POINT CHARGE SIMULATIONS OF THE

ELECTRIC FIELD GRADIENT

Physics 449 Undergraduate thesis

Author

Jordan Schultz
Department of Physics and Astronomy
University of British Columbia

Supervisor

Dr. R. Kiefl and Dr. W. A. MacFarlane
β-NMR Group
TRIUMF

April 2005
POINT CHARGE SIMULATIONS OF THE ELECTRIC FIELD GRADIENT
β-detected nuclear quadrupole resonance spectroscopy (β-NQR) is a powerful tool for condensed matter physicists to use in exploring the electrical properties of thin films and nano-structures. Recent β-NQR experiments performed on SrTiO$_3$ have led to speculation that there exists a non-axial electric field gradient (EFG) tensor at the interstitial site. To test these speculations, point charge simulations of the EFG were carried out under various perturbations in an attempt to theoretically re-produce these results.

Two important conclusions arose from the simulations. First, in the simplest approximation (where all ions were considered point charges at undistorted lattice positions) the theoretical EFG is a factor of three larger than the experimental value. Second, defects; such as a vacancy; need to be within 8Å in order to produce non-axial terms in the EFG large enough to be consistent with observations.
# TABLE OF CONTENTS

ABSTRACT .................................................................................................................. 3

TABLE OF CONTENTS ............................................................................................ 4

LIST OF FIGURES AND TABLES ............................................................................... 6

1. INTRODUCTION ..................................................................................................... 8
  1.1 – Introduction to NMR Theory and Applications .............................................. 8
  1.2 – A Conceptual Description of NMR and NQR ................................................. 9
  1.3 – A Mathematical Description of NQR .......................................................... 12
    1.3.1 – Simplified NMR ..................................................................................... 12
    1.3.2 – NQR ..................................................................................................... 14
  1.4 – Broadening NMR Applications with $\beta$ Decay ........................................... 17

2. MOTIVATION ......................................................................................................... 20
  2.0 – Overview ...................................................................................................... 20
  2.1 – Experimental Setup .................................................................................... 20
    2.1.1 – The Polarized Beam .......................................................................... 21
    2.1.2 – The Spectrometer .............................................................................. 22
    2.1.3 – Experimental Process ........................................................................ 23
  2.2 – Expected Results ......................................................................................... 24
  2.3 – Measured Results ....................................................................................... 26

3. SIMULATION ALGORITHM .................................................................................. 29
  3.0 – Overview .................................................................................................... 29
  3.1 – Basics of Crystal Structures ...................................................................... 29
  3.2 – Calculations of the EFG ............................................................................ 31
  3.3 – Simulation Algorithm ............................................................................... 33
    3.3.1 – Summation ......................................................................................... 33
    3.3.2 – Perturbations ...................................................................................... 35

4. RESULTS .............................................................................................................. 37
  4.0 – Overview .................................................................................................... 37
  4.1 – Bulk EFG Calculations .............................................................................. 38
    4.1.1 – Ti Substitutional Site ......................................................................... 38
    4.1.2 – Interstitial Site .................................................................................... 40
    4.1.3 – Perturbing the Nearest Neighbours ..................................................... 42
  4.2 – Vacancy Perturbations .............................................................................. 44
  4.3 – Near Surface Perturbations ...................................................................... 46
5. DISCUSSION
5.0 – Overview ...........................................................................48
5.1 – Physical Valuation of Simulation Results .........................................48
  5.1.1 – Bulk Calculations .................................................................48
  5.1.2 – Vacancy Calculations ............................................................50
  5.1.3 – Surface Calculations .............................................................51
5.2 – Sources of Error .......................................................................53
  5.2.1 – Computational Errors .............................................................54
  5.2.2 – Algorithmic Errors ...............................................................55
  5.2.3 – Theoretical Errors ...............................................................56

6. CONCLUSIONS ............................................................................58

APPENDIX A: SIMULATION CODE .................................................59
  • EFG_PC_Calc.cpp ........................................................................60
  • options.h ..................................................................................63
  • classes.h ..................................................................................64
  • supplement_functions.h ...............................................................68
  • initialize_functions.h .................................................................74
  • menu_functions.h ....................................................................78
  • sim_functions.h .......................................................................80
  • menu_options.h .....................................................................88
  • Lattice_info.dtt .......................................................................104

7. BIBLIOGRAPHY ..........................................................................105
LIST OF TABLES AND FIGURES

CHAPTER 1

- Figure 1.1: Spin and magnetic moment .............................................9
- Figure 1.2: The NMR Process ...........................................................10
- Figure 1.3: Quadrupole orientation energy ......................................11
- Figure 1.4: β Decays ....................................................................17

CHAPTER 2

- Figure 2.1: The Polarizer ...............................................................21
- Figure 2.2: The Spectrometer .........................................................22
- Figure 2.3: SrTiO₃ Crystal Structure .............................................24
- Figure 2.4: Axially symmetric transition energies .........................25
- Figure 2.5: Measured β-NQR Spectrum .......................................26
- Figure 2.6: Axially asymmetric transition energies .....................28

CHAPTER 3

- Figure 3.1: Translation and Primitive vectors ............................30

CHAPTER 4

- Table 4.1: SrTiO₃ Lattice and Basis parameters .........................37

- Figure 4.1: EFG Convergence in Ti Substitutional Site .............39
- Figure 4.2: EFG off-axis Convergence in Ti Site .......................39
- Figure 4.3: η Convergence in interstitial site .........................40
- Figure 4.4: νₜ Convergence in interstitial site .........................41
- Figure 4.5: νₜ Dependence on nearest neighbour position ........43
- Figure 4.6: νₜ Dependence on nearest neighbour flexing ..........44
- Figure 4.7: νₜ Dependence on distance to vacancy .................45
- Figure 4.8: η Dependence on distance to vacancy ..................45
- Figure 4.9: Perpendicular and Parallel interstitial sites ..........46
- Figure 4.10: νₜ Dependence on distance to surface .................46
- Figure 4.11: η Dependence on distance to surface .................47
CHAPTER 5

- Table 5.1: EFG calculations above the surface ..................................52
- Figure 5.1: Different Charge Distributions ...........................................56
- Figure 5.2: Shielding and Anti-shielding ............................................57
1

INTRODUCTION

1.1 – Introduction to NMR Theory and Applications

The discovery of nuclear magnetic resonance (or NMR) has been a fairly recent event, on the time scale of scientific discoveries, occurring around sixty years ago in the 1940’s. In this short time span, however, applications of NMR have appeared in nearly all corners of the scientific universe. Medicine and biology have found a tool of unprecedented potential in NMR tomography (MRI), while chemists studying in both organic and inorganic fields have applied NMR spectroscopy to study material make-up and elemental bonding with great success.

In the context of condensed matter, NMR is a primarily used to investigate the magnetic and electronic properties of materials. The information gathered is very specific, and is not necessarily limited by intensity or strength\(^1\). Furthermore, and more importantly for the purposes of this paper, NMR can be performed with zero magnetic field via nuclear quadrupole resonance (NQR). NQR occurs when a nucleus with a quadrupole moment resides an electric field that has non-zero derivatives in any direction. Studies of this nature are also highly sensitive to quantities of interest; so NQR provides useful tool for probing the fundamental properties of various materials.

\(^1\) Slichter provides a particularly compelling example of observing the nuclear paramagnetism of iron overtop a background of electronic ferromagnetism, see [1].
Before presenting a mathematical analysis of NMR theory, it will be useful to provide a heuristic outline of how NMR and NQR work. Mainly, this will provide those unfamiliar with NMR an idea of the motivating physical theories at work. It is also intended to act as a stepping-stone into the mathematics, hopefully making this step easier to follow. From there, I’ll present the slightly more in-depth derivation of these concepts. This will primarily be for the purposes of supporting the project motivation presented in section 2. Finally, with an understanding of basic NMR processes, I’ll present the motivation behind β-NMR and β-NQR.

1.2 – A Conceptual Description of NMR and NQR

Nuclear magnetic resonance has its roots, as do many recently discovered physical phenomena, in quantum mechanics. With quantum mechanics came the notion that some nuclei possess spin, a property that is described as a natural attribute intrinsic to the particle (somewhat, in this respect, like charge or mass). One very important effect of this spin, however, is that it gives the particle a slight magnetic dipole moment (see figure 1.1), and it is this property that is exploited in NMR.

As one might expect from a quality arising from quantum mechanics, a particle’s spin (and thus it’s magnetic moment) is quantized, able to exist in only a few discrete energy levels for simple nuclei. Thus, if one were to attempt to shift a nucleus from one energy level to another, they would require a very precise amount of energy. Since this energy is typically transferred by an oscillating magnetic field, this corresponds to a very...
specific resonant frequency (specifically, $E = h\nu$, where $\nu$ is the resonant frequency). Moreover, the energy required to induce these transitions in a certain nucleus is typically specific to that nucleus (based on the ‘gyromagnetic’ or ‘magnetogyric’ ratio), making it possible to target specific components of a material during measurements.

Figure 1.2 depicts the basic idea behind NMR spectroscopy. A material is placed in a strong, static magnetic field ($B_0$) in order to align the nuclear magnetic moments in one direction. Perpendicular to this static field, a smaller magnetic field ($B_1$) is set oscillating at some frequency in an attempt to induce transitions between the lower energy state (when the nuclear magnetic moment is in line with the static field) and the higher one (when the magnetic moment precesses with the oscillating field). By altering either the strength of the static field or the oscillation frequency of the weaker field, one can measure the net magnetic moment of the material to observe at what point resonance occurs, and use this information to infer various properties of the material\(^2\).

Nuclear quadrupole resonance is also founded on the principle of inducing changes in a population’s nuclear spins energy levels; however, in this case the splitting

\(^2\) Other information, such as the time it takes for the magnetic moments to re-align with the static field after the oscillating field is turned off (called the ‘relaxation time’), is also of general importance; however, as this data is not particularly relevant to the present work, these behaviours will not be discussed.
is obtained by the orientation of the nucleus’ quadrupole moment. The exact definition of a quadrupole moment will be presented later, but for now it should suffice to suggest that a quadrupole moment is a signature of an aspherical charge distribution. Given a particular electric potential, such a charge distribution will find it energetically favourable to place itself in a specific orientation (as in figure 1.3), which will then dictate the lowest spin state. Since altering the spin state would involve altering the quadrupole’s orientation, this requires a certain amount of energy.

Typically, this effect acts as a perturbation in NMR spectroscopy, but it can act as the primary method for separating the spin state energy levels. This latter case is typically referred to as “Zero Field” (or ZF) NQR, as the large static magnetic field is no longer required. The remainder of this paper will be concerned with only ZF NQR, so the “ZF” will not usually be mentioned from here on, and it will be considered implied.
1.3 – A Mathematical Description of NQR

As was mentioned, both NMR and NQR depend on the net angular momentum of a nucleus being quantized. It is typical, in quantum mechanics, to define the angular momentum $J$ in terms of an angular momentum vector operator, $I$, as follows

$$J = \hbar I$$  \hspace{1cm} (1.1)

where $I$ (1) is dimensionless; (2) commutes (in each component $I_x$, $I_y$, $I_z$) with $I^2$; (3) has, for each component $I_x$, $I_y$ and $I_z$, eigenvalues $m = -I, -I+1, \ldots, I-1, I$; and finally (4) satisfies $I^2$ having eigenvalues $I(I+1)$, such that $I$ is either a full or half integer.

These “facts” about $I$ form the basis for magnetic resonance. They suggest that a Hamiltonian that is dependent on $I$ will have a finite number of states, whose energy will be proportional to $m$ (resulting in discrete energy levels). More importantly, transitions between states $m$ and $m'$ will obey

$$\langle m' | I_x | m \rangle = \delta_{m,(m'\pm 1)}$$  \hspace{1cm} (1.2)

In other words, the angular momentum can only make transitions to the nearest eigenstates. I will attempt to briefly illustrate the usefulness of this using a simplified NMR example, before presenting the slightly more difficult case of quadrupole interactions.

1.3.1 – Simplified NMR

As was mentioned, the nuclear magnetic moment, $\mu$, arises from the nuclear spin. Mathematically, this relation is represented as

$$\mu = q I$$

These derivations are carried through with a little more care for detail in chapters 1 (for NMR) and 10 (for NQR) of [1], should one desire a more rigorous analysis of the relations examined here.
\[ \mu = \gamma I \]  

(1.3)

where \( \gamma \) is the gyromagnetic ratio mentioned earlier. Equation 1.3, in conjunction with equations 1.1 and 1.2, implies transitions between magnetic moment states must obey

\[ (m \mid \mu_s \mid m') = \gamma h (m \mid I_s \mid m') \]  

(1.4)

Thus, if one were to immerse a magnetic dipole in a magnetic field \( H = H_0 \mathbf{z} \) oriented in the \( \mathbf{z} \)-direction, the Hamiltonian of such a system would be

\[ H = -\mu \cdot H = -\gamma h H_0 I_z \]  

(1.5)

which yields the energy eigenvalues

\[ E = -\gamma h H_0 m, \ m = -I, \ldots, I \]  

(1.6)

being a set of \( 2I+1 \) eigenstates, separated by a constant energy (\( \gamma h H_0 \)). To induce transitions between two of these states requires a time dependent energy being introduced to the dipole

\[ \omega h = \Delta E \]  

(1.7)

where \( \omega \) is the angular frequency of supplied energy.

In NMR, this transition energy is achieved by applying a transverse magnetic field that oscillates at angular frequency \( \omega \). Equating this with the gap between energy levels derived from equation 1.6, we obtain a relation for the resonance frequency

\[ \omega h = \Delta E = \gamma h H_0 \]  

(1.8)

\[ \Rightarrow \omega = \gamma H_0 \]

At this frequency, transitions between all states will occur until the oscillating field is turned off (or the frequency changed), at which point the magnetic moments will re-align with the static field.
1.3.2 – NQR

To derive the quadrupolar interaction, we follow a very similar procedure to that used in section 1.3.1. We start, in this case, by stating the Hamiltonian of a charge distribution $\rho(r)$ placed in a potential field $V(r)$

$$ H = \int_{all \ space} \rho(r)V(r)d\tau $$

(1.9)

To make this integral more manageable, the potential is Taylor expanded about the center of charge, giving 1.9 as

$$ H = V(0)\int \rho d\tau + \sum_{i} V_{i} \int x_{i} \rho d\tau + \frac{1}{2!} \sum_{i,j} V_{ij} \int x_{i} x_{j} \rho d\tau + O(x^{3}) $$

(1.10)

where $i$ and $j = x,y,z$; and the sums are over these three co-ordinates. Furthermore, partial derivative notation has been adopted on the electric potential; thus

$$ V_{i} \equiv \frac{\partial V}{\partial x_{i}} \bigg|_{r=0} \quad \text{and} \quad V_{ij} \equiv \frac{\partial^{2}V}{\partial x_{i} \partial x_{j}} \bigg|_{r=0} $$

(1.11)

Examining the terms in equation (1.10) a little more closely can help simplify the problem. The first term is equivalent to the energy contribution of a point charge. The second term is the dipole interaction, which vanishes for nuclei. The third term represents the quadrupole interaction Hamiltonian, and as it is the first term that is not constant, it will now be the only subject of analysis. Since the nine-element tensor $V_{ij}$ will play a rather important role in these calculations, it will typically be referred to as the electric field gradient (or EFG) in future discussions.

