61784261X

TA
460
.S99
2826

NUMERICAL CALCULATION OF ELASTIC PROPERTIES OF FCC CRYSTALS USING MOLECULAR DYNAMICS SIMULATION

By

Robert Szydłowski, B. Eng.
Ryerson University, Toronto
2004

A THESIS PRESENTED TO RYERSON UNIVERSITY IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

IN THE PROGRAM OF

MECHANICAL ENGINEERING

Toronto, Ontario, Canada, 2006
© Robert Szydłowski 2006

PROPERTY OF
RYERSON UNIVERSITY LIBRARY

UMI Number: EC53542

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform EC53542

Copyright 2009 by ProQuest LLC

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

R. Szydłowski

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

R. Szydłowski

Numerical Calculation of Elastic Properties of FCC Crystals Using Molecular Dynamics Simulation

Master of Applied Science
In the Program of
Mechanical Engineering
2006

Robert Szydłowski

School of Graduate Studies
Ryerson University

ABSTRACT

The elastic constants and the bulk modulus of six FCC monoatomic metals were calculated using molecular dynamics simulation as functions of temperature. In addition, the coefficients of linear thermal expansion were also calculated for all six metals. A fully functional 3D molecular dynamics code capable of simulating single crystals composed of one type of atom was written for this study. Eight different embedded-atom method (EAM) interatomic potential models were used to simulate the properties of the six metals. The elastic constants were calculated using fluctuation formulas which are functions of both potential energy, as well as the momentum of the particles in the system. It was found that the temperature dependence of elastic constants is not well exhibited by all EAM potentials. However, the eight EAM potentials presented in this study show results comparable to the values and trends of experimental data.

ACKNOWLEDGEMENTS

I would like to extend my thanks and appreciation to my supervisors Dr. Zouheir Fawaz and Dr. Kamran Behdinan for their guidance and support through out this thesis. The time and helpful discussions of Dr. Jason Lassaline from Ryerson University, Quiping Bian, Dr. Marek Niewczas, and Li Pan from McMaster University were also invaluable. I would like to thank my colleagues Simon Buhajczuk, Kam Shahid, Goddard, and Brian Leonard for their suggestions throughout the course of this study.

I would also like to thank my parents Joanna and Jerzy, and my brothers Krzysztof and Andrzej Szydlowski for their support. Finally, I would like to acknowledge the encouragement, patience, and constant support of Valbona Zacaj. You were always there for me, you were my biggest supporter, and for that I thank you.

TABLE OF CONTENTS

AUTHOR'S DECLARATION	iii
ABSTRACT.....	v
ACKNOWLEDGEMENTS	vii
TABLE OF CONTENTS	ix
LIST OF FIGURES	xiii
LIST OF TABLES	xv
NOMENCLATURE.....	xvii

Chapter 1: Introduction	1
-------------------------------	---

Chapter 2: Molecular Dynamics Simulation.....	7
2.1 Introduction.....	7
2.2 Fundamentals	8
2.3 Dynamics	9
2.3.1 Newtonian Dynamics.....	9
2.3.2 Hamiltonian Dynamics	10
2.4 Phase-Space Trajectories	11
2.5 Boundary Conditions	12
2.6 Interatomic Potential Models.....	14
2.6.1 Truncated Potential	16
2.6.2 Shifted-Force Potential	17
2.7 Simulation.....	18
2.7.1 Preliminaries	19
2.7.2 Gear Predictor-Corrector Algorithm.....	19
2.7.3 Time-Step Size.....	22
2.7.4 Initial Conditions for Atoms	23
2.7.5 Equilibration	24
2.7.6 Identification of Equilibrium	25
2.7.7 Production.....	26
2.8 Temperature clamp	26
2.9 Microscopic Stress Tensor.....	27
2.10 Zero-Pressure Simulation.....	28

Chapter 3: Embedded-Atom Method Potential.....	31
3.1 Introduction.....	31
3.2 Equations used in MD simulation in EAM form.....	33
3.3 Developing EAM potentials	34
3.3.1 Mei's <i>et al.</i> EAM potential model	34
3.3.2 Johnson's and Voter's and Chen's EAM potentials	38
3.4 Form of EAM potentials	38

