Spring 2011

MUSCLE: A Simulation Toolkit Modeling Low Energy Muon Beam Transport in Crystals

Nazmus Saquib
Bard College, ns428@bard.edu

MUSCLE: A Simulation Toolkit Modeling Low Energy Muon Beam Transport in Crystals

A Senior Project submitted to
The Division of Science, Math, and Computing
of Bard College

by

Nazmus Saquib

Annandale-on-Hudson, New York
May 2011

Abstract

The project involves the development of MUSCLE (MUonS Cascade at Low Energy), a collection of programs written in C++ and Mathematica to numerically simulate the passage of low energy muon beams through crystals. Monte Carlo methods employing binary collision approximation calculations and appropriate molecular dynamics algorithms are implemented to construct the trajectories and determine the spatial distribution of stopped muons in single crystals. Channeling of muon particles along certain crystal planes are also found. Binary collision approximation and molecular dynamics algorithms are compared and the possible effect of channeling is discussed.

Acknowledgements

I am grateful to Allah for granting me the ability to achieve what I (and even my family) could not imagine to achieve even a few years ago. I would like to thank the entire physics department: Matthew Deady, for being an awesome adviser and super awesome teacher; Christian Bracher, for showing me what physics really is, and how to tackle it; Peter Skiff, for showing me what science is, in his History of Science class, and Burton Brody, for making me realize that what I learn is more important than what shows up in my transcript.

I am thankful to my summer REU mentor Dr. William Kossler (College of William & Mary) for advising me on my senior project at times. I would also like to thank Keith O'Hara for encouraging me to do computer vision research with him, and Greg Landweber for being a wonderful teacher. Becky Thomas introduced me to artificial intelligence, and Lauren Rose showed me mathematics is not only about solving problems from the book, there is more to it. I am grateful to both of them.

I would also like to thank Jyoti Dev for being such a good friend over the last four years. I am eternally indebted to my parents and my sisters for providing me support and courage in every single day of my life. May be I am far away from you, but you are always with me.

Contents

1

1 μSR: Methods and Applications

1.1 Introduction

Muons are unstable elementary particles that are abundant in space, and they can be produced in particle accelerators with much more intensity. At the atomic level, interactions between muons and surrounding particles such as the atoms and electrons of a particular material can provide wealth of information regarding the material such as microscopic magnetic properties. Muon science deals with such phenomena, and the methods mostly rely on the unique physical properties of this particle. The methods are collectively called μSR, which stands for muon Spin Rotation/Spin Relaxation/Spin Resonance techniques. Other than their usage in condensed matter physics, μSR is often used in biology to characterize protein by providing information about the microscopic level of electron transfer in proteins, and in medical physics to perform non-destructive elemental analysis of human bodies. This project involves the precise calculations of muons stopping in crystalline samples, which is crucial to every μSR studies as knowing the accurate position of stopped muons is the foremost step in the analysis. This section briefly describes the methods of muon spin relaxation, the kind of applications that can be employed using this technique, and the importance of precisely calculating the stopped muon sites in these experiments.

1.2 Properties and Behavior of Muons Inside Matter

Muons have some unique characteristics that make them particularly useful in applied science research.

- Muons have unique mass (0.114 amu), like a heavy electron and a light proton.
- They exhibit radioactivity with polarization phenomena.
- They exhibit electromagnetic interaction with matter without a strong interaction.

They can be found in two charge types (postive μ^+ and negative μ^-), with a spin of 1/2. Muons have a lifetime of of about 2.2 μs, with the following major decay modes:

$$\mu^+ \rightarrow e^+ + \overline{\nu}_\mu + \nu_e$$

$$\mu^- \rightarrow e^- + \nu_\mu + \overline{\nu}_e$$

where ν_e and ν_μ are the electron and muon neutrinos and $\overline{\nu}_e$ and $\overline{\nu}_\mu$ are the corresponding antineutrinos. We are mostly interested in μ^+since it is used as a "passive" probe to study the magnetic properties of the host. The stopping of muons in the material under consideration is divided into a few phases. Figure 1.1 [Nagamine, 41] provides a summary of the energy loss processes and depolarization mechanisms that occur during μ^+ stopping.

As seen in the figure, high energy muons beams produced in accelerators are slowed down to a few keVs by interaction with electrons. At 2-3 keV, the μ^+ particle may capture an electron to become a neutral Mu (muonium, a hydrogen-like atom composed of μ^+ and e^-). This might be the case when μ^+ travels through gases, insulators and most semiconductors. Then it is decelerated via elastic collisions with atoms and inelastic energy loss due to free electron cloud. While slowing down, Mu may also lose an electron through interaction with other atoms to become μ^+ again. After stopping (1-2 eV), the μ^+, Mu or μ^- are said to be *implanted* into the material. The initial polarization of μ^+ changes over time due to formation of Mu atoms and interaction with local magnetic fields, which is the basis of μSR studies. The prolonged half life of the particle is really useful as it does not decay during the stopping process. It is also important to note that such μ^+ probes operate in the low energy range, starting from 2-3 keV. Thus our concern is mostly about the stopping process associated with low energy regime.

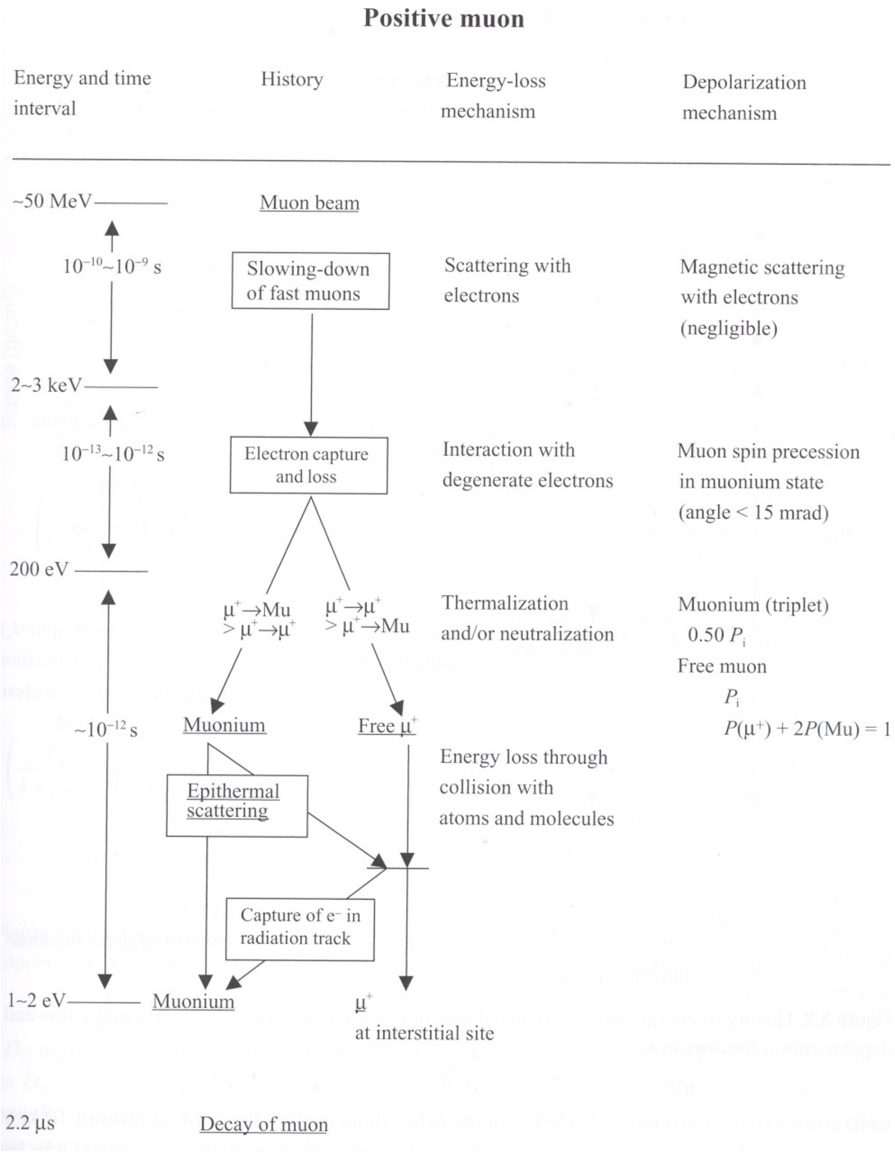

Positive muon
Energy and time interval
History
Energy-loss mechanism
Depolarization mechanism
~50 MeV
Muon beam
10^{-10}~10^{-9} s
Slowing-down of fast muons
Scattering with electrons
Magnetic scattering with electrons (negligible)
2~3 keV
10^{-13}~10^{-12} s
Electron capture and loss
Interaction with degenerate electrons
Muon spin precession in muonium state (angle < 15 mrad)
200 eV
$\mu^+ \to$ Mu $> \mu^+ \to \mu^+$
$\mu^+ \to \mu^+ > \mu^+ \to$ Mu
Thermalization and/or neutralization
Muonium (triplet)
0.50 P_i
Free muon
P_i
$P(\mu^+) + 2P(\mathrm{Mu}) = 1$
~10^{-12} s
Muonium
Free μ^+
Energy loss through collision with atoms and molecules
Epithermal scattering
Capture of e− in radiation track
1~2 eV
Muonium
μ^+
at interstitial site
2.2 μs
Decay of muon

Figure 1.1. Energy loss mechanisms involved with muon stopping.

1.3 μSR Experiments

The μ spin relaxation technique is based on the fact that the initial spin of a muon may be relaxed due to interaction with the local magnetic field distribution and its dynamic and random fluctuation. Due to such interactions, the projection of the muon spin along its initial spin direction changes over time, i.e. the longitudinal polarization relaxes. In order to observe this experimentally, two counters can be set in the backward and forward directions with reference to the initial direction of the incident muon to measure the forward/backward asymmetry, as shown in the following figure.

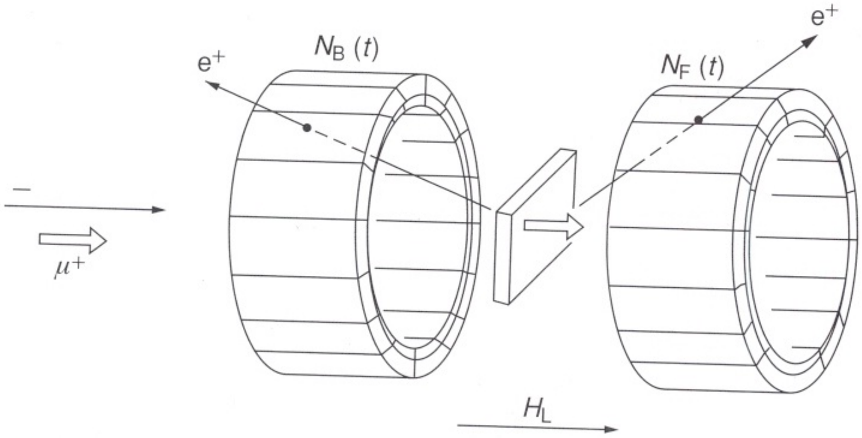


Figure 1.2. Detection of muon spin relaxation. [Nagamine, 105]

The relaxation can also be observed without the applied field $\vec{H_L}$, using zero-field μSR techniques (ZF-μSR). The stopped μ^+ decays inside the sample under study, and gives out e^+ that are detected in the counters. The time evolution of such anisotropic e^+ decay corresponds to the motion of the muon spin direction, which in turn can be related to the dynamic or static nature of local magnetic field using a one-to-one correspondence [Nagamine, 105 - 109].

All low energy μSR (LE-μSR) experiments involve the following steps:

1. Use of an energy degrader (usually a suitable material of certain thickness) to lower the energy of the muon before it enters the target.

2. Focusing of the beam to the target. From the previous step, certain amount of spread in the beam is introduced, along with some contamination. These deviations are minimized in this step using a collimator.

3. Detection of positron after the muon stops in the sample and decays.

4. Precise time difference measurement between the stopping of muon and detection of positrons.

5. Data collection and statistical calculations.

In the statistical calculation, the possible remaining contamination in the beam and the noise in the signals are characterized and removed. With the improved data, one can now deduce what happened at the relaxation site of the muon.

1.4 Applications in Condensed Matter Physics

Determining or predicting the location of stopped μ^+ is crucial for the later stages of μ^+SR calculations. The properties to be probed cannot be used here. So we have to rely solely on the properties of μ^+. The following figure shows a basic example of the kind of qualitative and quantitative inferences that can be made about the location of μ^+ from a μ^+SR experiment.

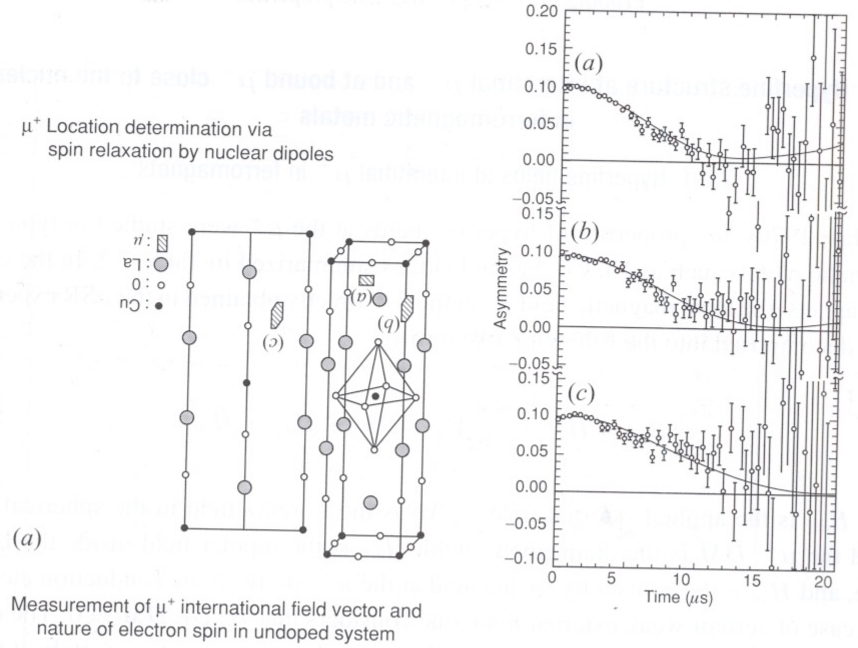


Figure 1.3. Determination of μ^+ site using asymmetry data. [Nagamine, 129]

The μ^+ location can be determined from the asymmetry data that essentially captures the spin relaxation scenario. It can be noticed that the asymmetry dies out exponentially and rises again in the cases labelled (a) and (b). The μ^+ sites at (a) and (b) in the crystal shows that it is likely for the magnetic field there to fluctuate dynamically that may cause such behavior in the relaxation process, as the muons at (a) or (b) are surrounded by atomic dipoles. On the other hand, (c) is at a location where dipole contributions from the surrounding atoms do not fluctuate much according to the experiment data (smooth relaxation curve). Thus, with some idea about the crystal structure and experimentally determined asymmetry functions, we can learn more about the μ^+ site, surrounding dipole contributions and the nature of local field distribution of a crystalline sample. This is the essential concept behind using muons as a "probe." Other μ^+SR (spin rotation/resonance) techniques are also used to construct hyperfine field vector profile and magnetic phase diagrams for a sample based on where the muon stops and what behavior it exhibits, and this may provide a complete picture of the magnetic properties [Nagamine, 128 - 129, 132 - 140].

An important concern in such studies, therefore, is predicting the location of a muon. This provides a useful background check for experimentally determined location. In case the experimental data cannot be used to locate muons, a good prediction may act as an equivalent of experimental observation. Ion beam simulation software is widely used for predicting the spatial distribution of stopped muons, and our goal is to come up with a reliable prediction of muon stopping locations using such simulation algorithms. However, there are problems associated with some existing simulation packages. Firstly, some do not take account of the channeling phenomena associated with ion beams passing through crystals. Secondly, those which do take account of this phenomena are often not reliable in terms of carrying out an accurate calculation of implantation depth profiles. These issues are addressed in the following sections.

1.5 Channeling

Experimental results have shown that ions and recoiling nuclei move in a crystal in a different way than in amorphous materials. In particular in the case of motion along crystallographic axes and planes, the so-called "channeling effect" can occur and the ions manifest an anomalous deep penetration into the lattice of the crystal.

The channeling effect can occur in crystalline materials due to correlated collisions of ions with target atoms. In particular, the ions through the open channels have ranges much larger than the maximum range they would have if their motion would be either in other directions or in amorphous materials. When a low-energy ion goes into a channel it transfers its energy mainly to electrons rather than to nuclei in the lattice and, thus, it usually penetrates much deeper into the crystal compared to its regular trajectory in an amorphous target. The figure below depicts the effect of channeling in a Sodium Iodide crystal.

.

.

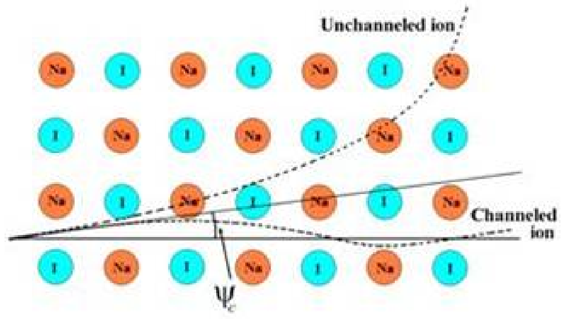


Figure 1.4. Channeling in Sodium Iodide crystal under a certain incident angle ψ_c. (Picture courtesy: http://statistics.roma2.infn.it/~dama/web/nai_dmp_20.html)

The ion in this example, under a certain critical angle ψ_c, enters a channel, deflects from different nuclei but still stays in the channel. On the other hand, the unchanneled ion behaves much like what it would do in an amorphous sample, i.e. scatter from random atoms.

The effects of channeling can be a very important factor in μSR studies. Often, the sample being studied is a multilayered one, composed of several samples stuck together in layers. If a good proportion of muons channel out of the first layer, then the overall multilayer spatial distribution would look considerably different from a simulation run for an amorphous sample. Other than such concerns, it should also be noted that even for a single layer sample, channeling of muons may result in greater depths being reached, and hence, affect the shape of the spatial distribution.

1.6 Existing Software

Monte Carlo algorithms have been extensively used to predict the behavior and trajectories of ions propagating through a solid. Ion Beam physics is mostly concerned about the stopping distribution of the incident ions, since this information can be used to characterize the properties of materials. There are different varieties of programs available from many different authors that find the ion trajectories, stopping distribution, damage calculation and sputtering yields. The programs mostly fall under two categories according to the treatment of a sample. The samples can be treated as amorphous targets, or crystalline targets. There are further variations in the implementation of the algorithms for each of the category. The binary collision approximation is generally well accepted in the μSR community [Dubman, 2009], which takes account of two ions scattering from each other and does not consider the influence of the other neighbor ions in a single event. There are other models such as the molecular dynamics model that deals with the problem from a many-body perspective.

1.6.1 Geant4

Geant4 is an open source C++ framework that is famous for its versatility and stability. It has been used in many physics applications that deal with some form of ion passage through matter. It is a recognized tool for high energy, medical, radiation and ion beam physics simulations. Our initial idea was to write a Geant4 simulation of muons passing through layered Iron or Niobium thin film samples. Geant4 has nice features for constructing any type of geometry that describes the experimental setup. Another useful feature of Geant4 is the ease of combining different physics processes in the program. For our task, we needed to take account of muon precession and decay processes. Geant4 includes models for all the physics processes involved with muons except the polarization property, which, according to the documentation, will be included in future releases. However, it does not track channeling of projectile particles, whereas our goal is to understand and find a measure of the channeling effect associated with muons traveling through crystals. As famous and useful as it is in the ion beam physics community, Geant4 was a dead end for our goal.

1.6.2 TRIM

TRIM, which stands for TRansport of Ions in Matter, has been used by many experimentalists and theoreticians for over 25 years. The specifications and description of the algorithms associated with the program are explained in a book by one of the authors [Ziegler, 1985]. After we ran a few simulations using TRIM, results were quite promising. We were successful in creating a simulation of layered thin film samples and generated depth distribution plots for several crystals. TRIM does not simulate crystalline structure. It uses random sampling of surrounding atoms to choose a collision partner at each step, which essentially means it simulates amorphous materials. For the purpose of simulating crystals, TRIM may not be very useful, but we have still studied the program in details and amended and implemented our own version of it in order to simulate crystal cells. The program uses binary collision approximation (BCA) methods, which will be discussed in detail in chapter 2.

1.6.3 Crystal-TRIM

While researching primary literature to know what other tools are currently being used in ion beam physics simulation, we came across the name Crystal-TRIM. This is a version of TRIM that deals with crystalline structures. The program is written in Fortran and, to our disappointment, had very limited options. The program takes account of the crystal structure of diamond and Silicon only, whereas we were more interested in elements that exhibit microscopic magnetic properties, such as Iron, Niobium or Copper. Thus Crystal-TRIM is not suitable for our investigation.

1.6.4 MARLOWE

MARLOWE is one of the very first computer simulation programs that dealt with ion beam physics. Its origin dates back to 1974. The current version of the program implements methods to simulate crystals using a modified form of binary collision approximation. Although it is said to use binary collision approximation, it actually takes account of multiple collisions at the same time step in order to increase the accuracy of the scattering and energy loss process. Thus this program is one of our primary investigation tools to investigate the spatial distribution of stopped muons. The details of its algorithms are described in chapter 2.

1.6.5 MUSCLE

We essentially need a program where we can change parameters of the simulation flexibly, and put in our own algorithms to test out our ideas. The programs described so far uses BCA algorithms, but recently the molecular dynamics (MD) model is also being considered in the ion beam physics community [Nordlund, 2008]. In fact, for our purpose, we think that it is a very good idea to try the MD model as it takes account of scattering contribution from all the surrounding atoms. Thus it is actually more accurate compared to the BCA model. To address all our needs, and also to test our own ideas, we have developed MUSCLE. MUonS Cascade at Low Energy (MUSCLE) is a collection of programs that we have written in bits and pieces over a year to simulate low energy muon passage in crystals. MUSCLE has implementation of both binary collision approximation and molecular dynamics algorithms, and in this project we compare the results from both algorithms and find out which one is more accurate and efficient. With the inclusion of MD algorithms, we demonstrate the channeling effect and discuss whether it contributes significantly to the determination of spatial distribution. Comparison of MUSCLE with MARLOWE and TRIM is also provided.

Chapter 2 describes the BCA method in more detail, and provides explanation of all the algorithms we have used in our investigation. Chapter 3 demonstrates the MD model, and our own version of the same model that reduces computation time and memory storage. Qualitative comparison between the models and our own conclusion is drawn in chapter 4.

2

2 Binary Collision Approximation

2.1 Introduction

Binary Collision Approximation methods are used in many simulation programs that treat the movement of the projectile particle in a solid as a series of consecutive binary collisions. In this chapter, we use the terms particle and projectile interchangeably to denote the moving atom. The primary idea is that the particles come into the solid and scatter from several atoms, which are assumed to be stationary. During each binary collision, the stationary atom recoils and thus absorbs energy. The projectile is deflected in the process and after enough collisions it comes below a threshold energy, when it is assumed that it is now at rest. In terms of classical mechanical treatment, during a binary collision the transfer of energy between the moving and stationary atoms depends on the speed and direction of the incoming atom, and the mass and charge of both the atoms. Using conservation of energy and momentum, the final velocities and equations for trajectories can be obtained. In the literature, it is known as the asymptotic orbit problem [Zeigler, 14]. Analytical solutions can be obtained for screened potentials between the particles.

2.2 Essentials of Two Particle Scattering

This section will briefly describe the essential mathematics and physics behind the binary scattering process. At first we look at the general process of elastic scattering between two atoms. We extend this process to consider the problem of two-body scattering due to a central force between them. Then we provide a brief description of how interatomic potentials for these calculations are found.

The section ends with a very effective and widely used formula that captures the scattering process very efficiently, and evaluates the scattering angle and energy transferred analytically. This approach is appropriate for the purpose of simulations.

2.2.1 Classical Two Particle Scattering

We are dealing with a low energy regime here, so we stick to non-relativistic calculations. Figure 2.1 and 2.2 show the two coordinate systems we will be using frequently from now. In the laboratory coordinate system, a projectile of mass M1 comes in, gets deflected from the atom of mass M2 making an angle of ϑ with the axis of incidence, and consequently M2 recoils with a velocity v_2 and an angle ϕ. The parameter p is defined as the impact parameter, and represents the perpendicular distance from the initial position of the target atom to the initial axis of incidence of the projectile atom.

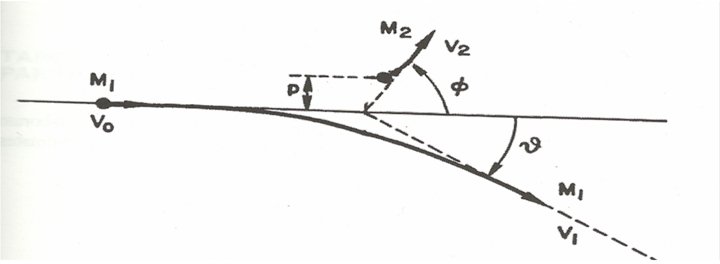


Figure 2.1 Scattering in the laboratory coordinate system. [Zeigler, 15]

For non-relativistic elastic collisions, using conservation of energy, we have the following relation for the initial kinetic energy E_0:

$$E_0 = \frac{1}{2}M_1v_0^2 = \frac{1}{2}M_1v_1^2 + \frac{1}{2}M_2v_2^2. \tag{2.1}$$

Using the conservation of momentum principle, we get two relations.

$$Longitudinal: \; M_1v_0 = M_1v_1cos\vartheta + M_2v_2cos\phi, \tag{2.2}$$

and

$$Lateral: \; 0 = M_1v_1sin\,\vartheta + M_1v_1sin\,\vartheta. \tag{2.3}$$

The problem, if reformulated in center of mass coordinate system, becomes simplified in several ways. The force function between the two atoms may become very complex, but if there is no transverse component (the force acts only on the line joining the two particles), the relative motion of the two atoms can be modeled as a single particle moving under the influence of a central potential (later we call it the interatomic potential). Thus there is an advantage of describing the problem using CM coordinate system when we describe the interaction of the two particles using a force field $V(r)$ that only depends on the interatomic separation r. The motion of both particles in the CM system is described using only one equation of motion for a particle that moves in a central force field $V(r)$. r is an independent variable in this equation. Figure 2.2 shows the scattering process in the CM system.

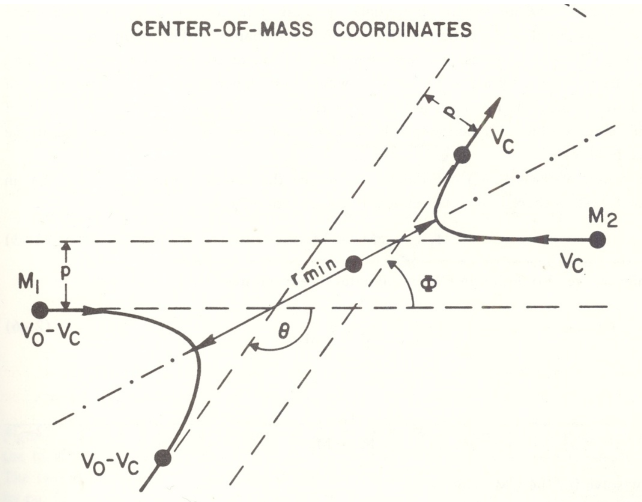


Figure 2.2 Scattering in Center of Mass coordinate system. [Zeigler, 15]

The CM system velocity is defined as v_c. v_c has to be defined in such a way that there is zero net momentum.

$$M_1 v_0 = (M_1 + M_2) v_c. \tag{2.4}$$

A reduced mass, M_c, is introduced in the CM system that simplifies the calculation.

$$\frac{1}{M_c} = \frac{1}{M_1} + \frac{1}{M_2}, \tag{2.5}$$

i.e.

$$M_c = \frac{M_1 M_2}{M_1 + M_2}. \tag{2.6}$$

Thus the velocity v_c is given by

$$v_c = \frac{v_0 M_c}{M_2}. \tag{2.7}$$

The velocities of the target and projectile atoms in terms of v_c are now given as

$$v_{projectile} = v_0 - v_c = \frac{v_0 M_c}{M_1}, \tag{2.8}$$

$$v_{target} = v_c = \frac{v_0 M_c}{M_2}. \tag{2.9}$$

For the purpose of simulation, we need to be able to convert the quantities from the CM system to the laboratory system when we want to calculate the loss of energy due to recoil. The target atom's recoil velocity is easy to convert because its initial velocity in laboratory frame is zero. We keep the total momentum of the system zero, so the velocity vector v_2 in laboratory system is related to the CM velocity vector $\vec{v_c}$ by the translation vector between the two systems, $\vec{v_c}$. This gives us an isosceles triangle and the angle of scatter in the CM frame is related to that of the the laboratory frame by

$$\Phi = 2\phi. \tag{2.10}$$

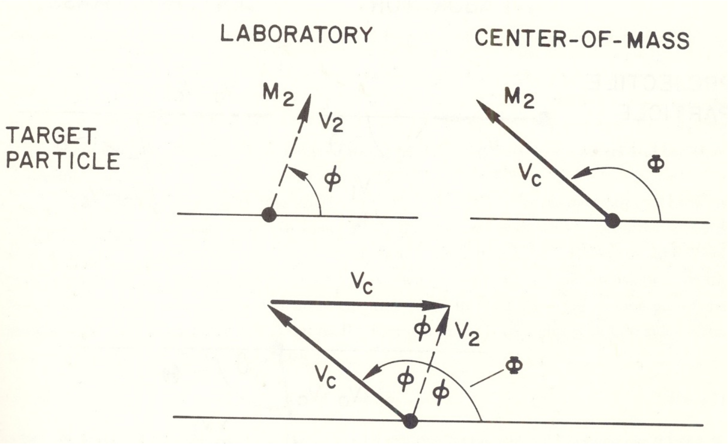


Figure 2.3 Conversion of the target recoil angle from the CM to the laboratory frame. [Zeigler, 17]

Using this relation, we apply the law of cosines to find the velocity v_2 in the laboratory coordinates.

$$v_2^2 = v_c^2 + v_c^2 - v_c^2\, cos\,(\pi - \Phi) = 2v_c^2\,(1 - cos\,\Theta). \tag{2.11}$$

Next, we simplify the expression and relate it to the laboratory angle of recoil by using $v_c = v_0 M_c/M_2$, and $\Phi = 2\phi$,

$$v_2 = 2v_0 \frac{M_c}{M_2}\, cos\,\phi, \tag{2.12}$$

thus relating the final recoil velocity to the angle of recoil in laboratory frame. The energy transferred, T, is simply the energy due to this recoil velocity v_2.

$$\begin{aligned} T &= \frac{1}{2} M_2 v_2^2 \\ &= \frac{1}{2} M_2 \left(\frac{2v_0 M_c cos\,\phi}{M_2}\right)^2 \\ &= \frac{2}{M_2}(v_0 M_c cos\,\phi)^2 \end{aligned}$$

It is important to be able to relate this quantity to the angle of scatter in laboratory frame by using the equation 2.10, giving us:

$$T = \frac{2}{M_2}\left(v_0 M_c\, sin\,\frac{\Theta}{2}\right)^2 = \frac{4E_c M_c}{M_2}\, sin^2\frac{\Theta}{2} = \frac{4E_0 M_1 M_2}{(M_1 + M_2)^2}\, sin^2\frac{\Theta}{2}. \tag{2.13}$$

We now have a basic treatment of two body elastic scattering process, along with expressions that give us the loss of energy involved. Another important quantity of interest is the scattering angle of the projectile. We need to find a relation that connects it with the CM angle of scatter. Figure 2.4 shows a conversion scheme.

