.25 in matrix form as below:

$$MC(\bar{x}) = P(\bar{x}) \qquad or \qquad C(\bar{x}) = M^{-1}P(\bar{x}) \tag{3.26}$$

where the entries of the matrix M are obtained by

$$M_{ij} = \int_{\Omega} p_i(\bar{r})\varphi(\bar{x}_k - \bar{r})p_j(\bar{r})\,d\bar{r} \qquad i,j = 1\cdots 6 \tag{3.27}$$

The matrix F in this section is defined as below:

$$F = \begin{pmatrix} p_1(\bar{x}_1) & p_1(\bar{x}_2) & \cdots & p_1(\bar{x}_{N_p}) \\ p_2(\bar{x}_1) & p_2(\bar{x}_2) & \cdots & p_2(\bar{x}_{N_p}) \\ \vdots & \vdots & \ddots & \vdots \\ p_6(\bar{x}_1) & p_6(\bar{x}_2) & \cdots & p_6(\bar{x}_{N_p}) \end{pmatrix} \tag{3.28}$$

Because of having 6 elements in the basis function vector equation 3.24, the matrix F in a two dimensions case is a $6 \times N_p$ matrix.

And also the $N_p \times N_p$ diagonal matrix W is defined by

$$W = \begin{pmatrix} \varphi(\bar{x}_k - \bar{x}_1)\Delta V_1 & 0 & \cdots & 0 \\ 0 & \varphi(\bar{x}_k - \bar{x}_2)\Delta V_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varphi(\bar{x}_k - \bar{x}_{N_p})\Delta V_{N_p} \end{pmatrix} \tag{3.29}$$

By defining M as $M = F\,W\,F^T$ and performing the needed mathematical calculations, we will extract both the continuous and discrete approximated forms of the unknown function as below:

$$\psi^a(\bar{x}) = \int_{\Omega} P^T(\bar{x})M^{-1}P(\bar{r})\varphi(\bar{x}_k - \bar{r})\psi(\bar{r})\,d\bar{r} \tag{3.30}$$

$$\psi^a(\bar{x}) = \sum_{I=1}^{N_p} N_I(\bar{x}) U_I \tag{3.31}$$

Where the shape function, $N_I(\bar{x}) = N_I(x,y)$, is given by

$$N_I(\bar{x}) = P^T(\bar{x}) M^{-1} P(\bar{x}_I) \varphi(\bar{x}_k - \bar{x}_I) \Delta V_I \tag{3.32}$$

Adopting the technique explained in [12] to avoid the multi-valued function problem, here the shape functions are computed at the node where the kernel function is centered. Recalling the Schrodinger equation with two variables, we need the second order derivatives of the shape functions as below

$$\frac{\partial^2}{\partial x^2} N_I(\bar{x}) \ , \ \frac{\partial^2}{\partial y^2} N_I(\bar{x}) \tag{3.33}$$

Considering the basis function as $P^T(\bar{x}) = [1, x, y, xy, x^2, y^2]$, the required derivatives are obtained as below:

$$\begin{aligned} \tfrac{\partial^2}{\partial x^2} N_I(\bar{x}) &= N_{I,xx}(\bar{x}) \\ &= \frac{\partial^2 P^T(\bar{x})}{\partial x^2} M^{-1} P(\bar{x}_I) \varphi(\bar{x}_k - \bar{x}_I) \Delta \\ &= [0,0,0,0,2,0] M^{-1} P(\bar{x}_I) \varphi(\bar{x}_k - \bar{x}_I) \Delta V_I \end{aligned} \tag{3.34}$$

$$\begin{aligned} \tfrac{\partial^2}{\partial y^2} N_I(\bar{x}) &= N_{I,yy}(\bar{x}) \\ &= \frac{\partial^2 P^T(\bar{x})}{\partial y^2} M^{-1} P(\bar{x}_I) \varphi(\bar{x}_k - \bar{x}_I) \Delta \end{aligned}$$

$$= [0,0,0,0,0,2]M^{-1}P(\bar{x}_I)\varphi(\bar{x}_k-\bar{x}_I)\Delta V_I$$

(3.35)

In sections 3.1.1 and 3.1.2 the fixed reproducing kernel approximation for one- and two-dimensional space were detailed. In next section we will expand the fixed reproducing kernel procedure to a three-dimensions case.

3.1.3 The fixed reproducing kernel approximation in three dimensions

In the 3D case we represent the domain Ω by $N_p = N_x \times N_y \times N_z$ points, where N_x, N_y and N_z are the number of nodes in each dimension. Considering $\bar{x} = (x,y,z)$ and $\bar{r} = (r,s,t)$ as the space position vectors and $d\bar{r} = dr\,ds\,dt$, the fixed reproducing approximation is given by

$$\psi^a(\bar{x}) = \int_\Omega \zeta(x,y,z,r,s,t)\varphi(\bar{x}_k-\bar{x})\psi(\bar{r})d\bar{r} \tag{3.36}$$

where the multi-dimensional kernel function, $\varphi(\bar{x}_k-\bar{x})$, is defined by:

$$\varphi(\bar{x}_k-\bar{x}) = \frac{1}{d_x}W(\frac{x_k-x}{dx})\frac{1}{d_y}W(\frac{y_k-y}{dy})\frac{1}{d_z}W(\frac{z_k-z}{dz}) \tag{3.37}$$

Similar to the routine explained in 3.1.2 we extract the formulas for a 3D case. $\zeta(x,y,z,r,s,t)$, the correction function is given by

$$\zeta(x,y,z,r,s,t) = P^T(\bar{r})C(\bar{x}) \tag{3.38}$$

For a $2nd$ order partial differential equation in 3D we define the quadratic basis function $P^T(\bar{r}) = P^T(r,s,t)$ by

$$P^T(\bar{r}) = [1,\, r,\, s,\, t,\, rs,\, rt,\, st,\, r^2,\, s^2,\, t^2] \tag{3.39}$$

The vector of correction function coefficients, $C(\bar{x})$ is determined by satisfying the reproducing condition given by

$$\int_{\Omega} P^T(\bar{r})C(\bar{x})\varphi(\bar{x}_k - \bar{r})\; p_i(\bar{r})\,d\bar{r} = p_i(\bar{x}) \;\; for \;\; i = 1\cdots 10 \tag{3.40}$$

The matrix form of equation 3.40 is given by

$$MC(\bar{x}) = P(\bar{x}) \qquad or \qquad C(\bar{x}) = M^{-1}P(\bar{x}) \tag{3.41}$$

where the matrix elements are obtained by

$$M_{ij} = \int_{\Omega} p_i(\bar{r})\varphi(\bar{x}_k - \bar{r})p_j(\bar{r})\,d\bar{r} \qquad i,j = 1\cdots 10 \tag{3.42}$$

The matrices F and W in 3D are defined as follows.

$$F = \begin{pmatrix} p_1(\bar{x}_1) & p_1(\bar{x}_2) & \cdots & p_1(\bar{x}_{N_p}) \\ p_2(\bar{x}_1) & p_2(\bar{x}_2) & \cdots & p_2(\bar{x}_{N_p}) \\ \vdots & \vdots & \ddots & \vdots \\ p_{10}(\bar{x}_1) & p_{10}(\bar{x}_2) & \cdots & p_{10}(\bar{x}_{N_p}) \end{pmatrix} \tag{3.43}$$

Since the vector of basis function in 3D case has 10 elements instead of the 6 elements we have in 3.1.2, in this section the matrix F is a $10 \times N_p$ matrix.

And the $N_p \times N_p$ diagonal matrix W is given by

$$W = \begin{pmatrix} \varphi(\bar{x}_k - \bar{x}_1)\Delta V_1 & 0 & \cdots & 0 \\ 0 & \varphi(\bar{x}_k - \bar{x}_2)\Delta V_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \varphi(\bar{x}_k - \bar{x}_{N_p})\Delta V_{N_p} \end{pmatrix} \tag{3.44}$$

Repeating the calculation done in 3.1.2 will result in the approximated forms (continuous and discrete) of the unknown function as

$$\psi^a(\bar{x}) = \int_\Omega P^T(\bar{x}) M^{-1} P(\bar{r}) \varphi(\bar{x}_k - \bar{r}) \psi(\bar{r})\, d\bar{r} \tag{3.45}$$

$$\psi^a(\bar{x}) = \sum_{I=1}^{N_p} N_I(\bar{x}) U_I \tag{3.46}$$

Where the shape function, $N_I(\bar{x}) = N_I(x,y,z)$, is given by

$$N_I(\bar{x}) = P^T(\bar{x}) M^{-1} P(\bar{x}_I) \varphi(\bar{x}_k - \bar{x}_I) \Delta V_I \tag{3.47}$$

Here again to avoid the multi-valued function problem we compute the shape functions at the node where the kernel function is centered. In the 3D case we have three variables in our governing partial differential equation. Recalling the Schrodinger equation we need the second order derivatives of the shape functions as below

$$\frac{\partial^2}{\partial x^2}N_I(\bar{x}) \ , \ \frac{\partial^2}{\partial y^2}N_I(\bar{x}) \ , \ \frac{\partial^2}{\partial z^2}N_I(\bar{x}) \tag{3.48}$$

Considering the basis function

$$P^T(\bar{x}) = [1, x, y, z, xy, xz, yz, x^2, y^2, z^2] \tag{3.49}$$

and following the calculation in 3.1.2 will result in the required derivatives as

$$\begin{aligned}&\frac{\partial^2}{\partial x^2}N_I(\bar{x}) = N_{I,xx}(\bar{x}) = \\ &\qquad [0,0,0,0,0,0,0,2,0,0]M^{-1}P(\bar{x}_I)\varphi(\bar{x}_k - \bar{x}_I)\Delta V_I\end{aligned} \tag{3.50}$$

$$\begin{aligned}&\frac{\partial^2}{\partial y^2}N_I(\bar{x}) = N_{I,yy}(\bar{x}) = \\ &\qquad [0,0,0,0,0,0,0,0,2,0]M^{-1}P(\bar{x}_I)\varphi(\bar{x}_k - \bar{x}_I)\Delta V_I\end{aligned} \tag{3.51}$$

$$\begin{aligned}&\frac{\partial^2}{\partial z^2}N_I(\bar{x}) = N_{I,zz}(\bar{x}) = \\ &\qquad [0,0,0,0,0,0,0,0,0,2]M^{-1}P(\bar{x}_I)\varphi(\bar{x}_k - \bar{x}_I)\Delta V_I\end{aligned} \tag{3.52}$$

So far we have obtained the shape functions and their required derivative for solving the Schrodinger equation with finite cloud method (FCM), in one-, two- and three- dimensions. Now we are ready to go to the next step which is discretizing the governing partial differential equation using a point collocation method.

3.1.4 Collocation

In FCM a point collocation method [13], [33] is used to discretize the partial differential equation and obtain the solution. An alternative for point collocation method would be to use the Galerkin method [34]. The latter is more accurate and has various applications in the literature such as element free galerkin method (EFG) and petrov galerkin diffuse element method (PG DEM). But due to the fact that the Galerkin approach is based on integrating over the intervals, it needs more computation time as well as connectivity information. In contrast, a point collocation method doesn't need any background grid; the main concept here is meeting the governing partial differential equation for each point in the represented domain. This feature makes it easy to enforce the boundary conditions in point collocation method. Suppose we have N_i interior nodes, N_d boundary nodes with Dirichlet boundary conditions and N_n nodes with Neumann boundary conditions in the domain Ω, where $N_p = N_i + N_d + N_n$. Taken from [18] here are the definitions of Dirichlet and Neumann boundary conditions. "In mathematics, a Dirichlet boundary condition (often referred to as a first-type boundary condition) imposed on an ordinary differential equation or a partial differential equation specifies the values a solution is to take on the boundary of the domain", "In mathematics, a Neumann boundary condition (named after Carl Neumann and also referred to as a third-type boundary condition) imposed on an ordinary differential equation or a partial differential equation specifies the values the derivative of a solution is to take on the boundary of the domain". In point collocation approach the interior nodes must satisfy the governing partial differential equation and the boundary nodes must meet the boundary conditions, so if we consider the general form of a partial differential equation given by

$$G u(\bar{x}) = f(\bar{x}) \tag{3.53}$$

Where G is the differential operator, then the collocation equations are given by

$$G u^a(\bar{x}_i) = f(\bar{x}_i) \qquad i = 1 \cdots N_i \qquad (interior\, nodes) \tag{3.54}$$

$$u^a(\bar{x}_i) = D(\bar{x}_i) \qquad i = 1 \cdots N_d \qquad (Dirichlet\, boundary\, nodes) \tag{3.55}$$

$$\frac{\partial u^a(\bar{x}_i)}{\partial n} = N(\bar{x}_i) \qquad i = 1 \cdots N_n \qquad (Neumann\, boundary\, nodes) \tag{3.56}$$

Substituting the discrete form of unknown function $u^a(\bar{x})$ in equations above will lead us to a matrix problem. Depending on the governing equation and the boundary conditions we will obtain different types of matrix problems and for each type there is a specific approach of solution.