---

4 See [2]

5 Reasoning for this is presented in chapter 10 of [1]
At this point, it becomes convenient to express the energy contribution due to quadrupole interactions in terms the quadrupole moment tensor of the nuclear charge distribution $\rho(r)$, $Q_{ij}$, which is defined as

$$Q_{ij} = \int (3x_i x_j - \delta_{ij} r^2) \rho(d\tau)$$

When this is placed in the 3rd term of (1.10), the following equation for the quadrupole Hamiltonian is obtained

$$H^{(Q)} = \frac{1}{6} \sum_{i,j} \left( V_{ij} Q_{ij} + V_{ij} \delta_{ij} \int r^2 \rho(d\tau) \right)$$

This equation can be even further simplified by noting a few special features of the EFG tensor. First, since it is based on a scalar potential, it is always possible to choose the co-ordinate system such that

$$V_{ij} = 0 \text{ if } i \neq j$$

The particular system that gives rise to (1.14) being true is called the “principle co-ordinate system”. Furthermore, since $V$ obeys Laplace’s equation

$$\nabla^2 V = 0 \Rightarrow \text{ (with 1.14) } \sum_i V_{ii} = 0$$

Incorporating 1.15 into 1.13 eliminates the 2nd term, thus

$$H^{(Q)} = \frac{1}{6} \sum_{i,j} V_{ij} Q_{ij}$$

---

6 Interestingly, the two constraints outlined in (1.14) and (1.15) reduce the 9-element EFG tensor down to two independent components. These components are conventionally named $v_4$ and $\eta$, which will be defined later.
Unfortunately, expressing the quadrupole tensor in terms of spin operators requires the use of some rather elaborate math (namely, the Wigner-Eckart theorem). Nevertheless, after a few pages of algebra it can be shown that

$$H^{(Q)} = \frac{eQ}{6I(2I-1)} \sum_{i,j} V_{ij} \left[ \frac{3}{2} (I_i I_j + I_j I_i) - \delta_{ij} I^2 \right]$$

(1.17)

where $e$ is the charge of a proton, and $Q$ is the quadrupole moment of the nucleus. Once again, substituting equations 1.14 and 1.15 in here allow for simplification of the quadrupole Hamiltonian; specifically

(1.14) $\Rightarrow H^{(Q)} = \frac{eQ}{6I(2I-1)} \left[ V_{xx} \left( 3I_x^2 - I^2 \right) + V_{yy} \left( 3I_y^2 - I^2 \right) + V_{zz} \left( 3I_z^2 - I^2 \right) \right]$  

(1.18)

and

(1.15) $\Rightarrow H^{(Q)} = \frac{eQ \cdot V_{zz}}{4I(2I-1)} \left[ \left( 3I_z^2 - I^2 \right) + \left( \frac{V_{xx} - V_{yy}}{V_{zz}} \right) \left( I_x^2 - I_y^2 \right) \right]$  

(1.19)

By convention, the diagonal elements of the EFG are ordered in ascending magnitude, i.e.

$$|V_{xx}| \leq |V_{yy}| \leq |V_{zz}|$$

(1.20)

which allows for the definition of an EFG asymmetry parameter, $\eta$, defined as

$$\eta = \frac{V_{xx} - V_{yy}}{V_{zz}} , \in [0,1]$$

(1.21)

In many instances, especially for cubic crystal structures, the EFG will be axially symmetric (Meaning $V_{xx} = V_{yy}$, and thus $\eta = 0$). In this case, equation 1.19 simplifies even further, yielding a very simple set of energy levels.
1.4 – NMR Applications with β Decay

There is no doubt that NMR’s reliance on atomic processes is a great strength in its application to spectroscopy, however in certain scenarios it can be equally detrimental. In particular, NMR requires a large number of nuclei ($\sim 10^{20}$) to produce a reasonable signal. While this is certainly not an obscene amount when one considers the relative size of an atom (for example: 1 mol of atoms, which is typically on the scale of 1 g in mass, contains over $10^{23}$ atoms), it is easy enough to envision circumstances that warrant a more sensitive form of measurement, such as the study of nano-structures, or of thin films, to name a few. As the reader has likely guessed by now, there does exist a method for treating these cases, which incidentally involves the emission of β particles from radioactive decay. Before explaining this method in further detail, then, it is worth briefly explaining the process of β decay.

β decay is a special kind of radioactive decay, which involves the emission of a β particle. For example, a neutron in a radioactive nucleus may decay into a proton, emitting an electron and neutrino in the process (The types of β decay are noted in figure 1.4). While these reactions have notable historical importance (such as contributing to the discovery of neutrinos), the key feature for the purposes of this project is the violation of parity. The β particle, it turns out, is emitted preferentially opposite the nuclear spin orientation.

This property of the β particle being emitted anti-parallel to the spin of the nuclei suggests a far more sensitive method of measuring the spin of β-emitting particles, for it

\[ e^- + ^{\text{1}_2 \text{Z}} \rightarrow ^{\text{1}_2 \text{Z}} + \nu_e \]
\[ e^+ + ^{\text{1}_2 \text{Z}} \rightarrow ^{\text{1}_2 \text{Z}} + \nu_e \]
\[ e^- + ^{\text{1}_2 \text{Z}} \rightarrow ^{\text{1}_2 \text{Z}} + \nu_e \]

Figure 1.4: Possible β decays: (1) electron absorption, (2) positron emission, (3) electron emission.

---

\(^7\) See [3] for more details on β decay processes.
reduces the relatively difficult task of detecting a small magnetization to the relatively easy task of detecting a high energy β-particle. Unfortunately, there are many materials studied in NMR Spectroscopy that do not sporadically emit β particles; so one final leap of imagination is necessary to complete this method.

The solution, which may or may not be obvious, is to simply implant the unstable nuclei. Experimenters using β-NMR typically implant these probe nuclei in a target and use the data from the resonance frequency and relaxation time of the probe to extract information regarding the various interstitial and substitutional sites inside the structure being examined\(^8\).

Following a similar train of thought as with conventional NMR, one might note that if a probe nucleus possessed a quadrupole moment, interactions with the local EFG could be used to split the polarization energy levels, in the same way as in conventional NQR. This is precisely the philosophy behind β-NQR, and so one could exploit all the relations derived for NQR to explore the properties of nano-structures and thin films using this technique\(^9\).

The fact that the NQR relations will still apply in β-NQR results in two rather important implications. First, β-NQR can be performed in zero-field. Second, in these ZF conditions, the strong dependence on the EFG can lead to clear resolution between the various sites where the probe nucleus comes to rest. Since the EFG is very sensitive to the local atomic structure, it is almost guaranteed that each site in a crystal structure is represented by a distinct tensor. Furthermore, any defects such as vacancies or impurities near certain sites would result in yet another tensor, and thus another resonant frequency.

\(^8\) See [4] for more details on the motivation and applications of β-NMR

\(^9\) See [5] and [6] for more details on the motivation and applications of β-NQR
Studying the various resonance peaks in a β-NQR spectrum, therefore, can supply one with a wealth of information regarding not only the number of sites in a crystal, but also the relative distribution of all the sites (or rather, the electronic properties resulting from these configurations).

As the focus of this paper is ZF β-NQR, I will from here on be speaking strictly of this phenomenon and experiments that make use of it. It is important to note, then, that not all the material presented in this introduction is directly pertinent to this subject. It was put forth both to lead the reader through the developments in NMR that led to β-NQR, and to provide some appreciation for the vast field of NMR, of which β-NQR makes up only a small subset.
MOTIVATION

2.0 – Overview

This project was spawned from the results of a β-NQR experiment recently performed at TRIUMF. In this experiment, low energy $^8$Li ions were implanted into a SrTiO$_3$ crystal, and an oscillating magnetic field used to induce transitions between spin states (as detailed in the previous section). Such a configuration should, in theory, have shown a shallow, narrow resonance peak. Instead, the results yielded a deep resonance dip, with an anomalous asymmetric lineshape. The goal of this project was to investigate a theory that attempts to account for this behavior.

In this section, I will briefly review the experimental setup that yielded these results, followed by a justification for expecting small, sharp resonance peaks. I will then go over the measured results of this experiment, and detail why the suggested theory could potentially explain the observations resulting from this experiment.

2.1 – Experimental Setup

The experiment was performed using an apparatus that is made up of two main components: a polarized beam, and a spectrometer. To quickly summarize this setup, an isotopically pure beam of $^8$Li$^+$ supplied to the polarizer, which aligns the nuclear spin of $^{10}$

---

\[10\] It should be noted that neither the setting up nor the carrying out of this experiment was part of the project being presented in this paper. Most of the information and figures in section 2 has been reproduced from a paper authored by the scientists who originally performed this experiment [5].
the ions in the beam. The polarized beam is then fed into the spectrometer, which houses a target sample of SrTiO$_3$, and is used to measure the polarization of the $^8$Li$^+$ in this target as an oscillating magnetic field is applied. This section will describe these two elements in more detail, and then briefly outline the procedure used in a $\beta$-NQR experiment.

2.1.1 – The Polarized Beam

Isotopes for this experiment were created in the ISAC facility at TRIUMF. Since this facility is capable of producing a large variety of isotopes, certain criteria were needed to choose a specific one. The primary criterion for a $\beta$-NQR probe nucleus is that it can be produced at high rates (> $10^4$ per second reaching the spectrometer), and it can be easily polarized. $^8$Li$^+$ ($I = 2$) was chosen for being the lightest isotope satisfying these conditions. Other redeeming features of this choice are that the average lifetime ($\tau = 1.2s$) is short enough to be compatible with spin relaxation times in solids, the quadrupole

![Diagram](image)

Figure 2.1: The incoming Li ion beam is neutralized, and orbital transitions of the valence electrons are induced to polarize the nucleus. The neutral Li is then re-ionized and guided to the spectrometer
moment \((Q = 33 \text{ mB})\) is fairly low, and the emitted \(\beta\) particles are released with more than enough energy to ensure easy detection \((E_{\text{ave}} = 6 \text{ MeV})\).

After the \(^8\text{Li}^+\) ions are created, they are accelerated to 30 keV \((\pm 2 \text{ eV})\), and guided electrostatically into the polarizer (see figure 2.1). The basic concept behind the polarizer is to place neutral atoms in a small magnetic field, and excite the valance electron from \(2s \rightarrow 2p\) orbital using a monochromatic, circularly polarized laser. After the valance electrons make a few transitions between orbitals (excited up by the laser, and spontaneously dropping back down), both the electronic and nuclear spin achieve a high degree of polarization\(^{11}\). Testing done on this beam showed typical polarizations of about 70%.

2.1.2 – The Spectrometer

The spectrometer (shown in figure 2.2) has two main functions: to create the magnetic field that will be used to induce transitions in the \(^8\text{Li}^+\) spin states, and detect the \(\beta\) particles emitted when the \(^8\text{Li}^+\) decays in order to determine the polarization. To achieve this, a target is mounted onto a

\(^{11}\) There are other techniques to this process, which are not discussed here. These involve neutralizing and re-ionizing the \(^8\text{Li}\) beam, and a fair amount of physics that I glossed over, which explain why this technique works. For more information on this, however, it is probably best to refer the reader back to the source \([3]\).
stage that is placed in an Ultra High Vacuum (UHV) chamber. Surrounding this stage are two small coils that are used to induce the alternating magnetic field in the sample. The entire system rests on a high-potential plate that is used to effectively slow the beam down to a few keV and control the implantation depth.

Two plastic scintillation detectors situated outside the UHV chamber were used to detect the \( \beta \) particles. As was mentioned, the \(^{8}\text{Li}^+\) decay process emits \( \beta \) particles with more than enough energy to escape the sample and pass through the steel windows (each 0.05 mm thick) to escape the UHV chamber.

Finally, a set of 3 coils just outside the chamber can be used to apply a static magnetic field between 0 and 15 mT. This is used to ensure zero magnetic field inside the sample with a precision of 5 \( \mu \)T.

**2.1.3 – Experimental Process**

Once the equipment is in place and operational, the experimental process is fairly straightforward. A trial consists of implanting a \(^{8}\text{Li}^+\) ions into the sample, which is held in a magnetic field that is oscillating at frequency \( \nu \). As the \( \beta \)-particles are counted by the detectors, the time-averaged nuclear polarization, \( P_z(\nu) \) is computed by the relation

\[
AP_z(\nu) = \frac{L(\nu) - R(\nu)}{L(\nu) + R(\nu)}
\]

(2.1)

where \( L(\nu) \) and \( R(\nu) \) are the total counts from the left and right detectors (respectively) at frequency \( \nu \). \( A \) is the constant of proportionality between the \( \beta \)-decay asymmetry and \( P_z(\nu) \). At the resonance frequency, the time averaged polarization is destroyed, giving rise to a dip in \( P_z(\nu) \).
2.2 – Expected Results

In this particular experiment, resonance is obtained when the polarization of the $^8\text{Li}^+$ particles is destroyed by inducing transitions between spin states. Since the quadrupole moment ($Q = 33\text{mB}$) and spin ($I = 2$) of the probe ions are fixed, equation 1.19 implies this frequency will be entirely determined by the EFG at the site that the $^8\text{Li}^+$ ions come to rest.

Furthermore, it is expected that the implanted Li ion will come to rest in a well-defined lattice site (either interstitial or substitutional)$^{12}$. The crystal structure of SrTiO$_3$ (figure 2.3) seems to suggest that regardless of which of these sites is occupied, the site will exhibit an axially symmetric crystal structure, and thus an axially symmetric EFG is anticipated.

Compiling this information regarding the Li ion and axially symmetric EFG serves to immensely simplify equation (1.19), so that in this case it may be reduced to

$$H^{(Q)} = h\nu_q (I_z^2 - 2)$$

(2.2)

---

$^{12}$ I have left a review of crystal structures to section three. If these terms are unfamiliar, interstitial in this case simply implies “between the atoms” (as is depicted in figure 2.2), while substitutional implies the Li ion replaces one of the ions already existing in the lattice.
where \( \nu_q = eQV_{zz} / 8h \) (for \( I=2 \)), and \( h \) = Planck’s constant = \( 6.626 \times 10^{-34} \) J·s. From equation 2.2, the energy eigenstates can be readily extracted (figure 2.4)
\[
E_m = h \nu_q (m^2 - 2) \quad (2.3)
\]
This implies there should exist two resonant frequencies, \( \nu_q \) and \( 3\nu_q \), corresponding to the allowed transitions between \( 0 \leftrightarrow \pm 1 \) and \( \pm 1 \leftrightarrow \pm 2 \) states, respectively.

Given this, it should now be possible to predict the size of the resonant polarization dip one would observe in this experiment. To do this, it is convenient to introduce a polarization vector \( p \), where \( p \) has 5 elements (\( p_m \)) that each represent the probability of a \(^8\)Li\(^+\) ion being in state \( m \). The polarization of the Li ions, \( P_z(v) \), can then be expressed as
\[
P_z(v) = \frac{1}{2} \sum_{m=-2}^{2} mp_m \quad (2.4)
\]
For example, consider a perfectly polarized beam, \( p = <1,0,0,0,0> \), and \( P_z(v) = 1 \).

Once \( B_1 \) oscillates at the resonant frequency, \( 3\nu_q \), transitions between the \( m=2 \) and \( m=1 \) state would result in an equal probability that the Li ion be in either of these states. This would reduce \( p \) to \(<\frac{1}{2}, \frac{1}{2}, 0, 0, 0>\), and thus \( P_z(v) \) would become \( \frac{3}{4} \). In such a case, then, one would expect to witness a resonance dip of 25\% in polarization\(^{13}\).

\(^{13}\) As was mentioned, the beam used in the experiment is not completely polarized, having \( P_z(0) \) closer to 0.7. Notice, however, that any configuration of \( p \) that satisfies this is still bound by at most a 25\% resonant dip at \( 3\nu_q \)
2.3 – Measured Results

Figure 2.5 shows the $\beta$-NQR spectrum about the $3v_q$ resonant frequency. There are two features to this figure that are in sharp contrast with the hypothetical scenario previously calculated: the large drop in polarization, nearly three times the predicted amount; and the large width of the resonance dip\textsuperscript{14}.

These results show strong evidence for a small perturbation in the basic quadrupolar Hamiltonian. This perturbation, however, would need to have two effects: it would need to split the $\pm m$ state degeneracy (to explain the width), and it would need to

\textsuperscript{14}Typically, the gyromagnetic ratio and the amplitude of the oscillating magnetic field determine the width of the resonance peak. In this case, the width should be $\gamma_{\perp} B_1 \sim 500$ Hz.
result in new eigenstates which were mixtures of the unperturbed eigenstates (to explain the large resonance amplitude)\(^{15}\).

There are at least two possible perturbations that could account for this behavior. The first of these is a perturbation due to a small, stray magnetic field perpendicular to the symmetry (\(z\)) axis. While this would account for the state mixing observed in the \(\beta\)-NQR spectrum, however, the degeneracy splitting would only occur in second order (proportional to \((\gamma B)^2\)) – likely too small an effect to account for the shift that was measured.

This is not to say the effect can be ignored. For one, this perpendicular term might be only one of many perturbations causing the anomalies in the spectrum. Also, if any component of the stray magnetic field were parallel to the symmetry axis, this would shift the degeneracy in first order (proportional to \(\gamma B\)). However, these possibilities were experimentally tested by artificially introducing a small field. It was concluded that they could not explain the observed results.