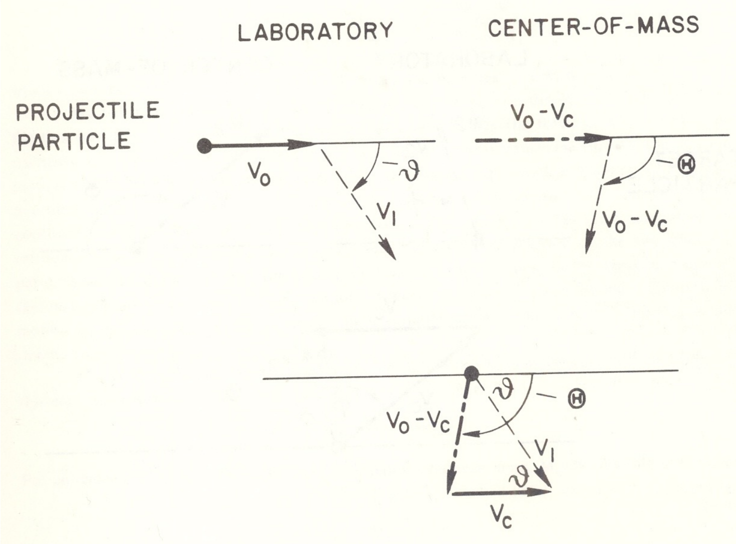


Figure 2.4 Conversion of the projectile scattering angle from the CM to the laboratory frame. [Zeigler, 18]

This time, the situation is complicated by the initial velocity v_0 of the particle. The angle of scatter in the laboratory frame is given by

$$tan\,\vartheta = \frac{(v_0 - v_c)\,sin\,\Theta}{v_c + (v_0 - v_c)\,cos\,\Theta}. \tag{2.14}$$

Now, we can use the conservation of momentum in the CM system to say that

$$(v_0 - v_c)/v_c = M_2/M_1. \tag{2.15}$$

Thus the angle relationship is now given as

$$tan\,\vartheta = \frac{(M_2/M_1)\,sin\,\Theta}{1 + (M_2/M_1)\,cos\,\Theta}, \tag{2.16}$$

or:

$$tan\,\vartheta = \frac{M_2\,sin\,\Theta}{M_1 + M_2\,cos\,\Theta}. \tag{2.17}$$

We now have figured out the basic physics of the elastic scattering between an initially moving particle and a stationary target. A more rigorous treatment is presented in the next section that deals with the interatomic potential between the two charged particles, and modification of the ideas presented in this section to build a basic understanding of binary collision approximation between charged bodies.

2.2.2 Two Body Central Force Scattering

The discussion so far is valid for all collisions that maintain the laws of conservation of energy and momentum. There are some inelastic energy loss due to electronic stopping, which will be discussed later. Let us look closer at the physics of two body central force scattering, which arises from the use of CM coordinates that essentially reduces the problem of two body scattering to that of a single body motion under the influence of a static potential field $V(r)$. The single body has a mass of M_c and possess a velocity v_c. The potential $V(r)$ is centered at the origin of the CM coordinates. This scheme works because of the underlying symmetry of the scattering process. In the CM system, the total linear momentum of the particles is always zero, and since the paths of both the particles are symmetric before and after scattering, the calculation for one particle's trajectory gives the trajectory of the other. After we find the scattering angles in CM frame, we can change them back to that of the laboratory frame using the equations (2.10) and (2.17).

In order to derive a trajectory equation for a particle, we resort to the use of polar coordinates as it makes the math much easier. There are only two particles to consider, and we assumed that there are no transverse forces involved in this interaction. So the scenario is essentially two dimensional in a plane defined by the target's initial position and the initial velocity vector of the projectile. Let us define the azimuthal polar coordinate Θ and radial coordinate r for the vector connecting the projectile and the target atom. Then the time differentials are given by $\dot{r} = dr/dt$ and $\dot{\Theta} = d\Theta/dt$. The CM energy of the system is given by

$$E_c = \frac{1}{2} M_c \, v_0^2, \tag{2.18}$$

so from conservation of energy of the system, we have the following:

$$E_c = \frac{1}{2} M_c \, (\dot{r}^2 + r^2 \dot{\Theta}^2) + V(r). \tag{2.19}$$

Using the conservation of angular momentum, we can also state the following for the polar coordinate system:

$$J_c = M_c\, r^2\, \dot{\Theta} \tag{2.20}$$

where J_c is the constant of angular momentum. It is worthwhile to also know this relation in the general coordinate system:

$$J_c = M_c\, v_0\, p, \tag{2.21}$$

where p is the impact parameter. In order to determine the radial equation of motion, we substitute equation (2.21) into equation (2.20) and solve for r^2.

$$r^2 = \frac{v_0\, p}{\dot{\Theta}}. \tag{2.22}$$

Putting this back to equation (2.19), and solving for $\dot{r}$,

$$\dot{r}^2 = \frac{2\,(E_c - V(r))}{M_c} - v_0\, p\, \dot{\Theta}. \tag{2.23}$$

We now have a relation leading to the radial equation of motion. It can be simplified using $M_c = 2\,E_c/v_0^2$, and

$$\dot{\Theta} = v_0\, p/r^2 \tag{2.24}$$

(by combining (2.21) and (2.20)), yielding

$$\dot{r}^2 = v_0^2 - \frac{V(r)}{E_c}\, v_0^2 - \frac{p^2}{r^2}. \tag{2.25}$$

i.e. the radial equation of motion is

$$\dot{r} = \frac{dr}{dt} = v_0 \left(1 - \frac{V(r)}{E_c} - \left(\frac{p}{r}\right)^2\right)^{1/2}. \tag{2.26}$$

Now combining the equations for $\dot{r}$ and $\dot{\Theta}$, we can solve for $d\Theta/dr$, as this will yield the scattering angle later.

$$\frac{d\Theta}{dr} = \frac{d\Theta}{dt}\frac{dt}{dr} = \frac{p}{r^2\left(1 - \frac{V(r)}{E_c} - \frac{p^2}{r^2}\right)^{1/2}}. \tag{2.27}$$

In order to find the scattering angle, we integrate the above relation over the entire collision path

$$\Theta = \pi - \int_{-\infty}^{\infty} \frac{p\,dr}{r^2\left(1 - \frac{V(r)}{E_c} - \frac{p^2}{r^2}\right)^{1/2}}. \tag{2.28}$$

The initial value of Θ is π, that is why the integral is subtracted from the initial value. The limits of the integral can be changed by taking account of the fact that there is a closest distance of approach between the particles, which is defined as r_{min}, and the path of the particle is symmetric (hence, we can simply integrate one portion and put in a factor of 2 in front of the integral). The integral is now

$$\Theta = \pi - 2\int_{r_{min}}^{\infty} \frac{p\,dr}{r^2\left(1 - \frac{V(r)}{E_c} - \frac{p^2}{r^2}\right)^{1/2}}. \tag{2.29}$$

This scattering angle can be used to evaluate the energy transferred from the projectile to the target by using equation (2.13). The above equation is known as the general orbit equation for two-body central force scattering, and also as the classical scattering integral. In order to apply this, we should make sure that the central force potential is not dependent on time or the motion of the particle, i.e. the potential must be spherically symmetric.

2.2.3 Interatomic Potentials

An accurate potential function is essential in the calculation of scattering angle and energy loss. In order to calculate the potential between two atoms, extensive studies have been carried out in the last 60 years or so, and a comprehensive list of such literature is provided by Zeigler [Zeigler, 1985]. The theory itself is detailed and it would require much larger space to explain all the developments. We summarize the essentials of the theory in this section.

Much of the theory relies on experimental results and statistical models. Some widely used potential functions are the Thomas-Fermi potential, the Moliere approximation and the Bohr potential. All these potentials are given in a Coulombic $1/r$ form multiplied by a screening function.

The actual Coulombic term value due to the positive nucleus is reduced by the screening due to the surrounding electron cloud, and the so called screening function ϕ attempts to capture this scenario. It is defined as the ratio of the actual atomic potential at radius r to the potential due to an unscreened nucleus.

$$\phi = \frac{V(r)}{Ze/r} \tag{2.30}$$

where $V(r)$ is the potential at the radius r, Z is the atomic number and e is the electronic charge. From the experimental data, it is much easier to find the screening function ϕ, and then derive the actual interatomic potential from it. There are other methods to calculate the interatomic potential too, but we stick to describing the method that relies on experimental data.

The general form for the total interaction potential is

$$V = V_{nn} + V_{en} + V_{ee} + V_k + V_a. \tag{2.31}$$

V_{nn} is the electrostatic potential energy between the projectile and target nuclei, V_{ee} is the pure electrostatic interaction energy between the electron distribution of the two atoms, V_{en} is the interaction energy between each nucleus and the other atom's electron distribution, V_k is the increase in kinetic energy due to Pauli excitation of the electrons because of overlapping of regions, and V_a is the net increase in exchange energy of electrons. Each term is evaluated based on specific theories and a full model for V is derived.

However, a screening function is generally used to express the potential. The interatomic screening function definition is given as

$$\phi_I = \frac{V(r)}{(Z_1 Z_2 e^2/r)}. \tag{2.32}$$

The general approach to express an interatomic screening function requires the use of a *reduced radius* R, which is the atomic radius divided by the screening length.

$$R = \frac{r}{a_U}, \tag{2.33}$$

where a_U, the universal screening length, is defined as

$$a_U = \frac{.8854\, a_0}{(Z_1^{.23} + Z_2^{.23})} \tag{2.34}$$

which is derived by fitting the primitive form of screening length expression to experimental data. With the reduced radius and the screening function ϕ, the potential can be written as

$$V(R) = \frac{Z_1\, Z_2\, e^2}{a\, R}\, \phi_I(R). \tag{2.35}$$

This is the form in which potential functions are used in simulations. As said before, the screening function is determined by fitting a guessed form an expression with experimental data. The universal screening function is given as

$$\phi_U = .1818e^{-3.2x} + .5099e^{-.9423x} + .2802e^{-.4028x} + .02817e^{-.2016x} \tag{2.36}$$

where x is the reduced radius. The word universal (used for screening length and screening function) does not actually mean that it is accurate and true for all atom pairs. In fact, these formulae are derived by selecting a large number of random pairs of atoms and adjusting the formula by means of a least squares fit with experimental data for all these pairs. This approximation is actually pretty accurate and works very well [Zeigler, 41 - 44, 48].

2.2.4 Magic Scattering Formula

For a Monte Carlo simulation, it is impractical to evaluate the scattering integral for all the collisions a projectile undergoes with selected atoms in the sample. Depending on the length of each step a projectile takes, there can as well be hundreds of collisions depending on the initial energy of the projectile, and the thickness and structure of the sample. Thus for the simulation purpose, another method of approximation was proposed by Biersack [Zeigler, 110], usually known as the Magic scattering formula.

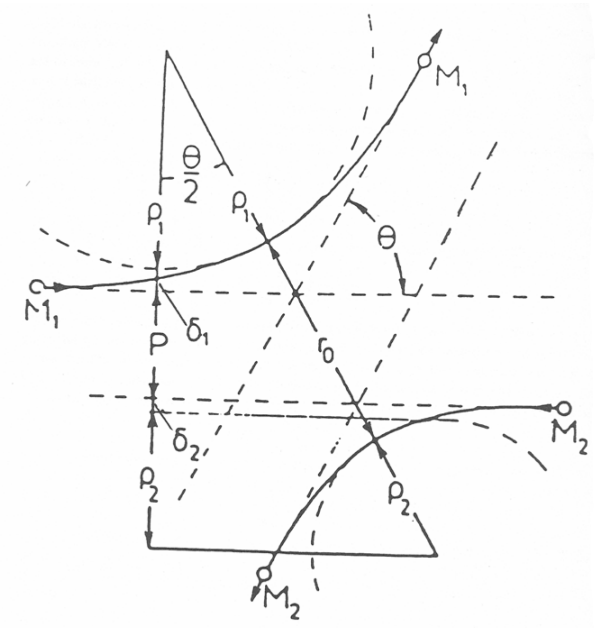


Figure 2.5. Scattering triangle depicting the scattering process in CM system. [Zeigler, 112]

For an analytical evaluation of the scattering angle, we formulate the problem according to the above figure, which depicts the scattering in a center-of-mass coordinate system. A projectile of mass M_1 and energy E scatters from a mass M_2 which is initially stationary. The angle of scattering is given by θ. A so called "scattering triangle" is constructed in the diagram which has some known parameters as its sides. These parameters are the impact parameter p, distance of closest approach r_0, radii of curvature of the trajectories at the closest approach defined as ρ_1 and ρ_2, and the terms δ_1 and δ_2 known as the correction terms that compensate for the deficiency of the lengths of the scattering triangle composed of the other parameters. We can find the angle θ from the following relation

$$cos\,\frac{\theta}{2} = \frac{\rho + p + \delta}{\rho + r_0}. \tag{2.37}$$

ρ is defined as the summation of ρ_1 and ρ_2, and $\delta = \delta_1 + \delta_1$. In order to obtain r_0, we set the radial equation of motion dr/dt to zero to find the minimum value of r. So r_0 is obtained by solving the equation

$$1 - \frac{V(r_0)}{E_c} - \left(\frac{p}{r_0}\right)^2 = 0, \tag{2.38}$$

where E_c is the energy of the projectile in CM system, and $V(r)$ is the interaction potential between the projectile and the target atoms. The above equation can be solved by using Newton's method if we reformulate the equation in the following manner. Let $f(r)$ be the right hand side of the equation.

$$f(r) = 1 - \frac{V(r)}{E_c} - \left(\frac{p}{r}\right)^2 = 0. \tag{2.39}$$

The derivative is given by

$$f'(r) = -\frac{V'(r)}{E_c} + 2\left(\frac{p^2}{r^3}\right) \tag{2.40}$$

Also, an approximation for the derivative $f'(r)$ is given by

$$f'(r) = \frac{f(r) - f(r_0)}{r - r_0}. \tag{2.41}$$

We know that $f(r_0)$ has a value of zero. Hence,

$$f'(r) = -\frac{f(r)}{r - r_0}. \tag{2.42}$$

Thus r_0 can be obtained from

$$r_0 = r - \frac{f(r)}{f'(r)}, \tag{2.43}$$

which is essentially the form for iterative solution using Newton's method. A few steps with an initial guess of r can yield a good value for the distance of closest approach.

The radius of curvature ρ is obtained from the following relation based on the fundamental rule for centripetal force f_c.

$$\rho = \rho_1 + \rho_2 = (M_1 v_1^2 + M_2 v_2^2)/f_c \tag{2.44}$$

Using $2(E_c - V(r_0))$ to represent the numerator which is double the kinetic energy, where E_c is the energy of CM system, we can say that

$$\rho = 2\frac{(E_c - V(r_0))}{-V'(r_0)} \tag{2.45}$$

with $-V'(r_0)$ representing the force. It is convenient to introduce a dimensionless energy ε at this point, which is basically the CM energy E_c expressed in the units of $Z_1\, Z_2\, e^2/a$.

$$\varepsilon = \frac{a\, E_c}{Z_1\, Z_2\, e^2}, \tag{2.46}$$

where Z_1 and Z_2 are atomic numbers of the projectile and target atoms, e is the electronic charge and a is the screening length.

In order to determine the correction term δ, it is a wide accepted practice to change the cosine formula for the scattering angle in the following manner. The parameters of scattering angle are expressed in units of the screening length a. The universal screening length is used

$$a = \frac{0.8853\, a_0}{\left(Z_1^{0.23} + Z_2^{0.23}\right)}, \tag{2.47}$$

where $a_0 = 0.529\,\text{Å}$ is the Bohr radius. Thus the parameters are now given as

$$B = p/a,\ R_0 = r_0/a,\ R_c = \rho/a,\ and\ \Delta = \delta/a. \tag{2.48}$$

Now the cosine relation is

$$cos\ \frac{\theta}{2} = \frac{B + R_c + \Delta}{R_0 + R_c}. \tag{2.49}$$

The parameter Δ is now to be determined. The authors of this method determined a formula by fitting it to precalculated scattering results, which is

$$\Delta = A\frac{R_0 - B}{1 + G} \tag{2.50}$$

where

$$A = 2\,\alpha\,\varepsilon\, B^{\beta},\ and\ G = \gamma\left((1 + A^2)^{1/2} - A\right)^{-1}. \tag{2.51}$$

Here,

$$\alpha = 1 + C_1 \varepsilon^{-1/2}, \tag{2.52}$$

$$\beta = \frac{C_2 + \varepsilon^{1/2}}{C_3 + \varepsilon^{1/2}}, \tag{2.53}$$

$$\gamma = \frac{C_4 + \varepsilon}{C_5 + \varepsilon}, \tag{2.54}$$

and C_1, ..., C_5 are fitting parameters which are statistically determined for the potential of interest. The solutions of classical scattering integral is calculated for a range of ε and B values using the desired potential function, and the parameters $C_1 - C_5$ are determined from a least squares fitting procedure. There is no particular derivation of the equation (2.50), although the term $R_0 - B$ was shown to give best fits and valid results. The essence of this formula comes from the fact that as ε becomes quite large, the quantities α, β, *and* γ approach unity. Thus at larger energy range, equation (2.50) produces the Rutherford scattering formula, which is valid at the high energy limit.

The universal interatomic potential is used in all the calculations.

$$V(R) = \frac{Z_1 \, Z_2 \, e^2}{a \, R} \, \phi(R), \tag{2.55}$$

where R is interatomic separation expressed in units of screening length a, $R = r/a$, and $\phi(R)$ is the universal screening function discussed before.

The analytic expression for the scattering angle essentially yields quite accurate values according to some studies [Zeigler, 114]. Thus this formula is widely used in Monte Carlo programs to save computation time and resources.

2.2.5 Validity of Classical Mechanical Treatment of BCA

We use classical equations of motion in all the BCA calculations. This is valid when quantum mechanical effects are negligible. A lower limit for the energy of the projectile is immediately evident from the fact that the wavelength of the moving atom must be smaller than the lattice dimensions. The wavelength λ of an atom with mass M, velocity v, and kinetic energy E is given by

$$\lambda[nm] = \frac{\hbar}{M\,v} = \frac{2.87 \times 10^{-2}}{\sqrt{M[amu]E[eV]}}. \tag{2.56}$$

For a muon, $M = 0.114$, and assuming that it goes through Iron, the lattice dimension of the crystal is 2.87 Å (0.287 nm). If $\lambda = 0.287\,nm$, then $E = 0.0877\,eV$. Thus for most of our purposes, we are safe. In many simulation, a projectile with energy below 2 - 3 eV is considered to be at rest. In that respect, it is ok to say that BCA calculations are perfectly valid for our purpose.

2.3 Inelastic Energy Loss

Atoms/particles going through a solid lose energy due to two types of interaction with electrons. This is the basis of inelastic or electronic energy loss. The first type of interaction is excitation or ionization in both the colliding atoms. Since it happens in the electronic shells of atoms, it is called local energy loss. The other type of energy loss is due to the electron gas in the solid (metal) which acts as a friction force to the projectile motion. It is known as continuous energy loss as the projectile loses energy to the electron cloud throughout its motion. The theory for local energy loss involves quantum mechanics, and the formulae are once again validated with experimental data by means of curve fitting. For our purpose, we do not need to worry about energy loss due to electronic shell interactions as the muon particle does not have a conventional atomic shell structure. Thus, even if it somehow manages to excite or ionize an atom, such events will be very rare and our code does not need to take account of local energy loss. Besides, ionization or excitation could occur for very high energy muons, and here we are dealing with low energy ones. Continuous electronic energy loss, on the other hand, is a very important factor in our simulation, as it is responsible for a major amount of energy loss of low energy muons.

2.3.1 Continuous Electronic Energy Loss

We consider two schemes of determining the continuous electronic energy loss. The first one is by Lindhard and Scharff [Eckstein, 66] and the second one is by James Zeigler. The continuous energy loss schemes are energy dependent; the amount of energy loss depends on the kinetic energy of the projectile.

Using the dielectric response of a solid, *Lindhard* and *Scharff* presented a formula for inelastic stopping cross section (given in $eV\text{Å}^{-1}$) that they derived on the basis of modeling the electron gas as a viscous medium:

$$S_{LS}(E) = 8\pi\sqrt{2}a_B\hbar\,\frac{Z_1^{7/6}Z_2}{(Z_1^{2/3}+Z_2^{2/3})^{3/2}}\sqrt{\frac{E_1}{M_1}}, \tag{2.57}$$

$$= K\sqrt{E} = 1.21\frac{Z_1^{7/6}Z_2}{(Z_1^{2/3}+Z_2^{2/3})^{3/2}}\sqrt{\frac{E_1}{M_1}}. \tag{2.58}$$

Here, E is given in eV/amu. Hence, $E = \frac{E_1}{M_1}$ is used. The constant K is adopted in such a way that it fits with experimental data. Z_1 and Z_2 are the atomic numbers of the projectile and the target atom, respectively. M_1 represents the mass of the projectile in atomic mass unit.

However, a better method is to rely on experimental data. A more comprehensive treatment over a wide variety of experimental proton stopping data is done by *Zeigler* et al. [Eckstein, 70]. The stopping for other atoms are usually found by means of careful extrapolation of proton stopping data. For our job, we stick to the expression found for proton stopping.

$$S_{low} = a_1\left(\frac{E_1}{M_1}\right)^{a_2} + a_3\left(\frac{E_1}{M_1}\right)^{a_4}, \qquad \frac{E_1}{M_1} < 25\,keV/amu \tag{2.59}$$

$$S_{high} = a_5\,\frac{ln\left(a_7\frac{M_1}{E_1}+a_8\frac{E_1}{M_1}\right)}{\left(\frac{M_1}{E_1}\right)^{a_6}} \qquad \frac{E_1}{M_1} \gg 25\,keV/amu. \tag{2.60}$$

The constants $a_1 - a_8$ are called the proton stopping coefficients which are found by curve fitting with experimental stopping data. Once the stopping at low and high energies are calculated, the average stopping $S_e(E)$ for a particular energy value is given by

$$\frac{1}{S_e(E)} = \frac{1}{S_{low}} + \frac{1}{S_{high}}. \tag{2.61}$$

This is the form we will be using for our simulation. The authors have also suggested a velocity proportional stopping at low energy regime. For $\frac{E_1}{M_1} < 25\,keV/amu$, the stopping becomes

$$S_e(E) \sim v_1^{0.75}. \tag{2.62}$$

This adjustment is required for electronic stopping to agree well with experimental data [Zeigler, 218].

2.4 Monte Carlo Simulation for Amorphous Samples

TRansport of Ions in Matter (TRIM) [Zeigler, 1985] is a standard Fortran program that is used to predict the slowing down and spatial distribution of ions in an amorphous sample. The latest version of TRIM is available for download from the author's (James F. Zeigler) website. The program has received a face lift over the years and has been transformed from a command window program to a nice Windows Graphical User Interface. The current version is called SRIM. We downloaded the program and modified its parameter files so that it recognizes muons as ions. We also found the ion mass parameter in the data file and modified its range so that the program allowed a lower mass limit, as the lowest mass that could be entered was the mass of proton.

The Monte Carlo algorithm is based on the physics of scattering and energy loss described above in the previous sections. The program follows the two dimensional trajectory of the projectile, i.e. the information about an axis is omitted. A few details of the implementation of this program will be described in this section, along with some examples showing the kind of results it produces. The program does not come with a very good documentation (although general explanations are given, many times the authors have not made it clear why they were using some certain formulas), and the following is our own interpretation of the original design of the authors. We will also add our own analysis to justify the usage of some formulas and numerical computation code in the program.

2.4.1 TRIM techniques

The structure of the program can be divided into four phases.

- Initial calculation of the properties of the projectile and the target material,
- electronic stopping calculation for the target material,
- Monte Carlo loop that simulates the transport and scattering of the projectile,
- and finally, calculation of quantities that provide statistical inference about ion beam implantation.

In our discussion, we will mainly focus on how the electronic stopping calculation and the Monte Carlo loop is implemented, which are the main essence of the program.

The stopping coefficients data is loaded from a text file that lists experimentally determined coefficient values for all the ninety two elements. The program allows incorporating up to three layers of different materials as the target. Hence, all the properties of the elements in the target, such as density, atomic number, atomic mass etc, are retrieved from the data file. The parameters for the calculation of the scattering angle, e.g. the reduced mass M_c, screening length a, initial CM energy E_c etc are calculated in the initial phase of the simulation.

In order to incorporate electronic stopping cross sections, the authors use a list of 1000 stopping values that are calculated before the main Monte Carlo loop. Our guess is that they wanted to make the computation faster during the Monte Carlo phase by taking this approach. The electronic stopping calculation proposed by Zeigler (section 2.3.1) is used in the program. So, instead of calculating $S_e(E)$ for the current energy E of the projectile that requires calling the electronic stopping method in every step of the simulation, the authors decided to precompute stopping for 1000 energy values, ranging from the initial maximum energy to zero, in equal steps. For a specific energy, the corresponding element in the stopping list is selected by rounding off the energy variable to decide which bin its integer value belongs.

The Monte Carlo loop manages the life cycle of a particle moving in the material. Based on a fixed length step, the particle travels a certain amount of distance in every execution of the loop. An atom is selected in every step that will act as the target atom from which the particle will scatter. In a multi-atomic material, an atom is chosen randomly from the set of available atoms, and the randomization scheme is weighted according to the proportion of the elements present. The energy loss due to scattering is calculated using the formula we previously derived

$$T_{ns} = 4M_1M_2(M_1+M_2)^{-2}E\ sin^2(\theta/2). \tag{2.63}$$

Here, E is current energy of the particle. The scattering angle is in the laboratory system is given as

$$\psi = arctan\left(\frac{sin\ \theta}{cos\ \theta + \frac{M_1}{M_2}}\right) \tag{2.64}$$

This angle lies on the plane defined by the scattering process. The azimuthal scattering angle is selected at random, using

$$\phi = 2\,\pi\,R_n, \tag{2.65}$$

where R_n is a random number with a value between 0 and 1.

In the program, the displacements are calculated with reference to a fixed axis, usually chosen to be the one perpendicular to the target surface. In order to determine how far the particle deviates from this axis, the angle the particle makes with the axis is determined after each collision by

$$cos\;\alpha_i = cos\;\alpha_{i-1}\;cos\;\psi_i + sin\;\alpha_{i-1}\;sin\;\psi_i\;cos\;\alpha_i. \tag{2.66}$$

The directional cosines for other axes in the lateral directions are determined if the programmer wants to follow the trajectory with reference to those directions.

The nuclear energy loss is subtracted from the current energy, along with the energy loss due to electronic stopping, in every step. Determining the length of each step is tricky as it needs to be adjusted for different energy range of the incoming projectiles. We focus our discussion only on the low energy regime. In order to determine the step length, the density of the material is taken into account by assuming that there is one target atom in every cylinder of volume N^{-1}, where N is the atomic density of the target (number of atoms per unit volume). Then the length of step L is given by the relation

$$\pi p_{max}^2\,L = N^{-1}, \tag{2.67}$$

where p_{max} is the maximum impact parameter. The maximum impact parameter is predetermined for a material by using numerical fitting with the parameters T_{min} (minimum transferred energy during a collision, usually around 5 eV), Z_1, $Z_{2,}$ M_1, M_2 and the screening length a, keeping in mind that the particle loses at least T_{min} amount of energy in every step.

The impact parameter is chosen randomly. In other words, the position of the target atom in the scattering plane is determined by a random scheme. A proportion of the maximum impact parameter is assigned as the value of the current impact parameter using the following

$$p = R_n \, p_{max}, \tag{2.68}$$

where R_n is again a random number between 0 and 1.

S_e is calculated in $eV \text{Å}^{-1}$, and the energy loss per step length is given by

$$T_{es} = L \, S_e(E) \tag{2.69}$$

The current energy of the particle is then reduced by the amount lost in nuclear scattering and electronic stopping.

$$E_{i+1} = E_i - T_{ns} - T_{es} \tag{2.70}$$

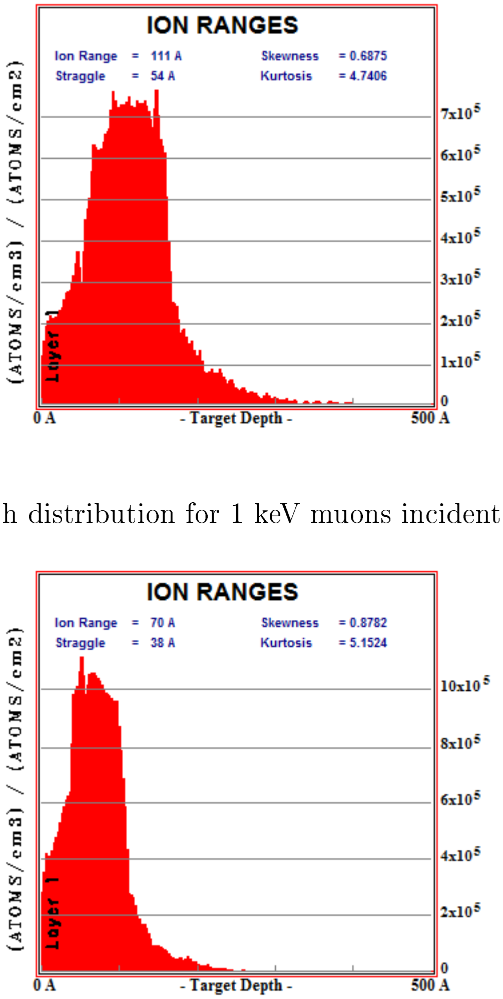
One scattering process occurs in every execution of the Monte Carlo loop. After every scattering, the new positions are calculated with reference to the fixed axis normal to the target surface. The loop continues until the energy of the particle comes below the threshold energy value, which is usually considered to be 5 eV. At this point the final coordinates are saved and a new particle is introduced. At the end of all the particles' journey, statistical calculations regarding the average penetration and lateral spread are carried out, which is not important for our discussion of the Monte Carlo scheme.

2.4.2 Implementation

The original TRIM program was written in Fortran 77, back in 1985. In my summer REU, I rewrote the above algorithm using C++. The data file containing the atomic properties and the stopping coefficients had to be reformatted for making it usable in my program. In order to visualize the trajectory, a preliminary trajectory viewer was also written using Processing, which is a Java wrapper for easy graphics and animation creation. Our concern is mostly about the spatial distribution of the muons. Mathematica was used to create histograms from the data obtained by running the simulation in a Linux machine. The current version of TRIM (available from its authors) was released in 2009, and comes with a nice GUI and a versatile configuration window where the necessary parameters for the simulation can be easily set.