3.2 Applying finite cloud method (FCM) on Schrodinger equation

For a one-electron quantum structure obtaining the eigenstates (eigenenergies and eigenfunctions) reduces to solving the one-electron time-independent Schrodinger equation. The purpose of this thesis is to find the eigenstates of nano-scale quantum structures by employing FCM to solve the Schrodinger equation.

As mentioned earlier in the technical information chapter, one of the essential conditions to have the desired accuracy and coverage using a meshless method is that the kernel weighting function, integrals to unity (equation 2.46). Therefore in this matter we have used a normalized Gaussian function as the kernel function to perform the interpolation step. The general form of a normalized Gaussian function is given by equation 2.52, where μ is the mean value and σ^2 is the variance of the function. The Gaussian function is a bell-shaped function which is symmetric around its mean value so we chose the mean value to be zero. Later when we center the kernel function at each node in Ω, having a zero mean value makes it easier to show the node where the kernel function is centered at. The other parameter in equation 2.52 is variance, sometimes referred to smoothing parameter. Due to the fact that variance controls the width of the function, it is a very important factor in controlling the accuracy of the approach. Here we wanted to mimic the Dirac delta function so we want to have σ^2 as small as we can. Using trial and error we chose $\sigma^2 = 0.1$ in our code, if σ^2 gets smaller we may face the singular matrix problem at some point and if it gets larger we won't get the desired accuracy so with this introductory explanation the chosen one dimensional kernel function is given by

$$f(x) = 1.2615 \exp(\frac{-x^2}{0.2}) \tag{3.57}$$

This kernel function is centered at every point in the domain Ω to weight the nodes that are in the cloud of the center point $\bar{x}_k$.

Another issue that we should discuss here is the cloud size. Once again we resort to trial and error to find the dimensionless factor in equation 3.1 ($\alpha_x = 1.02$). We chose to have three nodes in each cloud so while obtaining the desired accuracy of the results we won't face the singular matrix problem.

3.2.1 Quantum well

Consider a quantum well with infinite barriers and the width l_x. Because of the infinite potential barriers, the wavefunction $\psi(\bar{x})$ vanishes outside the well, hence the domain Ω is given by

$$\Omega : \quad 0 < x < l_x \tag{3.58}$$

The potential function inside the well can be any arbitrary function $V(x)$. To represent the infinite barriers we apply the Dirichlet boundary conditions at the boundary nodes as follows

$$\psi(0) = \psi(l_x) = 0 \tag{3.59}$$

For a one-electron quantum structure obtaining the eigenstates (eigenenergies and eigenfunctions) reduces to solving the one-electron time-independent Schrodinger equation. The time-independent Schrodinger equation for a one electron Q-well is given by equation 3.60.

$$-\frac{\hbar^2}{2m^*}\frac{d^2\psi(x)}{dx^2} + V(x)\psi(x) = E\psi(x) \tag{3.60}$$

E is the eigenenergy and $\psi(x)$ is the wave function of the electron. We represent the domain Ω by N_p uniformly distributed nodes.

$$\Delta_x = \frac{l_x}{N_p - 1} \tag{3.61}$$

Δ_x is the distance between two neighbor nodes. In our code $N_p = 25$. The quadratic basis function is given by equation 3.5. If we follow the procedure in section 3.1.1 we will get the approximated wavefunction, shape functions and the derivatives of the shape functions as shown in equations 3.12, 3.13, 3.14 and 3.20. Recalling the operator form of the schrodinger equation 2.28, the collocation equations can be written as

$$H\psi^a(x_i) = E\psi^a(x_i) \qquad i = 1 \cdots N_i \tag{3.62}$$

$$\psi^a(x_i) = 0 \qquad i = 1 \cdots N_d \tag{3.63}$$

By enforcing boundary conditions (equation 3.63) in equation 3.62, solving the schrodinger equation reduces to the eigenvalue equation given by

$$\widehat{H}\widehat{\psi} = E\widehat{\psi} \tag{3.64}$$

where $\widehat{H}$ is a submatrix of H and $\widehat{\psi}$ is the vector of the interior nodal unknowns.

3.2.2 Quantum wire

In this section we study a Q-wire with infinite barriers and the potential function $V(x,y)$ inside the wire; we follow the procedure in section 3.2.1, except the changes due to the added dimension. Assume $\bar{x} = (x,y)$ is the vector of position. The 2D domain Ω is given by

$$\begin{aligned} \Omega \;:\; & 0 < x < l_x \\ & 0 < y < l_y \end{aligned} \tag{3.65}$$

where l_x and l_y are the lengths of the Q-wire in coordinates x and y. The Schrodinger equation and the boundary conditions for a quantum wire are given by

$$-\frac{\hbar^2}{2m^*}\left(\frac{d^2\psi(\bar{x})}{dx^2} + \frac{d^2\psi(\bar{x})}{dy^2}\right) + V(\bar{x})\psi(\bar{x}) = E\psi(\bar{x}) \tag{3.66}$$

$$\psi(0,y) = \psi(l_x,y) = \psi(x,0) = \psi(x,l_y) \tag{3.67}$$

We represent the domain by $N_p = N_x \times N_y$ uniformly distributed nodes, where N_x and N_y are the number of points in coordinate x and y respectively (in our code $N_x = N_y = 16$)

$$\begin{aligned} \Delta_x &= \frac{l_x}{N_x - 1} \\ \Delta_y &= \frac{l_y}{N_y - 1} \end{aligned} \tag{3.68}$$

Δ_x and Δ_y are the distances between two neighbor nodes in direction x and y. The quadratic basis function is given by

$$P^T(\bar{x}) = [1, x, y, xy, , x^2, y^2] \tag{3.69}$$

If we follow the procedure in section 3.2.1 we will get the collocation equations as below.

$$H\psi^a(x_i,y_i) = E\psi^a(x_i,y_i) \qquad i = 1\cdots N_i \tag{3.70}$$

$$\psi^a(x_i,y_i) = 0 \qquad i = 1\cdots N_d \tag{3.71}$$

And finally the eigenvalue equation is obtained as

$$\widehat{H}\widehat{\psi} = E\widehat{\psi} \tag{3.72}$$

3.2.3 Quantum box

We consider a Q-box with infinite potential barriers. Once again we repeat the same procedure we have done earlier for Q-well and Q-wire with the difference that we now have to develop the 2D fixed reproducing kernel to a 3D technique. Due to this change the size of the matrices will be a lot larger and the formulations will be more complicated than the two first cases. Assume $\bar{x} = (x,y,z)$ is the vector of position. The 3D domain Ω for a Q-box is given by

$$\begin{aligned} \Omega \ : \ & 0 < x < l_x \\ & 0 < y < l_y \\ & 0 < z < l_z \end{aligned} \tag{3.73}$$

where l_x, l_y and l_z are the lengths of the Q-box in each coordinates. The Schrodinger equation and the boundary conditions in this case is given by

$$-\frac{\hbar^2}{2m^*}\nabla^2\psi(\bar{x}) + V(\bar{x})\psi(\bar{x}) = E\psi(\bar{x}) \tag{3.74}$$

$$\begin{aligned} &\psi(0,y,z) = \psi(x,0,z) = \psi(x,y,0) = \\ &\psi(l_x,y,z) = \psi(x,l_y,z) = \psi(x,y,l_z) = 0 \end{aligned} \tag{3.75}$$

The total number of nodes in this case is given by $N_p = N_x \times N_y \times N_z$ where N_x, N_y and N_z are the number of points in coordinate x, y and z respectively (in our code we have chosen $N_x = N_y = N_z = 14$). Note that the total number of points in 3D case is extensively large and this requires excessive time and memory for the computation. For instance if we have 10 nodes in each coordinate then $N_p = 1000$ and we obtain a matrix with an order of 1000 which consumes huge computing resources –both in time and memory– to do the operations on this matrix. In our implementation to overcome this problem, we have used a sub-matrix of the main matrix to find the eigenstates which reduces the substantial computing time and memory, however compared to the 1D and 2D cases the required computing time and memory is still considerable. Recalling the formulas in 3.1.3 and following the procedure done in 3.2.1 and 3.2.2 we have

$$\begin{aligned}
\Delta_x &= \frac{l_x}{N_x - 1} \\
\Delta_y &= \frac{l_y}{N_y - 1} \\
\Delta_z &= \frac{l_z}{N_z - 1}
\end{aligned} \tag{3.76}$$

Δ_x and Δ_y and Δ_z are the distances between two neighbor nodes in direction x, y and z. The resulting collocation equations are as follows

$$H\psi^a(\bar{x}_i) = E\psi^a(\bar{x}_i) \qquad i = 1 \cdots N_i \tag{3.77}$$

$$\psi^a(\bar{x}_i) = 0 \qquad i = 1 \cdots N_d \tag{3.78}$$

Using equation 3.78 and the discrete form of $\psi^a(\bar{x})$ in equation 3.77 will lead us to a eigenvalue problem

$$\widehat{H}\widehat{\psi} = E\widehat{\psi} \tag{3.79}$$

where $\widehat{H}$ is a submatrix of H and $\widehat{\psi}$ is the vector of the interior nodal unknowns.

In sections 3.2.1 to 3.2.3 we explained how to discretize the Schrodinger equation for a quantum well/ wire/ box and extract the eigenvalue equation. The final step is obtaining the eigenvalues and the corresponding eigenstates by solving the eigenvalue equation, which is done the same way for the three cases by our code. As mentioned earlier Matrix H is a large matrix specially for the 3D cases. To decrease the order of matrix H, we omit the unimportant rows of matrix H (the rows corresponding to the boundary nodes) and obtain the sub-matrix $\widehat{H}$. This makes the order of matrix $\widehat{H}$ a lot smaller than matrix H and that conserves some computing time and memory space. Then using the matrix operations we find the first five eigenvalues, which are the first five energy states of the Q- structure, and then with some more computation we obtain the wavefunction corresponding to the resulting energy states.

Implementing our program, we had to use very large matrices and operating on such large matrices required huge memory space as well as excessive computing time. As a practical solution we needed an algorithm that would do the computations in the most effective way. We therefore used the Java programming language to implement our scheme presented in appendix D. Java is a modern, object oriented, garbage collected language which offers a lot of flexibility to customize the code in many various ways. Specially when dealing with large matrices, the garbage collecting feature was very useful and a must. For the linear algebra operations we used the Jama package [35] which is a Java matrix package.

To demonstrate the validity of our approach we have applied our program on different quantum structures and compare the results with the theoretical results. In the next chapter we analyze the numerical results and discuss their validity.

4 Numerical Results

In order to prove the validity of our methodology, below we analyze some numerical examples. In our examples we consider one-electron infinite rectangular GaAs quantum structures with parameters defined in Table 4.1.

Quantum well structures in GaAs semiconductor

We consider two quantum wells with the parameters given in columns E and F of Table 4.1. The Schrodinger equation for an infinite well with width l_x is given by

$$-\frac{\hbar^2}{2m^*}\frac{\partial^2\psi(x)}{\partial x^2}+V(x)\psi(x)=E\psi(x) \tag{4.1}$$

$V(x)$, the potential function is ∞ outside the well and equals to V_0 inside the well. Because of having infinite barriers, the wavefunction must vanish outside the well so the Dirichlet conditions are as below.

$$\psi(0)=\psi(l_x)=0 \tag{4.2}$$

Defining $k=\sqrt{\frac{2m^*(E-V_0)}{\hbar^2}}$ and replacing $V(x)$ by V_0, the Schrodinger equation inside the well is as follows.

$$\frac{\partial^2\psi(x)}{\partial x^2}+k\psi(x)=0 \tag{4.3}$$

The general solution equation 4.3 is given by

$$\psi(x)=A\,\exp(ikx)+B\,\exp(-ikx) \tag{4.4}$$

Using equation 4.2 we have

$$\psi(0) = A + B = 0 \qquad \Rightarrow \qquad B = -A \tag{4.5}$$

and

$$\begin{aligned}
\psi(l_x) &= A\exp(ikl_x) + B\exp(-ikl_x) = 0 \\
&\Rightarrow A\sin(kl_x) = 0 \\
&\Rightarrow kl_x = n\pi \\
&\Rightarrow \sqrt{\frac{2m^*(E-V_0)}{\hbar^2}}\, l_x = n\pi \\
&\Rightarrow E = V_0 + \frac{\hbar^2\pi^2}{2m^*a^2}n^2 \qquad n = 1, 2, \cdots
\end{aligned} \tag{4.6}$$

The amplitude of the wavefunction is obtained by satisfying the normalization condition (equation 2.26) as below:

$$\begin{aligned}
&\textstyle\int_{-\infty}^{+\infty} \psi(x)\,\psi^*(x)\,dx = 1 && \Rightarrow \qquad (4.7)\\
&\textstyle\int_{-\infty}^{+\infty} 4A^2\sin^2(kx)\,dx = 1 && \Rightarrow \\
&\textstyle\int_{-\infty}^{+\infty} 4A^2\sin^2(\frac{n\pi x}{l_x})\,dx = 1 && \Rightarrow \\
&A = \sqrt{\tfrac{1}{2l_x}} && \Rightarrow
\end{aligned}$$

$$\psi(x) = \sqrt{\tfrac{2}{l_x}}\,\sin(\tfrac{n\pi}{l_x}x) \qquad for\, n = 1, 2, \cdots \tag{4.8}$$

Maple, a mathematics software tool, is employed to calculate the exact energies using formula 4.6 as well as to plot the exact wavefunctions using equation 4.8. We need to import the data of the wavefunctions obtained by running our code to Maple in order to plot the resulting wavefunctions. The Maple code for the quantum well is given in Appendix A.