A second possibility involves crystal imperfections that result in a non-axial EFG (meaning \(\eta \neq 0\)). Recall that in deriving equation (2.2) it was argued that the \(^{7}\)Li ions were likely coming to rest at an axially symmetric site inside the crystal. It is conceivable, however, that a local change in the crystal structure (such as a vacancy, or nearby surface) could destroy this symmetry, rendering equation (2.2) incorrect. It turns out, incorporating a non-zero \(\eta\) parameter results in both \(\pm m\) state mixing, and a splitting of

\(^{15}\) Further evidence for a perturbation that mixes the \(m=\pm 1\) states was obtained by 1) noting a nearly negligible resonance peak at the \(v_q\) frequency, and 2) measuring a polarization dependence on crystal orientation
the degenerate energy eigenvalues (see figure 2.6 for a schematic of these new energy levels, and the allowed transitions).

Since it is unclear exactly how drastic the imperfections need to be in order to explain the $\beta$-NQR spectrum, the proposed project is to simulate the EFG using a point-charge model of the crystal lattice. Once this is achieved, various scenarios under which an asymmetry in the EFG might arise can be simulated.
3

SIMULATION ALGORITHM

3.0 - Overview

Calculations of the electric field gradient were done, in this project, using the point charge model. The basic concept behind this model is to approximate each ion in a crystal lattice by its ionic charge, centered at a point of radius zero. There are, however, some other aspects to the algorithm used in this project that are worth discussing at this point, as well.

To begin this section, a brief overview of simple crystal lattice structures will be outlined. Following that, I will propose a general derivation of the EFG in a point charge lattice, and point out some properties that will prove useful in the simulation. With these two principles covered, it will be possible to discuss a practical method of calculating the EFG, which was the one used in this project.

3.1 - Basics of Crystal Structures

An ideal crystal, which is what will be dealt with in this project, is simply a periodic repetition of identical atomic structures. Such a crystal is made of two key components: a lattice, and a basis\(^\text{16}\). A lattice is the set of points in space at which each atomic structure is located. Given the periodic nature of an ideal crystal, one should be able to find (at least) three vectors, \(\mathbf{a}_1\), \(\mathbf{a}_2\), and \(\mathbf{a}_3\) (in 3 dimensions), such that translation along these three vectors “moves” from one lattice point to another. In fact, since this

\(^{16}\) See [7] for a detailed introduction to crystals
A new lattice point should be indistinguishable from the first, this operation should work any number of times, both forward and backwards.

In more precise terms, suppose a point \( \mathbf{r} \) was in the lattice. Another point in the lattice, then, will be found at

\[
\mathbf{r}' = \mathbf{r} + u_1\mathbf{a}_1 + u_2\mathbf{a}_2 + u_3\mathbf{a}_3
\]

(3.1)

for any integers \( u_1, u_2, \) and \( u_3 \). If (3.1) describes every point in a lattice, \( \{\mathbf{a}_i\} \) are called the \textit{primary vectors} for the lattice. This idea is depicted in two dimensions in figure 3.1.

In practice, it is convenient to define a lattice point at the origin, which allows the rest of the lattice points to be defined solely by three integers and the primary vectors.

An interesting quantity that arises from a set of primary vectors is called the \textit{unit cell}, which is simply put the volume swept out by the primary vectors. Since these unit cells need to fit together with no empty space left behind, there is an extremely limited number of physically possible cells (to be exact, there only exist seven systems, which make up fourteen different “Bravais” lattices).

Given a unit cell, defined by a set of primary vectors, one can now place atoms at any position inside the unit cell to define what is called a \textit{basis}. Since these unit cells will all fit together to perfectly fill the volume of the crystal, simply placing this basis in every unit cell will fulfill the original definition of an ideal crystal. To put this idea in more mathematical terms, a basis will consist of \( N \) atoms, with each atom located at a vector.

![Figure 3.1: Translation vectors. Notice the vectors \( \mathbf{a} = (a_1, a_2) \) cannot be considered primary vectors, however \( \mathbf{b} = (b_1, b_2) \) can.](image)
\[ \vec{r}_i = x_i \vec{a}_1 + y_i \vec{a}_2 + z_i \vec{a}_3 \] (3.2)

from the origin of the basis, where \( x_i, y_i, \) and \( z_i \) are fractions bounded between \([0,1)\). Thus, if we take the origin to be at a lattice point, the position of any atom in the crystal can be described by the vector

\[ \vec{r}_i = (x_i + u_1) \vec{a}_1 + (y_i + u_2) \vec{a}_2 + (z_i + u_3) \vec{a}_3 \] (3.3)

where \((u_1, u_2, u_3)\) is the position of the \( i^{th} \) atom’s basis.

### 3.2 - Calculations of the EFG

In general, the EFG is a difficult quantity to compute, since it must be derived by differentiating the potential \(^{17}\)

\[ V(r_0) = \frac{1}{4\pi\varepsilon_0} \int \frac{\rho(r')}{r} d\tau' \] (3.4)

where the integral is taken over a volume swept out by \( r' \), and the vector \( \vec{r} = \vec{r}' - \vec{r}_0 \). So long as the charge distribution, \( \rho(\vec{r}) \), is unknown, this integral cannot be solved. Since simulations in this project will assume a point charge distribution, however, (3.4) can be simplified to

\[ V(r_0) = \frac{1}{4\pi\varepsilon_0} \sum_k \frac{q_k}{r_k} \] (3.5)

by substituting in the charge distribution of \( k \) point charges

\[ \rho(\vec{r}) = \sum_k q_k \delta^3(\vec{r} - \vec{r}_k) \] (3.6)

\(^{17}\) The basics of E&M are found in chapter 2 of [2].
This equation for $V(r_0)$ is a quantity that is at least possible to differentiate, as can be easily shown:

$$V_{ij}(\vec{r}_0) = \frac{\partial^2}{\partial x_i \partial x_j} \left( \frac{1}{4\pi \varepsilon_0} \sum_k q_k \frac{1}{r_k^2} \right)$$

$$= \frac{1}{4\pi \varepsilon_0} \sum_k q_k \left( \frac{\partial^2}{\partial x_i \partial x_j} \frac{1}{r_k} \right)$$

$$= \frac{1}{4\pi \varepsilon_0} \sum_k V_{ij}^{(k)}$$ \hspace{1cm} (3.7)

where some algebra will yield

$$V_{ij}^{(k)} = q_k \left( \frac{3x_i x_j' - \delta_{ij} r_k^2}{r_k^5} \right)$$ \hspace{1cm} (3.8)

This quantity is the (unscaled) contribution of each charge towards the $(i,j)^{th}$ component of the EFG tensor.

This has two important implications regarding EFG calculations. First, to calculate the net EFG, one need only specify the partial EFG due to each point charge, and sum each contribution. Second, in the case of an infinite number of charges, an approximation to the EFG can be made over a finite number of $N$ charges

$$\tilde{V}_{ij} = \frac{1}{4\pi \varepsilon_0} \sum_N V_{ij}^{(k)}$$ \hspace{1cm} (3.9)

and the error can then be calculated as

$$\epsilon_{ij}^{(N)} = V_{ij} - \tilde{V}_{ij} = \frac{1}{4\pi \varepsilon_0} \sum_M V_{ij}^{(k)} \text{ s.t. } (k \in M) \text{ iff } (k \notin N)$$ \hspace{1cm} (3.10)

which can be made to rapidly decay by exploiting the $r^{-3}$ dependence of (3.8).
3.3 - Simulation Algorithm

As a preface to this section, it should be noted that the code used to carry out the simulations in this project is attached in appendix A. For a variety of reasons (not the least of which being that not everyone understands C++, and most of those who do prefer not to sift through pages of code), a brief description of the algorithms used to generate the results provided in section 4 would likely be useful. It should be noted, however, that only the key routines would be outlined here; so if an understanding of the less essential sub-routines (such as the diagonalization process) is sought, one should reference the code directly.

The main goal of this section is to put (3.7) in “computer friendly” terms, for once there is a well-defined algorithm for adding all charge contributions to the EFG, it should be fairly straight-forward to incorporate cases where only some charges are contributing to the tensor.

3.3.1 - Summation

The key to computing the EFG will lie in finding the most efficient algorithm for reducing the $\varepsilon^{(N)}$ tensor. This, in turn, will rely on how the atoms in the lattice are organized. Thus, the first step to the summation process is declaring how the lattice information is to be stored.

Begin by specifying the primary vectors for the lattice

$$\vec{a} = (\vec{a}_1, \vec{a}_2, \vec{a}_3)$$  \hspace{1cm} (3.11)

Since this matrix can completely describe the lattice, the next step is to specify the basis that is to be attached to each point on the lattice. This is done with the quantities
\[ \vec{Q} = (q_1, q_2, \ldots, q_N) \]  
\[ \vec{D}^u_k = (u_1 + x_k, u_2 + y_k, u_3 + z_k) \]  

where \( q_i \) describes the charge of the \( k \text{th} \) atom in a basis with \( N \) atoms, and \( D_k \) represents the distance from the origin to the \( k \text{th} \) atom in the basis located at the lattice point \( u = (u_1, u_2, u_3) \). With this notation, \( r_k \) (from equation (3.3)) can be expressed as \( a \cdot D_k \).

The final parameter needed to complete these definitions is the point where the EFG is to be calculated (denoted \( r_0 \)). The easiest way to define this parameter is in fractions of the basis vectors, as with atom positions in the unit cell, and place it in the basis corresponding to \( u = (0,0,0) \). With this definition, the net distance between the site of interest and the \( k \text{th} \) atom of a basis at the lattice point \( u \) can be expressed as \( (k)^-r^u \).

\[ (k)^-r^u = \left[ (D^u_k - r_0) \cdot \vec{a} \right] \]  

We are now in a position to calculate the EFG. Recalling (3.7),

\[ V_g = \frac{1}{4\pi \epsilon_0} \sum_u \sum_k \left[ q_k \left( 3 \cdot \left( \frac{(k)^-r^u}{(k)^-r^u} \right) \left( \frac{(k)^-r^u}{(k)^-r^u} \right) - \delta_0 \left( \frac{(k)^-r^u}{(k)^-r^u} \right)^2 \right] \]  

which, despite its messy appearance, is nearly the complete algorithm. The only problem with (3.15) is that the parameters \( u_1, u_2, \) and \( u_3 \) are unbounded (since, theoretically, the lattice could extend out to infinity). To remedy this, it is convenient to work with shells of lattice points. Due to the strong radial dependence of the EFG (\( r^{-3} \)), it seems logical
that the fastest convergence will be obtained by grouping the lattice points in spherical shells\textsuperscript{18}:

\[
S_{ij}^{(n)} = \sum_{k} V_{ij} \text{ s.t. } \|u\|^{2} = u_{1}^{2} + u_{2}^{2} + u_{3}^{2} = n \quad (3.16)^{19}
\]

Using this notation, the approximate EFG can be written as

\[
\tilde{V}_{ij}^{(N)} = \sum_{n=0}^{N} S_{ij}^{(n)} \quad (3.17)
\]

and the error as

\[
\mathcal{E}^{(N)} = \sum_{n=N+1}^{\infty} S_{ij}^{(n)} \quad (3.18)
\]

A discussion regarding the expected convergence of (3.18) is left for section 5.2.1.

**3.3.2 – Perturbations**

There are two perturbations that will become important in this project: vacancies and near surface calculations. The algorithms for these simulations are very similar to the one just outlined in 3.3.1, exploiting the additive property derived in equation (3.7).

Since these perturbations are of fairly central importance in the coming section, however, it seems worth briefly discussing how these calculations are carried out.

\textsuperscript{18} This statement requires that the primary vectors be of similar magnitude. Since, in this project, only a simple cubic lattice was considered, the primary vectors all possessed identical magnitudes; however, when this is not the case, spherical lattice point shells may no longer be the most efficient algorithm.

\textsuperscript{19} There may be some question as to whether or not this includes every lattice point, or if it misses a few. If you find yourself wondering this, consider the argument

\[ (x, y, z) \in \mathbb{Z}^{3} \Rightarrow (x^{2} + y^{2} + z^{2}) \in \mathbb{Z} \]

which basically suggests that if \(x, y, \) and \(z\) are all integers, then \(x^{2}+y^{2}+z^{2}\) is an integer, and thus the point \((x,y,z)\) must exist in some shell.
To obtain the vacancy perturbed EFG, it is best to re-adopt the system used in section 3.2 of sequentially enumerating every atom in the lattice. Suppose we wished the atom located at \( k = k' \) to be vacant, then (3.7) implies the EFG would be

\[
V_{ij}^* = \frac{1}{4\pi\epsilon_0} \sum_{k \neq k'} V_{ij}^{(k)}
\]

or, in other words, the contribution of the vacant atom, \( V_{ij}^{(k')} \), can simply be subtracted from the bulk EFG calculated by (3.17), \( V_{ij}^{(N)} \), to obtain the perturbed EFG, \( V_{ij}^{*(N)} \).

For the near surface calculations, one could – in theory – use this exact same approach, subtracting off all the contributions from atoms that fall outside the surface. This may not be the best approach, however. First, it would be a rather cumbersome calculation, especially if a large number of shells were included in the bulk calculation. Second, it would compound any rounding errors that resulted from representing irrational roots in terms of floating point numbers (an effect that will be further discussed in section 5.2.1, as well).

Instead, it seems much more efficient (and precise) to define the surface-bounded lattice as a volume of lattice points, \( L \), and redefine the spherical shells as

\[
S_{ij}^{(n)} = \sum_{u} V_{ij} \text{ s.t. } \|\vec{u}\|^2 = n \text{ & } (u_1, u_2, u_3) \in L
\]

In the case of a simple cubic lattice, this redefinition ends up being fairly straightforward if the surface normal lies along one of the primary vectors. In this case,

\[
L = \{(u_1, u_2, u_3) \text{ s.t. } (\vec{u} \cdot \vec{a})_i < d\}
\]

where \( i \) is the surface normal axis, and \( d \) is the distance to the surface.
4

RESULTS

4.0 – Overview

In this section, calculations of the EFG inside SrTiO$_3$ based on the relations derived in section 3 are presented. There are three main sections to these calculations: bulk lattice structures, isolated vacancies, and near surface lattice structures. While there are a number of variable parameters specific to each simulation, there are a few parameters basic to SrTiO$_3$ that are listed in table 4.1. These will be treated as fundamental to each simulation, and extra parameters will be specified as they become important.

<table>
<thead>
<tr>
<th>Lattice Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Vectors (Å)</td>
</tr>
<tr>
<td>3.91</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basis Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ion</td>
</tr>
<tr>
<td>Ti</td>
</tr>
<tr>
<td>O</td>
</tr>
<tr>
<td>O</td>
</tr>
<tr>
<td>O</td>
</tr>
<tr>
<td>Sr</td>
</tr>
</tbody>
</table>

Table 4 - 1: Lattice$^{20}$ and Basis parameters for SrTiO3

---

$^{20}$ The lattice constants for SrTiO$_3$ were obtained from [8]. See Lattice_info.dtt (Appendix A) for the parameter definitions actually used in simulation.
4.1 – Bulk EFG Calculations

For bulk crystal calculations, the crystal lattice is assumed to be an ideal crystal, stretching off to infinity in every direction. The main purpose behind carrying out such simulations is to check if the program returns reasonable results – both with regard to theoretical scenarios where the solution can be proven, and with regard to experimental results where the solution has been measured. It will also be useful for observing convergence in various scenarios.

4.1.1 – Ti Substitutional Site

In the SrTiO$_3$ lattice, the Ti atom rests in a site of cubic symmetry (recall figure 2.3 for Ti atom’s position in SrTiO$_3$). That it does can be observed by noting that if the entire lattice is rotated about any axis by $\pi/2$, the lattice seems unchanged from the Ti atom’s perspective. This leads to the conclusion that

$$V_{xx} = V_{yy} = V_{zz} \quad (4.1)$$

Recalling (1.15) ($V$ satisfies Laplace’s equation), (4.1) implies that each element must be identically 0.
This observation leads to an obvious first test of the simulation algorithm. Figure 4.1 shows the dependence of the nine EFG elements on shell radius, taken at the Ti substitutional site.