2.4.3 Sample Results

Simulations were run in the TRIM software for 10000 muons going into an Iron sample of thickness 500 angstroms. Initial energy values were set at 500 eV and 1000 eV for two runs.



(a) Depth distribution for 1 keV muons incident on Iron

(b) Depth distribution for 500 eV muons incident on Iron.

Figure 2.6. TRIM output for 1 keV and 500 eV muons.

As expected, 1 keV muons penetrate deeper into the sample compared to the 500 eV ones. The average depth reached by 1 keV muons, with an incident angle of 0, is 111 Å. With the same settings, 500 eV muons reach 70 Å on average. Note that the maximum range reached by 1 keV muons is ~400 Å. For 500 eV muons, the maximum range is ~260 Å.

The C++ version that we developed produces similar distributions. The following image is a snapshot of the visualization program written in Processing that loads trajectory data from the output file produced by the main simulation program.

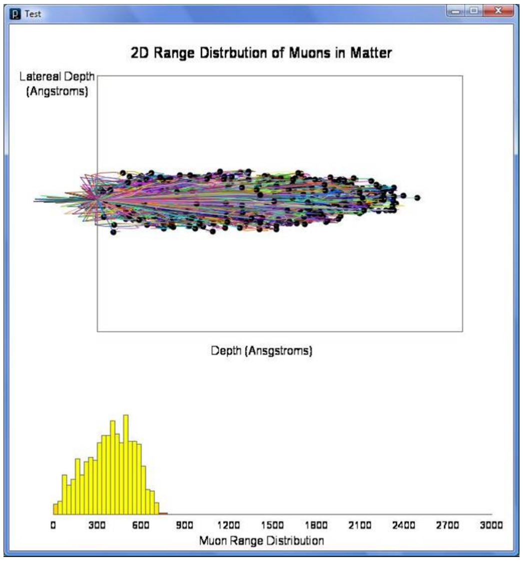


Figure 2.7. Visualization program written using Processing that animates the muon trajectories.

The program also calculates the depth histogram from the stopped muons' coordinates and displays it. The program outputs agrees well with the distributions produced from TRIM. The next task is to modify the same code to take account of crystalline structures to see if the distributions are the same as those produced from TRIM .

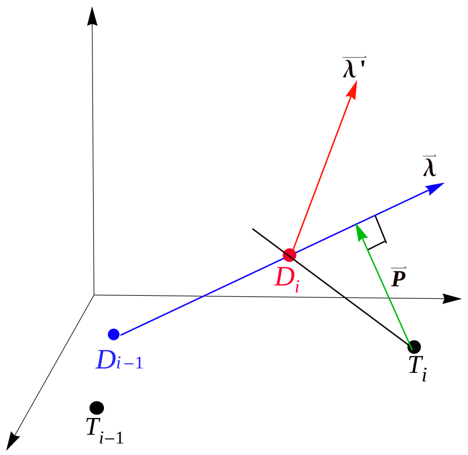
2.5 Monte Carlo Simulation for Crystalline Samples

TRIM takes account of amorphous samples only, whereas in many instances the samples under muSR study are crystalline. Although many muSR physicists [Dubman, 2009] rely on TRIM as the results match well with experimental data, our main goal is to establish a simulation for crystalline samples and investigate the effect of channeling. It is still possible to use the same BCA code to simulate a projectile's passage through crystalline materials, only this time we need to choose the target atom in each step carefully. The length of each step should be associated with the lattice constant of the crystal in some way. In addition, the usual 2-dimensional calculations done in TRIM (where only the geometry associated within the plane of scattering is considered) needs to be altered to take account of target atoms residing in fixed positions in a 3-dimensional crystal space. All these changes are implemented in our code to simulate the transport of muon in a crystal. We choose the body centered cubic crystal structure as an example in all the explanations.

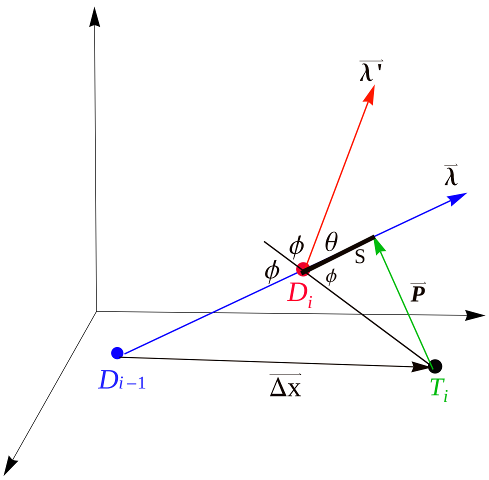
2.5.1 Going from 2D to 3D

In order to take account of the 3-d vector geometry associated with the scattering, we came up with the following calculations. Let us call the target atom T_i, the current direction of motion $\vec{\lambda}$, and the previous point of scattering D_{i-1}. Let us also denote the vector connecting $\vec{D_{i-1}}$ and $\vec{T_i}$ as $\vec{\triangle x}$. The projectile scattered previously from the atom T_{i-1}, and we are to determine the position of $\vec{D_i}$, the point where the next scattering will occur. We also need to determine $\vec{\lambda'}$, the new direction of motion after scattering at $\vec{D_i}$.

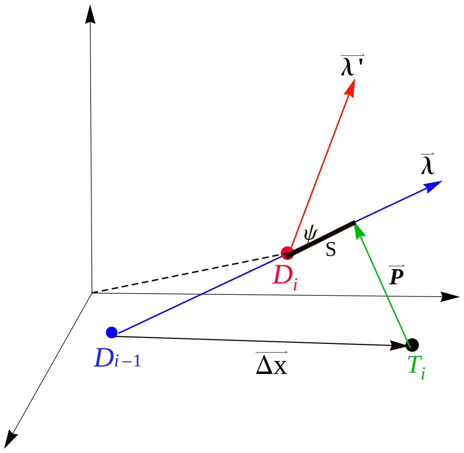
We begin by calculating the scattering angle in the center-of-mass coordinate system using the same formula(s) we used for the amorphous target. Once θ is found, we know that the angle between λ'-D_i-D_{i-1} is $\pi - \theta$ in the CM system. At the time of scattering, when the distance of approach between the projectile and the target is the closest, the symmetry of the problem allows us to safely say that the angle $\pi - \theta$ is bisected by the vector connecting the projectile and the target. If we denote $\phi = (\pi - \theta)/2$, then the angle between $\vec{\lambda}$ and the vector connecting D_i and T_i is also ϕ.



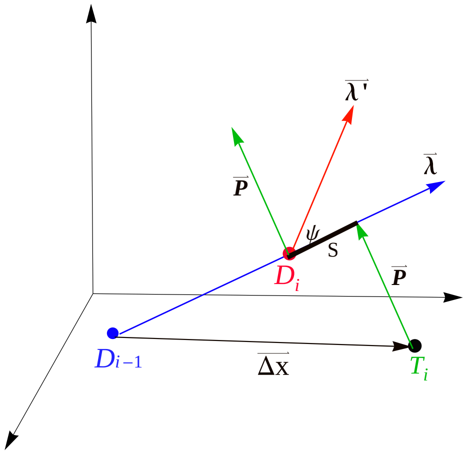
(a) $\vec{\lambda'}$ is the new direction of motion, after a particle scatters from a target atom T_i.



(b) The angle of scatter, θ, is calculated in center-of-mass coordinate system as before.



(c) Convert θ to the lab frame angle ψ, find s and calculate $\vec{D}_i$.



(d) Finding $\hat{\lambda}'$ using $\hat{P}$ and ψ.

Figure 2.8. Calculating $\vec{D_i}$ and $\hat{\lambda}$ for a scattering process where the target atom position is $\vec{T_i}$.

The magnitude of the impact parameter is given by

$$|\vec{P}|^2 = (\vec{\triangle x} \times \vec{\lambda})^2. \tag{2.71}$$

In order to find the vector $\vec{P}$, we use the following

$$\vec{P} = (\vec{\triangle x} \times \vec{\lambda}) \times \vec{\lambda}. \tag{2.72}$$

The value of the scalar distance s is found by

$$s = \frac{|\vec{P}|}{tan\,\phi} \tag{2.73}$$

Having found all these quantities, now we can calculate our desired parameters. The lab frame angle of scattering ψ is found from θ at first. Then the new position of scattering is given by

$$\vec{D_i} = \vec{D_{i-1}} + \vec{\triangle x} + \vec{P} - s\vec{\lambda}. \tag{2.74}$$

In order to find the new direction of motion $\hat{\lambda}'$, we find the unit vectors $\hat{P}$ and $\hat{\lambda}$. Then

$$\hat{\lambda}' = \hat{\lambda}\,cos\,\psi + \hat{P}\,sin\,\psi. \tag{2.75}$$

2.5.2 Modified Algorithm

The above calculations are enough to bring necessary changes to the amorphous TRIM algorithm. However, we have not discussed the most tricky part in our modification yet. As a muon enters a sample and scatters from different atoms, it is hard to determine which neighbor atoms it will scatter from. There is no good method to determine this, although several researchers have tried several techniques with some success (e.g. the program MARLOWE does a good job in this case). In my program, I employ a very simple condition that looks reasonable.

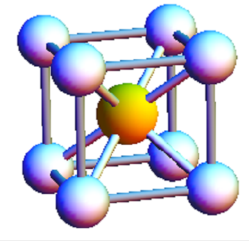


Figure 2.9. A unit cell of a body centered cubic crystal.

Let us assume that we are working with a crystal that has a bcc structure. There are 14 neighbor atoms surrounding the center atom in the unit bcc cell (the corner ones, and the center atoms of the adjacent cells). If we choose an initial target atom (based on proximity) when the muon enters the sample, then the next possible target atom must lie among the surrounding 14 neighbor atoms. We can exploit the symmetry of a bcc cell to deduce that any atom (not at the surface or boundaries) has 14 neighbor atoms. Hence, once the muon scatters from an atom residing in the surface, the next candidate for scattering is chosen from a list of neighbor atoms based on the condition

$$p_i < p_{max}, \quad i = 1,\, 2,\, ..., 14 \tag{2.76}$$

here p_i is the impact parameter of the $i-th$ atom in the neighbor list, and p_{max} is calculated beforehand using the formula that TRIM employs. Whichever atom in the list satisfies this condition at first, is chosen as the target. Thus this method is different from TRIM since the step length was fixed in the latter one. In this method, the step length is the distance between each scattering points, so it changes over time, based on which atom gets chosen from the list as the target. Whenever a candidate is chosen, the neighbor atoms list is updated to reflect the new position of the target atom and its neighbor atoms. The drawback for this method lies in the ordering of the atoms list. There is no good way (in our knowledge) to order the atoms in the list when using this method, and in the worst case this process may end up selecting an atom which is behind the muon (but still a neighbor) and does not contribute much in the muon's trajectory. Nonetheless, we tried this method and got some results which are not very promising.

2.5.3 Results

The following figure shows a range distribution produced from the modified program that simulates 500 eV muons stopping in Iron and Niobium samples.

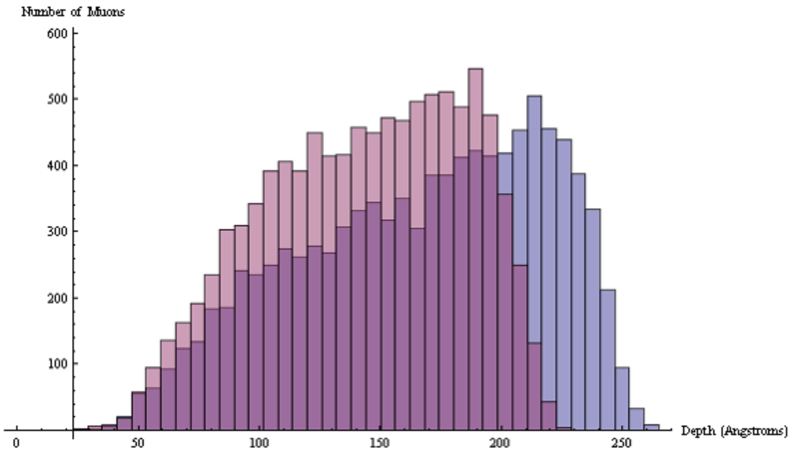


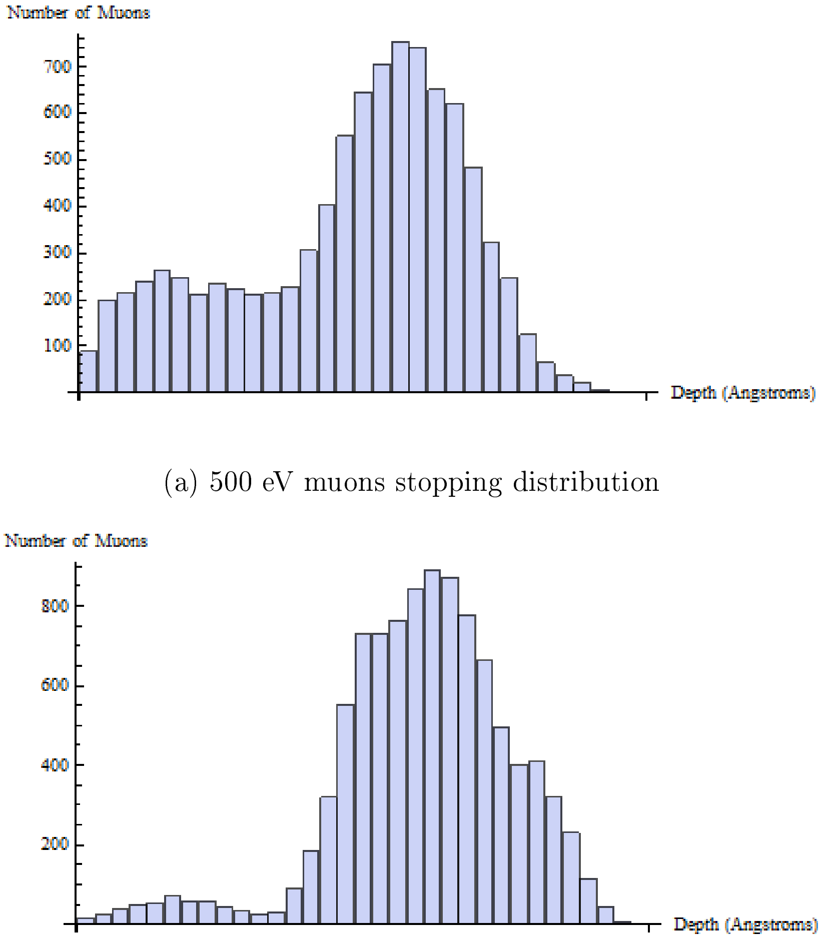
Figure 2.10. 500 eV muons stopping in Iron and Niobium.

As seen here, the ranges are quite high compared to TRIM outputs. It is as if only the electronic stopping was prominent in the stopping. In our investigation with raw simulation output data, we found that only a very negligible amount of nuclear recoil energy was lost during each muon's journey. This may be due to the fact that we could not come up with a rule to select a better candidate for scattering. The condition we use to find a neighbor atom for the next collision can be fulfilled by several atoms, but we could not find a good way to take account of all those atoms in the scattering process. The program MARLOWE takes account of multiple collision partners, so has more accuracy compared to our or any other BCA program.

2.5.4 MARLOWE Simulations

MARLOWE uses the same kind of geometry methods described in the previous section to find the point of scattering and new direction of motion after scattering [Eckstein, 104]. In addition, it finds the surrounding atoms which meet the condition $p < p_{max}$. Then the algorithm finds the momenta of all these atoms after scattering in laboratory frame. Conservation of momentum is then used to find the momentum of the projectile after scattering. Early versions of the program used to calculate the scattering processes individually for each selected neighbor atom, and used vectorial addition to find the final motion of the projectile. The momentum conservation technique is more accurate and gives better results.

Instead of using the magic scattering formula to find the scattering angle, MARLOWE solves the scattering integral using a 4-point Gauss-Mehler procedure [Eckstein, 106]. This comes with the expense of increased computation time, but yields much better and accurate results. The following distributions are produced by the program for 500 eV and 5 keV muons going into Iron.



(a) 500 eV muons stopping distribution

(b) 5 keV muons stopping distribution

Figure 2.11. Muon stopping distribution produced by MARLOWE.

The results show very promising signs of muons channeling in the sample. In the figures, the depth scale is not shown as MARLOWE generates bin information and depth data separately, and we have not figured out a way to merge the two data sets yet. The 500 eV muon distribution has two visible bumps, which is unusual compared to the TRIM distributions. This suggests that one group of muons have slowed down and come to rest at much earlier depth, whereas the other group continued to travel further due to channeling. For 5 keV muons, the first bump is less visible, and most of the muons have ended up in the second bump. This suggests that higher energy muons have traveled further into the sample because of their energy and also because of channeling, i.e. there are fewer muons which stopped at a smaller depth. The average depth of muons here are comparable to what TRIM estimates.

2.6 In Search of a Good Neighbor Selection Algorithm

In the course of the project, we have spent some time thinking about an efficient neighbor selection algorithm. In this section, we present our take on the problem and possible pitfalls in the method. The method employs probability and randomization to capture the very essence of the scattering process in reality. The success and failures of the method remains questionable as we will see from our results. The algorithm can be improved in several ways by taking account of some factors we ignored in order to make the coding process simple.

2.6.1 Basic Principle

The formulation of the problem remains the same. The scenario again has a target atom from which the muon scatters, and a list of potential candidate atoms surrounding the current target atom one of which will be selected as the next target. We consider a cylindrical volume that is enclosed by the muon and a possible target atom k, and which has a radius equal to the impact parameter p_k of the system, and a length equal to the distance between the muon and k, r_k.

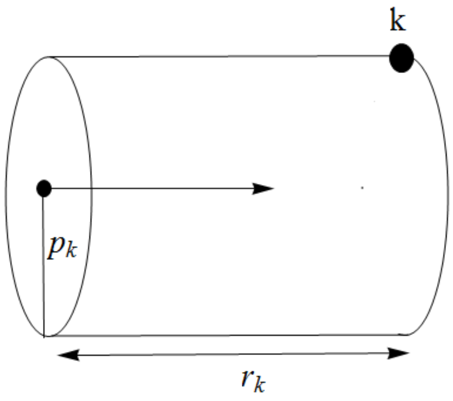


Figure 2.12. A cylindrical volume enclosed by the muon and a possible candidate for scattering, k.

The volume of this cylinder is

$$V_k = \pi \, p_k^2 \, r_k. \tag{2.77}$$

Our idea is that the bigger the volume of this cylinder is, the smaller is the probability of k being chosen as the target atom. We introduce the probability by

$$P(k) \propto \frac{1}{V_k}, \quad k = 1,\, 2,\, ...,\, 14 \tag{2.78}$$

and take out the factor π to write the probability of being chosen as

$$P(k) = \frac{1}{p_k^2 \, r_k} \tag{2.79}$$

The algorithm ranks each of the atoms in the neighbor list according to this probability and normalizes each $P(k)$ value by dividing it with the summation of all probabilities. Once this set of probability values is created, we treat the set as a collection of bins (where all the values add up to 1). A random value between 0 and 1 is generated, and by using a linear search in the probability set we determine which bin this value falls into. The bin widths are non-uniform because of different probability values, so the linear search is required to select the appropriate bin. The atom corresponding to the chosen bin is selected as the next target atom.

2.6.2 Algorithm

In order to generate the probability set, we do the following:

```
Begin
For nCount = 1 to number of neighbors
{
  Find the projectile's impact parameter p and radial distance r from neighbor[nCount];
  S[nCount] = 1/((p^2)r);
  sumS = sumS + S[nCount];
}
For nCount = 1 to number of neighbors
```

```
  Probability[nCount] = S[nCount]/sumS;
//Once we have the probability set, we generate a random number and see where in the set it belongs.
random_candidate = Random(0, 1);
low = 1;
high = Probability[2];
If random_candidate >= low AND random_candidate < high
{
  Select the 1st atom in the neighbor list;
  Break;
}
Else
{
  For i = 1 to number of neighbors
  {
    low = low + Probability[i];
    high += Probability[i+1];
    If random_candidate >= low AND random_candidate < high
    {
      Select the i-th atom;
      Break;
    }
  }
}
End
```

This algorithm is coded in our C++ version of TRIM that simulates crystalline structures. We replace the condition we employed before, $p < p_{max}$, and use this method to select the next target atom.

2.6.3 Results and Analysis

The results are, unfortunately, not very promising. We obtained a distribution that did not resemble the usual distributions produced by other simulations. Moreover, around one-fourth of the muons backscatter in each run. This is unusual, and this does not portray the real stopping process. The calculations are checked to make sure the algorithm is doing what it is supposed to do. The probability set generated at each step is calculated correctly - the individual elements of the set add up to 1.0. In order to investigate further, we tracked each muon's energy loss processes at each step. Similar to our last attempt (section 2.5.3), the energy loss due to nuclear recoil energy was found to be negligible, and most of the energy is lost due to electronic stopping.

One plausible explanation for this bizarre behavior is the probability formula we use. Small impact parameters usually give rise to backscattering of the projectile. Since the probability is inversely proportional to the impact parameter squared, the algorithm essentially selects atoms which have smaller impact parameter with the projectile (and thus have higher probability of being selected according to our formula). The probability function can be improved by taking account of the possibility of scattering from several neighbor atoms simultaneously. We are not sure how to incorporate this scenario with a single function though.

In general, it is not clear whether we can simulate muon passage through crystals and investigate the channeling effects with binary collision approximation methods. With some success in characterizing the channeling effect using MARLOWE, we next turn our attention to molecular dynamics models, which take account of the interaction between many bodies. Using these models, we may very easily see the effect of simultaneous scattering from all the neighbor atoms.

3

3 Molecular Dynamics Model

3.1 Introduction

Molecular dynamics techniques are considered to be more accurate than Binary Collision Approximation in the low energy range. The basic principle is to keep track of a number of projectiles and recoil atoms as they interact with one another in a simulation cell. The simulation process is time dependent. After a certain time step dt, the positions of all the atoms are recalculated and updated. The precision comes in exchange for longer computation time and larger memory storage. Unlike BCA, MD techniques require us to store information about all the particles in the simulation cell, and at each time step, this information is accessed and updated. Since all the interactions are taken into account, basic algorithms for MD techniques take account of interactions between n - 1 atoms with each atom that essentially yields $O(n^2)$ computation time. Despite the longer computation time, molecular dynamics techniques are getting more and more popular due to availability of parallel supercomputers. Even on a home machine that has several processor cores, fast MD programs can be executed in parallel which greatly reduces the computation time. The basic physics of MD is easy to implement, so it all boils down to intelligent use of data structures and fast computers when it comes to efficiency and reliability.

3.2 Existing Simulation Techniques

This section will describe the current techniques used in molecular dynamics simulations. This is a subject that has been well studied, and many different algorithms and schemes exist in the literature that address different issues. Molecular dynamics techniques are used in a wide variety of applications including cell biology, ion beam sputtering, radiation damage calculations etc. We will give a brief overview of the basic principles, equations and algorithms used in classical molecular dynamics simulations that explicitly deal with ion beams.

3.2.1 Basic Principles

Given an ensemble of n particles in a simulation cell, we are to find out the force they exert on one another over a certain length of simulation time, updating the positions of the particles as needed. As a projectile enters the target, the motion of the projectile is affected by the target particles that are nearby. If the projectile has enough speed, or has enough mass, it can knock off the target atoms from their lattice positions, and they become projectiles as well. Let the projectiles be denoted by the index i and the neighbor atoms which are exerting force on them be denoted by index j (note that these atoms may also be projectiles). Then the force on the projectile is given by the summation of all the forces from the neighbors.

$$M_i \frac{d^2\vec{r_i}(t)}{dt^2} = \sum_{j=1}^{N} \vec{F_{ij}} = \vec{F_i}(\vec{r_i}(t))\,, \tag{3.1}$$

where M_i stands for the mass of the projectile, and N is the total number of neighbors at a given instant of time. When looping through all the atoms, we should also note that an atom does not exert force on itself, and the magnitude of the force exerted by atom a on atom b is the same as that by b on a. The direction of the force is reversed.

$$\vec{F_{ij}} = -\vec{F_{ji}},\ j \neq i, \tag{3.2}$$

$$\vec{F_{ii}} = 0. \tag{3.3}$$

The forces are conservative, and hence, can be derived from a potential function $V(r)$ by the following formula:

$$M_i \frac{d^2\vec{r_i}(t)}{dt^2} = \vec{F_i}(\vec{r_i}) = -\sum_{j\neq i} \nabla V_{ij}(\vec{r_{ij}})\ , \tag{3.4}$$

where r_{ij} is given by

$$r_{ij} = (|\vec{r}_i|^2 - |\vec{r}_j|^2)^{1/2}. \tag{3.5}$$

There are several ways for solving these equations numerically, namely the Central Difference method, Average Force method, Euler-Cauchy scheme and the Verlet scheme. These are not very hard to implement in terms of programming, and usually yield a good approximation with a carefully chosen time step $\triangle t$.

A time step determines the accuracy and efficiency of the simulation. The popular rule of thumb [Eckstein, 39] in the literature is to choose a time step such that the fastest projectile will not traverse more than 5% of the distance equivalent to the lattice constant.

$$\triangle t = 0.05\, d\, \sqrt{M/2T_m}, \tag{3.6}$$

where T_m represents the kinetic energy of the projectile with mass M, and d is the lattice constant, i.e. the interatomic separation between unit cells, or equivalently, the length of one edge of the cell.

3.2.2 Potential Function

There are many varieties of potential functions which are used in both BCA and MD simulations. Some of the widely used are - Born-Mayer, Morse, Lennard-Jones, Johnson's etc. Born-Mayer potential is the simplest one we came across:

$$V(\vec{r}) = A_{BM}\, e^{-\frac{|\vec{r}|}{a_{BM}}} \tag{3.7}$$

where A_{BM} is an energy parameter given in eV, and a_{BM} is the screening length. The values of A_{BM} and a_{BM} are found by fitting them to the Thomas-Fermi-Dirac potential curves since these two potential functions behave almost in the same way [Eckstein, 46]. A table of values of these parameters is available [Eckstein, 47]. Morse potential is a bit more tricky which takes account of both small and large internuclear distances. If we are dealing with very low energy regime, attractive forces come into play. Morse proposed an attractive potential of the form

$$V(r) = D\, e^{-2\alpha'(r-r_0)} - 2D\, e^{-\alpha'(r-r_0)}. \tag{3.8}$$

The first term of the potential introduces a repulsive force, the second term dominates at bigger internuclear separations. D is a parameter (in eV) that gives the depth of the attractive well of the potential, and r_0 is a parameter that determines where the potential will reach 0, and it also determines the slopes of the curves. It has a similar value compared to the nearest neighbor distance in a lattice. These parameters are calculated and verified by curve fitting with experimental data.

Finding the correct potential for a specific task is tricky. Other than these simple potentials, there are many potentials which are just combinations of a few simpler potentials so that a larger range of internuclear distance and energy regime can be addressed. The idea of combined potentials comes from the fact that at low energies and larger separations, the interatomic force becomes attractive, and at smaller distances, repulsive force comes into action. So an trick is to use the Morse potential for larger distance, and any repulsive potential at smaller distance. To fit both potentials together, a cubic polynomial is used.

Another popular scheme for describing interactions between atoms is the Embedded Atom Method (EAM). This relies on the idea that the electron density surrounding an atom is a superposition of the electron densities of all the neighbor atoms. Due to the electrostatic repulsion, the total energy is approximated by

$$E = \sum_i F_i(\varrho_{h,i}) + \frac{1}{2} \sum_{i,j\,(i \neq j)} \phi_{ij}(r_{ij}), \tag{3.9}$$

where $F_i(\varrho_h)$ is the energy that is needed to attach atom i within the background electron density ϱ, and $\phi_{ij}(r_{ij})$ represents the repulsion between the cores of atoms i and j with interatomic separation r_{ij}. Using the total energy, the ground state properties of the solid can be calculated. With a good approximation function that describes $F(\varrho)$ and the pair potential ϕ_{ij}, we can calculate the exchange of energy between atoms to estimate their influence on one another.

3.2.3 Other Methods

The above sections stated the basic ideas used in MD simulations. People have tailored these schemes and algorithms according to their needs, which brings us to a brief discussion of other methods relevant to our simulation.

A method called Recoil Interaction Approximation only tracks the projectile and the surrounding atoms (the list of which dynamically evolves) in the simulation. This is particularly useful for ion beams simulation as we do not need to track what interactions go on between atoms other than the projectile and the neighbor atoms of the projectile which can be affected from the force exerted on them by the projectile while it moves along the crystal. Recoil Interaction Approximation is widely used in ion beams simulations nowadays [Nordlund, 1995].

3.2.4 Computational Efficiency

In order to increase the efficiency of calculation, Verlet introduced a method for bookkeeping [Eckstain, 39]. For a particle i, a table of all particles within a calculated distance r_m is produced, and only these particles are allowed to interact with the projectile in the next (n-1) time steps. Rather than keeping track of all the atoms in a solid (which would be a huge task), the simulations are usually done in a confined volume known as the simulation cell. The simulation cell contains a sample of the solid containg a few thousands to a few hundred thousands atoms (depending on the type of simulation), and as the projectile enters the simulation cell, it interacts with the atoms in the simulation cell. In many cases, the projectile atoms will come to a halt within the simulation cell volume. However, for atoms with very high velocity, or for simulation cells with lower number of atoms in the arrangement, i.e. a smaller cell, the contents of the cell needs to be updated as the projectile goes out of the cell. The projectile is then assumed to enter a similar simulation cell, which contains an updated list of atoms that could have entered the current cell from the previous cell's interactions, and also the new atoms in the cell which are supposed to be there. In this way, only one cell is needed to simulate the rest of the cells in the crystal. This method is convenient and adapted in many different forms by the researchers.

3.3 Discretization Technique

To tackle some of the computationally intensive tasks and also to overcome the burden of writing very big code modules, we decided to modify the existing MD models slightly, taking into account the properties of muons in low energy regime.

The simulation cell is reduced down to a very small volume that only considers atoms which are very close to the projectile. For increase in computational efficiency, we have also decided to discretize the cell volume. In other words, the simulation cell is imagined to be a box made up of much smaller cubes. Each of the cube's center is thought to represent the whole cube, i.e. any point inside the cube will be approximated as the center of the cube. The force field due to neighbors are calculated in each of these small blocks, and the projectile interacts with the field and moves from one block to the other in the simulation.

This is not a very good approximation, so we should be careful and clever enough to handle all the consequences that may result from it. The sections below deal with the different aspects of this concept.

3.3.1 Potential Function

We have decided to use the simplest potentials to begin with, as they are easier to program. One choice is the Born-Mayer potential:

$$V(\vec{r}) = A_{BM}\, e^{-\frac{|\vec{r}|}{a_{BM}}} \tag{3.10}$$

where A_{BM} is an energy parameter given in eV, and a_{BM} is the screening length. Some authors have listed all the values of these parameters in detail for every element [Eckstein, 47 - 51]. So we decided to use the values given by them. Morse potential is also made available in our program because it's easier to implement. The choice of potential, for our purpose, mostly was driven by the factor of simplicity and time consumption behind writing big code modules for the other potentials mentioned before, such as the EAM method or the combined potential.