The exact calculated energy levels, calculated in Appendix A, for the given wells are shown in columns B and E of Table 4.2 and the obtained results from running our code are shown in columns A and D of the same Table. Likewise in Figures 4.1 the wavefunctions of the second and forth energy states of well-I and in figure 4.2 the wavefunctions of the first and third energy states of well-II are illustrated. Comparing the results shown in Table 4.2 as well as comparing figures 4.1(a) with 4.1(b) and 4.2(a) with 4.2(b) we see that our results are in an excellent agreement with the exact theoretical results.

A quantum well with an applied Electric Field

In [36], B. V. Zeghbroeck describes an approximated solution for a Quantum well with an applied uniform electric field. It is shown that if E_{n0} is the n_{th} energy level of a Quantum well without the presence of an electric field then E_n, the Energy levels of the quantum well with an applied electric field (F), is approximated by equation 4.9. It is also stated that the equation 4.9 is a good approximation if $En \gg eFl_x$.

$$E_n = E_{n0} + \frac{eFl_x}{2} \qquad n_x = 1, 2, 3, \cdots \tag{4.9}$$

Here we have applied two electric fields of $5 \times 10^8 V/m$ and $5 \times 10^7 V/m$ to the well-I described in table 4.1. Using equation 4.9 and the data for E_{n0} given in column B of Table 4.2, we have calculated the energy levels of the quantum well-I in presence of the applied electric field. The calculated results are shown in columns B and E of Table 4.3. We have also run our code for the same well with the potential function $V(x) = V_0 + eFx$. The obtained results are shown in columns A and D of Table 4.3. Comparing the two sets of results validates the applicability of our approach.

Quantum wire structures in GaAs semiconductor

For the quantum wire structure, the two wires described in columns C and D of Table 4.1 are considered. According to quantum theory, the eigenenergies and eigenfunctions of an infinite quantum wire when variables are separable can be obtained as follows:

In case of having separable variables the wavefunction of the system can be written as

$$\psi(x,y) = \psi(x)\,\psi(y) \tag{4.10}$$

And also the potential function can be written as

$$V(x,y) = V(x) + V(y) \tag{4.11}$$

For a quantum wire with infinite barriers the Schrodinger equation inside the wire is given by

$$-\frac{\hbar^2}{2m^*}\left(\frac{\partial^2\psi(x,y)}{\partial x^2} + \frac{\partial^2\psi(x,y)}{\partial y^2}\right) + V(x,y)\psi(x,y) = E\psi(x,y) \tag{4.12}$$

If we substitute equation 4.10 and 4.11 in the Schrodinger equation 4.12, we'll have:

$$-\frac{\hbar^2}{2m^*}\left(\frac{\partial^2\psi(x)\psi(y)}{\partial x^2} + \frac{\partial^2\psi(x)\psi(y)}{\partial y^2}\right) + (V(x)+V(y))\psi(x)\psi(y) = (E_x+E_y)\psi(x)\psi(y) \tag{4.13}$$

By dividing the two sides of equation 4.13 by $\psi(x)\psi(y)$ the two-dimensional Schrodinger equation can be divided into two one-dimensional equations as below.

$$-\frac{\hbar^2}{2m^*}\frac{\partial^2\psi(x)}{\partial x^2}+V(x)\psi(x)=E_x\psi(x) \tag{4.14}$$

$$-\frac{\hbar^2}{2m^*}\frac{\partial^2\psi(y)}{\partial y^2}+V(y)\psi(y)=E_y\psi(y) \tag{4.15}$$

Where E_x is the eigenenergy in x direction and E_y is the eigenenergy in y direction and $E = E_x + E_y$ is the total energy levels of the wire. We solve equations 4.14 and 4.15 the same way we did for the quantum well and by using equations 4.10 and the fact that the total energy of the Quantum wire is given by $E = E_x + E_y$, we obtain the total energy levels as well as the wavefunction of the quantum wire as below.

$$E_n = V_0 + \frac{\pi^2\hbar^2}{2m^*}\left(\frac{n_x^2}{L_x^2}+\frac{n_y^2}{L_y^2}\right) \qquad n_x, n_y = 1,2,3,\cdots \tag{4.16}$$

$$\psi_n(\bar{x}) = \sqrt{\frac{4}{L_xL_y}}sin\left(\frac{\pi n_x x}{L_x}\right)sin\left(\frac{\pi n_y y}{L_y}\right) \qquad n_x, n_y = 1,2,3,\cdots \tag{4.17}$$

where V_0 is the constant potential function inside the wire. The Maple code for calculating the exact energy levels and plotting the wavefunctions of the given quantum wires with using formula 4.16 and 4.17 respectively, is shown in Appendix B. The expected theoretical eigenvalues and the ones obtained from running our program are shown in Table 4.4. Also in Figures 4.3 and 4.4 some wavefunctions of the given wires are illustrated. The validity of our approach is confirmed by comparing the results obtained by the two methods, given in Table 4.4 on the one hand and by comparing the figures 4.3(a) with 4.3(b) and 4.4(a) with 4.4(b) on the other hand.

A quantum wire with an applied Electric Field

In this example we analyzed the change in the energy levels of a nanowire in the presence of an applied electric field. If we expand Zeghbroeck's approximated solution for a Quantum well in an applied electrical field [36] to an approximated solution for a nanowire in an applied electrical field, the change in energy levels is calculated as

$$E_n - E_{n0} = \frac{eF_x l_x}{2} + \frac{eF_y l_y}{2} \qquad n = 1,2,3,\cdots \tag{4.18}$$

where F_x and F_y are the applied electric fields in directions x and y. Here we have applied electric fields of $F_x = F_y = 5 \times 10^7 V/m$ to the wire-I described in table 4.1. We have calculated the energy levels of the quantum wire-I in the presence of the applied electric field using Zeghbroeck's method. The calculated results are shown in column B of Table 4.5. The obtained results from running our code for the same wire with the potential function $V(x,y) = eF_x x + eF_y y$ are shown in column A of Table 4.5. Comparing the two sets of results validates the applicability of our approach.

Quantum box (dot) structures in GaAs semiconductor

As the last set of examples, we define two quantum dots as shown in Table 4.1. Similar to the Quantum wire example, in case of having separable variables we can re-write the wavefuntion and the potential of a Quantum box as below.

$$\psi(x,y,z) = \psi(x)\,\psi(y)\,\psi(z) \tag{4.19}$$

And also the potential function can be written as

$$V(x,y,z) = V(x) + V(y) + V(z) \tag{4.20}$$

If we follow the same procedure as the Quantum wire example, the theoretical energy levels and wavefunctions of an infinite quantum dot with the potential function V_0 inside the dot are given by

$$E_n = V_0 + \frac{\pi^2\hbar^2}{2m^*}\left(\frac{n_x^2}{L_x^2} + \frac{n_y^2}{L_y^2} + \frac{n_z^2}{L_z^2}\right) \qquad n_x, n_y, n_z = 1,2,3,\cdots \tag{4.21}$$

$$\psi_n(\bar{x}) = \sqrt{\frac{8}{L_xL_yL_z}} sin\left(\frac{\pi n_x x}{L_x}\right) sin\left(\frac{\pi n_y y}{L_y}\right) sin\left(\frac{\pi n_z z}{L_z}\right)$$
$$n_x, n_y, n_z = 1,2,3,\cdots \tag{4.22}$$

The Maple code for calculating the exact energy levels of the given quantum dots using formula 4.21 is shown in Appendix C. Plotting the wavefunctions of the Quantum dots results in a 4 dimensional plot which is difficult to visualize and can't be done by Maple. By employing our method on the given quantum dots, we have obtained the five lowest energy levels and the corresponding wavefunctions. Table 4.6 shows both the obtained and the exact energy levels as well as the percentage differences between them. Also in Figure 4.5 the obtained and exact wavefunctions of the quantum box-I corresponding to the first energy states are shown.

Once again comparing the results in Table 4.6 and also comparing Figures 4.5(a) with 4.5(b) show that the results obtained with our code are in agreement with the theoretical results which proves the validity and efficiency of our method. The drawback of simulating quantum boxes is the computing time it requires. In our program each dimension of the box is divided into 14 points. This results in large matrices in the order of $14 \times 14 \times 14 = 2744$ which in turn results in increased computing time and memory consumption. In section 5 some suggestions to overcome this problem are made (as future work).

Quantum wire structures in GaAs semiconductor with a sinusoidal potential function

To show that our code is capable of obtaining solutions for complex potential functions, we analyze below quantum wire-I in table 4.1 with a 2D sinusoidal potential function given by:

$$V(x,y) = 8(sin(x) + sin(y)) \times 10^{-12} \tag{4.23}$$

The obtained results from running our code for wire-I described in table 4.1 with the sinusoidal potential function given by equation 4.23 are shown in column A of table 4.7. To evaluate our results we need to find an approximated theoretical solution for this example. According to the Small Angle Approximation method in trigonometry $sin(x) = x$ when x approaches zero. In this example x and y are in domain Ω defined by

$$\begin{aligned} \Omega \ : \ & 0 < x < 10^{-9} \\ & 0 < y < 10^{-9} \end{aligned} \tag{4.24}$$

Therefore since x and y are very small, we can use the Small Angle Approximation method to approximate the sinusoidal potential function given by equation 4.23 with the simplified form given by

$$V(x,y) = 8(x+y) \times 10^{-12} \tag{4.25}$$

Noting the equivalence relation that exists between equations 4.23 and 4.25 we may conclude that the solution of 4.23 is equivalent to the solution of 4.25. Similar to what was done in the example for quantum wire-I in the presence of an applied electric field, we calculated the energy levels for wire-I in this example which are shown in column B of table 4.7. The comparison between the obtained and calculated results shows the accuracy and applicability of our approach.

Note that although for simplicity we have presented numerical examples of the quantum structures with simple potential functions, our implementation is capable of analyzing quantum structures with arbitrary potential functions.

Table 4.1: PARAMETERS OF THE Q-DOTs, Q-WIREs and Q-WELLs USED FOR THE EIGENSTATES CALCULATIONS

	Q-dot-I A	Q-dot-II B	Q-wire-I C	Q-wire-II D	Q-well-I E	Q-well-II F
l_x *nm*	1	10	1	10	1	40
l_y *nm*	1	10	1	100	-	-
l_z *nm*	1	10	-	-	-	-
m^* / m_0	0.067	0.067	0.067	0.067	0.067	0.067
V_0	4 *ev*	0 *ev*	0 *ev*	10 *mev*	0 *ev*	25 *mev*

Table 4.2: ENERGY LEVELS OF RECTANGULAR *GaAs* QUANTUM WELL STRUCTURES, DESCRIBED AT TABLE 4.1

	well-I			well-II		
number of level n	FCM ev A	Exact Solution ev B	Error % C	FCM mev D	Exact Solution mev E	Error % F
1	5.605	5.601	0.07	28.503	28.501	0.007
2	22.326	22.403	0.34	38.954	39.002	0.12
3	49.876	50.408	1.05	56.173	56.505	0.58
4	87.783	89.614	2.04	79.865	81.009	1.41
5	135.399	140.021	3.3	109.625	112.513	2.56

Table 4.3: THE EFFECT OF AN ELECTRIC FIELD ON ENERGY LEVELS OF QUANTUM WELL-I, DESCRIBED AT TABLE 4.1

	$F = 5 \times 10^7 V/m$			$F = 5 \times 10^8 V/m$		
number of level n	FCM ev A	Exact Solution ev B	Error % C	FCM ev D	Exact Solution ev E	Error % F
1	5.630	5.626	0.07	5.851	5.855	0.068
2	22.351	22.428	0.34	22.753	22.576	0.78
3	49.901	50.433	1.05	50.658	50.126	1.05
4	87.808	89.639	2.04	89.864	88.033	0.93
5	135.424	140.046	3.30	140.271	135.650	3.29

Table 4.4: ENERGY LEVELS OF RECTANGULAR *GaAs* QUANTUM WIRE STRUCTURES, DESCRIBED AT TABLE 4.1

	wire-I			wire-II		
number of level n	FCM ev A	Exact Solution ev B	Error % C	FCM mev D	Exact Solution mev E	Error % F
1	11.182	11.202	0.17	66.470	66.569	0.14
2	27.705	28.004	1.06	68.065	68.249	0.27
3	44.194	44.807	1.36	70.645	71.049	0.57
4	54.440	56.009	2.80	74.098	74.670	0.77
5	70.873	72.811	2.66	78.273	80.010	2.17