Chapter 4: Calculation of Elastic Constants and Bulk Modulus	39
4.1 Bulk modulus calculation at absolute zero	39
4.2 Calculation of elastic constants at absolute zero.....	40
4.3 Calculation of elastic constants at finite temperatures.....	41
4.4 Voigt notation	42
Chapter 5: Molecular Dynamics Simulation Code	45
5.1 Introduction.....	45
5.2 Numerical approaches.....	46
5.3 Validation of code.....	47
5.4 Simulation	48
Chapter 6: Results and Discussion	49
6.1 Simulation Parameters	49
6.2 Elastic Constants and Bulk Modulus	51
6.2.1 Cu - Copper.....	51
6.2.2 Ag – Silver	57
6.2.3 Au – Gold.....	59
6.2.4 Ni – Nickel.....	62
6.2.5 Pd – Palladium	64
6.2.6 Pt – Platinum.....	67
6.3 Coefficients of Linear Thermal Expansion.....	68
Chapter 7: Conclusion.....	71
7.1 Future work.....	72
References.....	73
APPENDIX A: Lennard-Jones and Shifted-Force Potential.....	77
A.1 Lennard-Jones Potential	77
A.2 Shifted-Force Potential	79
APPENDIX B: Bulk Modulus Derivation	81
APPENDIX C: EAM Potential Fitting Parameters.....	85
APPENDIX D: EAM Potential Function Plots for Cu.....	87
APPENDIX E: Tabulated Results of MD Calculations.....	89
E.1 Cu – Copper (MEI).....	89
E.2 Ag – Silver (MEI).....	90
E.3 Au – Gold (MEI)	91
E.4 Au – Gold (JON)	91
E.5 Ni – Nickel (MEI)	92
E.6 Ni – Nickel (VC)	93
E.7 Pd – Palladium (MEI).....	94
E.8 Pt – Platinum (MEI)	94

APPENDIX F: Simulation Procedures	95
F.1 Zero-stress simulation.....	95
F.2 Constant volume simulation	97
 APPENDIX G: Source Code.....	 101
G.1 Source code for MDROB.c.....	101
G.2 Source code for MDROB_BORN.c.....	129
G.3 Source code for mdrob.m.....	145
G.4 Source code for mdrob_disp_born.m.....	152
G.5 Source code for MEI EAM potential generation	153

LIST OF FIGURES

Figure 2.1: Two-dimensional representation of periodic boundary conditions. The primary cell containing all of the crystal atoms is in the center (filled circles). Image cells with image atoms (empty circles) of the primary cell surround it from all sides filling all empty space.	13
Figure 2.2: Comparison of Lennard-Jones potential and shifted-force potential versus the distance between a pair of atoms with a cutoff distance of $r_c = 2.5$. Note that the shifted-force potential was 'shifted' vertically upwards and the new potential is zero at $r^* = 2.5$. Units here are $r^* = r / \sigma$ and $u^* = u / \epsilon$ where $\sigma = 1$ and $\epsilon = 1$	17
Figure 2.3: Face-centered-cubic crystal [29].	23
Figure 3.1: Potential energy vs. lattice constant size plot for Cu using Mei's et al. EAM potential. Rose's et al. potential energy plot is included for comparison.	37
Figure 5.1: Total, potential, and kinetic energy as a function of iterations for Cu at 300K.	47
Figure 6.1: Elastic constants (a) and bulk modulus (b) for Cu as a function of temperature. MD results obtained using the MEI potential are compared to experimental work by Overton et al.[5] and Chang et al.[9].	51
Figure 6.2: Fluctuation (a) and Born term (b) components for Cu as a function of temperature using the MEI potential.	53
Figure 6.3: Comparison of elastic constants obtained using MD for Cu using the MEI potential and the Sutton-Chen potential based on work by Cagin et al. [26]. The two MD results are compared to experimental work by Overton et al. [5] and Chang et al. [9].	55
Figure 6.4: Elastic constants (a) and bulk modulus (b) for Ag as a function of temperature. MD results obtained using the MEI potential are compared to experimental work by Neighbours et al.[6] and Chang et al.[9].	57
Figure 6.5: Elastic constants (a) and bulk modulus (b) for Au as a function of temperature. MD results obtained using the MEI and the JON potential are compared to experimental work by Neighbours et al. [6] and Chang et al.[9].	59
Figure 6.6: Elastic constants (a) and bulk modulus (b) for Ni as a function of temperature. MD results obtained using the MEI and VC potentials are compared to experimental work by Alers et al. [11].	62
Figure 6.7: Elastic constants (a) and bulk modulus (b) for Pd as a function of temperature. MD results obtained using the MEI potential is compared to experimental work by Walker et al.[7] and Weinmann et al.[10].	64