3.3.2 Assumptions

Several assumptions have been made in order to reduce the computational load compared to the existing molecular dynamics simulations. Some of them take the advantage of the particular potential function we decided to use, and others simply follow from the physical properties of muon. There are certain advantages of using the lattice unit cell of a material as our basic simulation cell, which we discovered while implementing the simulation. These assumptions are described in detail in this section.

Effective Distance (Size of Cell)

The potential function we are using is basically an exponentially decaying curve, $V(\vec{r}) = A_{BM}\, e^{-\frac{|\vec{r}|}{a_{BM}}}$. Using the universal values proposed by Andersen and Sigmund [Eckstein, 45], $A_{BM} = 52.0\,(Z_1\, Z_2)^{3/4}$ eV and $a_{BM} = 0.219$ Å, and taking $Z_1 = 1$ for muon, we can plot the potential function over a certain range of $|\vec{r}|$ and get a feel for the strength of this particular potential for different elements (different values of Z_2). A small Mathematica script (see Appendix A) takes care of this to produce the following plots.

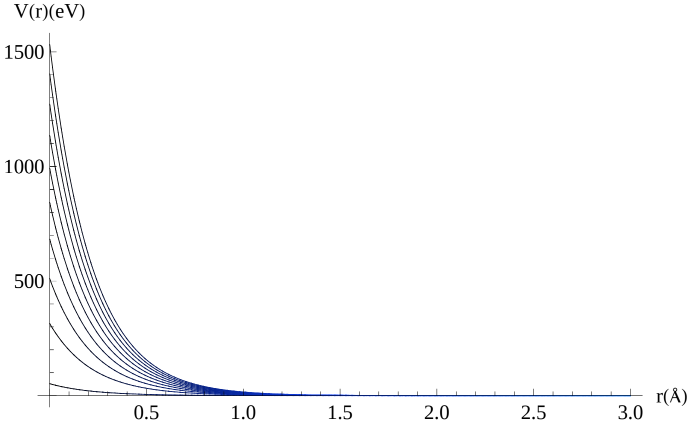


Figure 3.1. Potential strength (eV) vs. distance (Å) for elements with atomic number $Z_2 = 1$, 10, 20, ..., 90.

Zooming into a much smaller range provides us a clearer scenario.

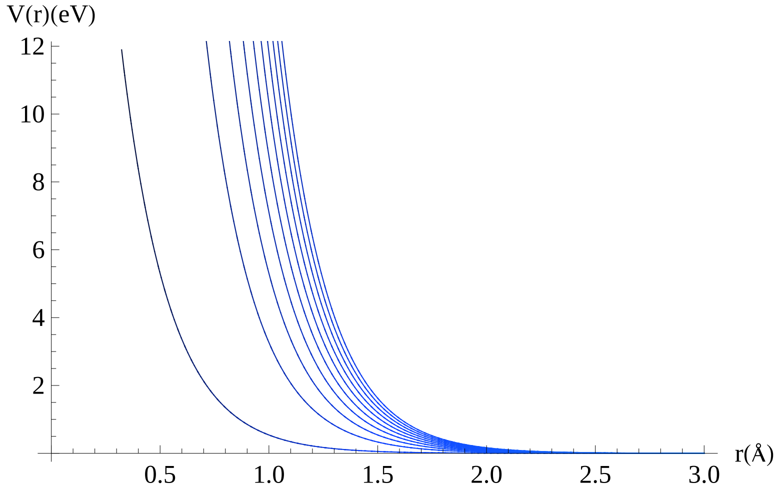


Figure 3.2. A closer look at the Born-Mayer potential function

From the figures we can deduce that the Born-Mayer potential acts only up to a certain range. Fortunately, for our purpose, this range is around the same size as the lattice constant of the elements we are mostly concerned about (for example, Iron - 2.87Å). In such a situation, we can argue that we really do not need to keep track of a few thousand atoms in a bigger simulation cell since we are mostly interested in the interaction between the projectile and the nearby lattice atoms that can affect it with enough force. Surely, a simulation cell large enough to include the neighbor atoms which are rougly one lattice constant distance away is sufficient to take account of the Born-Mayer interactions between the projectile and the neighbor atoms.

Negligible Recoil Energy Loss and Recoil Interaction Approximation

The previous assumption brings us to another important consideration. Most of the molecular dynamics simulations take account of all the interactions of all the atoms in the simulation cell. This is particularly useful for high energy ion beams since the projectiles have enough energy to knock off an atom from their almost fixed site in the crystal lattice, which eventually become another projectile and may knock off other atoms too. This eventual chain of collisions may create a disturbance in the whole system composed of thousands of atoms in the cell. Thus keeping track of all the atoms are necessary in such situations. Bulk of many MD codes are devoted to building efficient data structure to keep track of all the atoms, and needs a lot of computation time and resources to do so. Fortunately, for our purpose, the properties of muon comes to our rescue. We know that muons are not even as heavy as protons, whereas our material of choice, iron, is really heavy compared to muons. Moreover, we are concerned about the low energy regime which is around 25 eV - 1 keV. Thus, the momentum with which these low energy muons come in is small and they are not capable of knocking off an atom from its lattice site. In other words, the recoil energy loss during the collision is negligible since only a minute fraction of the muon's momentum is transferred to the heavy lattice atoms.

The standard method to take advantage of this situation is to use Recoil Interaction Approximation methods. In this technique, only the interaction of the projectile with the target atoms are considered. As there's no chance of knocking off an atom from its site, the interaction between the lattice atoms can be safely ignored. We decided to follow this technique in order to avoid the unnecessary burden of coding extra modules to take account of interatomic interactions in the crystal.

Inelastic Energy Loss - Continuous Electronic Energy Loss vs. Local Electronic Energy Loss

We have already defined the essential concepts of inelastic energy loss in chapter 2. The theory is complex and there are many formulations present in the literature. The energy loss mechanism in our MD technique must take account of the properties of muon. Once again, it is pretty much simplified. Muon is not an ion (as in an atom with empty valence shell) with electronic shell structure. So there is no question of any interaction in the electronic shell level when it collides with other atoms. Thus the only thing left to worry about is the non-local continuous electronic energy loss due to the electron gas in the metal. We decided to use the same electronic stopping code we used for BCA simulations.

The center of a block represents all the other points in the block

If the number of blocks in the simulation cell is enough so that the distance between the centers of two adjacent blocks are "small," we can approximate the center of a block to represent all the points in that block. The word small is quoted because the definition of it depends on us. What kind of accuracy are we looking for in our results? This issue is addressed in section 3.3.6 where we give a formal description of accuracy in the context of our model.

3.3.3 The Simulation Cell: General Structure

The simulation cell is a cuboid which has edges of length equal to the lattice constant. The idea is to divide the cell into blocks (smaller cubes) of constant edge lengths. The center of each block will represent all the other points inside the block. This means that for the approximation to work well, the dimension of each of the blocks should be sufficiently small enough. The dimension of these smaller blocks are determined by a scheme described in section 3.3.6.

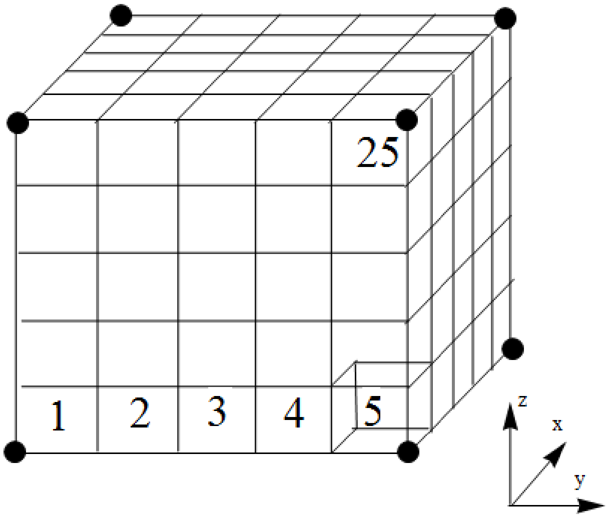


Figure 3.3. A simulation cell composed of a matrix of 5x5x5 blocks. The origin of the cell coordinates is located at the bottom left corner of the first layer.

In order to identify each of the blocks in the cell, we number them according to the following scheme, which makes the programming easier. The lower left corner of the cell (also the lower left corner of the block labeled 1) represents the origin. If the unit lattice cell dimension is equal in all three directions (e.g. bcc elements), then let the dimension be represented by L. Using a scheme we will describe later in section 3.3.6, we find the number of blocks we want in each direction n_b, and the length of one edge of the block is given by L/n_b. The blocks are numbered from the lower left corner to the right in an ascending order, and we go one step further in the z direction and continue our count. This makes the upper right corner of the first layer numbered 25. Then we continue with the same scheme for the consequent layers in ascending order, so that the block on the upper right corner of the last layer in x-direction is numbered 125.

3.3.4 Force Field

The force field in each of the smaller blocks can be calculated in several ways. We present two methods we devised that are particularly suitable for our simulation purpose.

Method 1: If we happen to know the individual potential at a certain point due to a few particles exerting force at that point, we can add up all the scalar potential values at that particular point. For the purpose of the simulation, we take the center of each of the blocks as the point that represents the potential of any other point in that box. Now, if the blocks are small enough so that the distance between the centers is quite small, we can approximate the force in the y-direction at the center of each of the blocks by

$$F_{Cy}=\frac{V(y2)-V(y1)}{y2\ -\ y1} \tag{3.11}$$

where y2 and y1 represent the coordinates of the centers of the blocks adjacent to the block of concern in the y-direction. Similarly, for z and x direction, we have

$$F_{Cz}=\frac{V(z2)-V(z1)}{z2\ -\ z1} \tag{3.12}$$

and

$$F_{Cx}=\frac{V(x2)-V(x1)}{x2\ -\ x1} \tag{3.13}$$

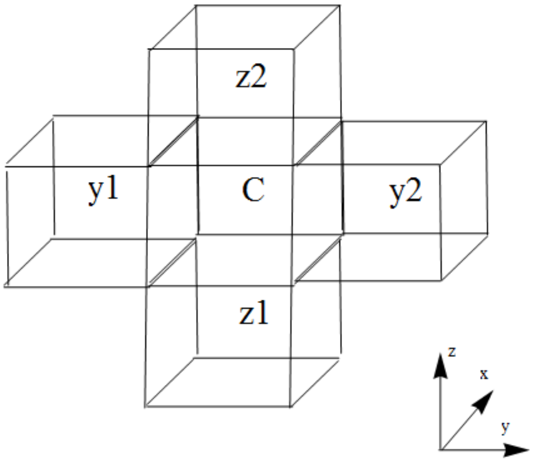


Figure 3.4. Calculation of force vector in the block labeled C from the adjacent blocks' potential

values. The blocks in the x-direction are not shown.

After finding the individual components of the force vector, we simply represent the vector by

$$\vec{F_C} = \{F_{Cx}, F_{Cy}, F_{Cz}\} \tag{3.14}$$

These approximation formulae are discrete versions of the general formula for deriving force from a conservative potential field. We start by noting that force is simply the gradient of the scalar potential

$$\vec{F} = -\nabla V = -(\frac{\partial V}{\partial x}, \frac{\partial V}{\partial y}, \frac{\partial V}{\partial z}). \tag{3.15}$$

We are to figure out the combined force vector at a certain point due to multiple bodies. Consider the hypothetical situation where we have n lattice atoms in the neighbor configuration, and we are trying to approximate the force at the mid point C between P1 and P2, the centers of two adjacent blocks.

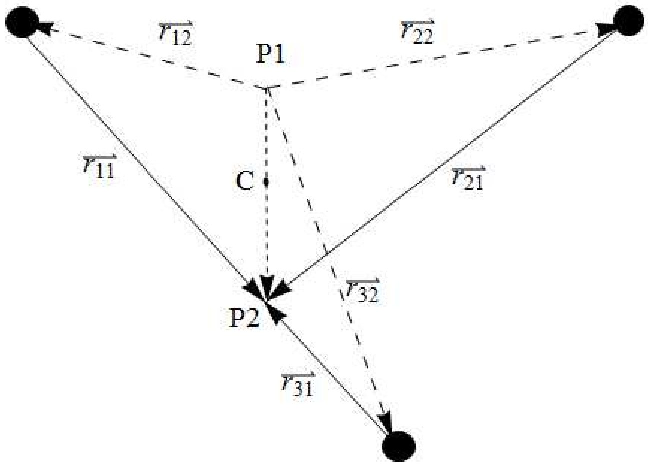


Figure 3.5. A hypothetical situation with n neighbor atoms (only three are shown). Their scalar potential at points P1 and P2 are known. The force at C (the mid point of P1P2) is to be approximated.

In general, the force vector for any two known potentials due to a particular neighbor (say 1) at P1 and P2 will be given by

$$\vec{F} = -\nabla V = -\frac{\partial V}{\partial r} \simeq -\frac{V_{11} - V_{12}}{|r_{11} - r_{12}|}, \tag{3.16}$$

where V_{11}= V($\vec{r_{11}}$) and V_{12}= V($\vec{r_{12}}$), and they are calculated from the Born-Mayer potential formula. Note that for all the other force vectors due to other $n-1$ neighbor atoms, the magnitude of ∂r in the formula is always same as the distance between P1 and P2 is constant. Now, the combined force vector at C is given by the summation of all $\frac{\partial V}{\partial r}$ values due to the n neighbor atoms. Thus

$$\vec{F_C} = -\sum_{k=1}^{n} \frac{\partial V_k(r_k)}{\partial r_k} \simeq -\sum_{k=1}^{n} \frac{V_{k1} - V_{k2}}{|r_{k1} - r_{k2}|}. \tag{3.17}$$

$$\vec{F_C} \simeq -\left(\frac{V_{11} - V_{12}}{|r_{11} - r_{12}|} + \frac{V_{21} - V_{22}}{|r_{21} - r_{22}|} + ... + \frac{V_{n1} - V_{n2}}{|r_{n1} - r_{n2}|}\right). \tag{3.18}$$

Note that all the denominator values are the same - the distance between P1 and P2. Let the distance be $\triangle r$. Then the combined force vector can be written as

$$\vec{F_C} \simeq -\left(\frac{V_{11} - V_{12}}{\triangle r} + \frac{V_{21} - V_{22}}{\triangle r} + ... + \frac{V_{n1} - V_{n2}}{\triangle r|}\right). \tag{3.19}$$

$$\vec{F_C} \simeq -\left(\frac{V_{11} + V_{21} + ... + V_{n1}}{\triangle r} - \frac{V_{12} + V_{22} + ... + V_{n2}}{\triangle r}\right). \tag{3.20}$$

Thus, the force vector is given by adding up all the potentials due to n neighbors at P1, dividing the number by the distance between P1 and P2, doing the same for P2 and finally subtracting the latter from the former. This is essentially captured by the equations stated earlier in this section.

In order to do this by writing a program, we need to -

1. Calculate the Born-Mayer potential at each of the block's center, due to all of the neighbor atoms we consider in the simulation cell,

2. Loop through each of the center points, and find the force according to the above formula. Calculating the potential requires us to go through every single point and add up the potential contribution of the neighbor atoms, which is roughly an $O(n^2)$ method. Finding the force costs us a linear time algorithm.

3. The force at the boundaries of the simulation cell require special attention that is dealt in 3.3.7. Otherwise, we stick to the general rule described above.

Algorithm

Input: *neighborAtomsList* (list of neighbor atoms' coordinates), *CoordList* (List of center coordinates).

Other Arrays: *Vlist* (list of potentials at the centers), V (temporary list to hold the individual potential due to each neighbor atoms), *forceList* (list of force vectors at each point)

```
Begin
 For i = 1 to Number of blocks
   $currpt = CoordList[i]$
   For j = 1 to number of neighbors
      $currNeighbor = neighborAtomsList[j]$
      $dist = FindDistanceBetween(currpt, currNeighbor)$
      $V[j] = A_{BM}\, e^{-dist/a_{BM}}$
      $sumV$ = Sum all the elements of V
      $Vlist[i] = sumV$
//force calculation
For i = 1 to Number of blocks
   If $CoordList[i][1]$ is not in boundary (here $CoordList[i][1]$ is the x coordinate)
      $fx = \frac{Vlist[i+1]-Vlist[i-1]}{2*StepVolume}$
   Else
      $fx = \frac{Vlist[i]-Vlist[i-1]}{StepVolume}$
   If $CoordList[i][2]$ is not in boundary ($CoordList[i][2]$ is the y coordinate)
      $fy = \frac{Vlist[i+1]-Vlist[i-1]}{2*StepVolume}$
   Else
      $fy = \frac{Vlist[i]-Vlist[i-1]}{StepVolume}$
   If $CoordList[i][3]$ is not in boundary ($CoordList[i][3]$ is the z coordinate)
      $fz = \frac{Vlist[i+1]-Vlist[i-1]}{2*StepVolume}$
   Else
```

```
        $fz = \frac{Vlist[i]-Vlist[i-1]}{StepVolume}$
    $forceList[i] = fx, fy, fz$
//end of For loop
 Return $forceList$
End
```

Method 2: This is a straightforward method that may yield more accurate results compared to the previous method. Following the derivation of the force $\vec{F_C}$ on a projectile in method 1, we take account of the fact that instead of using a discrete approximation for the change in potentials from adjacent blocks, we directly use the formula for $\frac{\partial V}{\partial r}$, i.e.

$$\vec{F_C^k} = -\sum_{i=1}^{n} \alpha_i^k \frac{\partial V(r_{ic})}{\partial r_{ic}}, \tag{3.21}$$

where α_i^k is the cosine of the angle between the vector $\vec{r}_{ic}$, the displacement from the i-th atom to the center of the block, and the k-th axis, where k $\in\{1, 2, 3\}$. The force $\vec{F_C^k}$ calculated in this way gives the force along the k-th axis. To determine α_i^k, we calculate the dot product between $\vec{r}_{ic}$ and a unit vector along the k-th direction, and divide the quantity by the product of magitudes of the vectors. In the case of a unit vector, which has a magnitude of 1, we simply divide the dot product by the magnitude of $\vec{r}_{ic}$. The summation of all the calculated $\frac{\partial V(r_{ic})}{\partial r_{ic}}$ values may give a better approximation compared to the scheme we described in method 1.

Algorithm

Input: *neighborAtomsList* (list of neighbor atoms' coordinates), *CoordList* (List of center coordinates).

Other Arrays: *Vlist* (list of potentials at the centers), *V* (temporary list to hold the individual potential due to each neighbor atoms), *forceList* (list of force vectors at each point)

```
Begin
 For i = 1 to Number of blocks
    $fx = 0.0$, $fy = 0.0$, $fz = 0.0$.
    $\vec{CP} = CoordList[i]$
    For j = 1 to number of neighbors
        $\vec{CN} = neighborAtomsList[j]$
```

```
      $dist = FindDistanceBetween(\vec{CP}, \vec{CN})$
      $dVdr = -\frac{A_{BM}}{a_{BM}} e^{-dist/a_{BM}}$
      $\vec{from} = \vec{CP} - \vec{CN}$
      $\vec{with} = \{1, 0, 0\}$
      $cs = \frac{\vec{from} \cdot \vec{with}}{|\vec{from}| * |\vec{with}|}$
      $fx = fx - dVdr * cs$
      $\vec{with} = \{0, 1, 0\}$
      $cs = \frac{\vec{from} \cdot \vec{with}}{|\vec{from}| * |\vec{with}|}$
      $fy = fy - dVdr * cs$
      $\vec{with} = \{0, 0, 1\}$
      $cs = \frac{\vec{from} \cdot \vec{with}}{|\vec{from}| * |\vec{with}|}$
      $fz = fz - dVdr * cs$
    $forceList[i] = fx, fy, fz$
  Return $forceList$
End
```

3.3.5 Energy Loss Mechanism

As mentioned in our previous discussion, a projectile, when it enters the crystalline material, can lose energy in three ways - lattice atom recoiling due to momentum transfer from the projectile, interactions between the electron shells of the projectile and the lattice atoms (local electronic energy loss), and finally, continuous electronic energy loss due to the free electron cloud in the material. For muons, we assume that there's a negligible amount of energy loss due to lattice atom recoil (3.3.2). The question of local electronic energy loss is also not relevant here as our projectile does not have any electronic shell structure. So in terms of energy loss procedure, we are only concerned about continuous electronic energy loss. For this, we use the same calculations and procedures employed for binary collision approximation. However, for molecular dynamics purpose, where we are more reliant on velocity calculations, we need to use different formulae to incorporate the stopping.

In every time step, the projectile advances a variable amount of distance. For BCA, we relied on finding the length of the step, multiplying that with the S_e value that corresponds to the current energy of the projectile, and finally subtracting that quantity from the current energy of the projectile. For MD, our main goal is to calculate new velocities due to the force field at each block the projectile travels to, and calculate the displacement over some definite time step due to that velocity. In order to incorporate S_e in this method, we calculate by how much the magnitude of velocity changes as it traverses some distance over the fixed time step. If S_e is given in the units of $eV\text{Å}^{-1}$, The change in speed in Ås^{-1} is given by

$$\triangle v = \triangle t \, \frac{S_e}{M_m}, \tag{3.22}$$

where M_m is the mass of muon and $\triangle t$ is the time step. The magnitude of velocity is decreased by this amount in every $\triangle t$ amount of time. Other than the stopping scheme proposed by Zeigler, the Lindhard-Scharff electronic stopping model is also included in our simulation to see whether it provides better results. Both the models are described in chapter 2.

3.3.6 Number of Blocks

The number of blocks needed for our approximation scheme to work is a crucial part in the design of this simulation. We do not want a huge number of blocks in the simulation cell as that defeats the purpose of designing a computationally less intensive program. We also want to have sufficient blocks in our cell in order for the force field calculation to be as precise as possible. Thus, choosing the right number is a trade-off between simulation speed and computational accuracy. For crystals with much bigger unit cell size, the issue becomes more important. We have devised a simple method that gives the user complete control over this trade-off process.

The issue essentially boils down to the fact that we are approximating the center as the representative of all the points in the block. How good or bad is this? The corner points of a block are the farthest from the center in the block, so we should find a measure of how good the center is when approximated as the corners. We cannot simply use distance as our measure here. The simulation cell can be of different size and can have different neighbor atom configurations that may make the potential and force field vector values drastically different for different crystals. Hence, we concentrate on how much the potential value changes from the corners to the center of the block.

Let n_B be the total number of blocks in the x, y and z direction of the simulation cell. For now, we will focus on having equal number of blocks in each direction. For a particular block, let $\vec{C_k}$ be any of the eight corners' position vector (in the cell coordinate space), where k∈{1, 2, ..., 8}, and let $\vec{a}$ be the position vector of the center. We define the value ϵ as a measure which will be used to determine how many blocks we need in each of x, y and z directions. Let us also choose a random atom from the nearest neighbor atom configuration, which has a position vector $\vec{n}$. Then the mean of the potential contribution from the chosen neighbor atom to the eight corners of the block is

$$\overline{V} = \frac{\sum_{k=1}^{8} V(\vec{n} - \vec{C_k})}{8}, \tag{3.23}$$

Where $V(\vec{r})$ is the Born-Mayer potential ($\vec{r} = \vec{n} - \vec{C_k}$). The standard deviation is:

$$\sigma 1 = \sqrt{\frac{\sum_{k=1}^{8} (V(\vec{C_k}) - \overline{V})^2}{8}}. \tag{3.24}$$

Now, we will assume the potential at the center of the block, $V(\vec{a})$, as the mean potential $\overline{V}$ calculated above. Then the standard deviation is:

$$\sigma 2 = \sqrt{\frac{\sum_{k=1}^{8} (V(\vec{C_k}) - V(\vec{a}))^2}{8}}. \tag{3.25}$$

We set $\epsilon = \sigma 1 - \sigma 2$, and observe its characteristic as we increase n_B. In order to find n_B, we choose any two consecutive points from $\vec{C_k}$ (assuming that they are ordered), say the first two points. Then n_B is given by:

$$n_B = \frac{L}{|\vec{C_1} - \vec{C_2}|}, \tag{3.26}$$

where L = length of an edge of the simulation cell. The evolution of ϵ for different values of n_B tells us a way to choose a suitable value of n_B. A Mathematica program that carries out the above calculation is written for this purpose.

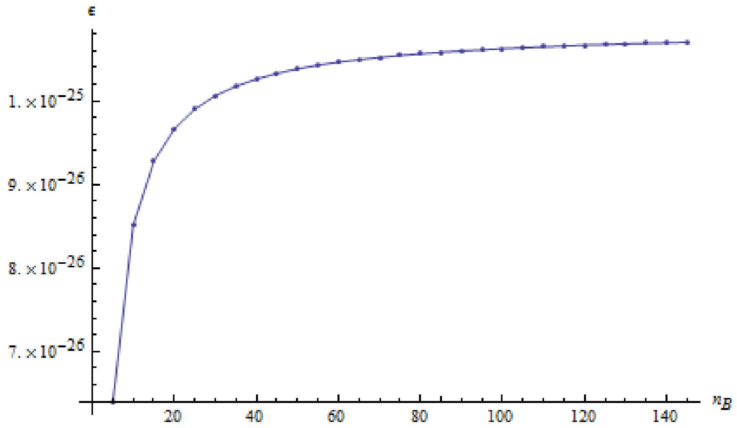


Fig. 3.6. ϵ vs. n_B graph generated from the Mathematica program for Born-Mayer Potential in bcc Iron. (Note: The vertical axis does not start from 0).

The graph shows the evolution of ϵ values as we increase the number of blocks starting with $n_B = 5$. The program output is written for the Born-Mayer potential interaction between muon and bcc iron lattice, although it can be easily modified for any other potential function and atoms. As seen from the graph, the value of ϵ starts to become steady after $n_B \simeq 45$. Certainly it does not make sense to choose a value for the number of blocks greater than 45 in each direction. For the purpose of speeding up the simulation, we choose a value between 25 and 30, which is right after the big jumps of ϵ values.

3.3.7 Boundary Conditions

Boundary conditions are always difficult to handle. An accurate boundary condition usually is a key factor for the reliability of a model. In usual MD methods, the boundary of the crystal is designed to have a restoring force to keep the crystal confined to a certain volume. A damped harmonic oscillator is a standard force for this purpose. Atoms at the boundary are governed by the equation

$$M_m \frac{d^2u}{dt^2} = ku - R\frac{du}{dt}. \tag{3.27}$$

The term ku represents a spring-like force that simulates the elastic response of the matrix. The damping term makes the excess kinetic energy of the model cluster disappear. Usually a critical damping model is used, for which R $= (4M_m k)^{1/2}$. The critical damping model makes a particle come back to the matrix with zero velocity.

For our model, since we are mostly interested in the trajectory of the projectile and since the lattice atoms are assumed to be fixed in their position, we decided that we do not need a boundary restoring force to keep the crystallite confined to a fixed volume. Since we have agreed to the fact that low energy muons only lose a very minute amount of energy to the heavy atoms of the crystal, there is no question of the atoms that are fixed in their position to move out from there and go out of the confined volume of the crystal.

For our simulation cell, however, we do need to take account of an accurate potential at the boundary for force field calculation by method 1. The force at block i is determined from the blocks $i+1$ and $i-1$ for a particular axis, but at the boundary there is either the $(i+1)th$ or the $(i-1)th$ block missing. For this purpose, we calculate the boundary potentials to be the potential at the mid point between the boundary block and the block immediately adjacent to it from the boundary, i.e.

$$F_{b_k} = \frac{V(\vec{r_{b_k}}) - V(\vec{r}_{(b\pm 1)_k})}{|\vec{r_{b_k}} - \vec{r}_{(b\pm 1)_k}|}, \tag{3.28}$$

where $\vec{r}_{b_k}$is the position vector of the center of the block at the boundary in $k-th$ direction, and $\vec{r}_{(b\pm 1)_k}$is the vector representing the center of the adjacent block in the postive or negative $k-th$ direction. This, again, is not a very good approximation as we are assuming that the average force at the mid point between $\vec{r}_{b_k}$ and $\vec{r}_{(b\pm 1)_k}$is the same as the force at $\vec{r}_{b_k}$, but that is the best we can do. This is justified to some extent due to the same reasoning we used in the previous section. The potential function we are using is a smooth curve and does not have a sharp change over the range we are considering here. Thus the force may not change a lot from the midpoint of $\vec{r}_{b_k}$ and $\vec{r}_{(b\pm 1)_k}$ to $\vec{r}_{b_k}$.

3.3.8 Trajectory Calculation

The heart of our simulation, after an accurate calculation of the force fields, is how precisely we calculate the trajectory of the muons. Although it seems that solving the force equations for the displacement term should do the job for us, which is pretty much straightforward, the inclusion of electronic stopping complicates the matter a little bit. Also, instead of using the trivial displacement and velocity equations,

$$\vec{r}(t) = \vec{r_0} + \vec{v_0}t + \frac{1}{2}\vec{a}t^2, \tag{3.29}$$

$$\vec{v}(t) = \vec{v_0} + \vec{a}t\,, \tag{3.30}$$

we use numerical integration schemes to solve the force equation dynamically during the simulation, which are slightly different from the above. Let the initial velocity of a muon particle be $\vec{v_i}$, and the initial energy be E_i. Realistically speaking, the particles do not come in with an incident angle of 0. The beam is spread over some area of a crystal face, hence we assume that the incident angle can range from 0° to 30° from the x (depth) axis. The relationship between v_i and E_i is -

$$v_i = \sqrt{\frac{2E_i}{M_m}}, \tag{3.31}$$

where M_m is the mass of the muon particle. The particle is incident on the $y - z$ face of the simulation cell, so we choose a random point $\vec{r_i}$on this face by choosing a random block B_i.

$$B_i = Random(1,\, Y_B Z_B), \tag{3.32}$$

where Y_B and Z_B are the number of blocks in y and z direction, respectively. $\vec{r_i}$ represents the center of the block B_i. Once the particle enters the crystal, we solve the equations of motion using numerical integration and find the new position $\vec{r}(t)$ at time t. There are several numerical integration schemes available. The most basic one is called the Central Difference method, which essentially reflects the general equations of motion.

This method tells us that the position at time $t + \triangle t$, $\vec{r}(t + \triangle t)$, is given by wherever the projectile is at time t, and a change in position due to the velocity of the projectile, which also changes in every time step as it interacts with a new force field. The velocity $d\vec{r}(t)/dt$ is determined at half the time step $\triangle t$,

$$\frac{d\vec{r}(t + \triangle t/2)}{dt} = \frac{d\vec{r}(t - \triangle t/2)}{dt} + \triangle t\, \frac{\vec{F_i}(t)}{M_m}, \tag{3.33}$$

$$\vec{r}(t + \triangle t) = \vec{r}(t) + \triangle t \, \frac{d\vec{r}(t + \triangle t/2)}{dt}. \tag{3.34}$$

The force $F_i(t)$ represents the force at the center of the current block the projectile resides in. The name comes from the fact that the velocity is found at $\triangle t/2$.