Table 4.5: THE EFFECT OF AN ELECTRIC FIELD ON ENERGY LEVELS OF QUANTUM WIRE-I, DESCRIBED AT TABLE 4.1

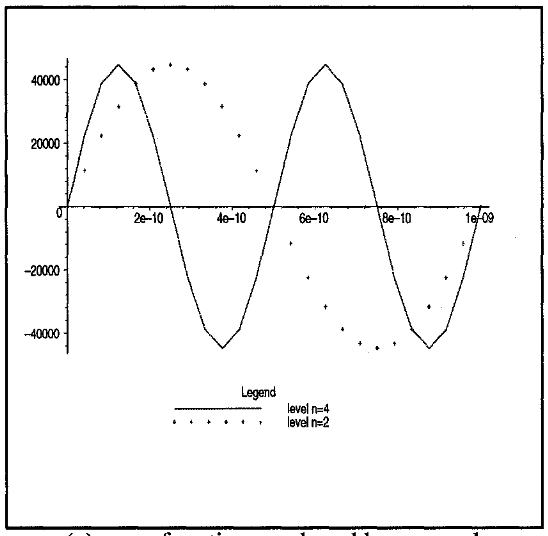
number of level n	FCM ev A	Exact Solution ev B	Error % C
1	11.232	11.252	0.18
2	27.755	28.054	1.06
3	44.244	44.857	1.37
4	54.490	56.059	2.80
5	70.923	72.861	2.66

Table 4.6: ENERGY LEVELS OF RECTANGULAR *GaAs* QUANTUM BOX STRUCTURES, DESCRIBED AT TABLE 4.1

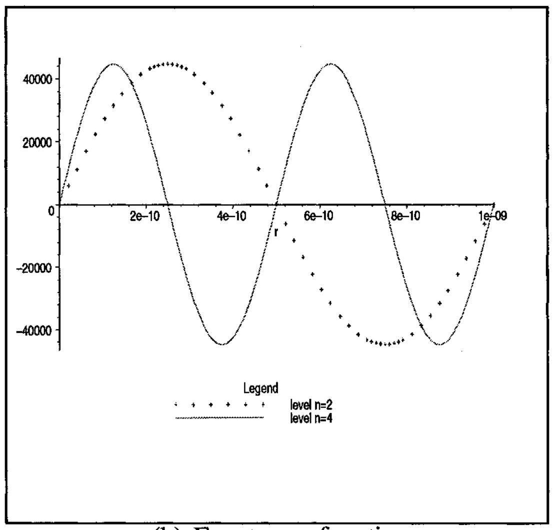
	box-I			box-II		
number of level n	FCM ev A	Exact Solution ev B	Error % C	FCM mev D	Exact Solution mev E	Error % F
1	20.743	20.803	0.29	0.167	0.168	0.59
2	37.146	37.605	1.22	0.331	0.336	1.49
3	53.504	54.407	1.66	0.495	0.504	1.78
4	63.424	65.609	3.33	0.594	0.616	3.57
5	69.818	71.210	1.95	0.658	0.672	2.08

Table 4.7: ENERGY LEVELS OF QUANTUM WIRE-I, DESCRIBED AT TABLE 4.1 WITH A SINUSOIDAL POTENTIAL FUNCTION

number of level n	FCM ev A	Exact Solution ev B	Error % C
1	11.232	11.252	0.18
2	27.755	28.054	1.06
3	44.244	44.857	1.37
4	54.490	56.059	2.80
5	70.923	72.861	2.66

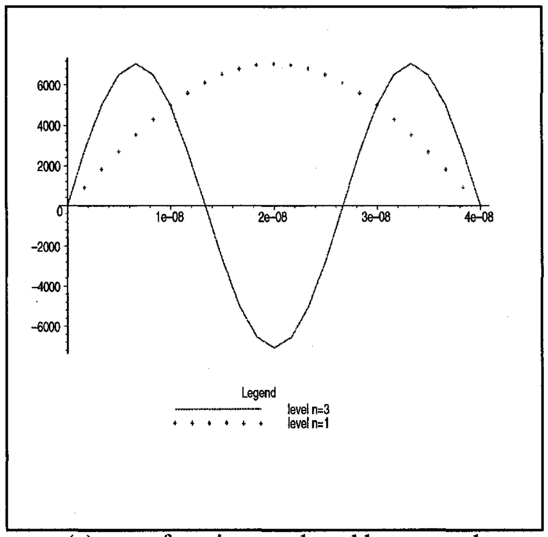


(a) wavefunction produced by our code

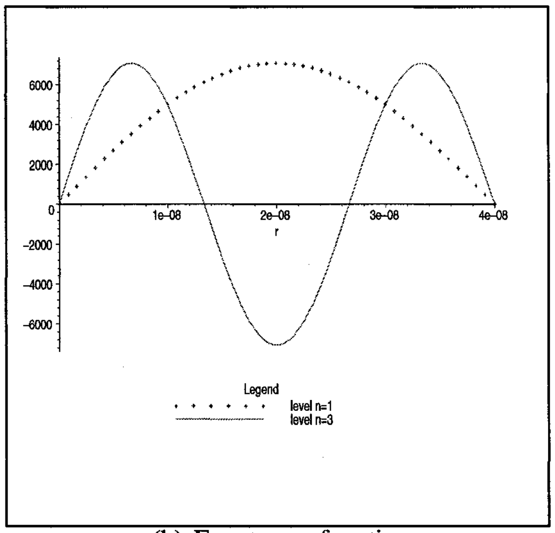


(b) Exact wavefunction

Figure 4.1: The (a) obtained and (b) exact wavefunctions corresponding to the second and the forth energy levels of a quantum well with width = $1\,nm$

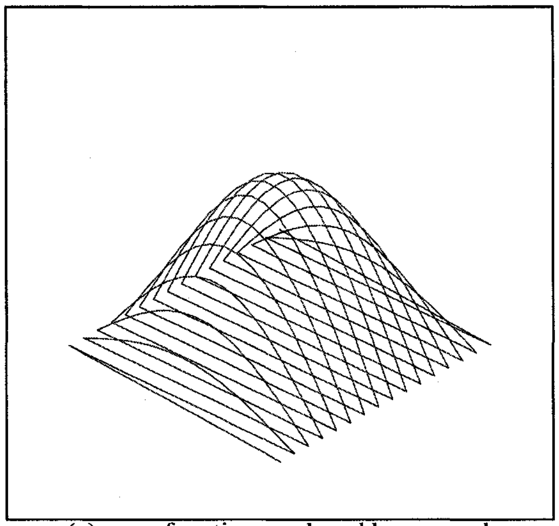


(a) wavefunction produced by our code

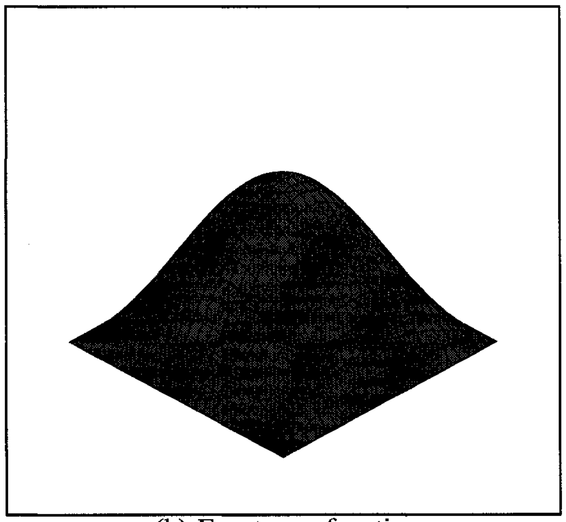


(b) Exact wavefunction

Figure 4.2: The (a) obtained and (b) exact wavefunctions corresponding to the first and the third energy level of a quantum well with width = $40\,nm$



(a) wavefunction produced by our code



(b) Exact wavefunction

Figure 4.3: The (a) obtained and (b) exact wavefunctions corresponding to the third energy level of a quantum wire with dimensions $1\,nm \times 1\,nm$

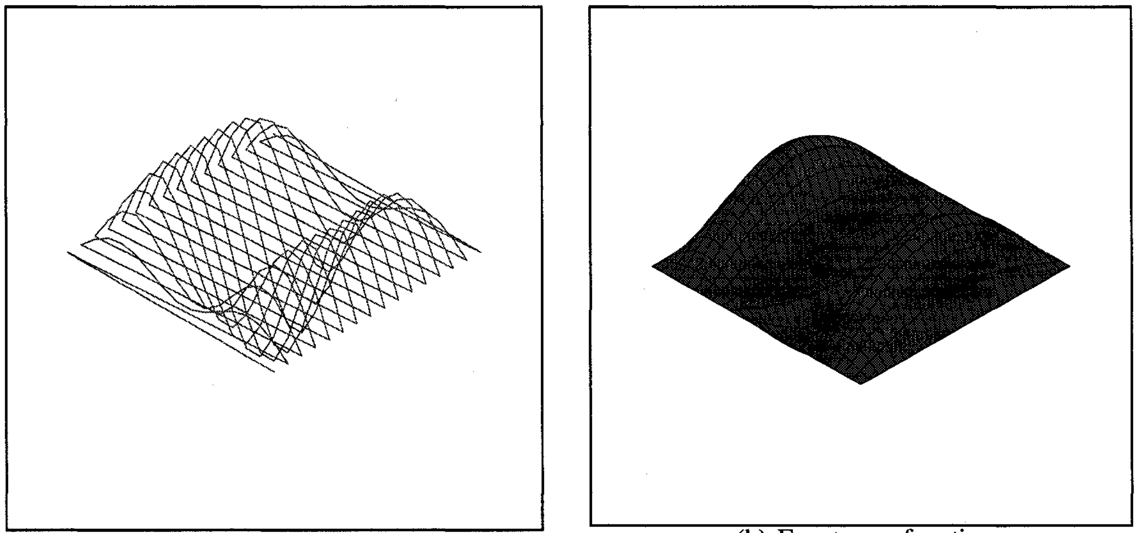

(a) wavefunction produced by our code (b) Exact wavefunction

Figure 4.4: The (a) obtained and (b) exact wavefunctions corresponding to the third energy level of a quantum wire with dimensions $10nm \times 100nm$

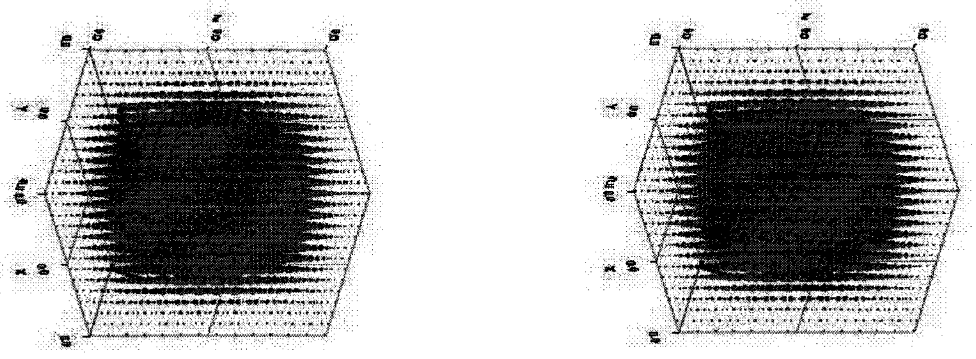

(a) wavefunction produced by our code (b) Exact wavefunction

Figure 4.5: The (a) obtained and (b) exact wavefunctions corresponding to the first energy level of a quantum dot (box) with dimensions $1\,nm \times 1\,nm \times 1\,nm$ and $V_0 = 4$ ev

5 Conclusions and remarks

We have presented an efficient, accurate and simple algorithm to solve the multi-dimensional time-independent Schrodinger equation for infinite quantum well/wire/dot structures. The algorithm works on arbitrary potential functions and finds the energy levels and the corresponding wavefunctions of the input structures. The algorithm is based on the Finite cloud method, which is a meshless method and as a consequence does not deal with the known complexities of mesh methods. To underscore the reasons for using a meshfree method in this thesis, it would be appropriate to make a comparison between mesh methods and meshless methods. In the mesh-based methods the first step is to discretize the physical geometry of the problem into a mesh which is a time consuming process. An expert analyst has to invest a lot of time and effort into the mesh generation process, since mesh generation can not be completely handled by a computer. The need for human expertise increases the cost of the simulation due to the simple fact that manpower is more expensive and valuable than computer time. Whereas in meshless methods distribution of nodes can be done easily, the nodes are independent from each other and there is no need to connectivity information between the nodes in these methods. Another important issue is the use of adaptive algorithms in the simulating process. Usually defining a very fine mesh or node distribution leads to very accurate results, however this is by no means guaranteed. Sometimes a very fine discretization or representation of the domain does not have a considerable effect on the accuracy of the results but nonetheless increases the computation time and eventually the cost of the whole project. If we use an adaptive algorithm in our simulation, the extra information can be ignored automatically to reduce memory consumption and computation time. The fact that adding/removing nodes in the meshless methods are easy and straightforward, provides more flexibility in implementing adaptive algorithms by these methods. The weak point of meshless methods is the fact that they are very new and haven't been thoroughly developed yet so there is as yet no commercial packages based on these methods which we can use in the modeling and simulation projects. Furthermore if we do not consider the mesh generation and node distribution steps, the computation time for the rest of process in FEM is less than that of the meshless methods. Besides, because of dealing with huge matrices, meshless methods are more memory intensive than many mesh methods.