As can be seen, the three axial components converge instantly; however, the off axis components ($V_{ij}$ for $i \neq j$) seem to converge a little slower, oscillating about the
origin as they converge to zero. This behavior is more clearly displayed in figure 4.2, which further shows that these factors eventually converge to zero as well. The cause of this convergence behavior is further discussed in section 5.2.2; however, it should be noted here that this behavior will persist in all simulations that will follow.

4.1.2 – Interstitial Site

The interstitial site in a SrTiO$_3$ crystal is the site at which it is expected the implanted Li ions will come to rest in the experiments outlined in section 2\textsuperscript{21}. Thus, the estimations of $q_i$ generated here can be compared to the experimentally gathered data to further check the validity of the simulation. First, however, it is useful to examine a value to which we can attach a theoretical value – the asymmetry parameter $\eta_i$. Recall in section 2.2 it was predicted that $\eta_i = 0$, again due to the lattice symmetry about this point.

![Log-Log $\eta$ convergence with increasing simulation radius](image)

Figure 4.3: Demonstrates convergence of $\eta$ in interstitial site to theoretical value of zero (due to axial symmetry of interstitial site)

---

\textsuperscript{21} Note that all future simulations (including those with perturbations) are done from this site.
Figure 4.3 confirms this prediction; however, note that the convergence rate is substantially slower than with the parameters observed in section 4.1.1. The cause of this lies in the subtraction of $V_{xx}$ and $V_{yy}$. This operation exaggerates the slight variance between the two values (that theoretically should not exist) due to rounding errors. Fortunately, the experimental results displayed in section 2.3 predict that $\eta$ should be on the order of $10^{-2}$ to adequately explain the anomalies observed. In this simulation, $\eta$ has converged far below this value in fewer than 10 shell iterations, so this can be considered a minor effect.

Figure 4.4 displays the convergence $v_q$ (which is proportional to $V_{zz}$, the largest principle value of the EFG) for the interstitial site. This figure contains both good and bad news. First, the good news: note that convergence is obtained in only a few shells. This is the $r^{-3}$ dependence coming into play; the contributions of the first few shells greatly outweigh those coming from the outer shells, and the changes in $v_q$ due to outer shell contributions are quickly unnoticeable.

Figure 4.4: Same convergence plot as in figure 4.3, however this time for $v_q$. Note the rapid convergence to theoretical value, indirectly implying a negligible error for even small $N$. 
The bad news involves the proximity of the simulation’s predicted $v_q$ to the experimentally determined value, $v_q^{(E)}$. The converged value of $v_q = 205.8$ kHz is about three times the experimental value $v_q^{(E)} \approx 76.6$ kHz\cite{4}. The point charge model, however, was intended to give some idea what dependences the EFG had on small perturbations, thus this discrepancy is not completely unexpected.

The error is mostly due to the ions in the crystal not really being point charges. In fact, the finite charge distributions are not even necessarily spherical. These realities introduce significant complications which are further discussed in 5.2.3; however, in the mean time it might be possible to liken the simulated value $v_q$ to $v_q^{(E)}$ by considering a simple inter-nuclear interaction model\textsuperscript{22}.

### 4.1.3 – Perturbing the Nearest Neighbours

The goal of this section is to determine how changes in the position of nearby ions affect the EFG. It is possible that, in the presence of a charged Li$^+$ ion, the nearest-neighbours (or even the next-nearest neighbours) of the interstitial site could be slightly distorted from their equilibrium positions in the unperturbed crystal. Such relaxation effects are known at impurities in crystals. If these are drastic enough, it might help explain the discrepancy resulting from figure 4.4.

The general algorithm for this procedure mimics the one used to calculate vacancy effects (equation (3.19)). Specifically, the EFG contributions due to the ions in

---

\textsuperscript{22} This “might” hinges on some kind of physical explanation, of course. See section 5.1.1 for a discussion on this.
question are subtracted from the bulk EFG, and the contributions are then re-added with the ion positions modified to reflect the desired flexing.

![Diagram](image)

**Figure 4.5:** Nearest neighbour distortion effects. As O moves away (negative distance) and Sr moves away (positive distance), the EFG decreases, while moving Ti away (positive distance) causes the EFG to increase.

Figures 4.5 depict $v_q$’s dependence on the nearest O, Sr and Ti atom flexing. In the case of O, the negative x-axis represents repulsion, whereas this effect is represented by the positive x-axis in the case of Sr and Ti. Figure 4.6 displays the net $v_q$ dependence on flexing all neighbours, where once again the positive x-axis represents repulsion (this time of all atoms, including O – this is done with the foresight of expecting Li-O repulsion. This too will be discussed in 5.1.1). If Ti repulsion is ignored, only $\sim \frac{1}{2}$ Å of flexing is required to match up $v_q$ with $v_q^{(E)}$. If Ti is included, the required repulsion is closer to $\sim 1$ Å.
4.2 – Vacancy Perturbations

The first of the perturbations that could conceivably cause a non-zero $\eta$ is proximity to a vacancy. These vacancies could arise from at least two situations: (1) Natural defects in the crystal, and (2) Damage caused by implanting the $^8$Li ions. For these calculations, a perfect crystal simulation in the interstitial site was carried out over 1000 shells, which according to the results in section 4.1.2 should give adequate convergence for all the parameters of interest. Figures 4.7 and 4.8 show the dependence of $v_q$ and $\eta$ (respectively) on vacancies in nearby Sr, Ti and O sites.
Figure 4.7: Dependence of $v_q$ on distance from vacancy. Note that within ~6 Å the effects on $v_q$ are more or less negligible.

Figure 4.8: $\eta$ dependence on distance to vacancy. This time, effects are significant up to 8-10 Å away. Note $\eta = 0$ for Sr vacancies.
4.3 – Near Surface Perturbations

For these simulations, each shell was redefined according to (3.20) (the shell definition in a near-surface lattice volume) before its contribution was added to the EFG. The surface normal was taken to be perpendicular to the site normal that was being considered (see figure 4.9 for the difference between the site normal being parallel and perpendicular to the surface normal), as this would be the only logical way to introduce an asymmetry to the EFG\textsuperscript{23}. Due to the number of iterations involved in recalculting the EFG for each simulation, only 500 shells were used for each simulation.

Figure 4.9: A crystal surface (denoted by dashed lines) is shown next to a site lying in a parallel plane (a) and a perpendicular plane (b).

Figure 4.10: Dependence of \( v_q \) on distance from surface. Note changes are < 1% by 2Å.

\textsuperscript{23} as a test, the surface normal was taken to be parallel to the face normal, and it was confirmed that no asymmetry was introduced this way.
The results of these calculations are shown in figure 4.10 (for the $v_q$ dependence) and figure 4.11 (for the $\eta$ dependence). These results are actually quite astounding, and take a fair amount of pondering to fully appreciate, for they seem to imply that unless the Li ion comes to rest directly on the surface, any effects on the EFG due to surface proximity can be more or less ignored.

![Graph](image)

Figure 4.11: $\eta$ dependence on distance to surface. Once again, surface effects are negligible at distances greater than 2Å.
5 DISCUSSION

5.0 Overview

The goal of this section is to analyze the results of section 4 in order to determine if they (1) are physically reasonable, and (2) sufficiently describe the experimental results. This is followed by a brief discussion regarding sources of error inherent to these computational simulations.

5.1 – Physical Valuation of Simulation Results

5.1.1 – Bulk Calculations

As was mentioned in 4.1.1, the bulk calculations were performed primarily as a check between the simulated results, and those obtained through either theoretical or experimental observation. Agreement between theoretical and simulated values has been shown to be reasonably good (recall figures 4.1 and 4.2), thus it is the contrast between experimental and simulated results that are of primary concern here.

Since the $\eta$ discrepancy is to be accounted for by the perturbations of the later sections, this section will examine only $v_q$ results. If we accept the experimentally determined value $v_q^{(E)} = 76.6$ kHz (which has been confirmed in a variety of experiments, making this assumption a reasonable one$^{24}$), what is sought here is an explanation as to why the simulated value $v_q = 205.8$ kHz is too large. In other words, reasons for

$^{24}$ See [5]
believing the simulated $v_q$ value is larger than one should expect will be presented, as opposed to calculations attempting to equate $v_q$ to $v_q^{(E)}$.

There are a few methods by which one might try to reconcile these two values. Perhaps the most convincing of these involves noting the errors introduced by assuming a point charge lattice distribution. These effects, however, are left for a later section (namely 5.2), for while they present convincing evidence as to why one should be cautious in blindly accepting the point charge model’s predictions, they do little to describe the nature of the error (like, for example, if the predicted value will be larger or smaller than one should expect).

A model that may provide a useful relation in this regard is the one that was presented in section 4.1.3, the flexing of nearest neighbour atoms. Figure 4.6 implies $v_q \rightarrow v_q^{(E)}$ only if all nearest neighbour atoms are slightly repelled (the exact repulsion distance required for complete agreement depends on the size of the perturbation). While the repulsion of the positive Sr and Ti ions seems natural, it appears counterintuitive to suggest the negatively charged O ions would similarly be repelled. The attraction between positive and negative ions, however, only exists at large distances; when ions get too close, their closed electron shell cores overlap, and there are no states for the electrons. This effect is called core-overlap, and the distance at which it occurs is loosely related to the ionic bonding length. In the case of Li and O, this distance is around 2 Angstroms$^{25}$.

Since the interstitial site lies slightly less than 2 Angstroms from the nearest O ions, such a Li-O bond length provides a mild degree of support for O repulsion in this case.

---

$^{25}$ Reasonable values for this quantity seem to range from 1.7-2.1 Angstroms. See [9], [10], and [11].
case. The other errors present in this model render an exact calculation of the repulsion distance a rather futile exercise; however, it is useful to point out such repulsion could exist, and thus at least partially account for the discrepancy in simulated and measured $v_q$.

Besides core repulsion, there are a few other plausible reasons for believing the nearest oxygen to the implanted Li ion is a bit farther than a half lattice constant. It could be that implanting the $^8\text{Li}^+$ ion decreases the effective charge of the oxygen atom. This redistribution of electrons in the crystal lattice might explain a slight repulsion between the Li and O atoms (however it does not rule attraction between the two ions. It is not understood what the exact effect would end up being).

Another possibility involves the delocalization of the $^8\text{Li}$ nuclear charge along the symmetry axis (towards the Sr atoms). This would actually represent the Li nucleus shrinking away from the O ions (as opposed to the O ions being pushed away from the Li). It is not immediately clear, however, that moving the Li charge towards the Sr ions doesn’t increase $v_q$ more than moving charge away from the O’s decreases it. However, these effects are likely minor, if even existent, as the energy required to distort a nucleus is enormous.

5.1.2 – Vacancy Calculations

There is not much to discuss regarding the physical viability of the vacancy calculations. It is difficult to say what the likelihood of a crystal imperfection is, for although crystalline purity and perfection can be characterized, implanting Li ions could knock out ions and cause damage to the crystal structure. Thus, ascertaining the probability that an oxygen atom (for example) is missing near the lithium implantation
site involves a fair amount of guesswork. Given the results of section 4.2, however, it is useful to consider whether they even could explain the experimental results outlined in section 2.

Naturally, if a neighbouring atom is removed from the structure, this has a fairly drastic effect on the EFG (in the 30 to 40% range, apparently). The more interesting question is how far a vacancy can be and still have an effect on the order of 1%. It turns out, this distance is about eight Angstroms – or roughly two lattice constants – for both $\eta$ and $v_q$.

As mentioned, it is difficult to say whether an 8Å influence radius is a reasonable magnitude for explaining the results being addressed here. It is interesting to note (with some foresight) that this radius is about four times larger than that seen in near surface calculations. This, however, is believed to be a characteristic of the latter situation, and so a discussion of these effects is left for the next section.

### 5.1.3 – Surface Calculations

By far the most interesting result of section 4 is the nearly negligible effects a nearby perpendicular\textsuperscript{26} surface has on the EFG. One might expect, rather naturally, that removing nearly half the lattice points in a simulation would have a drastic effect on the EFG, especially given the 8Å influence radius observed in vacancy perturbations. As the simulations presented in section 4.3 imply, however, this is certainly not the case. This

\textsuperscript{26} Where, again, the perpendicular vectors being referred to here are the normal of the surface and the normal of the plane that the interstitial site lies in. Refer back to figure 4.12 for more details.
discrepancy warrants a short argument in favor of why intuition is actually ill served in this case.

The $\eta$ dependence (or lack thereof) on distance to a perpendicular surface is most surprising when viewed alongside this same dependence for nearby vacancies. As figure 4.8 depicted for this latter case, even slight variances in the EFG caused by distant point charge vacancies can have noticeable effects on $\eta$. It turns out, however, that in subtracting an entire plane of charges, a symmetry is somehow re-introduced, which results in a much faster convergence of $\eta \to 0$ then was observed in the single vacancy case.

<table>
<thead>
<tr>
<th>EFG 1/2 lattice constant above X-Surface (TiO$_2$ surface)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5.15852E+20  8.26031E+14  1.19702E+15</td>
</tr>
<tr>
<td>8.26031E+14  5.56904E+20  -1.13849E+14</td>
</tr>
<tr>
<td>1.19702E+15  -1.13849E+14  -4.10525E+19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EFG 1 lattice constant above X-Surface (SrO surface)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7.00395E+16  -2.7484E+15  -2.38078E+15</td>
</tr>
<tr>
<td>-2.7484E+15  -1.35676E+19  7.90811E+14</td>
</tr>
<tr>
<td>-2.38078E+15  7.90811E+14  1.36376E+19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EFG 2 lattice constant above X-Surface (SrO surface)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-9.42152E+12  -2.22869E+15  -2.09919E+15</td>
</tr>
<tr>
<td>-2.22869E+15  -2.52972E+16  5.9187E+14</td>
</tr>
<tr>
<td>-2.09919E+15  5.9187E+14  2.53067E+16</td>
</tr>
</tbody>
</table>

Table 5.1: Calculations just above the SrTiO$_3$ surface. Notice the EFG over a TiO$_2$ surface represents the compliment of an EFG under a SrO surface (and vice-versa).

Probably the best way of coming to an understanding regarding this re-introduced symmetry is by considering the EFG just above the surface, instead of just below. The
reason this exercise is useful lies in the additive property of the EFG. This relation implies

\[ V_{ij}^{\text{bulk}} = V_{ij}^{\text{below surface}} + V_{ij}^{\text{above surface}} \]  

(5.1)

Table 5.1 includes some EFG calculations at increasing distances above the surface to exemplify this re-introduced symmetry.

One final remark on near surface calculations regards possible surface compositions. There is a significant amount of evidence that suggests a SrO surface composition cannot be formed\(^ {27}\). This implies that at least one layer of TiO\(_2\) must always exist above a perpendicular interstitial site (since all such interstitial sites lie in the SrO plane). Therefore, the first point in figure 4.11 (and 4.10, for that matter) is not a physically possible configuration. The largest contribution to \( \eta \) due to a near surface, then, is on the order of \( 10^{-2} \), which occurs only when the \(^8\)Li comes to rest directly under the surface. If the probe ion penetrates any deeper, it would seem the surface effects would be negligible.

### 5.2 – Sources of Error

The final object of this analysis will be to cover some possible sources of error that are present in various elements of the simulation. These will be grouped into three sub-headings: computational, algorithmic, and theoretical. The first group consists of errors that arise due to approximations made in the computation process. The second are those that arise from the algorithm used to compute the EFG. The third and final subject of error involves side effects due to the point charge assumption.

\(^{27}\) See [12] and [13] regarding physical SrTiO\(_3\) surfaces.
5.2.1 – Computational Errors

There are two main contributions to the error that arise from computational limitations. The first of these involve solving for the square and cube root of various values. Due to the irrationality of some roots, exact floating point representation becomes impossible, giving rise to a small degree of error. Fortunately, this error is bound by the precision of the floating point numbers being used to carry the root values, and so can be easily reduced by simply increasing the precision of the quantities. For these simulations, eight-digit precision was used for all rooted quantities, and thus these errors only became visible for values converging to zero (even then, however, the errors were seen to be minor).