The mean kinetic energy is important for finding the right amount of electronic stopping $S_e(E)$. The mean K.E. at time t can be calculated from the following formula,

$$E_{mean}(t) = \frac{M_m}{2} \sum_{k=1}^{3} [\vec{v}^k(t)]^2 \tag{3.35}$$

where $\vec{v}^k(t)$ represents the velocity in the k-th direction (x, y or z) at time t. Using the position $\vec{r}$ of the projectile at times $t + \triangle t$ and $t - \triangle t$, we can write the mean K.E. as

$$E_{mean}(t) = \frac{M_m}{2} \sum_{k=1}^{3} \left(\frac{r^k(t + \triangle t) - r^k(t - \triangle t)}{2 \triangle t} \right)^2 . \tag{3.36}$$

Once the mean K.E. is found, we can find the appropriate electronic stopping at that particular energy of the muon. If the stopping is given by S_e, we incorporate it in the trajectory by reducing the velocity of the particle by an amount $\triangle v$ given in section 3.3.5. Let $\frac{d\vec{r}(t+\triangle t/2)}{dt} = v_{\vec{\triangle t}/2}$, and the unit vector of $v_{\vec{\triangle t}/2}$ be $v_{\vec{unit}}$. The new position of the particle is then given by

$$\vec{r}(t + \triangle t) = \vec{r}(t) + \triangle t \, (|v_{\vec{\triangle t}/2}| - \triangle v) \, v_{\vec{unit}} \tag{3.37}$$

3.3.9 Determining the Current Block

The symmetry of body centered cubic or face centered cubic crystals provides us a unique way to reduce the computation time and memory. Instead of storing many neighbor atoms' positions, we track the projectile's position and determine (at the end of its travel after every time step) where it is in the simulation cell. If it goes out of the cell, we determine where it emerges in the next simulation cell by using modulus of the displacement and the length of the simulation cell. Since the arrangement of the neighbor atoms are still the same for the next cell, we use the same force field calculations done for the initial cell to calculate the new trajectories of the projectile in the next cell, and the process continues.

If the new position of the particle is given by $\vec{r}(t + \triangle t)$, or simply $\vec{r}$, then the relative position in the cell is given by

$$rcell_k = r_k \; mod \; L \tag{3.38}$$

where k = 1, 2, 3 (for x, y and z) and $rcell_k$ represents displacements inside the simulation cell in the x, y and z directions. L is the length of one edge of the simulation cell. In this case, L is the lattice dimension.

3.3.10 Time Step

We use the same formula for time step described in section 3.2.1.

$$dt = 0.05\, L_c \sqrt{\frac{M_m}{2E_i}}, \tag{3.39}$$

where M_m= mass of muon, and E_i = initial kinetic energy. Advanced MD simulations use variable time steps to achieve better accuracy for more complicated setup of crystals, but for our purpose a fixed time step should be good enough. We can modify this time step formula to establish a lower limit based on the design of our simulation. The particles with maximum velocity should travel more than the dimension of a small block in the cell, i.e.

$$dt > \frac{L_c}{n_B} \sqrt{\frac{M_m}{2E_i}} \tag{3.40}$$

with n_B being the number of blocks in the cell. If n_B is variable (different values for the x, y and z axes), then the lowest value of n_B should be chosen. This limit ensures that the projectile does not experience the same force field by remaining in the same block for the next step. In our case, with a lattice constant of 2.87 Å for Iron, n_B = 25 should be enough number of blocks to use equation (3.39) that will give a dt value well above the lower limit.

3.3.11 Keeping Track of Channeling

All the basic quantities are taken care of by now. Now, as the simulation progresses, it is our goal to keep track of muons which are channeling at a certain plane. The idea of several brute-force methods are presented here that keeps track of a muon's direction and tries to determine whether the particular muon is channeling.

At each time step, the particle traverses a variable distance along the direction of its velocity. In order for the particle to stay in a path that is devoid of major deflections, we would expect that its trajectory is almost straight when it channels. From the raw trajectory data, it is possible to determine whether the particle channeled for some time by discretizing its trajectory into pieces and finding the angles between consecutive pieces. If the angles remain under a certain angle, e.g. 0.01 radian (as determined by the TRIM authors to estimate collision free flight path [Zeigler, 118]), then we conclude that the muon was channeling.

An easier alternative is to compare the initial and final velocity directions, find the angle between them and look at the distribution of the spread of angles. The standard deviation should give us a good estimate of whether the particle channeled from the initial to the final position. However, there can be a good possibility that a particle enters a channel much after it enters the crystal. Thus, the above method should work better.

However, in practice, we found that the trajectory of a projectile can be very random. None of the above methods may correctly describe channeling. The first method may find channeling at different parts of the trajectory of a muon, but such information is not useful for our purpose. We are mainly interested in transmission of muons out of the sample due to channeling. The easiest way is to manually change the incident direction of the muons and run the simulation to collect data regarding the average range reached and number of transmitted particles. This is done in chapter 4 where we compare the data of three different simulation runs with different incident beam directions.

3.3.12 Bookkeeping: What Quantities are of Interest?

Careful memory usage may reduce the computation time greatly. We are not dealing with low level memory management here, but we want to keep track of quantities which are useful to us for analysis, and at the same time we want to get rid of extraneous data that are generated for each particle and are not useful later. Only a few important data are saved during the simulation. We are mostly interested in keeping track of the trajectory itself, along with the energy of the particle. Thus all the position data at each step are saved so that we can build and analyze a precise trajectory.

The final stopping position should be stored in order to analyze the spatial distribution. Information regarding backscattering and transmission of particles also needed to be stored for our purpose.

3.3.13 Implementation of Another Method

In order to compare and validate our discretization methods, we have written another version of the MD algorithm that employs recoil interaction approximation and all the other assumptions we made for the discretization techniques, except discretizing the simulation cell volume. The force acting on a particle is calculated in real time as the particle moves through the solid, as opposed to using precalculated force fields. The advantage of using this method mostly concerns accuracy. Although it is slower to calculate the force field in every step of the trajectory, we no more have to approximate the center as the representative of all the points present in a certain unit volume. Every other calculation remains same in the code. With this code, now we have a platform to compare our discretization approximations.

3.3.14 Results and Analysis

All three MD programs produce histogram outputs that show stopping depth distributions. In addition, the programs produce trajectory plots, which is useful in studying the behavior of channeled particles. Programs based on the discretization method produce a 3-d vector plot of the force fields, which proved to be useful in debugging the program.

Figure 3.7 shows a typical force field produced by the discrete MD program for a bcc iron lattice. Many vectors appear as dots in this plot. This is due to the automatic adjustments of relative magnitude (done my Mathematica) to show all the vectors in the same plot. This gives us a sense about the regions where the force is stronger.

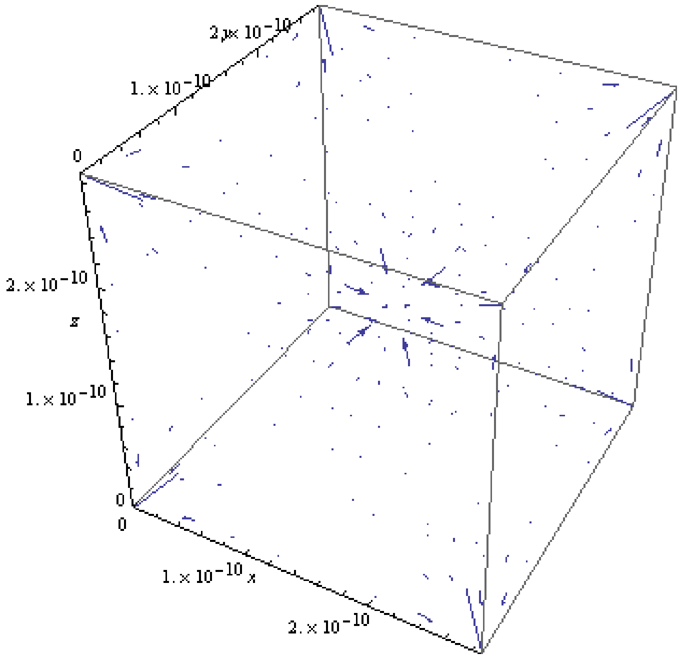


Figure 3.7. Force field vector plot for bcc iron lattice.

When the vectors are forced to be shown equal in magnitude using another command, the following plot (figure 3.8) is produced. This kind of plot is useful for studying the direction of the force field, and also for verifying the neighbor atoms positions. We have used such plots to see whether we left out any neighbor atom in our calculation. In that case, the direction of the fields at certain region changes, which is enough information to find out a discrepancy in the neighbor list. The density of vectors in this plot can be adjusted to make such debugging easier.

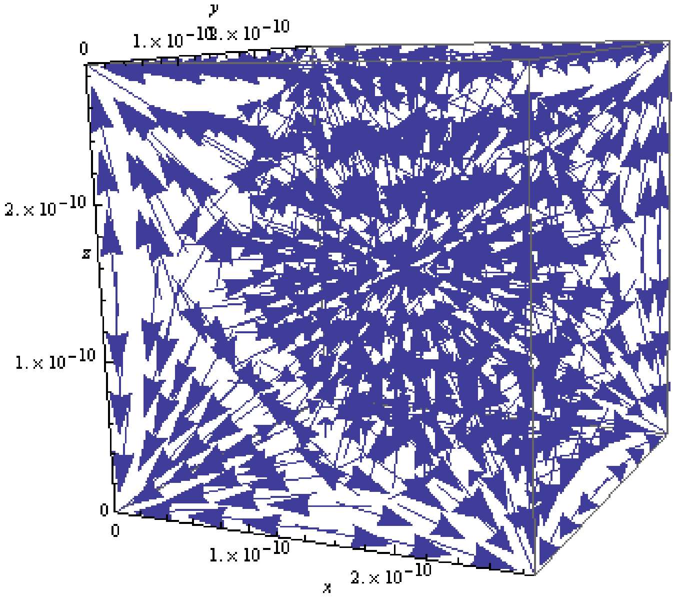


Figure 3.8. Force field vectors with unit magnitude, showing their directions only

Figure 3.9 shows a typical trajectory plot produced by the MD programs. This plotter program is not a built-in command in Mathematica. A custom program is written to collect all the trajectory data produced during the simulation, and plot each list in the same Graphics object in Mathematica.

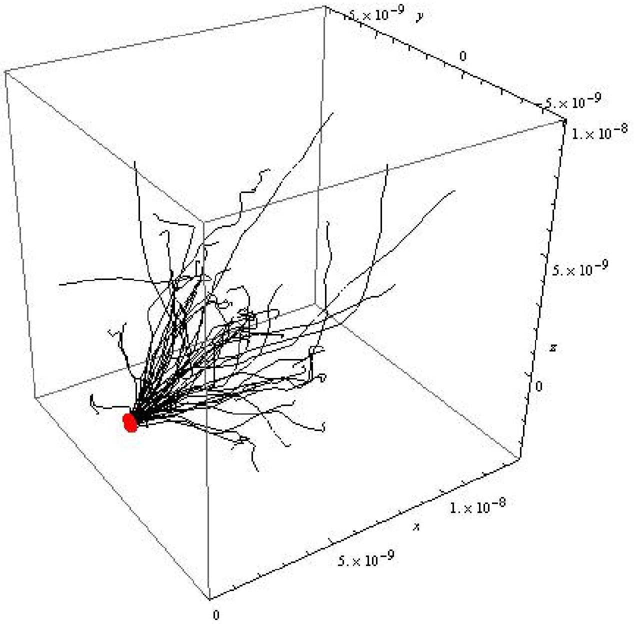


Figure 3.9. A trajectory plot for 500 eV muons going into iron. The red dot shows the entry area (the dot size is exaggerated).

A typical depth distribution produced by the programs is shown below. This particular run of the simulation was set up for 500 eV muons going into a bcc iron sample. The thickness of the sample is 500 Å. A total of a thousand particles were simulated in this case. The time taken for the simulation was around 15 minutes using the program that employs the method described in 3.3.13.

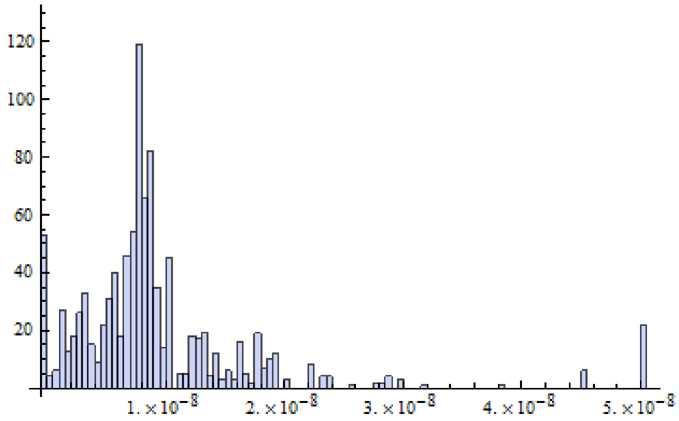


Figure 3.10. Range distribution produced from our molecular dynamics program. Depth is given in meters.

After running a few simulations using all three programs, we were certain that our methods are only valid with an incident energy lower than ~600 - 700 eV. Below this approximate threshold, the results come out to be remarkably close to TRIM or MARLOWE. Over this threshold, the particles do not tend to stop where they are supposed to stop, and continues much farther into the crystal. A possible explanation for such behavior is that we left out many factors and properties associated with the solid in order to keep our model simple. Phonon excitation of the lattice atoms could be taken into account, which would complicate the model. Our guess is that muons with incident energy below the threshold value cannot affect the lattice vibration significantly, whereas the ones coming in with greater energy may contribute more to the vibration.

4

4 Evaluation

4.1 Methods of Evaluation

In order to compare the models we designed and implemented, we ran simulations to collect key data, such as average depth reached in the sample, number of backscattered particles (particles that go out of the sample through the surface they come in initially) and the number of transmitted particles (particles that come out of the sample by processes other than backscattering). We do not evaluate the models we developed using BCA methods in crystalline samples as they were not really successful. The MD programs' outputs are compared with the data we collected from MARLOWE and TRIM. The MD programs are the two types of discretization (Method 1 & 2 in 3.3.4) and the recoil interaction approximation without discretization described in 3.3.13 (we will call it Method 3 from now on). Finally, we attempt to characterize channeling through different crystallographic directions in a body centered cubic metal.

4.2 Comparison of the Models

Simulations with the following properties were run for all five programs we are considering:

- Incident energy = 500 eV
- bcc iron sample
- Sample thickness = 500 Å (in the x direction)
- Incident angle = 0

- Number of muons = 1000

- Number of runs = 5

The following table summarizes the average values we obtained from the 5 runs.

Method	Average Depth (Å)	Backscattered Particles	Transmitted Particles
MD Method 1	76.0149	78	34
MD Method 2	83.9288	53	22
MD Method 3	69.3726	16	7
MARLOWE	68.786	81	N/A
TRIM	71	26	0

Table 4.1. Comparison of binary collision approximation and molecular dynamics programs.

The molecular dynamics methods 1 & 2 (the discretization techniques) make the muons penetrate more into the sample compared to other methods. The number of backscattered particles and transmitted particles are also higher. Method 2 gives rise to the highest average depth among all programs. Method 3, MARLOWE and TRIM seem to be producing consistent and similar values when it comes to average depth. As TRIM and MARLOWE are de facto standards in ion beam physics due to their consistency with experimental data, we may conclude that Method 3 is more accurate among all the MD techniques we employed. Method 1 & 2 may not have performed very well as they are only approximations, and it seems that channeling was more prominent in these simulations (as they produced the highest numbers of transmitted particles).

MARLOWE does not report the number of transmitted particles as we could not find a way to set the thickness of the sample in this program. However, it gave rise to more backscattering than any other programs. Comparing MD method 3 and TRIM, we can say that they agree very well in general. Some particles are transmitted when we use method 3, whereas none is transmitted in case of TRIM. This shows that channeling does occur when we consider the molecular dynamics model, but it is not very significant. The ratio of transmitted particles to the total number of particles is very small.

4.3 Channeling

We choose the bcc crystal iron for our investigation of channeling. We choose several crystallographic directions as incident directions of the particles. Other than the change in direction, the properties of the incoming muons are the same as before. The following table summarizes our findings.

Crystallographic Direction	Transmitted	Backscattered	Average Depth (Å)
[100]	9	16	69.3846
[110]	4	58	47.3662
[111]	27	71	89.2488

Table 4.2. Comparison of Crystallographic directions that are likely to give rise to channeling of muons.

The directions are given using Miller indices. Due to the symmetry of a bcc crystal, we may substitute the direction [110] with any of [101], [1$\bar{1}$0] and [10$\bar{1}$] etc. So we should treat the result for [110] similar to the results produced for any of the other directions mentioned. Along [100], there is a little bit of channeling as 9 particles are transmitted. Channeling is least through the [110] direction because the average depth reached is the lowest. However, incident beams in the [111] direction are more likely to channel as suggested by the higher number of transmitted particles and higher average depth.

The following histograms compare the distribution produced for [100] direction with those for [110] and [111].

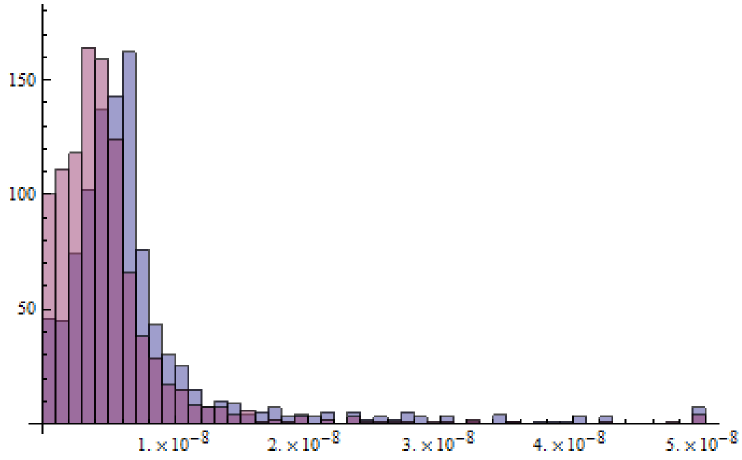


Figure 4.1. Stopping distributions for muons incident at [100] and [110] directions. The transparent histogram represents muons at [110] direction.

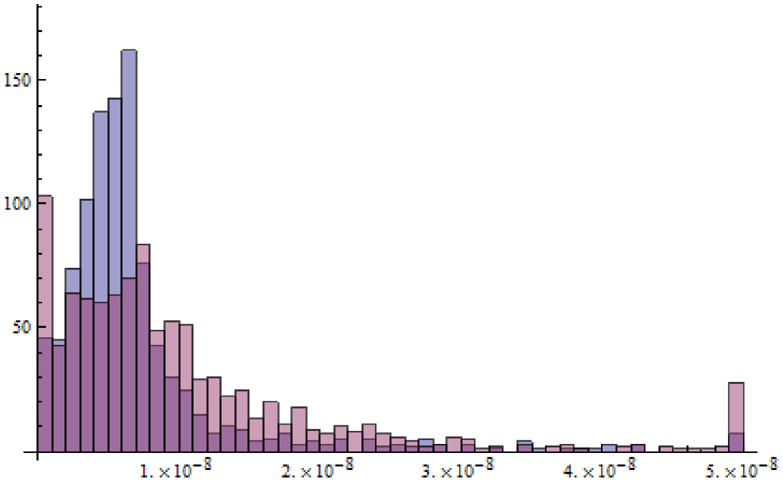


Figure 4.2. Stopping distributions for muons incident at [100] and [111] directions. The transparent histogram represents muons at [111] direction.

The depth is given in meters in both the graphs. As seen in figure 4.1, the peak of the distribution for [100] muons is located at a higher depth. On the other hand, muons incident at [111] angle does not have a sharp peak in their stopping distribution (figure 4.2). After around 100 Å (1.0×10^{-8} m), they dominate over the muons coming in at [100] direction in terms of reaching higher depth. This suggests that channeling is more prominent for the muons incident at [111] direction.

4.4 Conclusion

Based on the comparison of different programs, we can state that accurate molecular dynamics methods (in our case, method 3) do not provide significantly different results from those given by the BCA programs. The brute force recoil interaction approximation MD method (method 3) provided good results, whereas the discrete approximations were not very accurate. In general, we have confirmed that channeling of muons does occur, but we have also established that it does not have a very significant effect on the stopping distribution. Since MD programs take longer computation time, we have good reasons to use BCA programs for our needs. Nonetheless, MD programs can be useful in the investigation of channeling of muons in complex crystal structures.

4.5 Future Work

MUSCLE can be used to investigate the effect of channeling in multi layered, complex crystal structures; something we could not do due to time constraints. The MD programs can be changed to do parallel processing in order to take advantage of multiple CPU cores present in most computers nowadays. An attempt is taken to include parallelism in MUSCLE using the MPI (Message Passing Interface) library, but it is not yet finished. Simulation time can be greatly reduced once the parallel version is complete.

Bibliography

Zeigler, J. F. The Stopping and Range of Ions in Solids. New York: Pergamon Press Inc., 1985.

Nagamine, Kanetada. Introductory Muon Science. Cambridge: Cambridge University Press, 2003.

Eckstein, Wolfgang. Computer Simulation of Ion-Solid Interactions. Springer, 1991.

Dubman, M. *et. al.*, Low Energy μSR and SQUID Evidence of Magnetism in Highly Oriented Pyrolytic Graphite, Journal of Magnetism ad Magnetic Materials 322 (2009) 1228 - 1231.

Nordlund, K. Molecular Dyanamics for Ion Beam Analysis, Nuclear Instruments and Methods in Physics Research B 266 (2008) 1886–1891.

Nordlund, K. Molecular Dynamics Simulation of Ion Ranges in the 1 - 100 keV Energy Range, Comp. Mat. Sci 3, 1995 (448 -).

Appendix A

The appendix contains the simulation code we have written in Mathematica and C++ for the following programs:

- MD method 3 (brute force MD recoil interaction approximation)
- MD method 1 (Discretization method 1)
- MD method 2 (Discretization method 2)
- Evaluation of number of blocks in simulation cell
- Evaluation of Born-Mayer potential for different elements.
- Binary Collision Approximation neighbor selection code (as described in section 2.6).

MUSCLE: Molecular Dynamics Method #3 (Brute Force Method)

Mathematica Code

```
data =
  ReadList["C:\\Users\\Saquib\\Documents\\sp\\Senior_project_2nd\\scoef1.dat",
   Number, RecordLists -> True];

getAtomProperties[z_] := Module[
  {property = {}},
  property = data[[z]];
  Return[property]
 ]

getAtomProperties[6]

{6, 12, 12., 12.011, 2.2662, 11.364, 1., 1.03}

(*Proton Stopping Coefficients*)
getPCoef[z_] := Module[
  {pcoef = {}},
  For[i = 2, i <= Length[data[[z + 92]]], i++,
   pcoef = AppendTo[pcoef, data[[z + 92]][[i]]]
  ];
  Return[pcoef]
 ]

getPCoef[8]

{0.75253, 0.0050314, 4.0824, 0.30067, 2455.8, 1.0181, 5069.7, 0.017426}
```

```
(*Calculate electronic stopping*)
getSe[z_, eekev_] := Module[
  {pcoef = {}, se = {}, m1 = 0.114, e0 = 0.0, e = 0.0, PEO = 25.0, pe = 0.0,
   sl = 0.0, sh = 0.0, sed = 0.0, velpwr = 0.45, atrho = 0.0, dummy = {}},
  pcoef = getPCoef[z];
  dummy = getAtomProperties[z];
  atrho = dummy[[6]] * 10^22;
  (*Print[pcoef];*)
  e = eekev / m1; (* per atm. mass unit? *)
  pe = Max[PEO, e];
  sl = pcoef[[1]] * (pe^(pcoef[[2]])) + pcoef[[3]] * (pe^pcoef[[4]]);
  sh = (pcoef[[5]] / (pe^pcoef[[6]])) * Log[(pcoef[[7]] / pe) + pcoef[[8]] * pe];
  (*sh=pcoef[[5]]*Log[(pcoef[[7]]/pe)+pcoef[[8]]*pe]*(pe^pcoef[[6]]);*)
  sed = ((sl * sh) / (sl + sh));
  If[e > PEO,
   Return[sed * atrho * 10^-23]
   ,
   If[z ≤ 6, velpwr = 0.25, velpwr = 0.45];
   (*Print[sed," ",(e/PEO)^velpwr];*)
   sed = sed * ((e / PEO)^velpwr);
   Return[sed * atrho * 10^-23]
  ]
 ]

t = getSe[26, 1]

10.7606
```

```
getEStopList[z_, eekev_] := Module[
  {pcoef = {}, se = {}, m1 = 0.114, e0 = 0.0, e = 0.0, PEO = 25.0, pe = 0.0,
   sl = 0.0, sh = 0.0, sed = 0.0, velpwr = 0.45, atrho = 0.0, dummy = {}},
  pcoef = getPCoef[z];
  dummy = getAtomProperties[z];
  atrho = dummy[[6]] * 10^22;
  If[eekev < 10^-10,
   For[i = 1, i ≤ 1000, i++,
    se = AppendTo[se, 0]
   ];
   Return[se]
  ];
  e0 = 0.001 * eekev / m1;
  For[i = 1, i ≤ 1000, i++,
   e = e0 * i;
   pe = Max[PEO, e];
   sl = pcoef[[1]] * (pe^pcoef[[2]]) + pcoef[[3]] * (pe^pcoef[[4]]);
   sh = (pcoef[[5]] / (pe^pcoef[[6]])) * Log[(pcoef[[7]] / pe) + pcoef[[8]] * pe];
   sed = ((sl * sh) / (sl + sh));
   If[e > PEO,
    se = AppendTo[se, sed * atrho * 10^-23]
    ,
    If[z ≤ 6, velpwr = 0.25, velpwr = 0.45];
    sed = sed * ((e / PEO)^velpwr);
    se = AppendTo[se, sed * atrho * 10^-23]
   ]
  ];
  Return[se]
 ]


es = getEStopList[26, 0.5];
es[[1000]]
7.87725
```

```
(*Generate a list of electronic stopping list*)
getEStopping[z_, e_, e0kev_, estopList_] := Module[
  {ie = 0, see = 0.0},
  ie = Round[(e / e0kev)];
  If[ie > 1000,
   (*Print[">1000th element doesn't exist, using the 1000th value for -> e = ",
     e," eV, e0kev = ",e0kev];*)
   (*Print[">1000th element doesn't exist, using the getSe[]
       function for -> e = ",e," eV, e0kev = ",e0kev];*)
   Return[getSe[z, e / 1000]]
  ];
  see = estopList[[ie]];
  If[e < e0kev,
   see = estopList[[1]] * Sqrt[e / e0kev]
  ];
  Return[see]
 ]
```

```
getEStopping[26, 500, 0.5, es]
```

```
7.87725
```

```
(*Lindhard Scharff stopping*)
LSStopping[z1_, z2_, eev_] := Module[
  {m1 = 0.114, e0 = 0.0, se = 0.0, dummy = {}, atrho = 0.0},
  dummy = getAtomProperties[z2];
  atrho = dummy[[6]] * 10^22;
  e0 = eev / m1;
```

$$se = 1.21 * \frac{z1^{7/6} * z2}{\left(z1^{2/3} + z2^{2/3}\right)^{3/2}} * \sqrt{\frac{eev}{m1}};$$

```
  se = se * atrho * (10^-23);
  Return[se / 10](* /10 to make it ev/ang from ev/nm -
   see Ion solid interaction (Nastasy, Mayer) pg 110 sample calculations*)
 ]
```

```
LSStopping[1, 26, 1000]
```

```
8.1769
```

```
(*basic constants*)
kg = 1.0;
sec = 1.0;
m = 1.0;
ang = 10^-10 * m;
J = (kg * m^2) / sec^2;
eV = (1.6 * 10^(-19)) * J
MeV = eV * 10^6
c = (3.0 * 10^8) m * sec^-2
```

```
1.6 × 10^-19

1.6 × 10^-13

3. × 10^8
```

```
(*preliminaries*)

muonMass = 105.65836668 * MeV / c^2
atomMass = 0.055847 / (6.02 * 10^23) * kg
latticeConstant = 2.87 * ang
totalVolume = latticeConstant * latticeConstant * latticeConstant
Zmuon = 1
ZFe = 26
ZC = 6

ToeV[e_] := e / (1.6 * 10^-19)

FromeV[e_] := e * (1.6 * 10^-19)
```

```
(*neighbor atoms in the unit cube*)
lx = {latticeConstant / 2, 0, 0}
ly = {0, latticeConstant / 2, 0}
lz = {0, 0, latticeConstant / 2}
neighborIonsBCC = List[];
neighborIonsBCC = AppendTo[neighborIonsBCC, {0, 0, 0}];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * ly];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * ly + 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * ly];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * ly + 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, -ly + lx + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 3 * ly + lx + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly - lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly + 3 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, -lx + ly + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 3 * lx + ly + lz];
topLeft = 2 * lz
topright = 2 * ly + 2 * lz
bottomLeft = {0, 0, 0};
bottomRight = 2 * ly;
topLeft[[2]]


(*MD*)
Clear[distribution, d, dList, allDList, oldd, dbox,
  vv, oldvv, vvdir, trackHistory, selist, vmaglist, ke, F];
AbsoluteTiming[
 numMuons = 50;
 TotalDepth = 600 * ang;
 en = 500 * eV; (*energy in ev*)
 vthold = √2 * 5 * eV / muonMass
   ethold = 5 * eV;
 dt = 0.1 * latticeConstant * √muonMass / (2 * en) * sec; (*time step*)
 Print[en, " ", ethold, " ", dt];

 delv = 0.0;
 ls = 0.0;
 se = 0.0;
 currE = 0.0;
 vmag = 0.0;
 currEev = 0.0;
 numTransmission = 0;
```

```
numBackscatter = 0;

reducedv = {};
vvdir = {};
trackHistory = {};
selist = {};
vmaglist = {};
ke = {0.0, 0.0, 0.0};
distribution = List[];
depthList = {};
d = {};
dList = List[];
allDList = List[];
oldd = List[];
dbox = List[];
vv = {};
oldvv = List[];
F = {};

(*File Operation*)
f =
 OpenWrite["C:\\Users\\Saquib\\Documents\\sp\\Senior_project_2nd\\raw_data.txt",
  FormatType → OutputForm ];

(*Potential properties*)
(*Born-Mayer parameters*)
Abm = 52.0 * (Zmuon * ZFe)^(3/4) * eV;
abm = 0.219 * ang;

Monitor[
 For[nh = 1, nh ≤ numMuons, nh = nh + 1,
  numScatter = 0;
  transmitted = False;
  Clear[dList, selist, vmaglist, meanvSq, F];
  dList = List[];
  selist = {};
  vmaglist = {};
  meanvSq = {0.0, 0.0, 0.0};

  (*generate random v*)
  currE = en;
  diry = RandomReal[{0.0, 0.5}];
  dirz = RandomReal[{0.0, 0.5}];
  (*
  diry=0.0;
```

```
dirz=0.0;
*)
vr = √(2 * en / muonMass) ;
vy = vr * diry;
vz = vr * dirz;
vx = √(vr^2 - (vy^2 + vz^2)) ;

vv = {vx, vy, vz};
oldvv = {vx, vy, vz};

d = {0.0, RandomReal[{0.0, latticeConstant}],
  RandomReal[{0.0, latticeConstant}]};

dpdt = d;
dmdt = dpdt;
dbox = d;

F = {};

currT = 0.0;
count = 0;

minE = en;

(*
Print["Initializing Muon #",n,
 " velocity vector: ",vv, " vr: ",vr, " Threshold vel.: ",vthold];
Write[f,"Initializing Muon #",n," velocity vector: ",
 vv, " vr: ",vr, " Threshold vel.: ",vthold];
*)

(*loop until energy drops below threshold*)
While[(currE > ethold) && (d[[1]] < TotalDepth),

 If[count > 2,

  For[k = 1, k ≤ 3, k++,
   meanvSq[[k]] = ((dList[[count]][[k]] - dList[[count - 2]][[k]]) / (2 * dt))^2
  ];

  currE = (muonMass / 2) * Total[meanvSq];

  If[minE > currE,
   minE = currE
  ]
 ];
```

```
(*Calculate the force acting on the muon*)
fx = 0.0;
fy = 0.0;
fz = 0.0;

(*Go through each neighbor atom and add their potential contribution*)
For[n = 1, n ≤ Length[neighborIonsBCC], n = n + 1,
 currNeighbor = neighborIonsBCC[[n]];

 dist = √((dbox[[1]] - currNeighbor[[1]])^2 +
      (dbox[[2]] - currNeighbor[[2]])^2 + (dbox[[3]] - currNeighbor[[3]])^2);

 (*Born-Mayer potential*)
 dVdr = (-Abm / abm) * ⅇ^(-dist/abm);

 from = currNeighbor - dbox;

 with = lx;
 (*cs=Cos[VectorAngle[from,with]];*)
 cs = (Dot[from, with]) / ((Norm[from]) * (Norm[with]));
 fx = fx + dVdr * cs;

 with = ly;
 (*cs=Cos[VectorAngle[from,with]];*)
 cs = (Dot[from, with]) / ((Norm[from]) * (Norm[with]));
 fy = fy + dVdr * cs;

 with = lz;
 (*cs=Cos[VectorAngle[from,with]];*)
 cs = (Dot[from, with]) / ((Norm[from]) * (Norm[with]));
 fz = fz + dVdr * cs;
];

F = {-fx, -fy, -fz};

For[k = 1, k ≤ 3, k++,
 (*v(t+dt/2) = *)
 vv[[k]] = oldvv[[k]] + dt * F[[k]] / muonMass
];

(*Incorporate estopping, reduce velocity*)
vvdir = Normalize[vv];

(* Zeigler stopping *)
currEev = ToeV[currE];
```