Despite the shortcomings of the meshless methods, they still appeal to many analysts and researchers, because of the advantages they offer, in particular their overcoming the mesh generation problem. At the time of starting this thesis, there were several developed and less developed mesh free methods from which we chose the finite cloud method (FCM) to work with. FCM is a meshless method which uses fixed reproducing approximation to build the interpolation functions and uses a point collocation scheme to discretize the governing partial differential equation. The combination used in FCM gives unique characteristics to FCM as follows.

1- Due to the fact that the kernel function in fixed reproducing technique is centered at point $\bar{r}_k$, we have constant moment matrices, which makes the computation of the derivatives of the shape functions much more easier and faster than other reproducing methods.

2- As Li et al. [12] showed that using unity nodal volumes instead of exact nodal volume will result in almost identical shape functions which is a handy feature in case of having complex geometries.

3- In point collocation method boundary conditions are enforced easily and directly and there is no need for special treatment in dealing with them.

4- As noted in chapter 3 the shape functions constructed with the fixed reproducing kernel technique are multi-valued; this problem is overcome by using a point collocation method such that it limits the shape functions to be computed at the point where the kernel function is centered $(\bar{x}_k)$.

5- The most important advantage of the point collocation method which makes FCM truly meshless is the factor that it doesn't require any connectivity information for discretizing the governing equation as opposed to the Galerkin method which needs a background grid to do the required integrations.

5.1 Contribution

Li et al. [12] originally introduced FCM for two dimensional space. In this thesis we expanded FCM to three dimensions –which to our knowledge has not been done before– and implemented the algorithm in Java. Java has specific features which make it a powerful programming language, the most important of which for the purposes of our implementation was its automatic memory management (garbage collection) feature while we worked with huge matrices.

The contribution of this thesis is to develop a 3D method to solve the multi-dimensional Schrodinger equation for nanostructures as well as implementing a functional tool for simulating and analyzing quantum structures which gives both the eigenenergies (energy levels) and the corresponding eigenfunctions (wavefunctions) of the structures.

To demonstrate the validity of our method, we have applied our program to different quantum structures. The comparison between the obtained results from running our solver and the (expected) theoretical results shows the efficiency of our method.

5.2 Future Work

Following is a list of possible improvements to our implementation as future work.

1- Note that in our implementation we have used constant effective mass through the structure, using variable effective mass would be an obvious improvement for this implementation for the sake of the accuracy of the results.

2- In this algorithm we used rectangular clouds with the same size for the nodes, however the shape as well as the size of the clouds for each node are independent of each other. So up-scaling the algorithm to an adaptive algorithm which would determine the best cloud size for each node in order to get the best solution would be a huge improvement.

3- As mentioned earlier the order of matrices are fairly large specially in three dimensions and operating on large matrices requires extensive amounts of memory as well as computing power. In our implementation we omitted some unimportant rows of the matrix H to obtain the matrix $\widehat{H}$ which is much more smaller than H. An efficient way to make the resulting matrix $\widehat{H}$ even smaller would be to use an adaptive algorithm which would estimate the domain representation error with a priori or a posteriori error estimate and then determine the areas of the domain Ω where we need to add/remove nodes. In this case we could remove the unwanted nodes and keep only the useful ones which would lead to smaller matrices and as a consequence, to reduced memory and cpu utilization.

References

[1] K. Mutamba, M. Flath, A. Sigurdardottir, A. Vogt, and H. L. Hartnagel. A gaas pressure sensor with frequency output based on resonant tunneling diodes. *IEEE Trans. Instrum. Meas.*, 48(6):1333–1338, December 1999.

[2] F.Capasso, S. Sen, F. Beltram, L.M. Lunardi, and A.S. Vengurlekar. Quantum functional devices - resonant-tunneling transistors, circuits with reduced complexity, and multiple-valued logic. *IEEE Transactions on Electronic Devices*, 36(10):2065–2082, October 1998.

[3] Kenith E. Meissner and Adam Allen. Whispering gallery mode biosensors using semiconductor quantum dots. *IEEE Sensors J.*, October 2005.

[4] F. Capasso, K. Mohammed, and A. Y. Cho. Resonant tunneling through double barriers, perpendicular quantum transport phenomena in superlattices, and their device applications. *IEEE J. Quantum Electron.*, QE-22(9):1853–1869, September 1986.

[5] Emmanuel Anemogiannis, Elias N. Glytsis, and Thomas K. Gaylord. Bound and quasibound state calculations for biased/unbiased semiconductor quantum heterostructures. *IEEE J. Quantum Electron.*, 29(11):2731–2740, November 1993.

[6] Rade Vignjevic. Review of development of the smooth particle hydrodynamics (sph) method. *Cranfield University; School of Engineering*, December 2004.

[7] N. R. Aluru. A reproducing kernel particle method for meshless analysis of microelectromechanical systems. *Computational Mechanics*, 23(4):324–338, May 1999.

[8] Gregory M. Hulbert. Application of reproducing kernel particle methods in electromagnetics. *Computer Methods in Applied Mechanics and Engineering*, 139(1-4):229–235, December 1996.

[9] T. Belytschko, Y. Y. Lu, L. Gu, and M. Tabbara. Element-free galerkin methods for static and dynamic fracture. *International Journal of Solids and Structures*, 32(17-18):2547–2570, September 1995.

[10] P. Krysl and T. Belytschko. Element-free galerkin method: Convergence of the continuous and discontinuous shape functions. *Computer Methods in Applied Mechanics and Engineering*, 148(3-4):257–277, September 1997.

[11] S. N. Atluri and T. Zhu. A new meshless local petrov-galerkin (mlpg) approach in computational mechanics. *Computational Mechanics*, 22(2):117–127, August 1998.

[12] N. R. Aluru and G. Li. Finite cloud method: A true meshless technique based on fixed reproducing kernel approximation. *International Journal for Numerical Methods in Engineering*, 50(10):2373–2410, April 2001.

[13] N. R. Aluru. A point collocation method based on reproducing kernel approximation. *International Journal for Numerical Methods in Engineering*, 47(6):1083–1121, February 2000.

[14] Hendrik F. Hameka. *Quantum Mechanics: A Conceptual Approach.* John Wiley and Sons, Hoboken, New Jersey, 1st ed. edition, 2004.

[15] Eugen Merzbacher. *Quantum Mechanics.* Wiley, New York, 3rd ed. edition, c1998.

[16] Vladimir Mitin, Viacheslav Kochelap, and Michael A. Stroscio. *Quantum Heterostructures: Microelectronics and Optoelectronics.* Cambridge University Press, New York, 1st ed. edition, 1999.

[17] Jon Willis. Introduction to quantum physics, 2006. http://orca.phys.uvic.ca/ jwillis/teaching/teaching.html, last read on April 10, 2007.

[18] http://en.wikipedia.org/wiki/MainPage, last read on April 10, 2007.

[19] Kenji Nakamura, Akira Shimizu, Masanori Koshiba, and Kazuya Hayata. Finite-element analysis of quantum wells of arbitrary semiconductors with arbitrary potential profiles. *IEEE_J_JQE*, 25(5):889–895, May 1989.

[20] P. A. Knipp and T. L. Reinecke. Boundary element method for calculating electron and phonon states in quantum wires and related nanostructures. *Superlattices and Microstructures*, 16(2):201–204, September 1994.

[21] Willi Schnauer and Torsten Adolph. Higher order may be better or may not be better: Investigations with the fdem (finite difference element method). *Journal of Scientific Computing*, 17(1-4):221–229, December 2002.

[22] Willi Schnauer and Torsten Adolph. How we solve pdes. *Journal of Computational and Applied Mathematics*, 131(1-2):473492, June 2001.

[23] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl. Meshless methods: An overview and recent developments. *Computer Methods in Applied Mechanics and Engineering*, 139(1-4):347, December 1996.

[24] G. R. Liu. *Mesh free methods : moving beyond the finite element method.* CRC press, Boca Raton, Fla., 1st ed. edition, c2003.

[25] I. V. Singh. A numerical study of weight functions, scaling, and penalty parameters for heat transfer applications. *Numerical heat transfer. Part A, Applications*, 47(10):1025–1053, 2005.

[26] http://www.physics.utah.edu/ detar/phycs6720/handouts, last read on April 10, 2007.

[27] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82(12):1013–1024, December 1977.

[28] J. J. Monaghan. Why particle methods work. *SIAM Journal on Scientific Computing*, 3(4):422–433, December 1982.

[29] R.A. Gingold and J. J. Monaghan. Smooth particle hydrodynamics: theory and application to non-spherical stars. *Royal Astronomical Society*, 181(2):375–389, November 1977.

[30] W. K. Liu, Y. Chen, R. A. Uras, and C. T. Chang. Generalized multiple scale reproducing kernel particle methods. *Computer Methods in Applied Mechanics and Engineering*, 139:91–157, April 1996.

[31] L. Zhang J.X. Zhou, H.Y. Zhang. Reproducing kernel particle method for free and force vibration analysis. *Journal of Sound and Vibration*, 279(1-2):389–402, Nov 2005.

[32] M. J. Mitchell, R. Qiao, and N.R. Aluru. Meshless analysis of steady-state electroosmotic transport. *Journal of Microelectromechanical Systems*, 9(4):435–449, December 2000.

[33] N. R. Aluru. New approximations and collocation schemes in the finite cloud method. *Computers & Structures*, 83(17-18):1366–1385, June 2005.

[34] J. Dolbow and T. Belytschko. Numerical integration of the galerkin weak form in meshfree methods. *Computational Mechanics*, 23(3):219–230, April 1999.

[35] Joe Hicklin, Cleve Moler, Peter Webb, Ronald F. Boisvert, Bruce Miller, Roldan Pozo, and Karin Remington. Jama : A java matrix package, 1998-2005. http://math.nist.gov/javanumerics/jama/ , last read on April 10, 2007.

[36] B. Van Zeghbroeck. Principles of semiconductor devices, c2004. http://ece-www.colorado.edu/~bart/book, last read on April 10, 2007.

A maple code for quantum well

```
>  m_star:= 0.067 * 9.11 * 10^(-31):
>  h_hat := 1.054 * 10^(-34):
>  x:='x':
>  l_x:= 1.* 10^(-9):
>  V0:= 0:
>  E_01 :=((h_hat * 3.14)^2)/(2 * m_star * l_x^2):
>  # exact energy levels in Jule;
>  for n from 1 to 5 do Ej[n]:= V0 + E_01* n^2; end do;
```

$$Ej_1 := .8972580070\,10^{-18}$$

$$Ej_2 := .3589032028\,10^{-17}$$

$$Ej_3 := .8075322063\,10^{-17}$$

$$Ej_4 := .1435612811\,10^{-16}$$

$$Ej_5 := .2243145018\,10^{-16}$$

```
>  # exact energy levels in mev;
>  for n from 1 to 5 do (Ej[n])/(1.6*10^(-22)) ; end do;
```

5607.862544

22431.45018

50470.76289

89725.80069

140196.5636

```
>  # exact wave function is given as follow.
>  f:= r-> sqrt(2/l_x)* sin(Pi*r/l_x):
>  f(l_x/Pi);
```

$$37631.72646$$

```
>  plot([f(2*r),f(4*r)], r=0..l_x,style=[point,line]):
>  plot([f(1*r),f(3*r)], r=0..l_x,style=[point,line]):
```

B maple code for quantum wire

```
>  m_star:= 0.067 * 9.11 * 10^(-31);
>  h_hat := 1.054 * 10^(-34);
>  x:='x':
>  lx:= 1.* 10^(-9);
>  ly:= 1.* 10^(-9);
>  V0:=0.;
>  printlevel := 2;
```

$$m\_star := .6103700000\,10^{-31}$$

$$h\_hat := .1054000000\,10^{-33}$$

$$lx := .1000000000\,10^{-8}$$

$$ly := .1000000000\,10^{-8}$$

$$V0 := 0.$$

$$printlevel := 2$$

```
> i:=1;
```

> for nx from 1 to 5 do for ny from 1 to 5 do E_exact[i]:= (V0 + (((h_hat * 3.14)^2)/(2 $*$ $m\_star*lx^2$)) $*nx^2+(((h\_hat*3.14)^2)/(2*m\_star*ly^2))*ny^2)/(1.602*10^(-19)))*1:$ $i:=i+1: enddo: enddo:$

```
> sort([seq(E_exact[k],k=1..10)]);
```

$$i:=1$$

[11.20172293, 28.00430734, 28.00430734, 44.80689174, 56.00861469, 72.81119909, 95.21464493, 112.0172293, 145.6223982, 162.4249826]

```
> i := 1;
```

$$i:=1$$

```
> f:='f';
```

> f:=(r,s)- > sqrt((2.*2.)/(lx*ly)) * sin(1*Pi*r/lx)* sin(1*Pi*s/ly);

$$f:=f$$

$$f:=(r,s)\rightarrow\sqrt{4.\frac{1}{lx\,ly}}\sin(\frac{\pi r}{lx})\sin(\frac{\pi s}{ly})$$

```
> plot3d(f(r,s),r=0..10^(-9),s=0..10^(-9)):
> plot3d(f(2*r,s),r=0..10^(-9),s=0..10^(-9)):
> plot3d(f(1*r,3*s),r=0..10^(-9),s=0..10^(-9)):
> plot3d(f(2*r,2*s),r=0..10^(-9),s=0..10^(-9)):
```