Figure 6.8: Results of the MD calculations of the elastic constants for Pd using the MEI potential and EAM based on work by Wolf et al.[25]. Experimental values of the elastic constants and bulk modulus were given by Walker et al. [7] for temperatures from 0 K to 100 K, by Weinmann et al. [10] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.	66
Figure 6.9: Lattice constant size for Cu as a function of temperature using the MEI potential..	68
Figure A.1: Lennard-Jones (12,6) pair potential with the attractive and repulsive component. Units here are $r^* = r / \sigma$ and $u^* = u / \epsilon$ where $\sigma = 1$ and $\epsilon = 1$	78
Figure A.2: Lennard-Jones (12,6) pair potential and pair force functions	79
Figure D.1: Pair-potential for Cu as a function the distance between a pair of atoms based on the MEI potential.	87
Figure D.2: Electron density for Cu as a function of distance between a pair of atoms based on the MEI potential.	87
Figure D.3: Embedding function for Cu as a function of the total electron density of an atom based on the MEI potential.	88
Figure F.1: Shape identification for flow charts.	95
Figure F.2: Zero-pressure, variable volume simulation flow chart.....	96
Figure F.3: Constant volume simulation flow chart.	98

LIST OF TABLES

Table 2.1: <i>Values of α_i parameters in Gear's fifth-order predictor-corrector algorithm.</i>	21
Table 3.1: <i>Number of atoms in each shell for a perfect FCC crystal.</i>	35
Table 6.1: <i>Potentials used in this study.</i>	49
Table 6.2: <i>Results of the MD calculations of the elastic constants and bulk modulus for Cu using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Overton et al.[5] for temperatures from 0 K to 100 K, by Chang et al.[9] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.</i>	52
Table 6.3: <i>Potential energy of a Cu crystal as a function of lattice constant size at absolute zero.</i>	56
Table 6.4: <i>Results of the MD calculations of the elastic constants and bulk modulus for Ag using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Neighbours et al.[6] for temperatures from 0 K to 100 K, by Chang et al.[9] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.</i>	58
Table 6.5: <i>Results of the MD calculations of the elastic constants and bulk modulus for Au using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Neighbours et al.[6] for temperatures from 0 K to 100 K, by Chang et al.[9] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.</i>	60
Table 6.6: <i>Results of the MD calculations of the elastic constants and bulk modulus for Au using the JON potential. Experimental values of the elastic constants and bulk modulus were given by Neighbours et al.[6] for temperatures from 10 K to 100 K, by Chang et al.[9] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.</i>	60
Table 6.7: <i>Results of the MD calculations of the elastic constants and bulk modulus for Ni using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Alers et al.[11].</i>	63
Table 6.8: <i>Results of the MD calculations of the elastic constants and bulk modulus for Ni using the VC potential. Experimental values of the elastic constants and bulk modulus were given by Alers et al.[11].</i>	63

Table 6.9: <i>Results of the MD calculations of the elastic constants and bulk modulus for Pd using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Walker et al.[7] for temperatures from 0 K to 100 K, by Weinmann et al.[10] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.</i>	65
Table 6.10: <i>Results of the MD calculations of the elastic constants and bulk modulus for Pt using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by MacFarlane et al. [8].</i>	67
Table 6.11: <i>Coefficients of linear thermal expansion for all eight EAM potentials and their corresponding experimental values [40].</i>	69
Table B.1: <i>Constant C values for cubic lattices.</i>	83
Table C.1: <i>Coefficients c_i and model parameters for the six FCC crystals based on work by Mei et al. [22]. ϕ_e is in eV while all other parameters are dimensionless.</i>	85
Table C.2: <i>Material properties used in the creation of EAM potentials based on work by Mei et al. [22].</i>	85
Table E.1: <i>Results for Cu using the MEI potential.</i>	89
Table E.2: <i>Results for Ag using the MEI potential.</i>	90
Table E.3: <i>Results for Au using the MEI potential.</i>	91
Table E.4: <i>Results for Au using the JON potential.</i>	91
Table E.5: <i>Results for Ni using the MEI potential.</i>	92
Table E.6: <i>Results for Ni using the VC potential.</i>	93
Table E.7: <i>Results for Pd using the MEI potential.</i>	94
Table E.8: <i>Results for Pt using the MEI potential.</i>	94

NOMENCLATURE

a	- lattice constant size (length of a unit cube side)
a_0	- equilibrium lattice constant size
k_B	- Boltzmann's constant
m	- mass of atom, also a parameter used in the Lennard-Jones potential
n	- parameter used in the Lennard-Jones potential
$\mathbf{p}$	- momentum vector
r	- distance away from atom
$\mathbf{r}_a$	- position vector of atom a
r_{ab}	- distance between atoms a and b
r_c	- cut-off distance
t	- time
u	- potential energy
u_s	- shifted-potential energy
v	- velocity scalar
$\mathbf{v}$	- velocity vector
B	- bulk modulus
BC	- boundary condition
BCC	- body-centered cubic
C	- number of atoms in one lattice crystal
C_{ijkl}	- elastic constants
C_{ij}	- elastic constants in Voight notation
CPU	- central processing unit
E	- total energy
E_{coh}	- cohesive energy
E_{cry}	- energy of a crystal
E_k	- kinetic energy
E_{solid}	- internal energy of a solid
EAM	- embedded-atom method
Exp	- experimental results
F	- force scalar
$\mathbf{F}$	- force vector
FCC	- face-centered cubic
FEM	- finite element method
Fluct	- fluctuation term
JON	- EAM potential based on work by Johnson [30]
L_0	- initial length
MD	- molecular dynamics
MEI	- EAM potential based on work by Mei et al. [22]
MS	- multiscale
N	- number of atoms
P	- pressure
P_{ij}	- microscopic stress tensor in ij direction