The second source of computational error arises from the number of shells included in the simulation. Equation (3.18) gives an equation for this error, however it is of little practical use. Instead, one can predict that this error approaches zero by noting that, for \( r >> a \), atoms \( k_1 \) and \( k_2 \) in a given basis approach \( r_{k_1} \approx r_{k_2} \).\(^{28}\) This implies that for a basis located far from the point of interest,

\[
V_{ij}^{\text{Basis}} = \frac{1}{4\pi\varepsilon_0} \left( \frac{3x_i x_j - \delta_{ij}}{r^5} \right) \sum_{k=1}^{n} q_k
\]

(5.2)

Since a basis has net neutral charge, this equals 0. Thus, for every shell containing lattice points a distance \( r >> a \), \( S_{ij}^{(n)} = 0 \). If the simulation radius is chosen to be large enough so that all terms in the error satisfy this condition, we find

\[
\varepsilon^{(N)} = \left. \sum_{n=N+1}^{\infty} S_{ij}^{(n)} \right|_{r\gg a} = 0
\]

(5.3)

\(^{28}\) This, of course, is not meant as a proof. It is merely mentioned to explain why one should expect the error in this simulation to approach 0.
This behaviour is exemplified by the rapid convergence of \( v_q \) in figure 4.5.

### 5.2.2 – Algorithmic Errors

The algorithm of choosing shells based on lattice points, while very efficient due to the emphasis on radial contributions, fails to exploit the symmetry of many situations that would allow for faster convergence of the error. This effect is best described by means of a simple example.

Consider the calculation of an off axis elements \((i \neq j)\) of the EFG computed at the Ti substitutional site. The contribution from any atom located at \( \mathbf{r} = (x_1, x_2, x_3) \) is based on the relation

\[
V_{ij}^k \propto \frac{x_i x_j}{r^5}
\]

Since the Ti site exists in cubically symmetric site, there will exist an identical charge at \( \mathbf{r} = (\pm x_1, \pm x_2, \pm x_3) \), and thus the summed contribution \( V_{ij} \) from these 8 charges becomes

\[
V_{ij}^{\text{Sum}} \propto 4 \frac{x_i x_j}{r^5} - 4 \frac{x_i x_j}{r^5} = 0
\]

Recalling figure 4.3, however, one notices the oscillating convergence of \( V_{ij} \) with expanding shells. This effect arises because not all 8 symmetrically placed charges are included in the same shell, and thus their contributions are not completely cancelled until the next shell is included. Of course, at this point, farther out charges are included, and their adjoining canceling charges are not included until the shell after that. This continues as the contribution from the unmatched atoms approaches zero, thus resulting in the oscillating convergence observed in figure 4.2.
5.2.3 – Theoretical Errors

There are two main effects that I will consider in this section, however it should be noted there are likely many more that are not mentioned here. The effects discussed here arise from the point charge assumption, and the results of ignoring the finite radius of the atomic charge distributions.

The validity of a point charge approximation seems deceptively good when one notes the electric field due to a uniform sphere of charge is identical to that produced by a point charge (Gauss’ Law). However, while a point charge may be ideal for approximating spheres, it falls short on a few counts when it comes to approximating ions.

First, ions are not necessarily (nor even typically) spheres of uniform charge. Figure 5.1, for example, shows the electron cloud in a typical O$^{2-}$ ion. For such a distribution at close radius, the point charge approximation could be quite poor.

Another effect to consider with a finite charge distribution is interactions with the EFG. A non-homogenous EFG (i.e. one having non-zero derivatives over a finite site volume) could deform the electronic or nuclear charge distribution, giving rise to polarization or higher-order multipoles$^{29}$. These could, theoretically, have a significant impact on calculations of the EFG at the interstitial sites where the Li ions were coming to rest. It is unclear how severe these effects are in this

![Figure 5.1: EFG at r from various distributions. The results of a point charge (left) and spherical charge (middle) would be identical. A more realistic ion (right), however, would have a slightly different EFG at r](image)

---

$^{29}$ These polarization and multipole errors are introduced in [14].
particular case; however, it is rare that they play a significant role in EFG calculations.

Allowing for a finite radius in the Li charge distribution could give rise to shielding and anti-shielding\textsuperscript{30} factors. These effects arise from the electronic charge distribution surrounding the nucleus distorting in response to a non-homogenous EFG (as opposed to the nuclear charge distribution being warped). The net effect of this is to either shield or enhance the quadrupole moment of an ion (the two ways this is accomplished are outlined in figure 5.2). In the case of Li\textsuperscript{+}, however, the effect is considered a minor one, and so ignoring it in these simulations should not have a drastic effect on the results.

The results of section 4 yielded two important observations. The first, regarding bulk simulations, showed the predicted value of $n_q = 205.8$ kHz was of the same order of magnitude as, and only a factor of three larger than, the experimental value of $n_q^{(E)} = 76.6$ kHz. Ignoring errors involved with the point charge simplification, these two values can be reconciled by distorting the distance to the nearest neighbour atoms in response to implanting a $^8\text{Li}^+$ ion. Possible reasons this could occur involve core-overlap, reduced charge on the O ions, or a delocalized nuclear charge in the implanted Li ion.

Second, it was seen that perturbations invoked to create an asymmetry in the EFG needed to be in fairly close proximity to the site of interest to reproduce experimental results. In the case of a nearby vacancy, the site of interest needed to be within 8 Å to achieve the desired effect. For a nearby surface, the result was even more drastic, requiring a distance no greater than 2 Å for the correct asymmetry factor. In the latter case, factoring in the constraint of realistic surfaces (namely, the non-existence of a SrO surface), the implication is implanted Li ions must come to rest at the nearest possible site to the surface in order for a significant asymmetry to exist.
APPENDIX A

SIMULATION CODE

- EFG_PC_Calc.cpp .................................................................60
- options.h .................................................................63
- classes.h .................................................................64
- supplement_functions.h .............................................68
- initialize_functions.h .............................................74
- menu_functions.h .....................................................78
- sim_functions.h .....................................................80
- menu_options.h .....................................................88
- Lattice_info.dtt .....................................................104
#include "options.h"
#include "classes.h"
#include "supplement_functions.h"
#include "initialize_functions.h"
#include "menu_functions.h"
#include "sim_functions.h"
#include "menu_options.h"

int main()
{
    Basis basis;
    Lattice lattice;
    vec3 primitive[3];
    char choice,
    read;
    bool running = true;

    // STARTUP DISPLAY
    system("clear");
    cout << endl << endl << endl;
    cout << "               ELECTRIC FIELD GRADIENT CALCULATOR" << endl;
    cout << "                                             v1.0" << endl
    << endl;
    cout << "                    Written by Jordan Schultz      " << endl;
    cout << "                     Last updated: March 24, 2005" << endl;
    cout << "   Reading lattice data from file... " << endl << endl;

    // GET INFORMATION FROM LATTICE_INFO.INI
    string input;
    input = Read_file( );

    // CATCH ERROR
    if (input == "ERROR") return -1;

    // ELSE CONTINUE WITH PROGRAM
    read_primitive_vectors(input, primitive);
    basis = read_basis(input);

    // SET UP LATTICE
    lattice.set_a(primitive);
    lattice.set_basis(basis);

    // DISPLAY LATTICE
    cout << "   Done" << endl;
    cout << "   The following data was obtained from file:" << endl << endl;
    lattice.display();
cout << " Lattice data can be accessed and changed in 'Lattice_info.dtt'" << endl;
cout << endl << " Continue? ('n' to exit) ";
cin >> choice;
if ((choice == 'n') || (choice == 'N')){
  return(-1);
}
system("clear");

// DISPLAY PROGRAM OPTIONS

mat3 EFG;

while(running)
{
  cout << endl << endl << endl << endl;
  cout << "            Current EFG:" << endl;
  cout << "                         " << EFG[0] << endl;
  cout << "                         " << EFG[1] << endl;
  cout << "                         " << EFG[2] << endl << endl;
  cout << "            Simulations:" << endl << endl;
  cout << "                (a) Create a crystal" << endl;
  cout << "                (b) Increasing distance from surface" << endl;
  cout << " Functions:" << endl << endl;
  cout << "            (c) Set EFG" << endl;
  cout << "                (d) Rotate EFG about axis" << endl;
  cout << "                (e) Diagonalize EFG matrix" << endl;
  cout << "                (f) Add vacancy" << endl;
  cout << "                (g) Round down EFG" << endl;
  cout << "                (h) Summarize EFG" << endl;
  cout << "            Other Options:" << endl << endl;
  cout << "                (i) Flex Nearest Neighbours (SrTiO3 only)" << endl;
  cout << "                (x) Exit program" << endl;
  cin >> choice;
  cout << endl << endl;
  switch( choice )
  {
    case 'a': EFG = create_crystal(lattice); break;
    case 'b': step_surface(lattice); break;
    case 'c': EFG = set_EFG(); break;
    case 'd': EFG = rotate(EFG); break;
    case 'e': EFG = diagonalize(EFG); // NOTE - DIAGONALIZE FUNCTION
      break; // FOUND IN SUPPLEMENT_FUNCTIONS.H
    case 'f': EFG = add_vac(EFG, lattice); break;
    case 'g': EFG = scale(EFG); break;
  }
}
case 'h': system("clear");
    results(EFG);
    cin >> read;
    break;

case 'i': EFG = flex(EFG, lattice);
    break;

case 'x': running = false;
    break;
}

} system("clear");
}

return 0;
}
```c
#define PI 3.1415926535897932384626433832795028841971693993751058209749445923078164062862
#define EPS 8.8542E-12
#define ELEM 1.60217646E-19
#define SCALE ELEM/4/PI/EPS
#define H 6.6260755E-34
#define QUAD 33

#include "premade/algebra3_m.h"
#include "fstream.h"
#include "iostream.h"
#include "stdio.h"
#include <sstream>
#include <cstdlib>
```
# CLASSES.H

class Basis
{
private:
    float *charge;       // Charge of atom in basis
    vec3 *position;     // Position of atom in basis in terms of
                        // 0 < x, y, or z < 1
    vec3 *inter;        // List of possible interstitial sites
    int atom_ct;       // Number of atoms in basis
    int inter_ct;      // Number of interstitial sites
public:
    // INITIALIZATION
    Basis ( float _charge[],
            vec3 _position[],
            vec3 _inter[],
            int _atom_ct,
            int _inter_ct )
    {
        charge = _charge;
        position = _position;
        inter = _inter;
        atom_ct = _atom_ct;
        inter_ct = _inter_ct;
    }
    // SET VALUES
    void set_charge( float* _charge )
    {
        charge = _charge;
    }
    void set_position( vec3* _position )
    {
        position = _position;
    }
    void set_inter( vec3* _inter )
    {
        inter = _inter;
    }
    void set_atom_ct( int _atom_ct )
    {
        atom_ct = _atom_ct;
    }
    void set_inter_ct( int _inter_ct )
    {
        inter_ct = _inter_ct;
    }
    // GET VALUES
    float* get_charge()
return charge;
}

tvec3* get_position()
{
    return position;
}

tvec3* get_inter()
{
    return inter;
}

int get_atom_ct()
{
    return atom_ct;
}

int get_inter_ct()
{
    return inter_ct;
}

// DISPLAY INFO
void display()
{
    int i;
    cout << "Basis Information" << endl;
    cout << "Atom Charge Position" << endl;
    for( i=0; i<atom_ct; i++ )
    {
        cout << " " << i+1;
        cout.width(15);
        cout << charge[i];
        cout.width(13);
        cout << position[i] << endl;
    }
    cout << "Interstitial Sites Position" << endl;
    for( i=0; i<inter_ct; i++ )
    {
        cout << " " << i+1;
        cout.width(22);
        cout << inter[i] << endl;
    }
}

void display_int_sites()
{
    int i;
    cout << "Site Position" << endl;
    for( i=0; i<inter_ct; i++ )
    {
        cout << " " << i+1;
        cout.width(11);
        cout << inter[i] << endl;
    }
}

void display_sub_sites()
{
    int i;
    cout << "Site Position" << endl;
}
for( i=0; i<atom_ct; i++ )
{
    cout << " " << i+inter_ct+1;
    cout.width(11);
    cout << position[i] << endl;
}
}

class Lattice
{
    private:
        Basis basis;      //basis for lattice
        vec3 a[3];        //primitive vectors in lattice

    public:
        // INITIALIZATION
        Lattice ( Basis _basis = zero_basis,
                  vec3 _a[3] = zero_vectors )
        {
            basis   = _basis;
            a[0]    = _a[0];
            a[1]    = _a[1];
            a[2]    = _a[2];
        }

        // SET VALUES
        void set_basis( Basis _basis )
        {
            basis   = _basis;
        }

        void set_a( vec3 _a[3] )
        {
            a[0]    = _a[0];
            a[1]    = _a[1];
            a[2]    = _a[2];
        }

        // GET VALUES
        Basis get_basis()
        { return basis; }

        vec3* get_a()
        { return a; }

        // DISPLAY INFO
        void display()
        {
            cout << "        Lattice Information" << endl;
            cout << squeeze(_basis,12, ' ') << " Lattice Information" << endl;
        }
}
cout << "Primary Vectors" << endl;
cout << "   " << a[0] << endl;
cout << "   " << a[1] << endl;
cout << "   " << a[2] << endl;
basis.display();
cout << endl;
SUPPLEMENT_FUNCTIONS.H

// FUNCTION FOR CONVERTING INTEGERS TO STRINGS (COULDN'T FIND PREPROGRAMMED)
//

char *IntToString(int Number){
    const int MAXHOLD = 10;
    int Hundred = 1, Array = 0, Size = 0;
    bool Finished = false;
    char *String = new char[MAXHOLD];

    while(Finished == false){
        if(Number < 10){
            Finished = true;
            Size = 1;
            break;
        }
        Hundred *= 10;
        if(Number - Hundred >= 0 && Number - 10 * Hundred < 0){
            Finished = true;
        }
        Size++;
    }
    if(Size >= MAXHOLD){
        return 0;
    }

    while(Array <= Size){
        if(Hundred == 0){
            String[Array] = 0;
        }
        else{
            String[Array] = Number / Hundred + 48;
            Number -= Number / Hundred * Hundred;
            Hundred /= 10;
        }
        ++Array;
    }
    String[Array] = 0;
    return String;
}

// ABSOLUTE VALUE FUNCTION - AGAIN COULDN'T FIND PRE-PROGRAMMED

double abs( double num )
{
    return sqrt(num*num);
}

// DISPLAY FUNCTION: INFORM USER CURRENT OPTION IS NOT COMPLETED

void incomplete()
{ char wait;

cout << "This option has not been programmed" << endl;
cout << "Enter any character to return to menu ";
cin >> wait;
}

// DISPLAY FUNCTION: DETERMINE SIMULATION LOCATION
vec3 implant_location (Basis _basis)
{
  int i;
  double vx, vy, vz;
  vec3 pos;
  bool flag = true;

  while (flag)
  {
    cout << "*  Available Interstitial sites: " << endl;
    _basis.display_int_sites();
    cout << "*  Available Substitutional sites: " << endl;
    _basis.display_sub_sites();
    cout << "*  Site choice (Enter 0 to assign other position): ";
    cin >> i;
  }

  // SIMULATE AT CUSTOM SITE
  if( i == 0 )
  {
    cout << "*\n";
    cout << "* Custom position must be contained in [0,1]\n";
    cout << "*    Input X Co-ordinate: ";
    cin >> vx;
    while ( (vx < 0) || (1 < vx) )
    {
      cout << "*      Try Again: ";
      cin >> vx;
    }
    cout << "*    Input Y Co-ordinate: ";
    cin >> vy;
    while ( (vy < 0) || (1 < vy) )
    {
      cout << "*      Try Again: ";
      cin >> vy;
    }
    cout << "*    Input Z Co-ordinate: ";
    cin >> vz;
    while ( (vz < 0) || (1 < vz) )
    {
      cout << "*      Try Again: ";
      cin >> vz;
    }
    pos.set(vx,vy,vz);
    flag = false;
  }

  // SIMULATE AT PRE-PROGRAMMED INTERSTITIAL SITES
  if( (i > _basis.get_inter_ct()) &&
      (i <= (_basis.get_inter_ct() + _basis.get_atom_ct()) ) )
  {
pos = _basis.get_position()[i-_basis.get_inter_ct()-1];
flag = false;
}

// SIMULATE AT SUBSTITUTIONAL SITES
if( (i <= _basis.get_inter_ct()) &&
    (i > 0) )
{
pos = _basis.get_inter()[i-1];
flag = false;
}
}
return(pos);
}