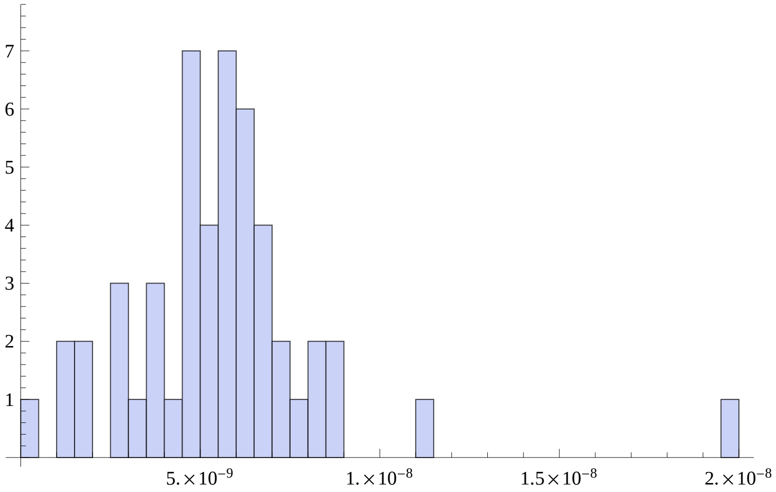
```
(*seev=getEStopping[ZFe,currEev,1.0,es];(*in eV/ang*) *)
seev = getSe[26, currEev];


(*
(*LS stopping *)
currEev=ToeV[currE];
seev=LSStopping[1,26,currEev];
*)

(*selist=AppendTo[selist,se];*)
(*Print[se];*)
se = seev * (1.6 * 10^(-19)) / (10^-10);

(*reduce velocity*)
delv = dt * se / muonMass;
vmag = Norm[vv];
vv = (vmag - delv) * vvdir;

(*new d *)
For[k = 1, k ≤ 3, k++,
 (*v(t+dt/2) = *)
 dpdt[[k]] = d[[k]] + dt * vv[[k]]
];

dList = AppendTo[dList, dpdt];

If[dpdt[[1]] < 0,
 numBackscatter = numBackscatter + 1;
 (*Print["Particle backscattered"];*)
 Break[]
];

(*abs[ls] might have solved the problem of getting stuck at a place*)
ls = Abs[Norm[dpdt] - Norm[d]];

(*
Print["e = ",currE," se = ",se," se(eV/ang) = ",getEStopping[ZC,ToeV[currE],
   1.0,es]," ls = ",ls," del v. = ",delv," Norm[vv] = ",Norm[vv]];
*)

(*Check Transmission*)
If[dpdt[[1]] ≥ TotalDepth,
 transmitted = True;
 numTransmission = numTransmission + 1;
 Break[];
```

```
   ];

   (*find the position in the simulation cell using modulus*)
   dbox[[1]] = Mod[dpdt[[1]], latticeConstant];
   dbox[[2]] = Mod[dpdt[[2]], latticeConstant];
   dbox[[3]] = Mod[dpdt[[3]], latticeConstant];

   currT = currT + dt;
   numScatter = numScatter + 1;

   count = count + 1;

   d = dpdt;

   (*new v = old v - del v + force in new block,
   which will be calculated in the next loop*)
   oldvv = vv;

  ];

  (*Print["final velocity: ",Norm[vv]];*)
  (*Print["final depth: ",d[[1]]];
  Print["Number of scatters: ",numScatter];*)
  (*Print["x: ",d[[1]]," y: ",d[[2]]," z: ",d[[3]]];*)
  (*save coord. for distribution data*)
  If[transmitted == True,
   trackHistory = AppendTo[trackHistory, {n, vmaglist, selist}]
  ];
  distribution = AppendTo[distribution, d];
  depthList = AppendTo[depthList, d[[1]]];
  allDList = AppendTo[allDList, dList];
 ]
 ,
 {"Muon #", nh, "distance: ", d,
  ProgressIndicator[Norm[d], {0, TotalDepth}], "Estopping: ", seev,
  "Resultant Velocity: ", ProgressIndicator[Norm[vv], {0, vr}],
  "Current Energy:", ProgressIndicator[currE, {0, en * 2}], "Min. Energy:", minE}
];
(*distribution*)
(*Average depth*)
Print["Average range = ", Total[distribution] / numMuons];
(*Transmission and backscattering*)
Print[numTransmission, " particles transmitted, ",
 numBackscatter, " particles backscattered"];
 Close[f]
]
```

```
Average range = {5.60003 × 10^-9, 1.2745 × 10^-9, 1.82551 × 10^-9}

0 particles transmitted, 0 particles backscattered
```

```
(*Code to plot all the trajectories together*)
Manipulate[
 Show[
  Flatten[
   Table[
    Graphics3D[{{ColorData[3, "ColorList"], Line[allDList[[n]]]},
      {Red, PointSize[Large], Point[allDList[[n]][[currpt]]]}},
     Axes → True, PlotRange → Automatic,
     AxesLabel → {x, y, z},
     PlotRangePadding → None,
     FaceGrids → None]
    , {n, nMax}]
  ]
 ],
 {{currpt, 1}, 1, Length[allDList[[nMax]]], 2},
 {{zoom, Max[allDList]}, Min[allDList], Max[allDList], 1},
 {{nMax, numMuons}, 1, numMuons, 1}
]
```

```
(*Plot depth distribution*)
Histogram[depthList, 50]
```



MUSCLE: Molecular Dynamics Method #1 (Discretization method 1)

Mathematica code

```
ToeV[e_] := e / (1.6 * 10^-19)

FromeV[e_] := e * (1.6 * 10^-19)

data =
  ReadList["C:\\Users\\Saquib\\Documents\\sp\\Senior_project_2nd\\scoef1.dat",
   Number, RecordLists -> True];

getAtomProperties[z_] := Module[
  {property = {}},
  property = data[[z]];
  Return[property]
 ]

getAtomProperties[6]

{6, 12, 12., 12.011, 2.2662, 11.364, 1., 1.03}

getPCoef[z_] := Module[
  {pcoef = {}},
  For[i = 2, i <= Length[data[[z + 92]]], i++,
   pcoef = AppendTo[pcoef, data[[z + 92]][[i]]]
  ];
  Return[pcoef]
 ]

getPCoef[8]

{0.75253, 0.0050314, 4.0824, 0.30067, 2455.8, 1.0181, 5069.7, 0.017426}
```

```
getSe[z_, eekev_] := Module[
  {pcoef = {}, se = {}, m1 = 0.114, e0 = 0.0, e = 0.0, PEO = 25.0, pe = 0.0,
   sl = 0.0, sh = 0.0, sed = 0.0, velpwr = 0.45, atrho = 0.0, dummy = {}},
  pcoef = getPCoef[z];
  dummy = getAtomProperties[z];
  atrho = dummy[[6]] * 10^22;
  (*Print[pcoef];*)
  e = eekev / m1; (* per atm. mass unit? *)
  pe = Max[PEO, e];
  sl = pcoef[[1]] * (pe^(pcoef[[2]])) + pcoef[[3]] * (pe^pcoef[[4]]);
  sh = (pcoef[[5]] / (pe^pcoef[[6]])) * Log[(pcoef[[7]] / pe) + pcoef[[8]] * pe];
  (*sh=pcoef[[5]]*Log[(pcoef[[7]]/pe)+pcoef[[8]]*pe]*(pe^pcoef[[6]]);*)
  sed = ((sl * sh) / (sl + sh));
  If[e > PEO,
   Return[sed * atrho * 10^-23]
   ,
   If[z ≤ 6, velpwr = 0.25, velpwr = 0.45];
   (*Print[sed," ",(e/PEO)^velpwr];*)
   sed = sed * ((e / PEO)^velpwr);
   Return[sed * atrho * 10^-23]
  ]
 ]

t = getSe[6, 0.1]

5.75291
```

```
getEStopList[z_, eekev_] := Module[
  {pcoef = {}, se = {}, m1 = 0.114, e0 = 0.0, e = 0.0, PEO  = 25.0, pe = 0.0,
   sl = 0.0, sh = 0.0, sed = 0.0, velpwr = 0.45, atrho = 0.0, dummy = {}},
  pcoef = getPCoef[z];
  dummy = getAtomProperties[z];
  atrho = dummy[[6]] * 10^22;
  If[eekev < 10^-10,
   For[i = 1, i ≤ 1000, i++,
    se = AppendTo[se, 0]
   ];
   Return[se]
  ];
  e0 = 0.001 * eekev / m1;
  For[i = 1, i ≤ 1000, i++,
   e = e0 * i;
   pe = Max[PEO, e];
   sl = pcoef[[1]] * (pe^pcoef[[2]]) + pcoef[[3]] * (pe^pcoef[[4]]);
   sh = (pcoef[[5]] / (pe^pcoef[[6]])) * Log[(pcoef[[7]] / pe) + pcoef[[8]] * pe];
   sed = ((sl * sh) / (sl + sh));
   If[e > PEO,
    se = AppendTo[se, sed * atrho * 10^-23]
    ,
    If[z ≤ 6, velpwr = 0.25, velpwr = 0.45];
    sed = sed * ((e / PEO)^velpwr);
    se = AppendTo[se, sed * atrho * 10^-23]
   ]
  ];
  Return[se]
 ]

es = getEStopList[26, 1];
es[[1000]]
10.7606
```

```
getEStopping[z_, e_, e0kev_, estopList_] := Module[
  {ie = 0, see = 0.0},
  ie = Round[(e / e0kev)];
  If[ie > 1000,
   (*Print[">1000th element doesn't exist, using the 1000th value for -> e = ",
     e," eV, e0kev = ",e0kev];*)
   (*Print[">1000th element doesn't exist, using the getSe[]
       function for -> e = ",e," eV, e0kev = ",e0kev];*)
   Return[getSe[z, e / 1000]]
  ];
  see = estopList[[ie]];
  If[e < e0kev,
   see = estopList[[1]] * Sqrt[e / e0kev]
  ];
  Return[see]
 ]
```

```
getEStopping[26, 1000, 1, es]
```

```
10.7606
```

```
(*basic constants*)
kg = 1;
sec = 1;
m = 1;
ang = 10^(-10) * m;
J = (kg * m^2) / sec^2
eV = (1.6 * 10^(-19)) * J
MeV = eV * 10^6
c = (3.0 * 10^8) m * sec^-2
```

```
1
```

1.6×10^{-19}

1.6×10^{-13}

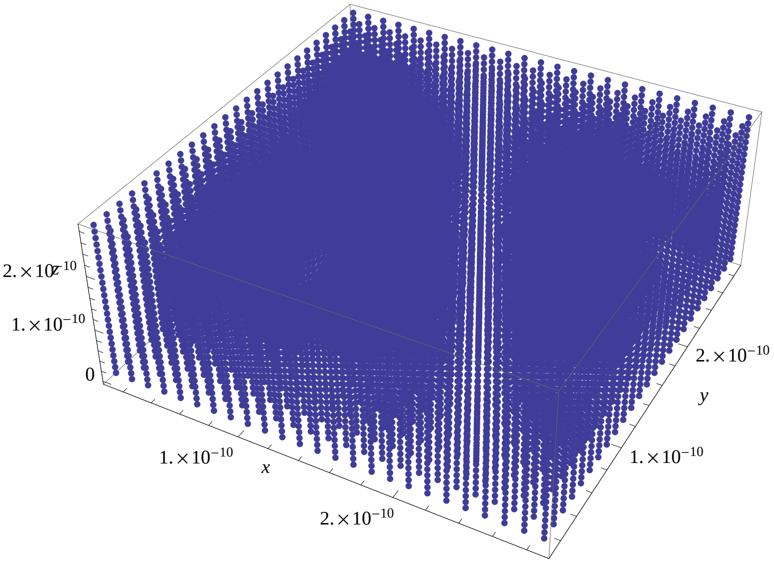
$3.\times10^{8}$

```
(*preliminaries*)
muonMass = 105.65836668 * MeV / c^2
atomMass = 0.055847 / (6.02 * 10^23) * kg
latticeConstant = 2.87 * ang
totalVolume = latticeConstant * latticeConstant * latticeConstant
xBound = 25
yBound = 25
zBound = 25
volStep  = (latticeConstant / xBound)
numUnitCube = (latticeConstant / volStep)^3
Zmuon = 1
ZFe = 26
ZC = 6
```

```
(*neighbor atoms in the unit cube*)
lx = {latticeConstant / 2, 0, 0}
ly = {0, latticeConstant / 2, 0}
lz = {0, 0, latticeConstant / 2}
neighborIonsBCC = List[];
neighborIonsBCC = AppendTo[neighborIonsBCC, {0, 0, 0}];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * ly];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * ly + 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * ly];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * ly + 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, - ly + lx + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 3 * ly + lx + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly - lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly + 3 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, - lx + ly + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 3 * lx + ly + lz];
topLeft = 2 * lz
topright = 2 * ly + 2 * lz
bottomLeft = {0, 0, 0};
bottomRight = 2 * ly;
topLeft[[2]]
```

```
{1.435 × 10^-10, 0, 0}

{0, 1.435 × 10^-10, 0}

{0, 0, 1.435 × 10^-10}

{0, 0, 2.87 × 10^-10}

{0, 2.87 × 10^-10, 2.87 × 10^-10}

0
```

```
(*Block coordinates inside the simulation cell*)
Clear[sc, sc2]
(*sc={0,0,0};*)
sc = List[];
sc2 = List[];
For[xc = 1, xc ≤ xBound, xc = xc + 1,
 For[zc = 1, zc ≤ zBound, zc = zc + 1,
  For[yc = 1, yc ≤ yBound, yc = yc + 1,
   sc = AppendTo[sc, {topLeft[[1]] + xc * volStep,
      topLeft[[2]] + yc * volStep, bottomLeft[[3]] + zc * volStep}]
  ]
 ]
]
For[i = 1, i ≤ Length[sc], i = i + 1,
 sc2 = AppendTo[sc2,
   {sc[[i]][[1]] - volStep / 2, sc[[i]][[2]] - volStep / 2, sc[[i]][[3]] - volStep / 2}]
]
ListPointPlot3D[sc, AxesLabel → {x, y, z}]
sc;
Length[sc2]
sc2;
(*,PlotRange→{-1.435,1.435},DataRange→{{0,2.87},{-1.435,1.435}}]*)
```



```
15 625
```

```
(*Returns the index of the block*)
detNode[num_, vstep_] := Module[{q = 0, r = 0, eps = (10^(-5)) * ang},
  q = Quotient[num, vstep];
  (*r=Mod[num,lattConst];*)
  r = num - q * vstep;
  (*Print[q," ",r];*)
  Which[
   r > eps, Return[q + 1],
   r ≥ 0 && q > 0, Return[q],
   r ≥ 0 && q == 0, Return[1],
   True, Print["Unable to determine the cell"]; Print[q, " ", r]; Return[-1]
  ]
 ]
(*Determine the block index based on x, y, and z values of a point*)
nodeCoord[x_, y_, z_, spacing_] := Module[{xk = 0, yk = 0, zk = 0},
  xk = detNode[x, spacing];
  yk = detNode[y, spacing];
  zk = detNode[z, spacing];
  If[(yk + (zk - 1) * zBound + (xk - 1) * (xBound)^2) > xBound * yBound * zBound,
   Print["Went Over ", xk, " ", x, " ", yk, " ", y, " ", zk, " ", z]
  ];
  Which[
   xk > 0 && yk > 0 && zk > 0, Return[yk + (zk - 1) * zBound + (xk - 1) * (xBound)^2],
   xk == 0 && yk == 0 && zk == 0, Return[1],
   xk == 0 && yk == 0 && zk == 1, Return[zBound + 1],
   yk == 0 && zk == 0 && xk == 1, Return[xBound^2 + 1]
  ]
 ]
(*Potential and force field Calculations*)

(*Born-Mayer parameters*)
Abm = 52.0 * (Zmuon * ZFe)^(3/4) * eV
abm = 0.219 * ang

(*Morse parameters*)
(*For Fe*)
DeV = 0.4174 * eV
alphap = 1.3885 / ang
r0 = 2.845 * ang

Clear[Vlist, V, forceList, vPlot]
Vlist = List[];
V = List[];
forceList = List[];
For[i = 1, i ≤ Length[sc2], i = i + 1,
 Clear[V];
 V = List[];
 (*Go through each neighbor atom and add their potential contribution*)
 For[n = 1, n ≤ Length[neighborIonsBCC], n = n + 1,
  currPt = sc2[[i]];
  currNeighbor = neighborIonsBCC[[n]];

  dist = √((currPt[[1]] - currNeighbor[[1]])^2 +
      (currPt[[2]] - currNeighbor[[2]])^2 + (currPt[[3]] - currNeighbor[[3]])^2);
  (*Born-Mayer potential*)
  V = AppendTo[V, Abm * ⅇ^(-dist/abm)]

  (*Morse Potential*)
  (*V=AppendTo[V, (DeV*ⅇ^((-2*alphap*(dist-r0)))-2*DeV*ⅇ^((-alphap*(dist-r0))))]*)
 ];

 sumV = Total[V];
 Vlist = AppendTo[Vlist, sumV];
]

Length[Vlist]

(*Now to force field*)
For[i = 1, i ≤ Length[sc2], i = i + 1,
 currx = sc2[[i]][[1]];
 curry = sc2[[i]][[2]];
 currz = sc2[[i]][[3]];
 (*Determine x-Boundary*)
 If[sc2[[i]][[1]] + volStep > latticeConstant,
  (*deeper-x-boundary*)
  fx = (Vlist[[i]] - Vlist[[nodeCoord[currx - volStep, curry, currz, volStep]]]) /
     (volStep);
  ,
  If[sc2[[i]][[1]] - volStep < 0,
   (*nearer-x-boundary*)
   fx = (Vlist[[nodeCoord[currx + volStep, curry, currz, volStep]]] - Vlist[[i]]) /
      (volStep);
   ,
   (*Not in the x-Boundary*)
   fx = (Vlist[[nodeCoord[currx + volStep, curry, currz, volStep]]] -
        Vlist[[nodeCoord[currx - volStep, curry, currz, volStep]]]) / (2 * volStep);
  ]
 ];

 (*Determine y-Boundary*)
 If[sc2[[i]][[2]] + volStep > latticeConstant,
  (*right-y-boundary*)
  fy = (Vlist[[i]] - Vlist[[nodeCoord[currx, curry - volStep, currz, volStep]]]) /
     (volStep);
  ,
  If[sc2[[i]][[2]] - volStep < 0,
   (*left-y-boundary*)
   fy = (Vlist[[nodeCoord[currx, curry + volStep, currz, volStep]]] - Vlist[[i]]) /
      (volStep);
   ,
   (*Not in the y-Boundary*)
   fy = (Vlist[[nodeCoord[currx, curry + volStep, currz, volStep]]] -
        Vlist[[nodeCoord[currx, curry - volStep, currz, volStep]]]) / (2 * volStep);
  ]
 ];

 (*Determine z-Boundary*)
 If[sc2[[i]][[3]] + volStep > latticeConstant,
  (*upper-z-boundary*)
  fz = (Vlist[[i]] - Vlist[[nodeCoord[currx, curry, currz - volStep, volStep]]]) /
     (volStep);
  ,
  If[sc2[[i]][[3]] - volStep < 0,
   (*lower-z-boundary*)
   fz = (Vlist[[nodeCoord[currx, curry, currz + volStep, volStep]]] - Vlist[[i]]) /
      (volStep);
   ,
   (*Not in the z-Boundary*)
   fz = (Vlist[[nodeCoord[currx, curry, currz + volStep, volStep]]] -
        Vlist[[nodeCoord[currx, curry, currz - volStep, volStep]]]) / (2 * volStep);
  ]
 ];
 If[(Not[NumberQ[fx]]) || ( Not[NumberQ[fy]]) || (Not[NumberQ[fz]]),
  Print["Not Numeric in Node ", i]];
 forceList = AppendTo[forceList, {fx, fy, fz}]
]

Length[Vlist]

Length[forceList]
vPlot = List[];
For[i = 1, i ≤ Length[forceList], i = i + 1,
 vPlot = AppendTo[vPlot, {sc2[[i]], forceList[[i]]}];
]
```

```
(*Code to plot the force field vectors*)
Manipulate[
 ListVectorPlot3D[vPlot, VectorScale → {Automatic, Automatic, Automatic},
  PlotRange → {{lox, hix}, {loy, hiy}, {loz, hiz}}, AxesLabel → {x, y, z}],
 {{lox, 0.0}, 0.0, latticeConstant / 2, 0.2 * ang},
 {{hix, 2.87 * ang}, latticeConstant / 2, latticeConstant, 0.2 * ang},
 {{loy, 0.0}, 0.0, latticeConstant / 2, 0.2 * ang},
 {{hiy, 2.87 * ang}, latticeConstant / 2, latticeConstant, 0.2 * ang},
 {{loz, 0.0}, 0.0, latticeConstant / 2, 0.2 * ang},
 {{hiz, 2.87 * ang}, latticeConstant / 2, latticeConstant, 0.2 * ang}
]


(*Main molecular dynamics simulation*)
Clear[distribution, d, dList, allDList, oldd,
  dbox, vv, oldvv, vvdir, trackHistory, selist, vmaglist, ke];
AbsoluteTiming[
 numMuons = 1000;
 TotalDepth = 1000 * ang;
 en = 500 * eV; (*energy in ev*)
 Print[ToeV[en]];
 vthold = √2 * 5 * eV / muonMass ;
 ethold = 5 * eV;
 dt = 0.05 * latticeConstant * √muonMass / (2 * en) * sec; (*time step*)
 Print[en, " ", ethold, " ", dt];

 delv = 0.0;
 ls = 0.0;
 se = 0.0;
 currE = 0.0;
 vmag = 0.0;
 currEev = 0.0;
 numTransmission = 0;
 numBackscatter = 0;

 reducedv = {};
 vvdir = {};
 trackHistory = {};
 selist = {};
 vmaglist = {};
 ke = {0.0, 0.0, 0.0};
 distribution = List[];
 depthList = {};
 d = {};
 dList = List[];
```

```
allDList = List[];
oldd = List[];
dbox = List[];
vv = {};
oldvv = List[];

(*File Operation*)
f =
 OpenWrite["C:\\Users\\Saquib\\Documents\\sp\\Senior_project_2nd\\raw_data.txt",
  FormatType → OutputForm ];

Monitor[
 For[n = 1, n ≤ numMuons, n = n + 1,
  numScatter = 0;
  transmitted = False;
  Clear[dList, selist, vmaglist];
  dList = List[];
  selist = {};
  vmaglist = {};
  meanvSq = {0.0, 0.0, 0.0};

  (*generate random v*)
  (*dirx=RandomReal[{0.0,1.0}];(*****)*)
  currE = en;
  diry = RandomReal[{0.0, 0.5}];
  dirz = RandomReal[{0.0, 0.5}];
  (*
  diry=0.0;
  dirz=0.0;
  *)
  vr = √2 * en / muonMass ;
  vy = vr * diry;
  vz = vr * dirz;
  vx = √(vr^2 - (vy^2 + vz^2)) ;
  (*choose a cell to start from*)
  initBoxIndex = RandomInteger[{1, yBound * zBound}];
  vv = {vx, vy, vz};
  oldvv = {vx, vy, vz};
  d = sc2[[initBoxIndex]];
  oldd = sc2[[initBoxIndex]];
  dbox = sc2[[initBoxIndex]];
  currT = 0.0;
  count = 0;
  chstep = 0;
  currIndex = initBoxIndex;
```

```
(*
Print["Initializing Muon #",n,
 " velocity vector: ",vv, " vr: ",vr, " Threshold vel.: ",vthold];
Write[f,"Initializing Muon #",n," velocity vector: ",
 vv, " vr: ",vr, " Threshold vel.: ",vthold];
*)

(*loop until energy drops below threshold*)
While[(currE > ethold) && (d[[1]] < TotalDepth),

 For[k = 1, k ≤ 3, k = k + 1,

  d[[k]] = oldd[[k]] + (vv[[k]]) * dt +
    (0.5 * (forceList[[currIndex]][[k]]) * (dt^2)) / muonMass;

  (*old v, or new v (follwing previous loop) see diagram*)
  vv[[k]] = oldvv[[k]] + (forceList[[currIndex]][[k]] * dt) / muonMass;

  (*
  Print["x: ",d[[1]]," y: ",d[[2]]," z: ",d[[3]]," vr: ",Norm[vv],
    " node: ",nodeCoord[dbox[[1]],dbox[[2]],dbox[[3]],volStep],
    " currIndex: ",currIndex];
  *)

  If[Not[NumberQ[vv[[k]]]],
   Print["Not Numeric in Node ", currIndex, " vv[[", k, "]]: ", vv[[k]]]

  ]
 ];

 (*Write[f,"x: ",d[[1]]," y: ",d[[2]]," z: ",
   d[[3]]," vr: ",Norm[vv]," currIndex: ",currIndex];*)

 dList = AppendTo[dList, d];

 If[d[[1]] < 0,
  numBackscatter = numBackscatter + 1;
  (*Print["Particle backscattered"];*)
  Break[]
 ];

 (*Kinetic energy*)
 If[count > 2,

  For[k = 1, k ≤ 3, k++,
   meanvSq[[k]] = ((dList[[count]][[k]] - dList[[count - 2]][[k]]) / (2 * dt))^2
  ];

  currE = (muonMass / 2) * Total[meanvSq];
 ];


(*currE=(1/2)*muonMass*(Norm[vv])^2;*)
 vvdir = Normalize[vv];

 (*abs[ls] might have solved the problem of getting stuck at a place*)
 ls = Abs[Norm[d] - Norm[oldd]];

 currEev = ToeV[currE];
 (*seev=getEStopping[ZFe,currEev,1.0,es];(*in eV/ang*) *)
 seev = getSe[26, currEev];
 selist = AppendTo[selist, se];
 (*Print[se];*)
 se = seev * (1.6 * 10^(-19)) / (10^-10);

 currE = currE - se * ls;

 delv = dt * se / muonMass;
 vmag = Norm[vv];
 vv = (vmag - delv) * vvdir;

 (*
 delv=dt*se/muonMass;
 vvdir=Normalize[vv];
 vmag=Norm[vv];
 vv=(vmag-delv)*vvdir;
 vmaglist=AppendTo[vmaglist,vmag];
 *)

 (*
 Print["e = ",currE," se = ",se," se(eV/ang) = ",getEStopping[ZC,ToeV[currE],
   1.0,es]," ls = ",ls," del v. = ",delv," Norm[vv] = ",Norm[vv]];
 *)

 (*Check Transmission*)
 If[d[[1]] ≥ TotalDepth,
  transmitted = True;
  numTransmission = numTransmission + 1;
  Break[];
 ];
```

```
   dbox[[1]] = Mod[d[[1]], latticeConstant];
   dbox[[2]] = Mod[d[[2]], latticeConstant];
   dbox[[3]] = Mod[d[[3]], latticeConstant];
   (*If[(Not[NumberQ[fx]]) ||( Not[NumberQ[fy]]) || (Not[NumberQ[fz]]),
     Print["Not Numeric in Node ",i]];*)
   currIndex = nodeCoord[dbox[[1]], dbox[[2]], dbox[[3]], volStep];

   If[currIndex == 0,
    Print["CurrIndex is 0 and dbox x, y, z: ",
     dbox[[1]], " ", dbox[[2]], " ", dbox[[3]]]
    (*currIndex=1*)];

   currT = currT + dt;
   numScatter = numScatter + 1;
   count = count + 1;

   oldd = d;

   (*new v = old v - del v + force in new block,
   which will be calculated in the next loop*)
   oldvv = vv;

  ];

  (*Print["final velocity: ",Norm[vv]];*)
  (*Print["final depth: ",d[[1]]];
  Print["Number of scatters: ",numScatter];*)
  (*Print["x: ",d[[1]]," y: ",d[[2]]," z: ",d[[3]]];*)
  (*save coord. for distribution data*)
  If[transmitted == True,
   trackHistory = AppendTo[trackHistory, {n, vmaglist, selist}]
  ];
  distribution = AppendTo[distribution, d];
  depthList = AppendTo[depthList, d[[1]]];
  allDList = AppendTo[allDList, dList];
 ]
 ,
 {"Muon #", n, "distance: ", d,
  ProgressIndicator[Norm[d], {0, TotalDepth}], "Estopping: ", seev,
  "Resultant Velocity: ", ProgressIndicator[Norm[vv], {0, vr}],
  "Curent Energy:", ProgressIndicator[currE, {ethold, en * 2}],
  "Current Block Index: ", ProgressIndicator[currIndex, {1, xBound^3}]}
];
(*distribution*)
(*Average depth*)
```

```
   Print["Average range = ", Total[distribution] / numMuons];
   (*Transmission and backscattering*)
   Print[numTransmission, " particles transmitted, ",
    numBackscatter, " particles backscattered"];

   Close[f]
  ]
```

```
500.

8. × 10^-17 8. × 10^-19 1.55483 × 10^-17

Average range = {8.60232 × 10^-9, 2.73856 × 10^-9, 7.53652 × 10^-10}
```

```
  (*Trajectory plotter code*)
  Manipulate[
   Show[
    Flatten[
     Table[
      Graphics3D[{{ColorData[3, "ColorList"], Line[allDList[[n]]]},
        {Red, PointSize[Large], Point[allDList[[n]][[currpt]]]}},
       Axes → True, PlotRange → Automatic,
       AxesLabel → {x, y, z},
       PlotRangePadding → None,
       FaceGrids → None]
      , {n, nMax}]
    ]
   ],
   {{currpt, 1}, 1, Length[allDList[[nMax]]], 2},
   {{zoom, Max[allDList]}, Min[allDList], Max[allDList], 1},
   {{nMax, numMuons}, 1, numMuons, 1}
  ]
```