C maple code for quantum box

```
> m_star:= 0.067 * 9.11 * 10^(-31);
> h_hat := 1.054 * 10^(-34);
> x:='x':
> lx:= 1.  * 10 ^(-9):
> ly:= 1.  * 10 ^(-9):
> lz:= 1.  * 10 ^(-9):
> NP:=14:
> V0:=0.;
> printlevel := 2;
> i:=1;
```

> for nx from 1 to 5 do for ny from 1 to 5 do for nz from 1 to 5 do E_exact[i]:= (V0 + $((((\text{h\_hat} * 3.14)^2)/(2*m\_star*lx^2))*nx^2 + (((h\_hat*3.14)^2)/(2*m\_star*ly^2))*ny^2 + (((h\_hat*3.14)^2)/(2*m\_star*lz^2))*nz^2)/(1.602*10^(-19)))*1 : i := i+1 : enddo : enddo : enddo :$

```
> sort([seq(E_exact[k],k=1..15)]);
```

$$m\_star := .6103700000\,10^{-31}$$

$$h\_hat := .1054000000\,10^{-33}$$

$$V0 := 0.$$

$$printlevel := 2$$

$$i := 1$$

[16.80258440, 33.60516880, 33.60516881, 50.40775321, 61.60947615, 61.60947616, 78.41206056, 78.41206056, 100.8155064, 106.4163679, 117.6180908, 145.6223982, 151.2232596, 168.0258440, 196.0301514]

D Java Code to Implement the Solver

Listing 1: Main.java

```
/*
* Main.java
*
* Created on March 29, 2006, 7:46 PM
*
*/
package javaapplication1;
import Jama.Matrix;
import Jama.EigenvalueDecomposition;
import Jama.QRDecomposition;
import Jama.LUDecomposition;
import Jama.SingularValueDecomposition;
import java.util.Date;
/**
* @author Sanam Moslemi-Tabrizi
*/
public class Main {
    /**
    * Creates a new instance of Main
    */
    public Main() {
    }
    public static void main(String[] args) {
        if (args != null && args.length > 0) {
            if (args[0].equals("well")) {
                Util.print("..............␣WELL........");
                System.out.println();
                QuantumWellEnergyLevelsFinder aQuantumWellEnergyLevelsFinder =
                        new QuantumWellEnergyLevelsFinder();
                aQuantumWellEnergyLevelsFinder.findQuantumWellEnergyLevels();
            }
            else
            if (args[0].equals("wire")) {
                Util.print("..............␣WIRE........");
                System.out.println();
                QuantumWireEnergyLevelsFinder aQuantumWireEnergyLevelsFinder =
                        new QuantumWireEnergyLevelsFinder();
                aQuantumWireEnergyLevelsFinder.findQuantumWireEnergyLevels();
            }
            else
            if (args[0].equals("box")) {
                Util.print("..............␣BOX........");
                System.out.println();
                QuantumBoxEnergyLevelsFinder aQuantumBoxEnergyLevelsFinder =
                        new QuantumBoxEnergyLevelsFinder();
                aQuantumBoxEnergyLevelsFinder.findQuantumBoxEnergyLevels();
            }
            else
            if (args[0].equals("eq")) {
                Util.print("..............␣Equation........");
                System.out.println();
                EquationsSolver aEquationsSolver = new EquationsSolver();
                aEquationsSolver.SolveEquations();
            }
            else
            if (args[0].equals("exact")) {
                Util.print("..............␣Exact........");
                System.out.println();
                ExactWaves aExactWaves = new ExactWaves();
                aExactWaves.FindExactWaves();
            }
            else {
                Util.print("Unknown␣option:␣" + args[0]);
                System.exit(1);
            }
        }
    }
}
```

Listing 2: QuantumBoxEnergyLevelsFinder.java

```
/*
* QuantumBoxEnergyLevelsFinder.java
*
* Created on April 10, 2006, 3:54 PM
*
*/
package javaapplication1;
import Jama.Matrix;
import Jama.EigenvalueDecomposition;
import Jama.QRDecomposition;
import Jama.LUDecomposition;
import java.util.Arrays;
import Jama.SingularValueDecomposition;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Date;
/**
*
* @author Sanam Moslemi-Tabrizi
*/
public class QuantumBoxEnergyLevelsFinder {
    /** Creates a new instance of QuantumBoxEnergyLevelsFinder */
    public QuantumBoxEnergyLevelsFinder() {
    }
    public static int numberOfPoints = 14;
    //public static int numberOfPoints = 6;
    public void findQuantumBoxEnergyLevels(){
        double[][] txx = {
            {
                0., 0., 0., 0., 0., 0., 0., 2., 0., 0.
            }
        }
        ;
        Matrix PTXX = new Matrix(txx);

        double[][] tyy = {
            {
                0., 0., 0., 0., 0., 0., 0., 0., 2., 0.
            }
        }
        ;
        Matrix PTYY = new Matrix(tyy);
        double[][] tzz= {
            {
                0., 0., 0., 0., 0., 0., 0., 0., 0., 2.
            }
        }
        ;
        Matrix PTZZ = new Matrix(tzz);
        double lengthInXDirection = 1 * Math.pow(10, -9);
        double lengthInYDirection = 1 * Math.pow(10, -9);
        double lengthInZDirection = 1 * Math.pow(10, -9);
```

```
int totalnumberOfPoints = numberOfPoints * numberOfPoints * numberOfPoints;
int m = 10;
double mm = 0.067;
double m_star = mm * Util.freeElectronMass;
double h_hat = 1.054 * Math.pow(10, -34);
double alpha=-(h_hat * h_hat)/(2.*m_star);
Matrix V = new Matrix(totalnumberOfPoints, 1);
Matrix pot = new Matrix(totalnumberOfPoints, 1);
Matrix temp2 = new Matrix(1, 1);
double x[] = new double[numberOfPoints];
double y[] = new double[numberOfPoints];
double z[] = new double[numberOfPoints];
x[0] = 0;
y[0] = 0;
z[0] = 0;
/* dx, dy, dz are the could sizes in each Dim.
* the result is much more accurate for smaller cloud sizes
*but if they be too small it would cause singular Matrix problem
*with 1.05 it is fine. */
double delta_x = lengthInXDirection / (numberOfPoints - 1);
//double dx = 1.25 * delta_x;
double dx = 1.02 * delta_x;
double delta_y = lengthInYDirection / (numberOfPoints - 1);
//double dy = 1.25 * delta_y;
double dy = 1.02 * delta_y;
double delta_z = lengthInZDirection / (numberOfPoints - 1);
//double dz = 1.25 * delta_z;
double dz = 1.02 * delta_z;
for (int i = 0; i < numberOfPoints; i++) {
    x[i] = x[0] + i * delta_x;
    y[i] = y[0] + i * delta_y;
    z[i] = z[0] + i * delta_z;
}
Matrix[] nod = new Matrix[totalnumberOfPoints];
int nd = 0;
for (int i = 0; i < numberOfPoints; i++)
{
    for (int j = 0; j < numberOfPoints; j++)
    {
        for (int k = 0; k < numberOfPoints; k++)
        {
            double tempx = x[i];
            double tempy = y[j];
            double tempz = z[k];
            double[][] temp1 = {
                {
                    tempx , tempy , tempz
                }
            }
            ;
            nod[nd] = new Matrix(temp1);
            nd = nd+1;
        }
    }
}
Matrix F = Util.get3DF(m, totalnumberOfPoints, nod);
Matrix FT = F.transpose();
Matrix A = new Matrix(totalnumberOfPoints, totalnumberOfPoints);
double U[] = new double[totalnumberOfPoints];
double ax;
double ay;
double az;
double fiX = 0;
double fiY = 0;
double fiZ = 0;
double phi[][] = new double[totalnumberOfPoints][totalnumberOfPoints];
double NXX[][] = new double[totalnumberOfPoints][totalnumberOfPoints];
double NYY[][] = new double[totalnumberOfPoints][totalnumberOfPoints];
double NZZ[][] = new double[totalnumberOfPoints][totalnumberOfPoints];
int numberOfNZPoints= totalnumberOfPoints − 6 ∗
    numberOfPoints ∗numberOfPoints + 12 ∗ numberOfPoints − 8;
int[] r = new int[numberOfNZPoints];
int ri = 0;
for (int k = 0; k < totalnumberOfPoints; k++)
{
    System.out.print("k=␣" +k);
    System.out.println("\t");
    for (int i = 0; i < totalnumberOfPoints; i++)
    {
        ax = (nod[k].get(0,0) − nod[i].get(0,0)) / dx;
        double tx = (−ax ∗ ax) / 0.2;
        fiX = 1.2618∗Math.exp(tx);
        ay = (nod[k].get(0,1) − nod[i].get(0,1)) / dy;
        double ty = (−ay ∗ ay) / 0.2;
        fiY = 1.2618∗Math.exp(ty);
        az = (nod[k].get(0,2) − nod[i].get(0,2)) / dz;
        double tz = (−az ∗ az) / 0.2;
        fiZ = 1.2618∗Math.exp(tz);
        phi[k][i] = fiX ∗ fiY ∗ fiZ;
        V.set(i,0,phi[k][i]);
        // V[i,0]= phi[k][i]
    }
    Matrix W = Util.getDia(V);
    Matrix M = F.times(W).times(FT) ;
    Matrix M_in = M.inverse();
    for(int n=0; n<totalnumberOfPoints; n++) {
        double x0 = 1;
        double x1 = nod[n].get(0,0);
        double x2 = nod[n].get(0,1);
        double x3 = nod[n].get(0,2);
        double x4 = nod[n].get(0,0) ∗ nod[n].get(0,1);
        double x5 = nod[n].get(0,0) ∗ nod[n].get(0,2);
        double x6 = nod[n].get(0,1) ∗ nod[n].get(0,2);
        double x7 = nod[n].get(0,0) ∗ nod[n].get(0,0);
        double x8 = nod[n].get(0,1) ∗ nod[n].get(0,1);
        double x9 = nod[n].get(0,2) * nod[n].get(0,2);
        double[][] temp1 = {
            {
                x0
            }
            ,{
                x1
            }
            ,{
                x2
            }
            ,{
                x3
            }
            ,{
                x4
            }
            ,{
                x5
            }
            ,{
                x6
            }
            ,{
                x7
            }
            ,{
                x8
            }
            ,{
                x9
            }
        }
        ;
        Matrix P = new Matrix(temp1);
        NXX[k][n] = phi[k][n]*((PTXX.times(M_in).times(P)).get(0,0)) ;
        NYY[k][n] = phi[k][n]*((PTYY.times(M_in).times(P)).get(0,0)) ;
        NZZ[k][n] = phi[k][n]*((PTZZ.times(M_in).times(P)).get(0,0)) ;
    }
    if (nod[k].get(0,0) == 0 || nod[k].get(0,0)== lengthInXDirection ||
    nod[k].get(0,1) == 0 || nod[k].get(0,1)== lengthInYDirection ||
    nod[k].get(0,2) == 0 || nod[k].get(0,2)== lengthInZDirection) {
        U[k] = 0;
        for(int i=0; i<totalnumberOfPoints; i++) {
            A.set(k,i,0);
            pot.set(k,0,0);
        }
    }
    else {
        for(int i=0; i<totalnumberOfPoints; i++) {
            A.set(k,i,alpha*(NXX[k][i] + NYY[k][i] + NZZ[k][i]));
            pot.set(k,0,Util.get3DPotential(nod[k].get(0,0) , nod[k].get(0,1) , nod[k].get(0,2)));
        }
        r[ri] = k;
        ri= ri+1;
    }
}
Matrix pot_func = Util.getDia(pot);
Matrix H = A. plus(pot_func);
Matrix H2 = H.getMatrix(r,r);
EigenvalueDecomposition D1 =
new EigenvalueDecomposition(H2);
double[] d1 = D1.getRealEigenvalues();
Matrix wave = D1.getV();
Map<Double, double[]> map = new LinkedHashMap<Double, double[]>();
double[][] waveArray = wave.transpose().getArray();
for(int i=0; i<numberOfNZPoints; i++)
{
    Double d = d1[i];
    double[] m4 = waveArray[i];
    // ith coloumn
    map.put(d, m4);
}
Arrays.sort(d1);
Matrix[] u_funcs = new Matrix[5];
Matrix[] wave_function = new Matrix[5];
for(int i=0; i<5; i++)
{
    wave_function[i] = new Matrix(1 , totalnumberOfPoints);
}
for(int i=0; i<5; i++)
{
    double[] m4 = map.get(d1[i]);
    double[][] mm4 = {
        m4
    }
    ;
    Matrix w_m4 = new Matrix(mm4);
    u_funcs[i] = w_m4;
}
double beta = Math.pow(1/(delta_x * delta_y * delta_z ), 0.5);
for (int i=0 ; i<5 ; i++)
{
    if (u_funcs[i].get(0,0)<
    0)
    {
        u_funcs[i].timesEquals(-beta ) ;
    }
    else
    {
        u_funcs[i].timesEquals(beta ) ;
    }
}
for (int i=0 ; i<5 ; i++)
{
    int u_funcs_index = 0;
```

```
        for (int j=0 ; j<totalnumberOfPoints ; j++)
        {
            if (nod[j].get(0,0) == 0 || nod[j].get(0,0)== lengthInXDirection ||
            nod[j].get(0,1) == 0 || nod[j].get(0,1)== lengthInYDirection ||
            nod[j].get(0,2) == 0 || nod[j].get(0,2)== lengthInZDirection)
            {
                wave_function[i].set(0,j,0);
            }
            else
            {
                wave_function[i].set(0,j,u_funcs[i].get(0,u_funcs_index));
                u_funcs_index = u_funcs_index + 1 ;
            }
        }
    }
    // make 5 matrixes of x, y, wavefunction for ploting the wavefunction
    Matrix[] plot_data = new Matrix[5];
    for(int i=0; i<5; i++)
    {
        plot_data[i] = new Matrix(totalnumberOfPoints , 4);
    }

    for(int i=0; i<5; i++)
    {
        for(int j=0; j<totalnumberOfPoints; j++)
        {
            plot_data[i]. set(j,0,nod[j].get(0,0));
            plot_data[i]. set(j,1,nod[j].get(0,1));
            plot_data[i]. set(j,2,nod[j].get(0,2));
            plot_data[i]. set(j,3,wave_function[i].get(0,j) );
        }
        System.out.print("wavefuntion␣level␣n␣=␣");
        System.out.println(i+1);
        System.out.println("␣wavefunction␣");
        for(int j=0; j < totalnumberOfPoints; j++)
        {
            Util.print(plot_data[i].get(j,0));
            System.out.print("␣");
            Util.print(plot_data[i].get(j,1));
            System.out.print("␣");
            Util.print(plot_data[i].get(j,2));
            System.out.print("␣");
            Util.print(plot_data[i].get(j,3));
            System.out.println();
        }

    }

    double[][] dd1 = {
        d1
    }
    ;
    Matrix E_H= new Matrix(dd1);
```

```
            E_H = E_H.times(6.25*Math.pow(10, 21)) ;
            Util.print("EIGENVALUES_OF_H_mev\n");
            Util.print(E_H , true);
    }
}
```