// FIND THE 3 ROOTS TO A CUBIC EQUATION
// http://www.1728.com/cubic2.htm

// Input coefficients to the equation
// 3  2
// Ax + Bx + Cx + D = 0

// with A = coef[0], D= coef[3].

vec3 cube_root( double coef[4] )
{
dooble   comp[10],
    x1=0,
    x2=0,
    x3=0;

//F
comp[0] = ( (3*coef[2]/coef[0]) -
    (coef[1]*coef[1]/coef[0]/coef[0]))
    / 3;

//G
comp[1] = (2*coef[1]*coef[1]/coef[0]/coef[0]/coef[0]) -
    (9*coef[1]*coef[2]/coef[0]/coef[0]) +
    (27*coef[3]/coef[0])
    / 27;

//H
comp[2] = (comp[1]*comp[1]) / 4 +
    (comp[0]*comp[0]*comp[0]) / 27;

if( (comp[2] < 0) ||
    (comp[2] == 0) &&
    (comp[0] != 0) ||
    (comp[1] != 0))
{
//I
//J
//K
//L
    comp[6] = -1*comp[4];
}
//M

//N
comp[8] = sqrt(3)*sin( comp[5] / 3 );

//P
comp[9] = -1*coef[1]/3/coef[0];
x1 = 2*comp[4]*comp[7] + comp[9];
x2 = comp[6]*(comp[7]+comp[8]) + comp[9];
x3 = comp[6]*(comp[7]-comp[8]) + comp[9];

}

if( (comp[0] == 0) && (comp[1] == 0) && (comp[2] == 0) )
{
x1 = -1*cbrt(coef[3]/coef[0]);
x2 = x1;
x3 = x1;
}

if(comp[2] > 0)
{
cout << "ncube_root function: Only 1 real root\n";
cout << "Imaginary parts: +/- ";

//R

//S

//T

//U
comp[6] = cbrt(comp[5]);
x2 = -1*(comp[4]+comp[6])/2 -coef[1]/3/coef[0];
x3 = (comp[4]-comp[6])*sqrt(3)/2;
cout << x3 << endl << endl;
x3 = x2;
}

vec3 solution(x1,x2,x3);
return(solution);
}

mat3 diagonalize( mat3 EFG )
{
double  coef[4];
vec3  d_elem;
mat3 _EFG;

coef[0] = 1;
coef[1] = -1*(EFG[0][0] + EFG[1][1] + EFG[2][2]);
coef[2] = EFG[0][0]*EFG[1][1] + EFG[0][0]*EFG[2][2] +
        EFG[1][1]*EFG[2][2]
    - EFG[2][1]*EFG[1][2] - EFG[0][1]*EFG[1][0];
coef[3] = EFG[0][0]*EFG[2][1]*EFG[1][2] +
        EFG[1][1]*EFG[2][0]*EFG[0][2] + EFG[2][2]*EFG[0][1]*EFG[1][0];

//R

//S

//T

//U
comp[6] = cbrt(comp[5]);
x2 = -1*(comp[4]+comp[6])/2 -coef[1]/3/coef[0];
x3 = (comp[4]-comp[6])*sqrt(3)/2;
cout << x3 << endl << endl;
x3 = x2;
}

vec3 solution(x1,x2,x3);
return(solution);
}
d_elem = cube_root(coef);

if( abs(d_elem[1]) < abs(d_elem[0]) )
    d_elem.set(d_elem[1],d_elem[0],d_elem[2]);
if( abs(d_elem[2]) < abs(d_elem[1]) )
    d_elem.set(d_elem[0],d_elem[2],d_elem[1]);
if( abs(d_elem[1]) < abs(d_elem[0]) )
    d_elem.set(d_elem[1],d_elem[0],d_elem[2]);

_EFG.set(d_elem[0], 0, 0,
    0, d_elem[1], 0,
    0, 0, d_elem[2]);

return (_EFG);
}

double eta(mat3 EFG)
{
    double _eta;
    mat3 _EFG;

    _EFG = EFG;

    if( (EFG[0][1] != 0) ||
        (EFG[0][2] != 0) ||
        (EFG[1][0] != 0) ||
        (EFG[1][2] != 0) ||
        (EFG[2][0] != 0) ||
        (EFG[2][1] != 0) )
    {
        _EFG = diagonalize(_EFG);
    }

    _eta = abs(( _EFG[1][1] - _EFG[0][0] ) / _EFG[2][2] );

    return (_eta);
using std::string;
std::ostringstream stream;

string Read_file() 
{ 
char* buffer;

ifstream readfile;
readfile.open("Lattice_info.dtt");
if (!readfile)
{ 
    cout << "Error opening 'Lattice_info.dtt'" << endl;
    cout << "Terminating Program" << endl;
    return("ERROR");
}

// GET LENGTH *START* //
int file_length=0;
readfile.seekg (0, ios::end);
file_length = readfile.tellg();
// GET LENGTH *END* //

// return to start
readfile.seekg (0, ios::beg);

// read entire file into buffer string
buffer = new char [file_length];
readfile.read ( buffer, file_length );

// close file
readfile.close();

return(buffer);
}

void read_primitive_vectors( string lattice_data, vec3 primitive_vectors[3] ) 
{ 
    string vector_flag ,
    vector_str ,
    cast ;
    float v[3] ;

    for(int i=0; i<3; i++)
    {
        // Find primitive vector
    }
cast = IntToString(i+1);
vector_flag = "PV" + cast + "<";
int pos1 = lattice_data.find ( vector_flag, 0);
int pos2 = lattice_data.find ( ">", pos1);

// Read vector
vector_str = lattice_data.substr(pos1 + 4, pos2 - (pos1 + 4) );
pos1 = vector_str.find(":", 0);
/*0 = atof((vector_str.substr(0, pos1)).c_str());
pos2 = vector_str.find(":", pos1 + 1);
*/
v[0] = atof((vector_str.substr(0, pos1)).c_str());
pos2 = vector_str.find(":", pos1 + 1);
v[1] = atof((vector_str.substr(pos1 + 1, pos2 - (pos1 + 1))).c_str());
/*2 = atof((vector_str.substr(pos2 + 1, 
*/
string::size_type(vector_str.length()) - ( pos2+1 ))).c_str());
primitive_vectors[i].set(v[0],v[1],v[2]);
}
}

Basis read_basis( string lattice_data )
{
Basis _basis  
string flag ,
buffer ,
cast ;
int atoms ,
inter ;
float* charges  ;
float v[3] ;
vec3 *pos ,
*int_p ;

// Find and set atom count
flag = "AT";
string::size_type pos1 = lattice_data.find (flag, 0);
string::size_type pos2 = lattice_data.find (">", pos1);
atoms = atoi((lattice_data.substr(pos1 + 3, pos2 - (pos1 + 3))).c_str());

// Find and collect atom data
charges = new float [atoms];
pos = new vec3 [atoms];
for( int i = 0; i < atoms; i++ )
{
// Get charge

cast = IntToString(i+1);
flag = "CH" + cast + "<";
pos1 = lattice_data.find (flag, 0);
pos2 = lattice_data.find (">", pos1);
charges[i] = atof((lattice_data.substr(pos1 + 4, pos2 - (pos1 + 4))).c_str());

// Get position

cast = IntToString(i+1);
flag = "PS" + cast + "<";
pos1 = lattice_data.find (flag, 0);
pos2 = lattice_data.find (">", pos1);
buffer = lattice_data.substr(pos1 + 4, pos2 - (pos1 + 4) );
pos1 = buffer.find(":", 0);
v[0] = atof((buffer.substr(0, pos1)).c_str());
pos2 = buffer.find(":", pos1 + 1);
v[1] = atof((buffer.substr(pos1+1, pos2-(pos1+1))).c_str());
v[2] = atof((buffer.substr(pos2+1,
string::size_type(buffer.length()) - ( pos2+1 ))).c_str());
pos[i].set(v[0],v[1],v[2]);
}

// Find and set interstitial count
flag = "IT";
pos1 = lattice_data.find (flag, 0);
pos2 = lattice_data.find (">", pos1);
inter = atoi((lattice_data.substr(pos1 + 3, pos2 - (pos1 + 3))).c_str());

// Find and collect interstitial data
int_p   = new vec3 [inter];
for( int i = 0; i < inter; i++ )
{
   // Get position
   cast = IntToString(i+1);
   flag = "IP" + cast + "<";
pos1 = lattice_data.find (flag, 0);
pos2 = lattice_data.find (">", pos1);
   buffer = lattice_data.substr(pos1 + 4, pos2 - (pos1 + 4 ) );
pos1 = buffer.find(":", 0);
v[0] = atof((buffer.substr(0, pos1)).c_str());
pos2 = buffer.find(":", pos1 + 1);
v[1] = atof((buffer.substr(pos1+1, pos2-(pos1+1))).c_str());
v[2] = atof((buffer.substr(pos2+1,
string::size_type(buffer.length()) - ( pos2+1 ))).c_str());
   int_p[i].set(v[0],v[1],v[2]);
}

// Set basis variables
_basis.set_atom_ct(atoms);
_basis.set_charge(charges);
_basis.set_position(pos);
_basis.set_inter_ct(inter);
_basis.set_inter(int_p);
return(_basis);
/// DIAGONALIZE FUNCTION
/// -INPUT EFG FROM SIMULATION
/// -OUTPUT DIAGONALIZED MATRIX ALONG WITH ROTATIONS
/// USED TO OBTAIN IT

mat3 transform_EFG(mat3 EFG, char axis, double angle)
{
    vec3 row[3];
    double c, s;

    // TRIG CALCULATION
    c = cos(angle * PI / 180);
    s = sin(angle * PI / 180);

    if( angle == 45 ) {
        c=sqrt(2);   // EXACT VALUES FOR COMMON ANGLES
        s=sqrt(2); }
    if( angle == 90 ) {
        c=0;
        s=1; }
    if( angle == 135 ) {
        c=-1*sqrt(2);
        s=sqrt(2); }
    if( angle == 180 ) {
        c=-1;
        s=0; }
    if( angle == -45 ) {
        c=sqrt(2);
        s=-1*sqrt(2); }
    if( angle == -90) {
        c=0;
        s=-1; }
    if( angle == -135 ) {
        c=-1*sqrt(2);
        s=-1*sqrt(2); }
    if( angle == -180) {
        c=-1;
        s=0; }

    switch(axis)
    {
        case 'x' : row[0].set ( 1 * EFG[0][0],
                               c * EFG[0][1] - s * EFG[0][2],
                               s * EFG[0][1] + c * EFG[0][2]);
                    row[1].set ( 1 * EFG[1][0],
                               c * EFG[1][1] - s * EFG[1][2],
                               s * EFG[1][1] + c * EFG[1][2]);
                    row[2].set ( 1 * EFG[2][0],
                               c * EFG[2][1] - s * EFG[2][2],
                               s * EFG[2][1] + c * EFG[2][2]);
    }
    s * EFG[2][1] + c * EFG[2][2]);
    break;
    case 'y': row[0].set ( c * EFG[0][0] - s * EFG[0][2],
    1 * EFG[0][1],
    s * EFG[0][0] + c * EFG[0][2]);
    row[1].set ( c * EFG[1][0] - s * EFG[1][2],
    1 * EFG[1][1],
    s * EFG[1][0] + c * EFG[1][2]);
    row[2].set ( c * EFG[2][0] - s * EFG[2][2],
    1 * EFG[2][1],
    s * EFG[2][0] + c * EFG[2][2]);
    break;
    case 'z': row[0].set ( c * EFG[0][0] - s * EFG[0][1],
    s * EFG[0][0] + c * EFG[0][1],
    1 * EFG[0][2]);
    row[1].set ( c * EFG[1][0] - s * EFG[1][1],
    s * EFG[1][0] + c * EFG[1][1],
    1 * EFG[1][2]);
    row[2].set ( c * EFG[2][0] - s * EFG[2][1],
    s * EFG[2][0] + c * EFG[2][1],
    1 * EFG[2][2]);
    break;
    }
    EFG.set (row[0][0], row[0][1], row[0][2],
    row[1][0], row[1][1], row[1][2],
    row[2][0], row[2][1], row[2][2]);
    return(EFG);
    }
mat3 calc_efg (  
    int x,  // U1  
    int y,  // U2  
    int z,  // U3  
    vec3 implant_v,  
    Lattice lattice,  
    double surface[2],  
    double damaged,  
    bool diag  
)
{
    int impur  ;  
    double dist2  ,  
    contrib  ;  
    mat3 EFG  ,  
    EFG_Add  ;  
    vec3 lattice_v  ,  
    basis_v  ,  
    atom_v  ,  
    implant  ,  
    r  ;

    // OPEN DIAGNOSTIC FILE, IF REQUESTED
    ofstream track;
    track.open("diag.rtf", ios::app);
    if(!track)
    {
        cout << "Error writing lattice point ";
        cout << "( " << x << " , " << y << " , " << z << " ) ";
        cout << "to diagnostic file." << endl;
    }

    lattice_v =  x*lattice.get_a()[0] +
                y*lattice.get_a()[1] +
                z*lattice.get_a()[2] ;

    implant =  implant_v[0]*lattice.get_a()[0] +
                implant_v[1]*lattice.get_a()[1] +
                implant_v[2]*lattice.get_a()[2] ;

    if (diag)
    {
        track << "Basis: " << x << "," << y << "," << z << endl;
        track << "Basis position: " << lattice_v << endl;
    }

    for( int i = 0 ; i < lattice.get_basis().get_atom_ct() ; i++ )
    {
        basis_v =
        lattice.get_basis().get_position()[i][0]*lattice.get_a()[0] +
        lattice.get_basis().get_position()[i][1]*lattice.get_a()[1] +
        lattice.get_basis().get_position()[i][2]*lattice.get_a()[2] ;

        ...
lattice.get_basis().get_position()[i][1]*lattice.get_a()[1] +
lattice.get_basis().get_position()[i][2]*lattice.get_a()[2] ;

atom_v = basis_v + lattice_v;
if (diag)
{
  track << " Atom " << i+1 << endl;
  track << " Vector from Basis: " << basis_v << endl;
  track << " Vector from origin: " << atom_v << endl;
}

// CALCULATE RADIUS (VALUES GIVEN IN ANGSTROM)

r = (atom_v - implant);
dist2 = r[0]*r[0]+r[1]*r[1]+r[2]*r[2];
if (diag)
{
  track << " Vector from implant: " << r << endl;
  track << " |r|^2 : " << dist2 << endl;
  track << " |r| : " << sqrt(dist2) << endl;
  track << " Charge: " << lattice.get_basis().get_charge()[i] << endl;
}

// Base Contribution

contrib = lattice.get_basis().get_charge()[i] / (dist2*dist2*sqrt(dist2));

// Check Substitutional Site

if (dist2 == 0)
{
  contrib = 0;
  if (diag) track << " *** Substitution site *** " << endl;
}
if (diag) track << " Base Contribution: " << contrib << endl;

// DECIDE IF ATOM MISSING DUE TO IMPURE CALCULATION

impur = (rand()%1000)+1;
if ( impur < (damaged*1000))
{
  contrib = 0;
  if (diag)
  {
    track << " *** Impurity calc: ignore atom *** " << endl;
    track << " % Chance: " << impur << endl;
    track << " Threshold: " << 1000*damaged << endl;
  }
}

// DECIDE IF ATOM MISSING DUE TO SURFACE

if (surface[0] != -1)
int axis = surface[0];
if( r[axis] > surface[1] )
{
    contrib = 0;
    if (diag)
    {
        track << " *** Surface calc: ignore atom *** " << endl;
        track << " Atom Distance: " << r[axis] << endl;
        track << " Crystal Edge:  " << surface[1] << endl;
    }
}