MUSCLE: Molecular Dynamics Method #2 (Discretization Method 2)

(Electronic stopping and some other code omitted)

```
(*Potential and force field Calculations*)

(*Born-Mayer parameters*)
Abm = 52.0 * (Zmuon * ZFe)^(3/4) * eV
abm = 0.219 * ang

(*Morse parameters*)
(*For Fe*)
DeV = 0.4174 * eV
alphap = 1.3885 / ang
r0 = 2.845 * ang

Clear[Vlist, V, forceList, vPlot, dVdr]
Vlist = List[];
V = List[];
forceList = List[];
dVdr = {};
For[i = 1, i ≤ Length[sc2], i = i + 1,
 Clear[V, dVdr];
 V = List[];
 dVdr = {};
 fx = 0.0;
 fy = 0.0;
 fz = 0.0;
 (*Go through each neighbor atom and add their potential contribution*)
 For[n = 1, n ≤ Length[neighborIonsBCC], n = n + 1,
  currPt = sc2[[i]];
  currNeighbor = neighborIonsBCC[[n]];

  (*Note: sign? which dir does the vector point to?********)

  dist = √((currPt[[1]] - currNeighbor[[1]])^2 +
      (currPt[[2]] - currNeighbor[[2]])^2 + (currPt[[3]] - currNeighbor[[3]])^2);
  (*Born-Mayer potential*)
  V = AppendTo[V, Abm * e^(-dist/abm)];
  (*dVdr=AppendTo[dVdr,(-Abm/abm)*e^(-dist/abm)];*)
  dVdr = (- Abm / abm) * e^(-dist/abm);

  (*If currneighbor is not equal to currpt then*)(*****)
  If[currNeighbor == currPt,
   Print["caught"];
   (*check where currpt actually is,
   may be important if the particle really falls on this*****)
   currPt = sc2[[i - 1]]
  ];

  from = currNeighbor - currPt;

  with = lx;
  (*cs=Cos[VectorAngle[from,with]];*)
  cs = (Dot[from, with]) / ((Norm[from]) * (Norm[with]));
  fx = fx - dVdr * cs;

  with = ly;
  (*cs=Cos[VectorAngle[from,with]];*)
  cs = (Dot[from, with]) / ((Norm[from]) * (Norm[with]));
  fy = fy - dVdr * cs;

  with = lz;
  (*cs=Cos[VectorAngle[from,with]];*)
  cs = (Dot[from, with]) / ((Norm[from]) * (Norm[with]));
  fz = fz - dVdr * cs;

  If[(Not[NumberQ[fx]]) || ( Not[NumberQ[fy]]) || (Not[NumberQ[fz]]),
   Print["Not Numeric in Node ", i];
   Print[fx, " ", fy, " ", fz];
   Print[dVdr, " ", cs, " ", Norm[from], " ", Norm[with]]
  ]

  (*Morse Potential*)
  (*V=AppendTo[V, (DeV*ⅇ^(-2*alphap*(dist-r0))-2*DeV*ⅇ^(-alphap*(dist-r0)))]*)
 ];

 forceList = AppendTo[forceList, {fx, fy, fz}];
 sumV = Total[V];
 Vlist = AppendTo[Vlist, sumV];
]

Length[Vlist]

Length[forceList]
vPlot = List[];
For[i = 1, i ≤ Length[forceList], i = i + 1,
 vPlot = AppendTo[vPlot, {sc2[[i]], forceList[[i]]}];
]


(*Molecular dynamics method*)
Clear[distribution, d, dList, allDList, oldd,
  dbox, vv, oldvv, vvdir, trackHistory, selist, vmaglist, ke];
AbsoluteTiming[
 numMuons = 5;
 TotalDepth = 1000 * ang;
 en = 500 * eV; (*energy in ev*)
 Print[ToeV[en]];
 vthold = √2 * 5 * eV / muonMass
   ethold = 5 * eV;
 dt = 0.05 * latticeConstant * √muonMass / (2 * en) * sec; (*time step*)
 Print[en, " ", ethold, " ", dt];

 delv = 0.0;
 ls = 0.0;
 se = 0.0;
 currE = 0.0;
 vmag = 0.0;
 currEev = 0.0;
 numTransmission = 0;
 numBackscatter = 0;

 reducedv = {};
 vvdir = {};
 trackHistory = {};
 selist = {};
 vmaglist = {};
 ke = {0.0, 0.0, 0.0};
 distribution = List[];
 depthList = {};
 d = {};
 dList = List[];
 allDList = List[];
 oldd = List[];
 dbox = List[];
 vv = {};
 oldvv = List[];

   (*File Operation*)
   f =
  OpenWrite["C:\\Users\\Saquib\\Documents\\sp\\Senior_project_2nd\\raw_data.txt",
   FormatType → OutputForm ];

 Monitor[
  For[n = 1, n ≤ numMuons, n = n + 1,
   numScatter = 0;
   transmitted = False;
```

```
Clear[dList, selist, vmaglist];
dList = List[];
selist = {};
vmaglist = {};
meanvSq = {0.0, 0.0, 0.0};

(*generate random v*)
(*dirx=RandomReal[{0.0,1.0}];(*****)*)
currE = en;
diry = RandomReal[{0.0, 0.5}];
dirz = RandomReal[{0.0, 0.5}];
(*
diry=0.0;
dirz=0.0;
*)
vr = √(2 * en / muonMass) ;
vy = vr * diry;
vz = vr * dirz;
vx = √(vr^2 - (vy^2 + vz^2)) ;
(*choose a cell to start from*)
initBoxIndex = RandomInteger[{1, yBound * zBound}];
vv = {vx, vy, vz};
oldvv = {vx, vy, vz};
d = sc2[[initBoxIndex]];
oldd = sc2[[initBoxIndex]];
dbox = sc2[[initBoxIndex]];
currT = 0.0;
count = 0;
chstep = 0;
currIndex = initBoxIndex;

(*
Print["Initializing Muon #",n,
 " velocity vector: ",vv, " vr: ",vr, " Threshold vel.: ",vthold];
Write[f,"Initializing Muon #",n," velocity vector: ",
 vv, " vr: ",vr, " Threshold vel.: ",vthold];
*)

(*loop until energy drops below threshold*)
While[(currE > ethold) && (d[[1]] < TotalDepth),

 For[k = 1, k ≤ 3, k = k + 1,

  d[[k]] = oldd[[k]] + (vv[[k]]) * dt +
    (0.5 * (forceList[[currIndex]][[k]]) * (dt^2)) / muonMass;
    (*old v, or new v (follwing previous loop) see diagram*)
  vv[[k]] = oldvv[[k]] + (forceList[[currIndex]][[k]] * dt) / muonMass;

  (*
  Print["x: ",d[[1]]," y: ",d[[2]]," z: ",d[[3]]," vr: ",Norm[vv],
    " node: ",nodeCoord[dbox[[1]],dbox[[2]],dbox[[3]],volStep],
    " currIndex: ",currIndex];
  *)

  If[Not[NumberQ[vv[[k]]]],
   Print["Not Numeric in Node ", currIndex, " vv[[", k, "]]: ", vv[[k]]]

  ]
 ];

   (*Write[f,"x: ",d[[1]]," y: ",d[[2]]," z: ",
   d[[3]]," vr: ",Norm[vv]," currIndex: ",currIndex];*)

 dList = AppendTo[dList, d];

 If[d[[1]] < 0,
  numBackscatter = numBackscatter + 1;
    (*Print["Particle backscattered"];*)
  Break[]
 ];

 If[count > 2,

  For[k = 1, k ≤ 3, k++,
   meanvSq[[k]] = ((dList[[count]][[k]] - dList[[count - 2]][[k]]) / (2 * dt))^2
  ];

  currE = (muonMass / 2) * Total[meanvSq];
 ];


   (*currE=(1/2)*muonMass*(Norm[vv])^2;*)
 vvdir = Normalize[vv];

   (*abs[ls] might have solved the problem of getting stuck at a place*)
 ls = Abs[Norm[d] - Norm[oldd]];

 currEev = ToeV[currE];
   (*seev=getEStopping[ZFe,currEev,1.0,es];(*in eV/ang*) *)
 seev = getSe[26, currEev];
 selist = AppendTo[selist, se];
```

```
 (*Print[se];*)
 se = seev * (1.6 * 10^(-19)) / (10^-10);

 currE = currE - se * ls;

 delv = dt * se / muonMass;
 vmag = Norm[vv];
 vv = (vmag - delv) * vvdir;

 (*
 Print["e = ",currE," se = ",se," se(eV/ang) = ",getEStopping[ZC,ToeV[currE],
    1.0,es]," ls = ",ls," del v. = ",delv," Norm[vv] = ",Norm[vv]];
 *)

 (*Check Transmission*)
 If[d[[1]] ≥ TotalDepth,
  transmitted = True;
  numTransmission = numTransmission + 1;
  Break[];
 ];

 dbox[[1]] = Mod[d[[1]], latticeConstant];
 dbox[[2]] = Mod[d[[2]], latticeConstant];
 dbox[[3]] = Mod[d[[3]], latticeConstant];
 (*If[(Not[NumberQ[fx]]) ||( Not[NumberQ[fy]]) || (Not[NumberQ[fz]]),
   Print["Not Numeric in Node ",i]];*)
 currIndex = nodeCoord[dbox[[1]], dbox[[2]], dbox[[3]], volStep];

 If[currIndex == 0,
  Print["CurrIndex is 0 and dbox x, y, z: ",
   dbox[[1]], " ", dbox[[2]], " ", dbox[[3]]]
  (*currIndex=1*)];

 currT = currT + dt;
 numScatter = numScatter + 1;
 count = count + 1;

 oldd = d;

 (*new v = old v - del v + force in new block,
 which will be calculated in the next loop*)
 oldvv = vv;

];

(*Print["final velocity: ",Norm[vv]];*)
```

```
    (*Print["final depth: ",d[[1]]];
    Print["Number of scatters: ",numScatter];*)
    (*Print["x: ",d[[1]]," y: ",d[[2]]," z: ",d[[3]]];*)
    (*save coord. for distribution data*)
    If[transmitted == True,
     trackHistory = AppendTo[trackHistory, {n, vmaglist, selist}]
    ];
    distribution = AppendTo[distribution, d];
    depthList = AppendTo[depthList, d[[1]]];
    allDList = AppendTo[allDList, dList];
    (*Pause[2];*)
   ]
   ,
   {"Muon #", n, "distance: ", d,
    ProgressIndicator[Norm[d], {0, TotalDepth}], "Estopping: ", seev,
    "Resultant Velocity: ", ProgressIndicator[Norm[vv], {0, vr}],
    "Curent Energy:", ProgressIndicator[currE, {ethold, en * 2}],
    "Current Block Index: ", ProgressIndicator[currIndex, {1, xBound^3}]}
  ];

  (*distribution*)
  (*Average depth*)
  Print["Average range = ", Total[distribution] / numMuons];
  (*Transmission and backscattering*)
  Print[numTransmission, " particles transmitted, ",
   numBackscatter, " particles backscattered"];

  Close[f]
 ]
```

MUSCLE: Evaluation to find a suitable number of blocks for MD Discretization methods.

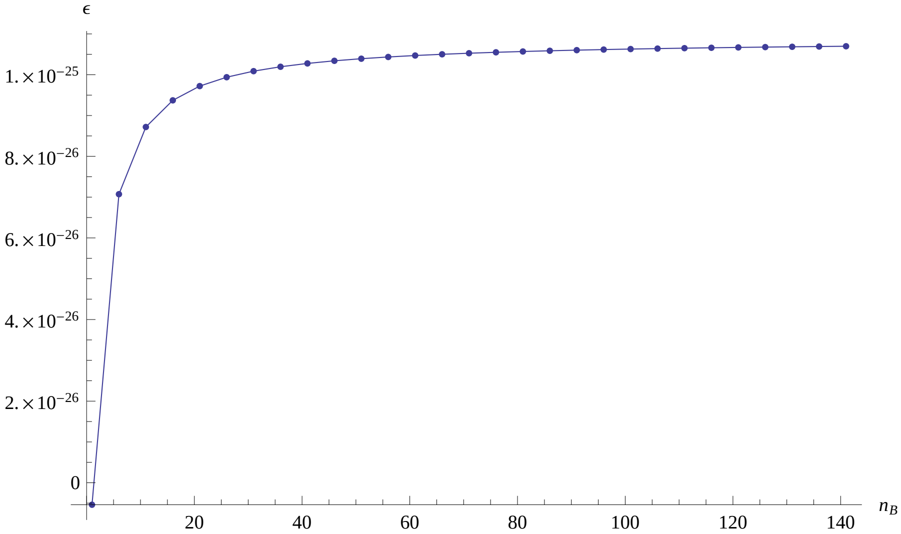
Mathematica Code

```
ang = 10^-10;
latticeConstant = 2.87 * ang;
eV = 1.6 * 10^-19;

(*neighbor atoms in the unit cube*)
lx = {latticeConstant / 2, 0, 0}
ly = {0, latticeConstant / 2, 0}
lz = {0, 0, latticeConstant / 2}
neighborIonsBCC = List[];
neighborIonsBCC = AppendTo[neighborIonsBCC, {0, 0, 0}];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * ly];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * ly + 2 * lz];
(*wrong!->neighborIonsBCC=AppendTo[neighborIonsBCC,lx-ly-lz];*)
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * ly];
neighborIonsBCC = AppendTo[neighborIonsBCC, 2 * lx + 2 * ly + 2 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, - ly + lx + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 3 * ly + lx + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly - lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, lx + ly + 3 * lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, - lx + ly + lz];
neighborIonsBCC = AppendTo[neighborIonsBCC, 3 * lx + ly + lz];

{1.435 × 10^-10, 0, 0}

{0, 1.435 × 10^-10, 0}

{0, 0, 1.435 × 10^-10}

getNumberOfBlocksEvaluation[ABM_, aBM_, L_, neighbor_] := Module[
  {Vr = 0.0, dis = 0.0, done = False, numB = 1, sig1 = 0.0,
   sig2 = 0.0, ep = 0.0, eps = 3.0 * 10^(-8), edgex = {}, edgey = {},
   edgez = {}, corners = {}, origin = {}, center = {}, eplist = {}},

  (*While[done==False,*)
  For[i = 1, i < 30, i++,
   (*Initial edge length*)
   edgex = {L / numB, 0.0, 0.0};
   edgey = {0.0, L / numB, 0.0};
   edgez = {0.0, 0.0, L / numB};
   origin = {0.0, 0.0, 0.0};
```

```
 center = (edgex + edgey + edgez) / 2;

 Clear[corners];
 corners = {};

 (*8 corners*)
 corners = AppendTo[corners, origin];
 corners = AppendTo[corners, edgex];
 corners = AppendTo[corners, edgey];
 corners = AppendTo[corners, edgez];
 corners = AppendTo[corners, edgex + edgey];
 corners = AppendTo[corners, edgex + edgez];
 corners = AppendTo[corners, edgey + edgez];
 corners = AppendTo[corners, edgex + edgey + edgez];
 Vr = 0.0;
 sig1 = 0.0;
 sig2 = 0.0;
 For[s = 1, s ≤ Length[corners], s = s + 1,
  dis =
    √((corners[[s]][[1]] - neighbor[[1]])^2 + (corners[[s]][[2]] - neighbor[[2]])^2 +
       (corners[[s]][[3]] - neighbor[[3]])^2);
  Vr = Vr + ABM * ⅇ^(-dis/aBM);
 ];
 Vr = Vr / 8;
 For[s = 1, s ≤ Length[corners], s = s + 1,
  sig1 = sig1 + (ABM * ⅇ^(-Norm[corners[[s]]]/aBM) - Vr) ^2;
  sig2 = sig2 + (ABM * ⅇ^(-Norm[corners[[s]]]/aBM) - ABM * ⅇ^(-Norm[center]/aBM)) ^2;
 ];
 sig1 = sig1 / 8;
 sig1 = Sqrt[sig1];
 sig2 = sig2 / 8;
 sig2 = Sqrt[sig2];
 ep = sig1 - sig2;
 (*If[ep>eps,
     numB=numB+5
     ,
     done=True
    ];
  ];
 *)
 eplist = AppendTo[eplist, {numB, ep}];
 numB = numB + 5
];
ListPlot[eplist, Joined → True, Mesh → All,
 AxesLabel → {n_B, ϵ}, PlotRange → All, AxesOrigin → {0, Min[eplist]}]
```

```
  (*Return[eplist]*)
 ]
```

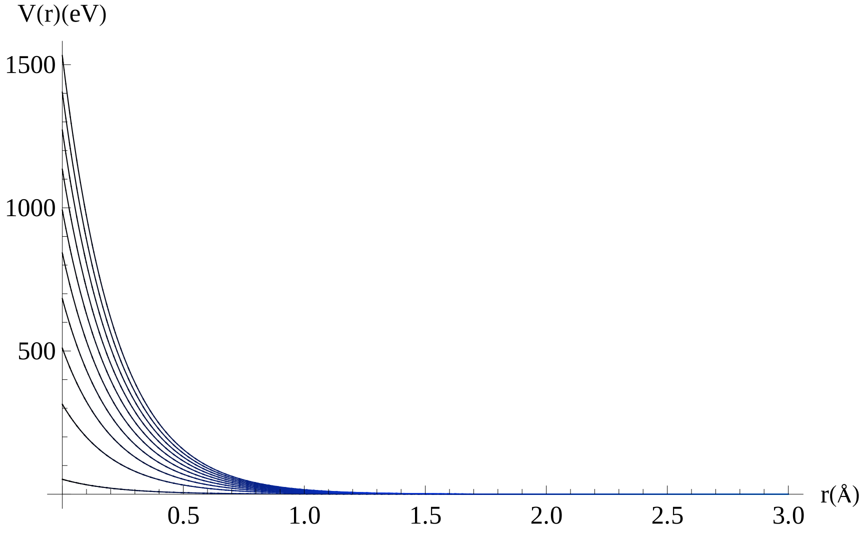
```
getNumberOfBlocksEvaluation[52.0 * (26)^{3/4} * eV,
 0.219 / ang, 2.87, neighborIonsBCC[[10]] / ang]
```



MUSCLE: Born-Mayer Potential plot for different elements

Mathematica Code

```
Show[
 Flatten[
  Table[
   Graphics[
    Plot[(z2)^(3/4) * 52.0 * ⅇ^(-r/0.219), {r, 0.0, 3.0},
     AxesLabel → {"r(Å)", "V(r)(eV)"}, LabelStyle → Directive[Medium],
     ColorFunction → Function[{z2}, {z2 * 0.1, z2 * 0.5, z2 * 2}],
     PlotRange → All
    ]
   ],
   {z2, 1, 92, 10}]
 ]
]
```



MUSCLE: Binary Collision Approximation code (C++)

(Random neighbor selection code as presented in the last section of Chapter 2, loosely based on the Fortran code for TRIM Monte Carlo simulation)

Globals.h (global functions and variables)

```
#include <iostream>
#include <fstream>
//#include <cstdlib>
//#include <cmath>

#include <math.h>
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

using namespace std;

#ifndef talk
#define talk 0
#endif

#ifndef PEO
//in PSTOP (i.e. calculateSE) vel. proportional stopping below velocity PEO (Pg-219,
ln 1210)
#define PEO 25.000;
#endif

void exitOnError(char* errmsg);

//numeric functions that need to be redefined

double Max(double a, double b);
double Min(double a, double b);
int Max(int a, int b);
int Min(int a, int b);
//int Abs(int a);
double Abs(double a);

//utilities

int whichBin (double e, double maxe, int numBins);

double generateRandom(int seed);

void printarray(char* arrayname, double array[], int size);
void printarray(char* arrayname, int array[], int size);

void testarray();
```

Globals.cpp

```
#include "Globals.h"

void exitOnError(char* errmsg)
{
    cout << errmsg << endl;
    exit(1);
}

double Max(double a, double b)
{
    return (b<a)? a:b;
}

double Min(double a, double b)
{
    return (a<b)? a:b;
}

int Max(int a, int b)
{
    return (b<a)? a:b;
}

int Min(int a, int b)
{
    return (a<b)? a:b;
}

double Abs(double a)
{
    if(a < 0.0) return -a;
}

//function to decide which bin an epsilon value falls into
int whichBin (double e, double maxe, int numBins)
{
    double interval = maxe / numBins;

    for(int i = 0; i < numBins; i++)
    {
        double low = i*interval;
        double high = (i + 1)*interval;

        if((e>=low) && (e<high))
        return i;
    }
}


void printarray(char* arrayname, double array[], int size)
{
    cout << endl << "Printing " << arrayname << endl;
    for (int i = 0; i <= size; i++)
    {
        cout << "\t " << array[i];
    }
    cout << endl;
    cout << "Done with printing " << arrayname << endl;
}

void printarray(char* arrayname, int array[], int size)
{
    cout << endl << "Printing " << arrayname << endl;
    for (int i = 0; i <= size; i++)
    {
        cout << "\t " << array[i];
    }
    cout << endl;
    cout << "Done with printing " << arrayname << endl;
}

void testarray()
{
    int row = 3, column = 7;
    cout << endl << "Printing array.." << endl;
    for (int i = 0; i <= row; i++)
    {
        cout << "row " << i << ": ";
        for (int j = 0; j <= column; j++)
        {
            //activate this line with the proper array name
            //cout << "\t " << mt[i][j];
        }
        cout << endl;
    }
    cout << "Done with printing array" << endl;

    //int **p = zt;
    //double **c = mt;
    //print2darray("zt", p, 3, 7);
    //print2darray("mt", c, 3, 7);
}

double generateRandom(int seed)
{
     //srand(time(NULL));
     //srand(seed);
    srand(seed);

    double e;

    //random epsilon, generates between the range (0,1]
    e = (double)rand()/((double)(RAND_MAX)+(double)(1));

    return e;

}
```

---

rstop.h (electronic stopping module)

```
#include "Globals.h"
```

```
struct rstopData
{
    double se[1000];
    //double sn;
    double vfermi; //set this 0 when initialized. (for the time being, I dont
calculate this)
};

void getrStop(int z1, int z2, double ee, int units, double lfctr, double vfermi,
rstopData& rstopdata);

double calculaterSE(double m1, double m2, int z1, int z2, double e, double pcoef[]);
```

rstop.cpp

```
#include "rstop.h"
#include "scoef.h"

//Calculate electronic stopping cross section using data from scoef1.dat

//ee - ion energy in keV

//note: we won't use the parameter lfctr and vfermi b/c they are not used in the
proton calculation.

void getrStop(int z1, int z2, double ee, int units, double lfctr, double vfermi,
rstopData& rstopdata)
{

    if (z1 > 92) exitOnError("Error: atomic number is greater than 92. Exiting..");

    if (ee < pow(10,-10))
    {
        for (int i = 1; i <= 1000; i++) rstopdata.se[i] = 0;
        return;
    }

    scoefData ionScoef, targetScoef;

    getstructScoef(z1, ionScoef, "scoef1.dat");
    getstructScoef(z2, targetScoef, "scoef1.dat");

// m1 and mm1 corrections (pg-217, line 620 and 630) are not included. we can change
it from inside if need arises.

    //m1 in this case is a proton
    double m1 = 1.0078; //ionScoef.m1;

    double e0 = 0.001 * ee/m1; //for 1000 values of stopping

    if (e0 > 100000) exitOnError("(Ion Energy/atomic mass)*0.001 ratio is bigger than
100000! Exiting..");

    for (int i = 1; i <= 1000; i++)
    {
        double e = e0 * i;

    //calculate electronic stopping with atomic weight of solid (M2 column in
scoef.dat)
    //be careful, m1 and m2 are not what they seem in the fortran code (pg217). Don't
        confuse between targetScoef's or ionScoef's m1 and m2.

    //we calculate only the proton's electron stopping here. (PSTOP)
    rstopdata.se[i] = calculaterSE(m1, targetScoef.m2, z1, z2, e, targetScoef.pcoef);

    //convert to ev-angstrom...I don't care about the parameter 'units' right now!
    //do the following for testing. The last test value should match the stopping
table,      //which is in ev-angstrom.
    if (talk == 4)
    {
        double test = rstopdata.se[i];
        test = test * targetScoef.atrho * pow(10, -23);
    //print every 10th value
    if ((i%10)==0)
    {
        //cout << "bin: " << i << ", Se = " << test << "\n";
        cout << test << ",";
    }
    }

    //trim85 takes rstop values in ev-Ang.2, so convert to that format
    rstopdata.se[i] = rstopdata.se[i] * 10;

    }

    //check the pcoef values in targetData
    if( talk == 2)
    {
        cout << "Test the PCoef values in targetScoef" << endl;

        cout << "\t" << targetScoef.pcoef[1] << "\t" << targetScoef.pcoef[2] << "\t"
<< targetScoef.pcoef[3] << "\t" << targetScoef.pcoef[4] << "\t" <<
targetScoef.pcoef[5] << "\t" << targetScoef.pcoef[6] << "\t" << targetScoef.pcoef[7]
<< "\t" << targetScoef.pcoef[8] << "\n" << endl;

    }

    //we are done here..
}

//This is PSTOP subroutine in pg-219
double calculaterSE(double m1, double m2, int z1, int z2, double e, double pcoef[])
{
    double se;
    double peo = (double)PEO; //had to do this because compiler is not accepting PEO
as double!
    double pe = Max(peo, e);
    double sl = pcoef[1]*(pow(pe,pcoef[2])) + pcoef[3]*(pow(pe,pcoef[4]));
    double sh = ( pcoef[5] / ( pow(pe,pcoef[6]) ) ) * log( (pcoef[7]/pe) + pcoef[8]*pe
);
    se = ((sl*sh)/(sl+sh));

    //PEO is defined in Globals.h
    if(e > peo) return se;
    else
    {
        double velpwr = 0.45;
        if (z2 <= 6) velpwr = 0.25;

        se = se * pow((e/peo),velpwr);

        return se;
    }
}
```

---

scoef.h (to read data from file)

```
#include "Globals.h"

struct scoefData
{
    int z1;
    double mm1, m1, m2, rho, atrho, vfermi;
    double lfctr;
    double pcoef[8];
};

//this was just a test function
void scoef(int z1, double mm1, double m1, double m2, double rho, double atrho, double
vfermi, double lfctr, double pcoef[]);

//this is the one we will be using
void getstructScoef(int zz, scoefData& scoefdata, char* filename);
```