Listing 3: QuantumWireEnergyLevelsFinder.java

```
/*
* QuantumWireEnergyLevelsFinder.java
*
* Created on April 7, 2006, 10:25 PM
*
*/
package javaapplication1;
import Jama.Matrix;
import Jama.EigenvalueDecomposition;
import Jama.QRDecomposition;
import Jama.LUDecomposition;
import Jama.SingularValueDecomposition;
import java.util.Arrays;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Date;
/**
*
* @author Sanam Moslemi−Tabrizi
*/
public class QuantumWireEnergyLevelsFinder {
    /** Creates a new instance of QuantumWireEnergyLevelsFinder */
    public QuantumWireEnergyLevelsFinder() {
    }
    /** set Nx = Ny = 16**/
    public static int numberOfPoints = 16;
    public void findQuantumWireEnergyLevels(){
        double[][] txx = {
            {
                0., 0., 0., 0., 2.,0.
            }
        }
        ;
        Matrix PTXX = new Matrix(txx);
        double[][] tyy = {
            {
                0.,0.,0.,0.,0.,2.
            }
        }
        ;
        Matrix PTYY = new Matrix(tyy);
        /** define the lenghts of the wire**/
        double lengthInXDirection = 1 * Math.pow(10, −9);
        double lengthInYDirection = 1 * Math.pow(10, −9);
        double mm = 0.067;
        int totalnumberOfPoints = numberOfPoints * numberOfPoints;
        int m = 6;
        double m_star = mm * Util.freeElectronMass;
        double h_hat = 1.054 * Math.pow(10, −34);
        double alpha=−(h_hat * h_hat)/(2.*m_star);
        Matrix V = new Matrix(totalnumberOfPoints, 1);
Matrix pot = new Matrix(totalnumberOfPoints, 1);
Matrix temp2 = new Matrix(1, 1);
double x[] = new double[numberOfPoints];
double y[] = new double[numberOfPoints];
x[0] = 0;
y[0] = 0;
/** set the cloud size**/
double delta_x = lengthInXDirection / (numberOfPoints − 1);
double dx = 1.02 * delta_x;
double delta_y = lengthInYDirection / (numberOfPoints − 1);
double dy = 1.02* delta_y;
for (int i = 0; i < numberOfPoints; i++) {
    x[i] = x[0] + i * delta_x;
    y[i] = y[0] + i * delta_y;
}
Matrix[] nod = new Matrix[totalnumberOfPoints];
int nd=0;
for (int i = 0; i < numberOfPoints; i++) {
    for (int j = 0; j < numberOfPoints; j++) {
        double tempx = x[i];
        double tempy = y[j];
        double[][] temp1 = {
            {
                tempx , tempy
            }
        }
        ;
        nod[nd] = new Matrix(temp1);
        nd = nd+1;
    }
}
Matrix F = Util.get2DF(m, totalnumberOfPoints, nod);
Matrix FT = F.transpose();
Matrix A = new Matrix(totalnumberOfPoints, totalnumberOfPoints);
double U[] = new double[totalnumberOfPoints];
double ax;
double ay;
double fiX = 0;
double fiY = 0;
double phi[][] = new double[totalnumberOfPoints][totalnumberOfPoints];
double NXX[][] = new double[totalnumberOfPoints][totalnumberOfPoints];
double NYY[][] = new double[totalnumberOfPoints][totalnumberOfPoints];
int numberOfNZPoints= totalnumberOfPoints − 4 * numberOfPoints + 4;
int[] r = new int[numberOfNZPoints];
int ri = 0;
for (int k = 0; k < totalnumberOfPoints; k++)
{
    for (int i = 0; i < totalnumberOfPoints; i++)
    {
        ax = (nod[k].get(0,0) − nod[i].get(0,0)) / dx;
        double tx = (−ax * ax) / 0.2;
        fiX = 1.2618*Math.exp(tx);
        ay = (nod[k].get(0,1) − nod[i].get(0,1)) / dy;
        double ty = (−ay * ay) / 0.2;
        fiY = 1.2618*Math.exp(ty);
        phi[k][i] = fiX * fiY;
        V.set(i,0,phi[k][i]);
        // V[i,0]= phi[k][i]
    }
    Matrix W = Util.getDia(V);
    Matrix M= F.times(W).times(FT) ;
    Matrix M_in = M.inverse();
    for(int n=0; n<totalnumberOfPoints; n++)
    {
        double x0 = 1;
        double x1 = nod[n].get(0,0);
        double x2 = nod[n].get(0,1);
        double x3 = nod[n].get(0,0) * nod[n].get(0,1);
        double x4 = nod[n].get(0,0) * nod[n].get(0,0);
        double x5 = nod[n].get(0,1) * nod[n].get(0,1);
        double[][] temp1 = {
            {
                x0
            }
            ,{
                x1
            }
            ,{
                x2
            }
            ,{
                x3
            }
            ,{
                x4
            }
            ,{
                x5
            }
        }
        ;
        Matrix P = new Matrix(temp1);
        /**NXX and NYY are the second derivatives of shape functions **/
        NXX[k][n] = phi[k][n]*((PTXX.times(M_in).times(P)).get(0,0)) ;
        NYY[k][n] = phi[k][n]*((PTYY.times(M_in).times(P)).get(0,0)) ;
    }
    /**constructing the H matrix and H2 submatrix**/
    if (nod[k].get(0,0) == 0 || nod[k].get(0,0)== lengthInXDirection ||
    nod[k].get(0,1) == 0 || nod[k].get(0,1)== lengthInYDirection)
    {
        U[k] = 0;
        for(int i=0; i<totalnumberOfPoints; i++)
        {
            A.set(k,i,0);
            pot.set(k,0,0);
        }
        }
        else
        {
            for(int i=0; i<totalnumberOfPoints; i++)
            {
                A.set(k,i,alpha*(NXX[k][i] + NYY[k][i]));
                pot.set(k,0,Util.get2DPotential(nod[k].get(0,0) , nod[k].get(0,1)));
            }
            r[ri] = k;
            ri= ri+1;
        }
    }
    Matrix pot_func = Util.getDia(pot);
    Matrix H = A. plus(pot_func);
    Matrix H2 = H.getMatrix(r,r);
    EigenvalueDecomposition D1 =
    new EigenvalueDecomposition(H2);
    double[] d1 = D1.getRealEigenvalues();
    Matrix wave = D1.getV();
    Map<Double, double[]> map = new LinkedHashMap<Double, double[]>();
    double[][] waveArray = wave.transpose().getArray();
    for(int i=0; i<numberOfNZPoints; i++)
    {
        Double d = d1[i];
        double[] m4 = waveArray[i];
        // ith coloumn
        map.put(d, m4);
    }
    Arrays.sort(d1);
    Matrix[] u_funcs = new Matrix[5];
    Matrix[] wave_function = new Matrix[5];
    for(int i=0; i<5; i++)
    {
        wave_function[i] = new Matrix(1 , totalnumberOfPoints);
    }
    for(int i=0; i<5; i++)
    {
        double[] m4 = map.get(d1[i]);
        double[][] mm4 = {
            m4
        }
        ;
        Matrix w_m4 = new Matrix(mm4);
        u_funcs[i] = w_m4;
    }
    double beta = Math.pow(1/(delta_x * delta_y ), 0.5);
    for (int i=0 ; i<5 ; i++)
    {
        if (u_funcs[i].get(0,0)<
        0)
        {
            u_funcs[i].timesEquals(-beta ) ;
        }
        else
        {
            u_funcs[i].timesEquals(beta ) ;
        }
    }
    for (int i=0 ; i<5 ; i++)
    {
        int u_funcs_index = 0;
        for (int j=0 ; j<totalnumberOfPoints ; j++)
        {
            if (nod[j].get(0,0) == 0 || nod[j].get(0,0)== lengthInXDirection ||
            nod[j].get(0,1) == 0 || nod[j].get(0,1)== lengthInYDirection)
            {
                wave_function[i].set(0,j,0);
            }
            else
            {
                wave_function[i].set(0,j,u_funcs[i].get(0,u_funcs_index));
                u_funcs_index = u_funcs_index + 1 ;
            }
        }
    }
    // make 5 matrixes of x, y, wavefunction for ploting the wavefunction
    Matrix[] plot_data = new Matrix[5];
    for(int i=0; i<5; i++)
    {
        plot_data[i] = new Matrix(totalnumberOfPoints , 3);
    }
    for(int i=0; i<5; i++)
    {
        for(int j=0; j<totalnumberOfPoints; j++)
        {
            plot_data[i]. set(j,0,nod[j].get(0,0));
            plot_data[i]. set(j,1,nod[j].get(0,1));
            plot_data[i]. set(j,2,wave_function[i].get(0,j) );
        }
        System.out.print("wavefuntion level n = ");
        System.out.println(i+1);
        System.out.println(" wavefunction ");
        for(int j=0; j<totalnumberOfPoints; j++)
        {
            System.out.print(plot_data[i].get(j,2));
            System.out.println(",");
        }
    }
    double[][] dd1 = {
        d1
    }
    ;
    Matrix E_H= new Matrix(dd1);
    E_H = E_H.times(6.25*Math.pow(10, 21)) ;
    Util.print(" EIGENVALUES OF THE QUANTUM WIRE mev  \n");
    Util.print(E_H, true);
    }
}
```