// Calculate EFG from Atom, then add to EFG from Basis
EFG_Add.set( 3*r[0]*r[0] - dist2, 3*r[0]*r[1],
            3*r[0]*r[2], 3*r[1]*r[0], 3*r[1]*r[1] - dist2, 3*r[1]*r[2],
            3*r[2]*r[0], 3*r[2]*r[1], 3*r[2]*r[2] - dist2);
EFG_Add = contrib*EFG_Add;
EFG = EFG + EFG_Add;

if (diag)
{
    track << " EFG from Atom " << i+1 << ": " " << EFG_Add[0] << endl;
    track << " " << EFG_Add[1] << endl;
    track << " " << EFG_Add[2] << endl;
    track << " Net EFG from Basis: " << EFG[0] << endl;
    track << " " << EFG[1] << endl;
    track << " " << EFG[2] << endl << endl;
}

if (diag)
{
    track << "Final EFG from Basis: " << EFG[0] << endl;
    track << " " << EFG[1] << endl;
    track << " " << EFG[2] << endl << endl;
}
track.close();
return EFG;

mat3 loop
{
    int x ,
    int y ,
    int z ,
    vec3 implant ,
    Lattice lattice ,
    double surface[2],
    double damaged,
    bool diag
}
int xtwo = 2 ,
ytwo = 2 ,
ztwo = 2 ,
i ,
j ,
k ;
mat3 EFG ;
if (x == 0) xtwo = 1;
if (y == 0) ytwo = 1;
if (z == 0) ztwo = 1;

x = -1*x;
y = -1*y;
z = -1*z;

for (i = 0 ; i < xtwo ; i++)
{
    for ( j = 0 ; j < ytwo ; j++)
    {
        for ( k = 0 ; k < ztwo ; k++)
        {
            EFG = EFG + calc_efg(x , y , z , implant , lattice, surface,
                                  damaged, diag);
            z = -1*z;
        }
        y = -1*y;
    }
    x = -1*x;
}

return EFG;

mat3 add_shell ( int n,
                vec3 implant, Lattice lattice, Lattice d_lat,
                char shell_type,
                double surface[2],
                double damaged,
                double doped,
                bool diag )
{
    mat3 EFG ;
    vec3 rmax ;
    int x = 0 ,
    y = 0 ,
    z = 0 ;
    double root_chk ,
    dope ;

    // OPEN DIAGNOSTIC FILE, IF REQUESTED
    if (diag)
    {
        ofstream track;
        track.open("diag.rtf", ios::app);
        if(!track)
        {
            cout << "Error writing Shell ";
cout << n;
cout << " to diagnostic file." << endl;
}

track << endl << endl << "**************SHELL " << n << "**************" << endl << endl;
track.close();
}

if( shell_type == 'c' )
{
    while ( x <= n )
    {
        while( (x + y) <= n )
        {
            z = n - x - y;
            dope = (rand()%1000)+1;
            if ( dope < (doped*1000))
            {
                if (diag)
                {
                    ofstream track;
                    track.open("diag.rtf", ios::app);
                    track << " *** Doping calc: Switch basis *** " << endl;
                    track << " % Chance: " << dope << endl;
                    track << " Threshold: " << 1000*doped << endl;
                }
                EFG = EFG + loop(x , y , z , implant , d_lat, surface, damaged, diag);
            }else EFG = EFG + loop(x , y , z , implant , lattice, surface, damaged, diag);
        }y++;
    }x++;
}

if( shell_type == 's' )
{
    while ( (x*x) <= n )
    {
        while( (x*x + y*y) <= n )
        {
            z = sqrt(n - x*x - y*y);
            if( (x*x + y*y + z*z) == n )
            {
                dope = (rand()%1000)+1;
                if ( dope < (doped*1000))
                {
                    if (diag)
                    {
                        ofstream track;
                        track.open("diag.rtf", ios::app);
                        track << " *** Doping calc: Switch basis *** " << endl;
                        track << " % Chance: " << dope << endl;
                    }
                    EFG = EFG + loop(x , y , z , implant , d_lat, surface, damaged, diag);
                }else EFG = EFG + loop(x , y , z , implant , lattice, surface, damaged, diag);
            }y++;
        }x++;
    }
track << "            Threshold: " << 1000*doped <<
endl;
    track.close();
}
EFG = EFG + loop(x, y, z, implant, d_lat, surface, damaged, diag);
else EFG = EFG + loop(x, y, z, implant, lattice, surface, damaged, diag);
    y++;
}
y = 0;
x++;
}

// CHECK SIZE OF EFG, TRY TO ELIMINATE ROUNING ERRORS
// IF NO ERRORS, RETURN VALUE OF EFG
rmax = (n+1)*lattice.get_a()[0] +
      (n+1)*lattice.get_a()[1] +
      (n+1)*lattice.get_a()[2];

root_chk = 1/(rmax.length()*rmax.length()*rmax.length()) ;

if (diag)
{
    ofstream track;
    track.open("diag.rtf", ios::app);
    if(!track)
    {
        cout << "Error writing Shell ";
        cout << n;
        cout << " to diagnostic file." << endl;
    }
    track << "Final EFG from Shell:       " << EFG[0] << endl;
    track << "                            " << EFG[1] << endl;
    track << "                            " << EFG[2] << endl << endl;
    track.close();
}

// DIAGNOSTIC FILE FORMATTING

if (diag)
{
    ofstream track;
    track.open("diag.rtf", ios::app);
    track << "  --------------------------------------------
";
    track << "  |             ROUNDING ERRORS
";
    track << "  |
";
    track << "  | Expected Lower-Bound of EFG = " << root_chk << endl;
    track << "  |
";
    track << "  | Expect rounding error at (i,j) when \n"
    track << "  |
";
    track << "  | EFG(i,j) < Lower-Bound*E-8\n"
    track << "  |
";
    track << "  | Ignoring shell contribution at:\n"
    track.close();
}

// ERASE ROUNING ERRORS FROM EFG
for( int i=0; i<3; i++ )
{
for( int j=0; j<3; j++ )
{
    if( (EFG[i][j] != 0) && (abs(EFG[i][j]) < root_chk*1E-8) )
    {
        if (diag)
        {
            ofstream track;
            track.open("diag.rtf", ios::app);
            track << "  |  EFG[" << i+1 << "][" << j+1 << "]";
            track << "  =  " << EFG[i][j] << endl;
        }
        EFG.set_value(i,j,0);
    }
}

// DIAGNOSTIC FILE FORMATING
if (diag)
{
    ofstream track;
    track.open("diag.rtf", ios::app);
    track << "  |
  --------------------------------------------
" << EFG[0] << endl;
    track << "Final EFG from Shell: " << EFG[1] << endl;
    track << " " << EFG[2] << endl;
    track.close();
}
return EFG;
mat3 create_crystal  (Lattice lattice)
{
    int    loop ,
    d_atoms ,
    atom ,
    seed ;
    char  read ,
    sim_type ,
    loop_type ;
    vec3   *d_pos ,
    pos(0,0,0) ;
    mat3    EFG ,
    tmp ;
    bool   save   = true,
    sv_vq   = true,
    sv_EFG  = true,
    sv_EFG_shell = true,
    sv_eta   = true,
    sv_eta_shell = true,
    bad_loop_type = true,
    diag   = false ;
    double  surface[2] ,
    damaged ,
    doped ,
    x,y,z ,
    ave_dist ;
    float   *d_charge ,
    Basis  *d_bas ;
    Lattice d_lat ;

    // DETERMINE SAVE FILE CONTENTS
    system("clear");
    cout << "

    *********************************************************
    *                     SIMULATION PARAMETERS
    *  Enter type of simulation:
    *      (a) Perfect Crystal
    *      (b) Near surface of Perfect Crystal
    *      (c) Impure Crystal
    *      (d) Near surface of Impure Crystal
    *      (e) Doped Crystal
    *   Choice: ";
    while( bad_loop_type )
    {
        cin >> sim_type;
        if( (sim_type == 'a') ||
            (sim_type == 'b') ||
            (sim_type == 'c') ||
            (sim_type == 'd') ||
            (sim_type == 'e') ) bad_loop_type = false;
        else cout << "* Choice: ";
    }
bad_loop_type = true;

surface[0] = -1;
if ( (sim_type == 'b') || (sim_type == 'd') )
{
  cout << "\n* Surface Info: \n";
  cout << "* Enter Axis normal to surface: ";
  while( bad_loop_type )
  {
    cin >> read;
    switch( read )
    {
      case 'x':
        surface[0] = 0;
        bad_loop_type = false;
        break;
      case 'y':
        surface[0] = 1;
        bad_loop_type = false;
        break;
      case 'z':
        surface[0] = 2;
        bad_loop_type = false;
        break;
      cout << "*   ";
    }
  }
  bad_loop_type = true;
  cout << "* Enter distance to surface: ";
  cin >> surface[1];
}

damaged = -1;
if ( (sim_type == 'c') || (sim_type == 'd') )
{
  cout << "\n* Damage info: \n";
  cout << "* Enter % of missing atoms: ";
  cin >> damaged;
  cout << "* Enter seed for generating r.v.: ";
  cin >> seed;
}
doped = -1;
if (sim_type == 'e')
{
  cout << "\n* Doping info: \n";
  cout << "* Enter % of doped bases: ";
  cin >> doped;
  cout << "* Enter seed for generating r.v.: ";
  cin >> seed;
  d_atoms = lattice.get_basis().get_atom_ct();
  d_charge = new float [d_atoms];
  d_pos = new vec3 [d_atoms];
  for (int i=0 ; i<d_atoms ; i++)
  {
    d_charge[i] = lattice.get_basis().get_charge()[i];
    d_pos[i] = lattice.get_basis().get_position()[i];
  }
  d_bas.set_atom_ct(d_atoms);
  d_bas.set_charge(d_charge);
  d_bas.set_position(d_pos);
  while( bad_loop_type )
  {
    d_bas.display();
    cout << "\n* Change which atom? (0 for none): ";
    cin >> atom;
    atom = atom - 1;
  }
}
if ((atom < 0) || (atom >= d_atoms)) bad_loop_type = false;
else
{
    cout << "*    New charge: ";
    cin >> d_charge[atom];
    cout << "*    New position:\n";
    cout << "*      x - ";
    cin >> x;
    cout << "*      y - ";
    cin >> y;
    cout << "*      z - ";
    cin >> z;
    d_pos[atom].set(x, y, z);
}
}
bad_loop_type = true;
d_lat.set_a(lattice.get_a());
d_lat.set_basis(d_bas);

// DETERMINE OUTPUT OPTIONS

cout << "*\n*  Save calculation to file? ";
cin >> read;
if (read == 'd')
{
    diag = true;
    cout << "*\n*  Diagnostic Output Enabled";
    cout << "\n*  Save? ";
cin >> read;
}
if ( (read == 'n') || (read == 'N') ) save = false;
else
{
    cout << "*    Save Vq? ";
    cin >> read;
    if ( (read == 'n') || (read == 'N') ) sv_vq = false;
    cout << "*    Save net EFG? ";
    cin >> read;
    if ( (read == 'n') || (read == 'N') ) sv_EFG = false;
    cout << "*    Save shell EFG? ";
    cin >> read;
    if ( (read == 'n') || (read == 'N') ) sv_EFG_shell = false;
    cout << "*    Save net Asymmetry? ";
    cin >> read;
    if ( (read == 'n') || (read == 'N') ) sv_eta = false;
    cout << "*    Save shell Asymmetry? ";
    cin >> read;
    if ( (read == 'n') || (read == 'N') ) sv_eta_shell = false;

    if ( !sv_vq &&
        !sv_EFG &&
        !sv_EFG_shell &&
        !sv_eta &&
        !sv_eta_shell ) save = false;
}
cout << "*\n";
ofstream writefile;
if (save)
{
    writefile.open("output.txt");
    if (!writefile)
{  
cout << "Error Opening 'output.txt'
" << endl;
cout << "Aborting Calculation...
" << endl;
return EFG;
}
pos = implant_location(lattice.get_basis());
cout << "*
";

// DETERMINE LOOP TYPE
while (bad_loop_type)
{
cout << "*  Loop Type: " << endl << "*    (C)ubic" << endl << "*  (S)pherical" << endl;
cout << "*    Choice: ";
cin >> loop_type;
switch( loop_type )
{
case 'C':  loop_type = 'c';
case 'c':  bad_loop_type = false;
           break;
case 'S':  loop_type = 's';
case 's':  bad_loop_type = false;
           break;
}
if (bad_loop_type) cout << "*  Unknown loop type
*  Unknown loop type\n*\n";
}
bad_loop_type = true;

// DETERMINE NUMBER OF LOOPS
cout << "*\n*  How Many shells? ";
cin >> loop;
cout << "*\n*********************************************************";

// CLEAR AND INITIALIZE DIAGNOSTIC FILE
ofstream track;
track.open("diag.rtf");
if(!track)
{
cout << "Error Opening diagnostic file"
;
}
if (!diag) track << "No diagnostic file requested\n\nPress 'd' at save query to activate diagnostic summary\n\n\n";
else
{
track << "EFG CALCULATOR DIAGNOSTIC FILE" << endl << endl;
track << "NOTE: All values unscaled until final calculation!" << endl << endl;
track << "Simulation Info: 
" << endl;
track << " Simulation type: ";
switch(sim_type)
{
case 'a':  track << "Perfect Crystal" << endl;
           break;
case 'b':  track << "Near Surface" << endl;
           track << " Surface Axis: " << surface[0] << endl;
           track << " Distance: " << surface[1] << endl;

}
break;
case 'c':
    track << "Damaged Crystal" << endl;
    track << " % of atoms missing: " << damaged <<
    endl;
    track << " r.v. seed: " << seed << endl;
break;
case 'd':
    track << "Near Surface and Damaged Crystal" <<
    endl;
    track << " Surface Axis: " << surface[0] <<
    endl;
    track << " Distance: " << surface[1] <<
    endl;
    track << " % of atoms missing: " << damaged <<
    endl;
    track << " r.v. seed: " << seed << endl;
break;
case 'e':
    track << "Doped Crystal" << endl;
    track << " % of doped bases: " << doped << endl;
    track << " r.v. seed " << seed << endl;
    track << " Doped Basis Info: " << endl;
    track << " Atom Charge Position" << endl;
    for( int i=0; i<lattice.get_basis().get_atom_ct(); i++ )
    {
        track << " " << i+1;
        track.width(15);
        track << d_bas.get_charge()[i];
        track.width(13);
        track << d_bas.get_position()[] << endl;
    }
    break;
}
track << endl;
track << " Implant Position: " << pos << endl;
track << " Type of loop: " << loop_type << endl;
track << " Number of loops: " << loop << endl << endl;
track << "Lattice Info: " << endl;
track << " Primary Vectors: " << endl;
track << " << lattice.get_a()[0] << endl;
track << " << lattice.get_a()[1] << endl;
track << " << lattice.get_a()[2] << endl << endl;
track << " Basis Info: " << endl;
track << " Atom Charge Position" << endl;
for( int i=0; i<lattice.get_basis().get_atom_ct(); i++ )
    {
        track << " " << i+1;
        track.width(15);
        track << lattice.get_basis().get_charge()[i];
        track.width(13);
        track << lattice.get_basis().get_position()[] << endl;
    }
}
track.close();

// CALCULATE EFG OVER EACH SHELL, WRITE TO OUTFILE IF REQUESTED
srand(seed);
double log_switch;
log_switch = log(10);
for( int i=0; i<=loop; i++ )
{
    tmp = add_shell(i, pos, lattice, d_lat, loop_type, surface,
    damaged, doped, diag);
    EFG = EFG + tmp;
    if (save)
    {
      if (loop_type == 's')
      {
        ave_dist = sqrt(i)*(lattice.get_a()[0].length() +
          lattice.get_a()[1].length() +
          lattice.get_a()[2].length())/3;
        writefile << ave_dist << "  ";
        writefile << log(ave_dist)/log_switch << " *-*-*  ";
      }
      if (sv_vg) writefile <<
          diagonalize(EFG)[2][2]*SCALE*ELEM*QUAD*1E-4/8/H;
      if (sv_vg) writefile << "  ";
      if (sv_EFG) writefile << EFG[0]*SCALE*1E30 << "  ";
      if (sv_EFG) writefile << EFG[1]*SCALE*1E30 << "  ";
      if (sv_EFG) writefile << EFG[2]*SCALE*1E30 << "  ";
      if (sv_EFG) writefile << "  ";
      if (sv_EFG_shell) writefile << tmp[0]*SCALE*1E30 << "  ";
      if (sv_EFG_shell) writefile << tmp[1]*SCALE*1E30 << "  ";
      if (sv_EFG_shell) writefile << tmp[2]*SCALE*1E30 << "  ";
      if (sv_EFG_shell) writefile << "  ";
      if (sv_eta) writefile << eta(EFG);
      if (sv_eta) writefile << "  ";
      if (sv_eta_shell) writefile << eta(tmp);
      writefile << endl;
    }
    cout << endl;
    if (save)
    {
      writefile.close();
      cout << "Data saved to 'output.txt'" << endl;
    }
}
// COMPLETE DIAGNOSTIC FILE
if (diag)
{
  track.open("diag.rtf", ios::app);
  if(!track)
  {
    cout << "Error Opening diagnostic file";
  }

  track << "\n\n\n\n";
  track << "*******************************************************************\n";
  track << "\n";
  track << "\n";
  track << "EFG = " << EFG[0] << endl;
  track << "\n";
  track << "\n";
  track << "Scaling Factor = " << 1E30*SCALE << endl;
  track << "\n";
  track << "\n";
}
track << " Actual EFG = " << EFG[0]*SCALE*1E30 << endl;
track << "                " << EFG[1]*SCALE*1E30 << endl;
track << "                " << EFG[2]*SCALE*1E30 << endl;
track << "*
";
track << "*
";
track << "**************************************************************";

track.close();
}
EFG = EFG*SCALE*1E30;
return EFG;
}
mat3 rotate
(mat3 EFG)
{
char axis;
double angle;
bool check = true;
mat3 _EFG;