---

scoef.cpp

```
#include "scoef.h"

void getstructScoef(int zz, scoefData& scoefdata, char* filename)
{
    if (talk == 2) cout << "Reading scoeff data file for atmoic no. " << zz << endl;

    ifstream instream;
    instream.open(filename);

    if (!instream)
    {
        exitOnError("Unable to open file ");
    }

    int j = 0, i = 0, z = zz;
    //dummy var
    double xx, zero;

    //iterate to the definite row - 1
    for( i = 1; i <= 92; i++)
    {
        instream >> j >> xx >> xx >> xx >> xx >> xx >> xx >> xx;
            if (j == z-1) break;
    }

    //now input the desired row to our structure

    instream >> scoefdata.z1 >> scoefdata.mm1 >> scoefdata.m1 >> scoefdata.m2 >>
scoefdata.rho >> scoefdata.atrho >> scoefdata.vfermi >> scoefdata.lfctr;

    //cout << "Testing the input.." << endl << "Z = " << scoefdata.z1 << endl << "rho
= " << scoefdata.rho << endl;

    instream.close();

    //find proton stopping power coefficients in the second data set

    instream.open(filename);

    if (!instream)
    {
        exitOnError("Unable to open data file ");
    }

    //dummy var
    int k; double s;

    //skip 92 lines
    for( i = 1; i <= 92; i++)
    {
        instream >> k >> s >> s >> s >> s >> s >> s >> s;
    }

    for( i = 1; i <= 92; i++)
    {
        instream >> j >> scoefdata.pcoef[1] >> scoefdata.pcoef[2] >>
scoefdata.pcoef[3] >> scoefdata.pcoef[4] >> scoefdata.pcoef[5] >> scoefdata.pcoef[6]
>> scoefdata.pcoef[7] >> scoefdata.pcoef[8];

        //cout << j << "\t" << pcoef[0] << "\t" << pcoef[1] << "\t" << pcoef[2] <<
"\t" << pcoef[3] << "\t" << pcoef[4] << "\t" << pcoef[5] << "\t" << pcoef[6] << "\t"
<< pcoef[7] << "\n" << endl;

        if (j == z) break;
    }

    //cout << "Testing the input on second data set (proton coefficients)" << endl <<
"Z = " << scoefdata.z1 << endl << "pcoef[1] = " << scoefdata.pcoef[1] << endl;

    instream.close();

    //multiply atrho by 10^22
    double temprho = scoefdata.atrho;
    temprho = temprho * 1.0 * pow(10, 22);
    scoefdata.atrho = temprho;

    if (talk == 2) cout << "Done loading data from " << filename << " for atomic no. "
<< zz << endl;
 }
```

monte.h (BCA Monte Carlo Simulation)

```
#include "Globals.h"

//functions

void Initialize();
void calculateAvgMassOfLayer();
void getStoppingForTarget();
void setInitialConditions();
void MonteCarlo();
void printFinalDetails();
void writeToFile();
```

monte.cpp

```
//============================================================================
// Name        : monte.cpp
// Author      : Nazmus Saquib
// Version     :
// Copyright   :
// Description : BCA code in C++
//============================================================================

#include “Globals.h”
#include “scoef.h”
#include “rstop.h”
#include “muscle_bca.h”
#include “vectorGeometry.h”


const int numXBin = 100;

const int numIons = 1;

int xBin[numXBin+1] = {0};

int selectedXBin = 0;

double rho[4]={0.0};

int n[]={0,0,0,0};

    double e0kev, m1, cw, ed, latticeConst;
    int z1, hn, iy, nowout;
    double dx[3] = {0.0};

    int zt[3][7] = {0};
    double mt[3][7] = {0.};
    double t[3][7] = {0.};

    int Layer; //use this for number of layers rather than the L in the program;
    //remember this cannot be set to 0. Other loops/arrays may start from
    //0, but keep this >= 1;

//the arrays..

double my[3][7]= {0},
 ai[3][7]= {0},
 fi[3][7]= {0},
 ec[3][7]= {0},
 io[3][7]= {0},
 k[3]= {0},
 kl[3][7]= {0};

double vf[3][7]= {0},
 mu[3]= {0},
 ioniz[3]= {0},
 h[3]= {0},
 xx[3]= {0},
 m2[3]= {0.0},
 z2[3]= {0},
 c[3]= {0},
 epsbk[3]= {0},
 arho[3]= {0};

double a[3]= {0},
 f[3]= {0},
 lm[3]= {0},
 pmax[3]= {0},
 fd[3]= {0},
 kd[3]= {0},
 sbk[3]= {0},
 lf[3]= {0},
 yy[8]= {0};

double se[3][1000]= {0.0},
 seo[1000]= {0.0},
  epsdg[3]= {0.0};

double ls = 0., lo = 0., maximum = 0.;

double xsum = 0, x2sum = 0, x3sum = 0, x4sum = 0, plsum = 0, pl2sum = 0;

double avex = 0, vari = 0, sigma = 0, v = 0, v2 = 0, Gamma = 0, beta = 0, y = 0, avepl
= 0, sigpl = 0, avecol = 0;

int i = 0, j = 0;

int ib = 0, it = 0;

double eb = 0.0, et = 0.0;
int icsum = 0;

double y2sum = 0, xy2sum = 0, x2y2su = 0, y4sum = 0;

double tau = 0;

int iii = -1;
```

```
//my dummy vars
int w; double q;

//my customized data structures or vars
scoefData scoefz1;

rstopData rstp;

//these vars (continuing from this line) are added later as needed, vars that were not
declared in the initialization of trim85 but showed up in the middle.
double e0, ef;

int iz;

double alfa, alpha;

double tmin, da;

double L0;

int iz1;
double ee;

int izt;

double nh;

double epso;

int ih;

double e;


//boolean for keeping track of a particle being transmitted or backscattered
int transmitted = 0;
int backscattered = 0;

//boolean for keeping track of channeling
int insideChannel = 1;

//boolean for determining whether to scatter from neighbor atoms
int neighborFlag = 0;


//main function
int main()
{
    cout << "Initializing" << endl;
    Initialize("IronInput.dat");
    calculateAvgMassOfLayer();
    getStoppingForTarget();
    setInitialConditions();
    MonteCarlo();

/* *********************** Testing Ground for arrays and variables
******************* */

    for (w = 1; w <= 8; w++) cout << "\t " << yy[w];
    for (w = 1; w <= 3; w++) cout << “\n\t “ << xx[w];
    cout << “CW or L0: “ << L0;

    //(trouble! The values of n[] elements are changing like crazy)
    cout << endl << n[0] << “\t” << n[1] << “\t” << n[2] << “\t” << n[3] << endl;

    cout << endl << rho[0] << “\t” << rho[1] << “\t” << rho[2] << “\t” << rho[3] <<
“\t” << rho[4] << endl;

    //test z2[], m2[] arrays

    for (w = 1; w <= Layer; w++) cout << “\t “ << z2[w];
    for (w = 1; w <= Layer; w++) cout << “\n\t “ << m2[w];

    //test the se values

    cout << endl << se[1][1000] << “ “ << se[2][1000] << “ “ << se[3][1000] << endl;

    //test the se[] values by printing 10th element from the array (to be implemented)

    //for (w = 1; w <= 100; w++)
    //{
        //for (int sss = 1; sss <= 50; sss++)
        //{
            //cout << “\t” << mpart[w][sss];
        //}
    //}

/* ********************** End of Testing Ground ************************* */
    cout << endl;

    //Now print values of each bin in xBin

    //for (w = 1; w <= numXBin; w++) cout << xBin[w] << “,”;

    //print final x values

    cout << endl;

    cout << “Number of Backscattered Ions: “ << ib << endl;
    cout << “Number of Transmitted Ions: “ << it << endl;

    return 0;
}

//end of main

//Helper functions follow from here

//initialize variables and setup each layer over here
void Initialize(char* filename)
{
    //Read command file
    ifstream instream;
    instream.open(filename,ios::in);
    cout << filename;

    if (!instream)
    {
```

```
    exitOnError(filename);
}

instream >> e0kev >> z1 >> m1 >> latticeConst >> hn >> cw >> ed >> iy >> nowout;

instream >> dx[1] >> rho[1];

instream >> zt[1][1] >> mt[1][1] >> t[1][1];

instream >> n[1];

instream >> dx[2] >> rho[2];

instream >> zt[2][1] >> mt[2][1] >> t[2][1];

instream >> n[2];

instream >> dx[3] >> rho[3];

instream >> zt[3][1] >> mt[3][1] >> t[3][1];

instream >> n[3];

for (w = 0; w <= 3; w++)
{ if (n[w] != 1) n[w] = 1; }

instream.close();

Layer = 3;

//done with primary (crude) setup, off to reading scoef.dat file

iz = z1;

getstructScoef(iz, scoefz1, “scoef1.dat”); //used to get pcoef data yy in trim85

//so we put the values in yy here immediately
for (w = 1; w <= 8; w++)
{ yy[w] = scoefz1.pcoef[w]; }

//note: although yy turns out to be just a dummy array

e0 = e0kev*1000; //convert to ev.

if (ed == 0.) ed = 25.0;

ef = Max(5.0, e0kev*0.1);

//alfa = angle of incidence, alpha = radian of alfa
alfa = 0.;
alpha = alfa*Pi/180;

tmin = 5.0;

tau = 0.0;

da = 3.0;

if (iy == 0) iy = 16381;
```

```
    //now calculate the total depth of each layer = xx(l), and grid spacing cw

    xx[1] = dx[1];

    for (w = 2; w <= 3; w++) { xx[w] = dx[w] + xx[w-1]; }

    if (cw == 0) cw = 0.01*xx[3];

    L0 = cw;

    //take care of any custom variable or struct I made
    rstp.vfermi = 0.0; //set this to 0. for the time being (see log for explanation)

    //now, off to avg mass of layer in the next void function
}

//avg mass and atomic number of each layer
void calculateAvgMassOfLayer()
{
    for (int LL = 1; LL <= Layer; LL++)
    {
        int ii = n[LL];
        for (w = 1; w <= ii; w++)
        {
            h[LL] = h[LL] + t[LL][w];
            //cout << “testing here..” << rstp.vfermi << endl;
        }
    }
    for (int LL = 1; LL <= Layer; LL++)
    {
        int ii = n[LL];
        for (w = 1; w <= ii; w++)
        {
            t[LL][w] = t[LL][w]/h[LL];
            m2[LL] = m2[LL] + t[LL][w] * mt[LL][w];
            z2[LL] = z2[LL] + t[LL][w] * zt[LL][w];

            //note: z2 wont be an integer once t has a value other than 1.0??
            //so are we calculating average atomic number here? why?
        }
    }

    //done with this, off to finding electronic stopping powers in the next function
}

void getStoppingForTarget()
{
    iz1 = z1; ee = 0;

    for (int LL = 1; LL <= Layer; LL++)
    {
        arho[LL] = rho[LL] * 0.6022/(m2[LL]);
        mu[LL] = m1/(m2[LL]);
        int ii = n[LL];
        for (int nn = 1; nn <= ii; nn++)
        {
            //set rstp.se[1..1000] = 0. Clear it for the next loop
            for (w = 1; w <= 1000; w++)
            { rstp.se[w] = 0.; }

            izt = zt[LL][nn];
            //calling getrstop now. units, lfctr, vfermi = 1 (doesnt matter)
            getrStop(iz1, izt, e0kev, 1, 1., 1., rstp); //rstp defined in header

            //set this anyway, though I dont calculate this in RSTOP
            vf[LL][nn] = rstp.vfermi; //which is just set 0 in the struct def.

            for (w = 1; w <= 1000; w++)
            {
                se[LL][w] = se[LL][w] + rstp.se[w] * t[LL][nn] * arho[LL];
            }
        }
    }

    //now, off to setting up initial conditions next function

}

void setInitialConditions()
{
    nh = hn; //number of histories

    for (int LL = 1; LL <= Layer; LL++)
    {
        a[LL] = 0.5292 * 0.8853 / ( pow(z1, 0.23) + pow(z2[LL], 0.23) );

        //now calculate the mean flight path with the conditions given in trim85
        f[LL] = a[LL] * m2[LL] / ( z1 * z2[LL] * 14.4 * (m1 + m2[LL] ) );

        epso = e0 * f[LL];

        epsdg[LL] = tmin * f[LL] * pow( (1.0 + mu[LL]) , 2) / (4.0 * mu[LL]);

        fd[LL] = 0.01 * pow( z2[LL], (-7.0/3.0) );

        kd[LL] = 0.1334 * pow ( z2[LL], (2.0/3.0) ) / sqrt( m2[LL] );
    }

    for (int LL = 1; LL <= Layer; LL++)
    {
        int ii = n[LL];

        for (w = 1; w <= ii; w++)
        {
            my[LL][w] = m1/mt[LL][w];

            ec[LL][w] = 4.0 * my[LL][w] / pow( ( 1.0 + my[LL][w] ), 2);

        ai[LL][w] = 0.5292 * 0.8853 / ( pow(z1,0.23) + pow(zt[LL][w],0.23) );

    fi[LL][w] = ai[LL][w] * mt[LL][w] / ( z1 * zt[LL][w] * 14.4 * ( m1 + mt[LL][w] ) );

        }

    }

    cout << “Setup finished. Starting Monte Carlo Loops..”;
    //off to monte carlo loop in next routine
}

void MonteCarlo()
{

//custom variables and arrays for this section

double e;
double cosin = 0.0, siny = 0.0, sine = 0.0, cosy = 0.0;
double pl = 0.0;
int ic;
int LL = 1;

double eps;
double eeg;
double p;
double b;
int ie, ia; //not sure if ie or ia should be double. They are used to access elements
of the m[][] array at some point
double see;
double dee;
double s2, c2, ct, st;
double r, rr;
double ex1, ex2, ex3, ex4;
double v, v1;
double fr, fr1;
double q;
double roc, sqe;
double cc, aa, ff;
double delta, co;
double den;
double phi, psi;
double x1;
int ip;

    //variables for crystal calculations
    double sep = 0.0;
    int ionCounter = 0;
    double p1, p2;
    double Theta = 0.0;
    double rTheta = 0.0, rPhi = 0.0;
    double thetaThreshold = 0.5;
    double crystalMuonY = 0.0, crystalMuonZ = 0.0;

    //amount of translations in y and z axes
    double translationY = 0.0, translationZ = 0.0;

    //Scatter Plot variables
    const int numScatterPlotBins = 5;
    double scatterPlot[numScatterPlotBins + 1] = {0.0};

    //Vector declarations

    //lattice constant of Target (input from file)
    double latticeConstant = latticeConst;

    Vector3 ions[4];
    Vector3 ionsOrdered[4];

    Vector3 neighborIonsBCC[14];
    Vector3 neighborIonsFCC[14];

    Vector3 ionTranslationY(0,translationY,0);
    Vector3 ionTranslationZ(0,0,translationZ);

    Vector3 unitzplus, unitzminus, unityplus, unityminus, unitxplus, unitxminus;
    unitzplus.z = 1;
    unitzminus.z = -1;
    unityplus.y = 1;
    unityminus.y = -1;
    unitxplus.x = 1;
    unitxminus.x = -1;

    Vector3 initialDirection(1,0,0);

    Vector3 d;
    d.x = 1;
    d *= latticeConstant/2;

    Vector3 lx(latticeConstant/2, 0, 0);
    Vector3 ly(0, latticeConstant/2, 0);
    Vector3 lz(0, 0, latticeConstant/2);

    Vector3 lambda;
    Vector3 lambdaPrime;
    Vector3 pVector;
    Vector3 pUnitVector;
    Vector3 sepVector;

    Vector3 Di;
    Vector3 DiPrev;
    //Vector3 DiPrevToDi;
    Vector3 delX;
    Vector3 delX1, delX2;

    Vector3 dummy1, dummy2, temp;

    Vector3 scatterIonPos;

//initialize the random number generator

    srand(time(NULL));

//open file to write output

    ofstream outStream;
    outStream.open(“coords0.txt”);

    if(outStream.fail())
    {
        exitOnError(“Could not open Output file”);
    }

    //write basic information in the output file
    //number of ions, ion energy, total depth, depth of each layer
    outStream << nh << “\t” << e0kev << “\t” << xx[Layer] << “\t” << dx[1] << “\t”
    << dx[2] << “\t” << dx[3] << endl;

//open scatter plot file to write current Y and Z coordinates of muons at designated
intervals

    ofstream scatterStream;
    scatterStream.open(“scatterOut1.txt”);

    if(scatterStream.fail())
    {
        exitOnError(“Could not open Scatter Plot Output file”);
    }

//open range distribution file to write final X coordinates of muons

    ofstream rangeStream;
    rangeStream.open(“rangeOut1.txt”);

    if(rangeStream.fail())
    {
        exitOnError(“Could not open Range Distribution Output file”);
    }

//Open general information dump file

    ofstream infoStream;
    infoStream.open(“info1.txt”);

    if(infoStream.fail())
    {
        exitOnError(“Could not open general information Output file”);
    }


//Entering the target
//First set up for the top layer

for (ih = 1; ih <= nh; ih++)
{
    avex = xsum / Max(1.0, (float)(ih - ib - it - 1));
    if (talk == 2) cout << “Average ion range so far: “ << avex << “ angstroms.”
<<endl;

    if (talk > 2) cout << “Now starting ion number “ << ih << endl;

    e = e0;

    //set scatterPlot array (the intervals)
    scatterPlot[0] = 10;
    scatterPlot[1] = 30;
    scatterPlot[2] = 50;
    scatterPlot[3] = 100;
    scatterPlot[4] = 150;
    scatterPlot[5] = 200;

    /*
    for(int ccc = 0; ccc <= numScatterPlotBins; ccc++)
    {
        cout << “scatter plot “ << ccc << “: “ << scatterPlot[ccc] << endl;
    }
    */

    //Initial Ion Positions
    ions[0] = d + ionTranslationZ + unitzminus * (latticeConstant/2);
    ions[1] = d + ionTranslationZ + unitzplus * (latticeConstant/2);

    ions[2] = d * 2 + ionTranslationY + unityminus * (latticeConstant/2);
    ions[3] = d * 2 + ionTranslationY + unityplus * (latticeConstant/2);

    //Create a polygon that resides on the lateral axes.
    //The points are put on anticlockwise order, which is important for
    //testing whether the test point lies on this polygon

    ionsOrdered[0] = ions[0];
    ionsOrdered[1] = ions[2];
    ionsOrdered[2] = ions[1];
    ionsOrdered[3] = ions[3];

    //generate random theta and phi angles.
    rTheta = ( (double)rand()/((double)(RAND_MAX)+(double)(1)) ) * thetaThreshold;
    rPhi = ( (double)rand()/((double)(RAND_MAX)+(double)(1)) ) * 2 * Pi;

    //find corresponding x, y and z components of the direction vector.
    //radius of the direction vector is 1
    initialDirection.x = cos(rTheta);
    initialDirection.z = sin(rTheta) * cos(rPhi);
    initialDirection.y = sin(rTheta) * sin(rPhi);

    //set the counter to 0 for a new ion
    ionCounter = 0;

    pl = 0.0;
    ic = 0;

    //set initial DiPrev - the origin
    DiPrev.x = 0.0; DiPrev.y = 0.0; DiPrev.z = 0.0;

    //set initial lambda, the direction of motion. Normalize it.
    lambda.clear();
    lambda = initialDirection;
    lambda.normalize();

    //set initial delX to origin
    delX.clear();

    //clear the dummy delX vectors, set them to origin
    delX1.clear();
    delX2.clear();

    dummy1.clear();
    dummy2.clear();
    temp.clear();

    LL = 1;

    //set transmitted and backscattered to false
    transmitted = 0;
    backscattered = 0;
```

```
    //set channeling to true
    insideChannel = 1;

    neighborFlag = 0;

    //write the initial coordinates to the output file
    outStream << DiPrev.x << “\t” << DiPrev.y << “\t” << DiPrev.z << endl;
    //cout << endl << “Initial: “;
    //cout << DiPrev.x << “\t” << DiPrev.y << “\t” << DiPrev.z << endl;

    //cout << “r1 = “ << r1 << endl;

    //cycle for each collision until the energy of the particle becomes too low, or
the particle backscatters, or it goes out of the last layer (transmission)

    //needs a do while loop here,
    //which I will mention as the ‘mother loop’ from now.

    do
    {

    ic = ic + 1;

    eps = e * f[LL];

    eeg = sqrt(eps*epsdg[LL]);

    //pmax[LL] = a[LL] / (eeg + sqrt(eeg) + 0.125 * pow( eeg, 0.1) );
    pmax[LL] = sqrt(3) * (latticeConstant / 2) * 0.7;

    //Calculate impact parameter and choose the atom to scatter from.
    //Do this for ion pairs 0,1 and 2,3.

    if (ionCounter == 0)
    {
        delX1 = ions[0] - DiPrev;
        delX2 = ions[1] - DiPrev;

        dummy1 = delX1 % lambda;
        dummy2 = delX2 % lambda;

        p1 = sqrt( dummy1.scalarProduct( delX1 % lambda ) );
        p2 = sqrt( dummy2.scalarProduct( delX2 % lambda ) );

        if(p2 > p1)
        {   //swap ion ordering
            temp = ions[0];
            ions[0] = ions[1];
            ions[1] = temp;
            //cout << “Vertical Ions swapped” << endl;
        }
    }

    if (ionCounter == 2)
    {
        delX1 = ions[2] - DiPrev;
        delX2 = ions[3] - DiPrev;
        dummy1 = delX1 % lambda;
        dummy2 = delX2 % lambda;

        p1 = sqrt( dummy1.scalarProduct( delX1 % lambda ) );
        p2 = sqrt( dummy2.scalarProduct( delX2 % lambda ) );

        if(p2 > p1)
        {   //swap ion ordering
            temp = ions[2];
            ions[2] = ions[3];
            ions[3] = temp;
            //cout << “Horizontal Ions swapped” << endl;
        }
    }

    //now calculate impact parameter

    if(neighborFlag == 0)
    {
        delX = ions[ionCounter] - DiPrev;

        dummy1 = delX % lambda;

        p = sqrt( dummy1.scalarProduct( delX % lambda ) );

        //find impact parameter vector and it’s unit vector
        pVector = dummy1 % lambda;
        pUnitVector = pVector.unit();
    }
    else if(neighborFlag == 1)
    {
        double impact[13] = {0.0};
        double radial[13] = {0.0};
        double S[13] = {0.0};
        double sumS = 0.0;
        double Probability[13] = {0.0};
        double rnd_candidate = 0.0;
        int selected_candidate = -1;

        for(int ncount = 0; ncount <= 13; ncount++)
        {
            delX = neighborIonsBCC[ncount] - DiPrev;

            dummy1 = delX % lambda;

            p = sqrt( dummy1.scalarProduct( delX % lambda ) );

            //find impact parameter vector and it’s unit vector
            pVector = dummy1 % lambda;
            pUnitVector = pVector.unit();

            //scatterIonPos = neighborIonsBCC[ncount];

            impact[ncount] = p;
            radial[ncount] = delX.magnitude();
            S[ncount] = 1 / ( pow(impact[ncount],2) * radial[ncount] );
            sumS += S[ncount];

            //general scheme of selecting the neighbor ion
```

```
            // if(p < pmax[LL])
            // {
                // scatterIonPos = neighborIonsBCC[ncount];
                // //cout << “Neighbor “ << ncount << “ is selected” << endl;
                // break;
            // }
        }

        for(int ncount = 0; ncount <= 13; ncount++)
        {
            Probability[ncount] = S[ncount] / sumS;
        }

        //print out the probability array
        cout << endl;
        for(int ncount = 0; ncount <= 13; ncount++)
        {
            //cout << Probability[ncount] << “ “;
            infoStream << Probability[ncount] << “ “;
        }
        infoStream << endl;
        //cout << endl;


        //random number between 0 and 1
        rnd_candidate = ( (double)rand()/((double)(RAND_MAX)+(double)(1)) );

        //cout << “rand_candidate: “ << rnd_candidate << endl;

        //choose the candidate for scattering
        selected_candidate = whichNonUniformBin(rnd_candidate, Probability, 13);

        //cout << “Ion: “ << ih << “\tSelected Candidate: “ << selected_candidate <<
endl;
        infoStream << “Ion: “ << ih << “\tSelected Candidate: “ << selected_candidate
<< endl;

        //assign the scatterIonPos variable to the selected neighbor
        scatterIonPos = neighborIonsBCC[selected_candidate];

        //cout << “Scattering Ion Position: “; scatterIonPos.printVector();

        //find the essential quantities for the selected neighbor
        delX = neighborIonsBCC[selected_candidate] - DiPrev;

        dummy1 = delX % lambda;

        p = sqrt( dummy1.scalarProduct( delX % lambda ) );

        //find impact parameter vector and it’s unit vector
        pVector = dummy1 % lambda;
        pUnitVector = pVector.unit();
    }

    //find eps and b using fi[LL][nn], using nn that I was supposed to find from above
    //here im deliberately using nn = 1

    eps = fi[LL][1] * e;
    b = p / ai[LL][1];
```

```
if (eps > 10) //rutherford scattering
{
    s2 = 1.0 / (1.0 + (1.0 + b * (1.0 + b)) * pow((2.0 * eps * b), 2) );

    c2 = 1.0 - s2;

    ct = 2.0 * c2 - 1.0;

    st = sqrt(1.0 - ct * ct);
}
else //magic formula
{
    r = b;

    rr = -2.7 * log(eps * b);

    if (rr >= b)//note >= sign instead < in trim85
    {
        rr = -2.7 * log(eps * rr);

        if (rr >= b)//note >= sign instead < in trim85
        {
            r = rr;
        }
    }

    //do while loop that replaces line 330 loop
    do
    {

        ex1 = 0.18175 * exp(-3.1998 * r);

        ex2 = 0.50986 * exp(-0.94229 * r);

        ex3 = 0.28022 * exp(-0.4029 * r);

        ex4 = 0.028171 * exp(-0.20162 * r);

        v = (ex1 + ex2 + ex3 + ex4) / r;

    v1 = -(v + 3.1998 * ex1 + 0.94229 * ex2 + 0.4029 * ex3 + 0.20162 * ex4) / r;

        fr = b * b / r + v * r / eps - r;

        fr1 = -b * b / (r * r) + (v + v1 * r) / eps - 1.0;

        q = fr / fr1;

        r = r - q;
    }
    while( (Abs(q / r)) > 0.001 );


    roc = -2.0 * (eps - v) / v1;

    sqe = sqrt(eps);

    //5 parameter magic scattering calculation
```

```
        //below is for universal potential

        cc = (0.011615 + sqe) / (0.0071222 + sqe);

        aa = 2.0 * eps * (1.0 + (0.99229 / sqe) ) * ( pow(b, cc) );

        ff = ( sqrt(aa * aa + 1.0) - aa) * ( (9.3066 + eps) / (14.813 + eps) );


        delta = (r - b) * aa * ff / (ff + 1.0);

        co = (b + delta + roc) / (r + roc);

        c2 = co * co;

        s2 = 1.0 - c2;

        ct = 2.0 * c2 - 1.0;

        st = sqrt(1.0 - ct * ct);

    }

    Theta = acos(ct);
    //we are done finding theta (in CM system). So calculate all other quantities.

    //find separation and the separation vector.
    phi = (Pi - Theta) / 2;
    sep = p / tan(phi);
    sepVector = lambda * sep;

    //find theta in laboratory frame - psi
    psi = atan(st / (ct + my[LL][1] ) );
    //note: change my[LL][1] to my[LL][nn] when the above section is fixed.

    if (psi < 0 ) psi = psi + Pi;   //should I do this for crystals?

    //find Di, the scattering point vector
    Di = DiPrev + delX + pVector - sepVector;

    //find new direction of motion
    lambdaPrime = lambda * cos(psi) + pUnitVector * sin(psi);
    lambdaPrime.normalize();

    //find length of step, ls = distance of Di from DiPrev
    ls = Di.getDistance(DiPrev);

    //find energy lost due to electronic stopping, dee
    ie = (int)(e/e0kev+0.5);    //should it be 0.5? or less so that ie <=1000?

    see = se[LL][ie];

    if (e < e0kev) see = se[LL][1] * sqrt(e/e0kev);

    dee = ls * see;

    // den = energy transferred to recoil

    den = ec[LL][1] * s2 * e; //note: I am using ec[LL][1] here instead of [LL][nn].
    //cout << “den = “ << den << “, dee = “ << dee << endl;
    infoStream << “den = “ << den << “, dee = “ << dee << endl;

    e = e - den - dee;

    //cout << endl<< “current ion energy: “ << e << endl;

    if (dee > maximum) maximum = dee;

    pl = pl + ls - tau;

    //write the ion position to output file
    outStream << Di.x << “\t” << Di.y << “\t” << Di.z << endl;

    if((ic%30)==0)
    {
        //cout << Di.x << “\t” << Di.y << “\t” << Di.z << endl;
    }

    //determine which scatter plot Di’s x value belongs to.
    //output the y and z coordinates to scatter plot file accordingly.
    for(int cc = 0; cc <= numScatterPlotBins; cc++)
    {
        if( Di.x >= (scatterPlot[cc]) )
        {
            crystalMuonY = Di.y - translationY;
            crystalMuonZ = Di.z - translationZ;
        //cout << “scatter plot “ << ct << “: “ << scatterPlot[ct] << endl;
        scatterStream <<  scatterPlot[cc] << “\t” << crystalMuonY << “\t” <<
crystalMuonZ << endl;

            //set scatterPlot[ss] to a big number
            scatterPlot[cc] = 10000;
            //break out of this for loop
            break;
        }
    }

    //determine if Di is in the channeling region.
    //insideChannel = Di.isInsidePolygon(ionsOrdered, 4);

    //break out of parent loop if not inside the channel
    if(insideChannel == 0)
    {
        cout << “Ion “ << ih << “ is out of Channel” << endl;
        cout << Di.x << “\t” << Di.y << “\t” << Di.z << endl;
        break;
    }

    //determine which layer the next collision will be in

    if (Di.x < 0.0) //particle is backscattered
    {
         backscattered = 1; //cout << “ion number “ << ih << “ backscattered.” << endl;

         infoStream << “ion number “ << ih << “ backscattered.” << endl;

         ib = ib + 1;
         eb = eb + e;

         break; //break out of ‘mother do loop’ and continue with the next session of 
for loop.
    }

    //here we set the current or next layer the particle will be in
    for (w = 1; w <= Layer; w++)
    {
        if ( (Di.x <= xx[w]) && (w == 1) )
        {
            LL = 1;

            //cout << endl<<”ion is in layer “ << LL << endl;

            break; 
        //break out of this For loop and go check if the particle is transmitted.
        }
        else if ( (Di.x <= xx[w]) && (Di.x > xx[w-1]) )
        {
            LL = w;

            //cout << “ion is in layer “ << LL <<endl;

            break; //break out of this For loop and go check if the particle is 
transmitted.
        }
    }

    //now, check for particle transmission, i.e. whether the particle went out of the 
last layer.

    if(Di.x >= xx[Layer])
    {
        //particle is transmitted, take care of appropriate variables and break

        transmitted = 1;//cout << “ion number “ << ih << “ transmitted.” << endl;

        it = it + 1;

        et = et + e;

        ia = 57.295779 * acos(cosin) / da + 1.0;

        ie = 100 * e / e0 + 1.0;

        //m[ie][ia] = m[ie][ia] + 1;//note: how is this possible? ie and ia should be 
integers in order to access the elements of the array m[][]. But we calculate them as 
doubles here!

        break;  //break out of the ‘mother’ do loop
    }

    //now take care of ionCounter and other variables for the next scattering
    if(neighborFlag == 0)
    {
        if(ionCounter == 3)
        {
            //ionCounter = 0;

            neighborFlag = 1;
            scatterIonPos = ions[3];

            //cout << endl << “got out of first two layers” << endl;
        }
        else
        {
            ionCounter++;
        }
    }

    if(neighborFlag == 1)
    {
        //cout << “Updating Neighbor Ions” << endl;

        neighborIonsBCC[0] = scatterIonPos + lx - ly + lz;
        neighborIonsBCC[1] = scatterIonPos + lx + ly + lz;
        neighborIonsBCC[2] = scatterIonPos + lx + ly - lz;
        neighborIonsBCC[3] = scatterIonPos + lx - ly - lz;
        neighborIonsBCC[4] = scatterIonPos + lx * 2;
        neighborIonsBCC[5] = scatterIonPos - lx - ly - lz;
        neighborIonsBCC[6] = scatterIonPos - lx + ly - lz;
        neighborIonsBCC[7] = scatterIonPos - lx + ly + lz;
        neighborIonsBCC[8] = scatterIonPos - lx - ly + lz;
        neighborIonsBCC[9] = scatterIonPos + lz * 2;
        neighborIonsBCC[10] = scatterIonPos - lz * 2;
        neighborIonsBCC[11] = scatterIonPos + ly * 2;
        neighborIonsBCC[12] = scatterIonPos - ly * 2;
        neighborIonsBCC[13] = scatterIonPos - lx * 2;
    }

    //set DiPrev to Di
    DiPrev = Di;

    //update current lambda to lambdaPrime
    lambda = lambdaPrime;

    //now the while condition of the mother do loop checks if the particle has lesser
energy than our lowest energy limit, ef.
    }
    while(e > ef);

    //since we are out of the mother do loop now, the particle must have come to a
stop. So, increase the final particle distributions if the particle has not been
transmitted or backscattered.

    if( ((transmitted == 0)) && ((backscattered == 0)) && ((insideChannel == 1)) )
    {

        ip = (int)(pl/cw + 1.0);

        if(ip > 100) ip = 100;

        //ipl[ip] = ipl[ip] + 1;

        xsum = xsum + Di.x;
        //my own bin function

        selectedXBin = whichBin(Di.x, xx[Layer], numXBin);

        //write the selected bin to the output file
        //outStream << selectedXBin << endl;

        //cout << x << endl << xx[Layer] << endl << numXBin << endl << selectedXBin;

        xBin[selectedXBin] = xBin[selectedXBin] + 1;

        //print final x value for plotting histogram
        //cout << “ion “ << ih << “ final x: “ << Di.x << endl;
        infoStream << “ion “ << ih << “ final x: “ << Di.x << endl;

        //cout << Di.x << “,”;
        rangeStream << Di.x << endl;

        plsum = plsum + pl;

        icsum = icsum + ic;

        //ipl is the ion path length - the total avg. distance the ion travels
regardless of direction before it comes to stop

    }

    //that brings us to the end of one ion’s journey, now go to next ion by going back
to the for loop’s beginning..

}

//and this ends the monte carlo loop function. Take care of necessary structures and
variables that need to be cleared/deleted

    outStream.close();

    scatterStream.close();

    infoStream << “Number of Backscattered Ions: “ << ib << endl;

    infoStream.close();

    rangeStream.close();


}


int whichNonUniformBin(double e, double arr[], int numBins)
{
    double low = 0, high = arr[0];

    if((e>=low) && (e<high))
        return 0;
    else
    {
        for(int i = 0; i < numBins; i++)
        {
            low += arr[i];
            high += arr[i+1];

            //cout << “\tlow: “ << low << “, high:” << high << endl;

            if((e>=low) && (e<high))
            return i;
        }
    }
}
```