Listing 4: QuantumWellEnergyLevelsFinder.java

```
/*
* QuantumWellEnergyLevelsFinder.java
*
* Created on April 7, 2006, 9:51 PM
*
*/
package javaapplication1;
import Jama.Matrix;
import Jama.EigenvalueDecomposition;
import Jama.QRDecomposition;
import Jama.LUDecomposition;
import java.util.Arrays;
import Jama.SingularValueDecomposition;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Date;
/**
*
* @author Sanam Moslemi-Tabrizi
*/
public class QuantumWellEnergyLevelsFinder {
    /** Creates a new instance of QuantumWellEnergyLevelsFinder */
    public QuantumWellEnergyLevelsFinder() {
    }
    /** set Np = 25**/
    public static int numberOfPoints = 25;
    public void findQuantumWellEnergyLevels(){
        double[][] txx = {
            {
                0., 0., 2.
            }
        }
        ;
        Matrix PTXX = new Matrix(txx);
        /** set lenght of the well **/
        double lengthInXDirection = 1 * Math.pow(10, -9);
        int m = 3;
        double mm = 0.067;
        double m_star = mm * Util.freeElectronMass;
        double h_hat = 1.054 * Math.pow(10, -34);
        double alpha=-(h_hat * h_hat)/(2.*m_star);
        Matrix V = new Matrix(numberOfPoints, 1);
        Matrix pot = new Matrix(numberOfPoints, 1);
        Matrix temp2 = new Matrix(1, 1);
        double x[] = new double[numberOfPoints];
        double E_exact[] = new double[numberOfPoints];
        x[0] = 0;
        double delta_x = lengthInXDirection / (numberOfPoints - 1);
        /** define cloud size **/
        double dx = 1.02 * delta_x;
        for (int i = 0; i < numberOfPoints; i++) {
            x[i] = x[0] + i * delta_x;
```

```
}
Matrix F = Util.getF(3, numberOfPoints, x);
Matrix FT = F.transpose();
Matrix A = new Matrix(numberOfPoints, numberOfPoints);
double U[] = new double[numberOfPoints];
double ax;
double phi[][] = new double[numberOfPoints][numberOfPoints];
double NXX[][] = new double[numberOfPoints][numberOfPoints];
int numberOfNZPoints= numberOfPoints − 2;
int[] r = new int[numberOfNZPoints];
int ri = 0;
for (int k = 0; k < numberOfPoints; k++) {
    for (int i = 0; i < numberOfPoints; i++) {
        ax = (x[k] − x[i]) / dx;
        double tx = (−ax ∗ ax) / 0.2;
        phi[k][i] = 1.2618∗Math.exp(tx);
        V.set(i,0,phi[k][i]);
        // V[i,0]= phi[k][i]
    }
    Matrix W = Util.getDia(V);
    /** M is the Moment Matrix**/
    Matrix M= F.times(W).times(FT) ;
    Matrix M_in = M.inverse();
    for(int n=0; n<numberOfPoints; n++) {
        double x0 = 1;
        double x1 = x[n];
        double x2 = x[n] ∗ x[n];
        double[][] temp1 = {
            {
                x0
            }
            , {
                x1
            }
            , {
                x2
            }
        }
        ;
        Matrix P = new Matrix(temp1);
        /** NXX is second derivative the shape function**/
        NXX[k][n] = phi[k][n]∗((PTXX.times(M_in).times(P)).get(0,0)) ;
    }
    /** Constructing the hamiltonian matrix, H and its submatrix H2**/
    if (x[k] == 0 || x[k]== lengthInXDirection) {
        U[k] = 0;
        for(int i=0; i<numberOfPoints; i++){
            A.set(k,i,0);
            pot.set(k,0,0);
        }
    }
    else {
        for(int i=0; i<numberOfPoints; i++){
                A.set(k,i,alpha*NXX[k][i]);
                pot.set(k,0,Util.getPotential(x[k]));
            }
            r[ri] = k;
            ri= ri+1;
        }
    }
    Matrix pot_func = Util.getDia(pot);
    Matrix H = A. plus(pot_func);
    Matrix H2 = H.getMatrix(r,r);
    /**obtaining the eigenvalues and eigenfunctions**/
    EigenvalueDecomposition D1 =
    new EigenvalueDecomposition(H2);
    double[] d1 = D1.getRealEigenvalues();
    Matrix wave = D1.getV();
    Map<Double, double[]> map = new LinkedHashMap<Double, double[]>();

    double[][] waveArray = wave.transpose().getArray();
    for(int i=0; i<numberOfPoints-2; i++)
    {
        Double d = d1[i];
        double[] m4 = waveArray[i];
        // ith coloumn
        map.put(d, m4);
    }
    Arrays.sort(d1);
    Matrix[] u_funcs = new Matrix[5];
    Matrix[] wave_function = new Matrix[5];
    for(int i=0; i<5; i++)
    {
        wave_function[i] = new Matrix(1 , numberOfPoints);
    }
    for(int i=0; i<5; i++)
    {
        double[] m4 = map.get(d1[i]);
        double[][] mm4 = {
            m4
        }
        ;
        Matrix w_m4 = new Matrix(mm4);
        u_funcs[i] = w_m4;
    }
    double beta = Math.pow(1/(delta_x ), 0.5);
    for (int i=0 ; i<5 ; i++)
    {
        if (u_funcs[i].get(0,0)<
        0)
        {
            u_funcs[i].timesEquals(-beta ) ;
        }
        else {
            u_funcs[i].timesEquals(beta ) ;
        }
        }
        for (int i=0 ; i<5 ; i++)
        {
            int u_funcs_index = 0;
            for (int j=0 ; j<numberOfPoints ; j++)
            {
                if (x[j] == 0 || x[j]== lengthInXDirection)
                {
                    wave_function[i].set(0,j,0);
                }
                else
                {
                    wave_function[i].set(0,j,u_funcs[i].get(0,u_funcs_index));
                    u_funcs_index = u_funcs_index + 1 ;
                }
            }
        }
        // make 5 matrixes of x, y, wavefunction for ploting the wavefunction
        Matrix[] plot_data = new Matrix[5];
        for(int i=0; i<5; i++)
        {
            plot_data[i] = new Matrix(numberOfPoints , 3);
        }
        for(int i=0; i<5; i++)
        {
            for(int j=0; j<numberOfPoints; j++)
            {
                plot_data[i]. set(j,0,x[j]);
                plot_data[i]. set(j,2,wave_function[i].get(0,j) );
            }
            System.out.print("wavefunion␣level␣n␣=␣");
            System.out.println(i+1);
            for(int j=0; j<numberOfPoints; j++)
            {
                Util.print(plot_data[i].get(j,2));
                System.out.println("␣,␣");
            }
        }
        double[][] dd1 = {
            d1
        }
        ;
        Matrix E_H= new Matrix(dd1);
        E_H= E_H.times(6.25*Math.pow(10, 21)) ;
        Util.print("EIGENVALUES␣OF␣H␣mev\n");
        Util.print(E_H, true);
    }
}
```

Listing 5: Util.java

```
/*
* Util.java
*
* Created on April 7, 2006, 9:54 PM
*
*/
package javaapplication1;
import Jama.Matrix;
import Jama.EigenvalueDecomposition;
import Jama.QRDecomposition;
import Jama.LUDecomposition;
import Jama.SingularValueDecomposition;
import java.util.Date;
/**
*
* @author Sanam Moslemi−Tabrizi
*/
public class Util {
    /** Creates a new instance of Util */
    public Util() {
    }
    public static double freeElectronMass = 9.11 * Math.pow(10, −31);
    public static double ElectronCharge = 1.6 * Math.pow(10, −19);
    static Matrix getDia(Matrix v) {
        int size = v.getRowDimension();
        assert size > 0;
        assert v.getColumnDimension() == 1;
        Matrix a = new Matrix(size, size);
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (i == j) {
                    a.set(i, j, v.get(i, 0));
                }
                else {
                    a.set(i, j, 0.);
                }
            }
        }
        return a;
    }
    /**
    * Generate magic square test matrix. *
    */
    public static Matrix magic(int n) {
        double[][] M = new double[n][n];
        // Odd order
        if ((n % 2) == 1) {
            int a = (n + 1) / 2;
            int b = (n + 1);
            for (int j = 0; j < n; j++) {
                for (int i = 0; i < n; i++) {
                    M[i][j] = n * ((i + j + a) % n) + ((i + 2 * j + b)
                    % n) +
                    1;
                }
            }
            // Doubly Even Order
        }
        else if ((n % 4) == 0) {
            for (int j = 0; j < n; j++) {
                for (int i = 0; i < n; i++) {
                    if (((i + 1) / 2) % 2 == ((j + 1) / 2) % 2) {
                        M[i][j] = n * n - n * i - j;
                    }
                    else {
                        M[i][j] = n * i + j + 1;
                    }
                }
            }
            // Singly Even Order
        }
        else {
            int p = n / 2;
            int k = (n - 2) / 4;
            Matrix A = magic(p);
            for (int j = 0; j < p; j++) {
                for (int i = 0; i < p; i++) {
                    double aij = A.get(i, j);
                    M[i][j] = aij;
                    M[i][j + p] = aij + 2 * p * p;
                    M[i + p][j] = aij + 3 * p * p;
                    M[i + p][j + p] = aij + p * p;
                }
            }
            for (int i = 0; i < p; i++) {
                for (int j = 0; j < k; j++) {
                    double t = M[i][j];
                    M[i][j] = M[i + p][j];
                    M[i + p][j] = t;
                }
                for (int j = n - k + 1; j < n; j++) {
                    double t = M[i][j];
                    M[i][j] = M[i + p][j];
                    M[i + p][j] = t;
                }
            }
            double t = M[k][0];
            M[k][0] = M[k + p][0];
            M[k + p][0] = t;
            t = M[k][k];
            M[k][k] = M[k + p][k];
            M[k + p][k] = t;
        }
        return new Matrix(M);
    }
```

```
/**
* Shorten spelling of print. *
*/
public static void print(String s) {
    System.out.print(s);
}
public static void print(double d) {
    System.out.format("%3.3E", d);
}
/**
* Format double with Fw.d. *
*/
public static String fixedWidthDoubletoString(double x, int w,
                int d) {
    java.text.DecimalFormat fmt = new java.text.DecimalFormat();
    fmt.setMaximumFractionDigits(d);
    fmt.setMinimumFractionDigits(d);
    fmt.setGroupingUsed(false);
    String s = fmt.format(x);
    while(s.length() <
    w) {
        s = "␣" + s;
    }
    return s;
}
/**
* Format integer with Iw. *
*/
public static String fixedWidthIntegertoString(int n, int w) {
    String s = Integer.toString(n);
    while(s.length() <
    w) {
        s = "␣" + s;
    }
    return s;
}
public static void print(Matrix a, boolean f) {
    for (int i = 0; i < a.getRowDimension(); i++) {
        System.out.print("row:␣" + i + "␣==>");
        for (int j = 0; j < a.getColumnDimension(); j++) {
            System.out.print(a.get(i, j) + "␣");
            if ((j%1)==0)
            {
                System.out.println();
            }
        }
        System.out.println();
    }
}
public static void print(Matrix a) {
    for (int i = 0; i < a.getRowDimension(); i++) {
        for (int j = 0; j < a.getColumnDimension(); j++) {
            System.out.format("%.3f␣", a.get(i, j));
            }
            System.out.println();
        }
    }
    /** we define potnetial functions as below **/
    /** potential functions are in unit of J. **/
    public static double getPotential(double x) {
        return 8 * x * Math.pow(10.,−11.);
    }
    public static double get2DPotential(double x, double y) {
        return 0;
    }
    public static double get3DPotential(double x, double y, double z) {
        return 0;
    }
    static Matrix getF(int r, int c, double[] x) {
        Matrix F = new Matrix(r, c);
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++) {
                if (i == 0) {
                    F.set(i, j, 1);
                }
                else {
                    double t = Math.pow(x[j], i);
                    F.set(i, j, t);
                }
            }
        }
        return F;
    }
    static Matrix get2DF(int r, int c, Matrix[] nod) {
        Matrix F = new Matrix(r, c);
        for (int i = 0; i <r ; i++) {
            for (int j = 0; j < c; j++) {
                if (i == 0) {
                    F.set(i, j, 1);
                }
                else if (i == 1){
                    F.set(i, j, nod[j].get(0,0));
                }
                else if (i == 2){
                    F.set(i, j, nod[j].get(0,1));
                }
                else if (i == 3){
                    F.set(i, j, nod[j].get(0,0) * nod[j].get(0,1) );
                }
                else if (i == 4){
                    F.set(i, j, nod[j].get(0,0) * nod[j].get(0,0) );
                }
                else if (i == 5){
                    F.set(i, j, nod[j].get(0,1) * nod[j].get(0,1) );
                }
            }
        }
        return F;
    }
    static Matrix get3DF(int r, int c, Matrix[] nod) {
        Matrix F = new Matrix(r, c);
        for (int i = 0; i <r ; i++) {
            for (int j = 0; j < c; j++) {
                if (i == 0) {
                    F.set(i, j, 1);
                }
                else if (i == 1){
                    F.set(i, j, nod[j].get(0,0));
                }
                else if (i == 2){
                    F.set(i, j, nod[j].get(0,1));
                }
                else if (i == 3){
                    F.set(i, j, nod[j].get(0,2));
                }
                else if (i == 4){
                    F.set(i, j, nod[j].get(0,0) * nod[j].get(0,1) );
                }
                else if (i == 5){
                    F.set(i, j, nod[j].get(0,0) * nod[j].get(0,2) );
                }
                else if (i == 6){
                    F.set(i, j, nod[j].get(0,1) * nod[j].get(0,2));
                }
                else if (i == 7){
                    F.set(i, j, nod[j].get(0,0) * nod[j].get(0,0) );
                }
                else if (i == 8){
                    F.set(i, j, nod[j].get(0,1) * nod[j].get(0,1) );
                }
                else if (i == 9){
                    F.set(i, j, nod[j].get(0,2) * nod[j].get(0,2) );
                }
            }
        }
        return F;
    }
}
```