_EFG = EFG;
system("clear");
while(check)
{
    cout << "\n\nChoose Axis: ";
    cin >> axis;
    if (axis == 'X') axis = 'x';
    if (axis == 'Y') axis = 'y';
    if (axis == 'Z') axis = 'z';
    if( (axis == 'x') || (axis == 'y') || (axis == 'z') ) check = false;
    if(check) cout << "Unknown Axis";
}
check = true;
while(check)
{
    cout << "\n\nChoose Angle: ";
    cin >> angle;
    if ( (-180 <= angle) && (angle <=180) ) check = false;
    if(check) cout << "Angle must be contained in [-180,180]";
}

_EFG = transform_EFG(_EFG , axis, angle);
cout << "\n\nNew EFG:\n" << _EFG;
cout << "\n\nSave? ";
cin >> axis;
if( (axis == 'n') || (axis == 'N') ) return(EFG);
else return(_EFG);
}
mat3 set_EFG()
{
mat3 EFG;
double xx = 0,
    xy = 0,
    xz = 0,
yx = 0,
yy = 0,
yz = 0,
zx = 0,
zy = 0,
zz = 0;

system("clear");
cout << "Input Format: | xx xy xz |
              | yx yy yz |
              | zx zy zz |

cout << "Input EFG by row: ";
cin >> xx >> xy >> xz;
cout << "                  ";
cin >> yx >> yy >> yz;
cout << "                  ";
cin >> zx >> zy >> zz;

EFG.set( xx, xy, xz,
yx, yy, yz,
zx, zy, zz);

return(EFG);

mat3 scale (mat3 EFG)
{
    mat3 _EFG;
    double precision,
    max_val;
    int max[2],
    i,
    j;
    char read;

    _EFG = EFG;
    max[0]=0;
    max[1]=0;

    for( i = 0; i < 3 ; i++ )
    {
        for( j = 0; j < 3 ; j++ )
        {
            if (abs(_EFG[i][j]) > abs(_EFG[max[0]][max[1]]))
            {
                max[0]=i;
                max[1]=j;
            }
        }
    }

    max_val = abs(_EFG[max[0]][max[1]]);

    system("clear");
    cout << "\n\n Enter precision: ";
cin >> precision;
cout << endl << endl;
    max_val = max_val*precision;
cout << "Threshold: " << max_val << endl << endl;
for( i = 0; i < 3 ; i++ )
{
    for( j = 0; j < 3 ; j++)
    {
        if( abs(_EFG[i][j]) < abs(max_val) ) _EFG[i][j] = 0;
    }
}

cout << "New EFG:\n" << _EFG;
cout << "\n\nSave? ";
cin  >> read;
if( ( read == 'n' ) || ( read == 'N' ) ) return(EFG);
else return(_EFG);
}

void results(mat3 EFG)
{
    mat3 _EFG;
    double _eta,
            Vzz;
    _EFG = diagonalize(EFG);
    _eta = eta(EFG);
    Vzz = _EFG[0][0];
    if( abs(_EFG[1][1]) > abs(Vzz) ) Vzz = _EFG[1][1];
    if( abs(_EFG[2][2]) > abs(Vzz) ) Vzz = _EFG[2][2];
    cout << "\nDiagonalized EFG: \n\n" << _EFG;
    cout << "\n    Q = " << QUAD << " mb";
    cout << "\n    nu = " << Vzz*ELEM*QUAD*1E-34/8/H << " KHz";
    cout << "\n    q";
    cout << "\nEta = " << _eta << endl << endl;
}

mat3 flex(mat3 EFG, Lattice lat)
{
    Basis tmp_bas;
    Lattice tmp_lat;
    vec3  pos,
          tmp_pos[5][10];
    float charge[5][10];
    mat3  _EFG;
    double flex,
            sflex,
            fake1[3],
            fake2 = -1;
    bool fake3 = true,
    loop = true;
    char  read;
    int   i, j,

    _EFG = EFG;
    charge[0][0] = 2;
    charge[0][1] = 2;
    charge[0][2] = 2;
    charge[0][3] = 2;
    charge[1][0] = -2;
charge[1][1] = -2;
charge[2][0] = -4;
charge[2][1] = -4;
charge[2][2] = -4;
charge[3][0] = 2;
charge[3][1] = 2;
charge[3][2] = 2;
charge[3][3] = 2;
charge[3][4] = -2;
charge[3][5] = -2;
charge[4][0] = 2;
charge[4][1] = 2;
charge[4][2] = 2;
charge[4][3] = 2;
charge[4][4] = -2;
charge[4][5] = -2;
charge[4][6] = -4;
charge[4][7] = -4;
charge[4][8] = -4;
charge[4][9] = -4;

tmp_pos[0][0].set(.5,0,0);
tmp_pos[0][1].set(0,.5,0);
tmp_pos[0][2].set(.5,1,0);
tmp_pos[0][3].set(1,.5,0);
tmp_pos[1][0].set(.5,.5,.5);
tmp_pos[1][1].set(.5,.5,-.5);
tmp_pos[2][0].set(0,0,0);
tmp_pos[2][1].set(0,1,0);
tmp_pos[2][2].set(1,0,0);
tmp_pos[2][3].set(1,1,0);
tmp_pos[3][0].set(.5,0,0);
tmp_pos[3][1].set(0,.5,0);
tmp_pos[3][2].set(.5,1,0);
tmp_pos[3][3].set(1,.5,0);
tmp_pos[3][4].set(.5,.5,.5);
tmp_pos[3][5].set(.5,.5,-.5);
tmp_pos[4][0].set(.5,0,0);
tmp_pos[4][1].set(0,.5,0);
tmp_pos[4][2].set(.5,1,0);
tmp_pos[4][3].set(1,.5,0);
tmp_pos[4][4].set(.5,.5,.5);
tmp_pos[4][5].set(.5,.5,-.5);
tmp_pos[4][6].set(0,0,0);
tmp_pos[4][7].set(0,1,0);
tmp_pos[4][8].set(1,0,0);
tmp_pos[4][9].set(1,1,0);

c[0] = 4;
c[1] = 2;
c[2] = 4;
c[3] = 6;
c[4] = 10;

while( loop )
{
    system("clear");
    cout << endl << endl;
    cout << "   Nearest Neighbours to Loop\n";
    cout << "    (1) O\n";
    cout << "    (2) Sr\n";
    cout << "    (3) Ti\n";
    cout << "    (4) SrO\n";
cout << "    (5) SrTiO
\n";
cout << "    Selection: ";
cin  >> atom;
atom = atom - 1;
if( (0 <= atom) && (atom <= 4) ) loop = false;
}
cout << "    Enter Flex parameter: ";
cin  >> flex;
sflex = sqrt(2) * flex;
pos = implant_location(lat.get_basis());
tmp_lat.set_a(lat.get_a());
tmp_bas.set_atom_ct(ct[atom]);
tmp_bas.set_charge(charge[atom]);
tmp_bas.set_position(tmp_pos[atom]);
cout << endl << endl;
tmp_bas.display();
tmp_lat.set_basis(tmp_bas);

_EFG = _EFG +
calc_efg(0,0,0,pos,tmp_lat,fake1,fake2,fake3)*SCALE*1E30;
cout << "\n\n  Cleared EFG: \n" << _EFG << endl << endl;
for( j=0 ; j<5 ; j++ )
{
    for( i=0 ; i<ct[atom] ; i++ )
    {
        charge[j][i] = -1*charge[j][i];
    }
}
tmp_pos[0][0].set(.5,flex,0);
tmp_pos[0][1].set(flex,.5,0);
tmp_pos[0][2].set(.5,1-flex,0);
tmp_pos[0][3].set(1-flex,.5,0);
tmp_pos[1][0].set(.5,.5,.5+flex);
tmp_pos[1][1].set(.5,.5,.5+flex);
tmp_pos[1][2].set(-sflex,-sflex,0);
tmp_pos[1][3].set(-sflex,1+sflex,0);
tmp_pos[1][4].set(-sflex,1+sflex,0);
tmp_pos[2][0].set(.5,flex,0);
tmp_pos[2][1].set(flex,.5,0);
tmp_pos[2][2].set(.5,1-flex,0);
tmp_pos[2][3].set(1-flex,.5,0);
tmp_pos[2][4].set(.5,.5,.5+flex);
tmp_pos[2][5].set(.5,.5,.5+flex);
tmp_pos[2][6].set(-sflex,-sflex,0);
tmp_pos[2][7].set(-sflex,1+sflex,0);
tmp_pos[2][8].set(1+sflex,-sflex,0);
tmp_pos[2][9].set(1+sflex,1+sflex,0);
tmp_pos[3][0].set(.5,flex,0);
tmp_pos[3][1].set(flex,.5,0);
tmp_pos[3][2].set(.5,1-flex,0);
tmp_pos[3][3].set(1-flex,.5,0);
tmp_pos[3][4].set(.5,.5,.5+flex);
tmp_pos[3][5].set(.5,.5,.5+flex);
tmp_pos[4][0].set(.5,flex,0);
tmp_pos[4][1].set(flex,.5,0);
tmp_pos[4][2].set(.5,1-flex,0);
tmp_pos[4][3].set(1-flex,.5,0);
tmp_pos[4][4].set(.5,.5,.5+flex);
tmp_pos[4][5].set(.5,.5,.5+flex);
tmp_pos[4][6].set(-sflex,-sflex,0);
tmp_pos[4][7].set(-sflex,1+sflex,0);
tmp_pos[4][8].set(1+sflex,-sflex,0);
tmp_pos[4][9].set(1+sflex,1+sflex,0);

tmp_bas.set_charge(charge[atom]);
tmp_bas.set_position(tmp_pos[atom]);
cout << endl << endl;
tmp_bas.display();
tmp_lat.set_basis(tmp_bas);
calc_efg(0,0,0,pos,tmp_lat,fake1,fake2,fake3)*SCALE*1E30;
cout << "\n\n New EFG: \n" << _EFG << endl << endl;
results(_EFG);
cout << "Save? ";
cin >> read;
if( (read == 'n') || (read == 'N') ) return(EFG);
return(_EFG);
}
mat3 add_vac(mat3 EFG, Lattice lat)
{
  mat3 _EFG;
  int x=0,
      y=0,
      z=0,
      atom;
  Basis tmp_bas;
  Lattice tmp_lat;
  vec3 tmp_pos[1],
       pos;
  float charge[1];
  char read;
  _EFG = EFG;
  system("clear");
pos = implant_location(lat.get_basis());
cout << "\n\n Enter lattice point: \n  x co-ord: ";
cin >> x;
cout << "  y co-ord: ";
cin >> y;
cout << "  z co-ord: ";
cin >> z;
cout << endl << endl;
lat.get_basis().display();
cout << "\n Enter atom to remove: ";
cin >> atom;
  // REVERSE CHARGE TO COUNTER ATOM ADDED LATER ON
  charge[0] = -1*lat.get_basis().get_charge()[atom-1];
  // SET THE REST OF THE VARIABLES
  tmp_lat.set_a(lat.get_a());
tmp_bas.set_atom_ct(1);
tmp_pos[0] = lat.get_basis().get_position()[atom-1];
tmp_bas.set_charge(charge);
tmp_bas.set_position(tmp_pos);
tmp_lat.set_basis(tmp_bas);
  bool fake3 = false;
  double fake1[3],
          fake2 = -1;
fake1[0]=-1;

_EFG = _EFG + calc_efg(x,y,z,pos,tmp_lat,fake1,fake2,fake3)*SCALE*1E30;

cout << "

  New EFG:
" << _EFG << endl << endl;
results(_EFG);

cout << "Save? ";
cin >> read;
if( (read == 'n') || (read == 'N') ) return(EFG);
return(_EFG);
}

int step_surface(Lattice lattice)
{
  mat3  EFG;
  vec3  pos;
  bool  bad_loop_type = true,
        bool_false = false;
  char  read;
  double surface[2],
         start = 0,
         step = 1;
  int    loop = 0,
         sim_loop = 0;
  char  save_file[8],
        shell_type = 's';
  Lattice d_lat;

  system("clear");
  cout << "
       SURFACE DEPENDENCE

";
  pos = implant_location(lattice.get_basis());

cout << "
Surface Info: 
";
  cout << "   Enter Axis normal to surface: ";
  while( bad_loop_type )
  {
    cin >> read;
    switch( read )
    {
      case 'x': surface[0] = 0;
                bad_loop_type = false;
                break;
      case 'y': surface[0] = 1;
                bad_loop_type = false;
                break;
      case 'z': surface[0] = 2;
                bad_loop_type = false;
                break;
    }
  }
  cout << "   Enter Initial distance to surface: ";
  cin >> start;
  cout << "   Enter distance step size: ";
  cin >> step;
  cout << "   Enter Number of steps: ";
  cin >> loop;
  cout << "   Enter Number of iterations per simulation: ";
  cin >> sim_loop;
cout << "\nData file name: ";
cin  >> save_file;

ofstream writefile;
writefile.open(save_file);
if (!writefile)
{
    cout << "Error Opening " << save_file << endl;
    cout << "Aborting Calculation..." << endl;
    return(1);
}

double log_switch = log(10),
    vq,
    _eta;

writefile << "Loop# * Dist Log(Dist) * vq Log(vq) * eta Log(eta)\n\n";

for (int i = 0; i < loop; i++)
{
    for (int j = 0; j < sim_loop; j++)
    {
        EFG = EFG + add_shell(j, pos, lattice, d_lat, shell_type,
                               surface, -1, -1, bool_false);
    }
    writefile << i+1 << "    *    ";
    writefile << surface[1] << "    ";
    writefile << log(surface[1])/log_switch << "    *    ";
    vq = diagonalize(EFG)[2][2]*SCALE*ELEM*QUAD*1E-4/8/H;
    writefile << vq << "    " << log(vq)/log_switch << "    *    ";
    _eta = eta(EFG);
    writefile << _eta << "    " << log(_eta)/log_switch << endl;
    EFG.set(0,0,0,0,0,0,0,0,0);
}

writefile.close();

return(0);
}
LATTICE_INFO.DTT

// Storage file for Lattice and Basis info
// Input only one lattice
// For each property, attach the appropriate prefix (below) and enclose
// the value in "<$
// Separate vector values by ":"
// Note: all sites MUST be symmetric, => if charge c exists at (x,y,z)
in lattice
// then there exists charge c at
// (-x,-y,-z)

// PV# = primitive vectors (# = 1, 2 or 3)
// AT = number of atoms in basis
// CH# = Charge for atom '#' in basis
// PS# = position for atom '#' in basis
// IT = number of interstitial sites in basis
// IP# = position of interstitial site '#' in basis

PV1<3.91:0:0>
PV2<0:3.91:0>
PV3<0:0:3.91>

IT<3>

IP1<0.5:0.5:0>
IP2<0.5:0:0.5>
IP3<0:0.5:0.5>

AT<5>

CH1<4>  PS1<0:0:0>
CH2<-2>  PS2<.5:0:0>
CH3<-2>  PS3<0:.5:0>
CH4<-2>  PS4<0:0:.5>
CH5<-2>  PS5<.5:.5:.5>

EOF