SF	- scaling factor
T	- temperature
T_{Corr}	- temperature correction term
U	- total potential energy
V	- volume
VC	- EAM potential based on work by Voter and Chen [32],[33]
Δt	- time-step size
ΔL	- change in length
α	- coefficient of linear thermal expansion
$\mathcal{H}$	- Hamiltonian function
σ	- standard deviation, also distance to zero in Lennard-Jones potential
μ	- mean
λ	- embedding function
ϕ_{ab}	- pair-potential between atoms a and b
ρ^{at}	- atomic electron density
$\rho_{a,total}$	- electron density at atom a due to all surrounding atoms
$\rho_{total,e}$	- electron density at atom a due to all surrounding atoms at equilibrium
χ	- reduction factor used in velocity scaling
Ω	- equilibrium atomic volume
β	- Unitless fitting parameter in the MEI potential
γ	- Unitless fitting parameter in the MEI potential
δ	- Unitless fitting parameter in the MEI potential
ϕ_e	- Fitting parameter in the MEI potential

Chapter 1: Introduction

In today's world, there is a need to study materials and their properties at both large and small length scales. New materials such as nanocomposites and nanotube reinforced materials are showing increasing potential due to their advantageous properties such as increased strength, corrosion and fire resistance [1], to name a few. Other aspects include the need to study phenomena such as nanosized crack propagation in order to increase the lifespan and safety of developed products. Mechanical properties of metals have been an important area of study due to the wide use of metals in engineering applications. Understanding the mechanics of these solids allows for the development of stronger and tougher materials.

Numerical simulations have begun to play an important and crucial role in the study of these materials. Molecular dynamics (MD) simulations have been widely used in the study and analysis of the materials and their properties at the molecular level. The embedded-atom method (EAM) introduced by Daw and Baskes [2] is being increasingly used in the development of interatomic potentials utilized in simulations.

The procedure of developing EAM potentials usually consists of fitting potential functions to some material properties at 0 K. The potentials are then used at a prescribed temperature, usually room temperature, for which they have been developed. Although these potentials have served well in various applications, there is a lack of information regarding the temperature dependence of the material properties using the EAM potentials.

The aim of this study is to fill this gap and provide a set of six monoatomic crystals and their properties as functions of temperature using MD simulations. Material properties include elastic constants, bulk modulus, and coefficients of linear thermal expansion, which are compared to experimental data.

A numerical simulation approach was chosen as it is relatively inexpensive to implement and easier to apply to a wide range of problems. However, MD simulations are not entirely independent of empirical data. As mentioned before, the interatomic potentials are fitted to data at 0 K, which are derived and approximated through empirical means.

Experimental testing has always been the first source of data for material properties. Testing single crystals has been the priority in the early stages as it was the simplest for

comparison to theory. A number of techniques have been devised to study single crystals and their properties. These include the oscillator technique used at liquid nitrogen temperatures to study sodium [3] and aluminum [4]. Ultrasonic pulse techniques based on the theory of plane elastic wave propagation were used to measure elastic constants in pure copper [5], silver, gold [6], palladium [7] and platinum [8] below room temperature. This technique has been extended to higher temperatures for copper, silver, gold [9] and palladium [10]. Another method composed of cooling of a material was used to measure the changes in the relative alignment of the magnetic moments which in turn changed the energy levels in the material. These changes along with the measurement of applied strains were used to calculate the elastic constants of nickel from 4.2 K to 760 K [11]. Most of the above mentioned data can be found in a compilation made by Simmons and Wang [12]. More recently, nanoindentation [13] has been used to quantitatively measure the local modulus of elasticity and hardness of bulk materials.

However, experimental data might be hard to obtain in some cases as either it may be too expensive to perform the experiments or other factors such as impurities may invalidate the results. This is the reason why numerical simulations have been introduced to overcome these issues in the past but have not really taken off until recently as more powerful computers are becoming readily available.

Finite element methods (FEM) have been successfully applied to larger scale analysis and have been used in a wide variety of applications. However, at smaller length scales, going right down to the molecular composition of matter, molecular dynamics methods have become the standard for simulating models.

The fundamental task in MD simulations is to solve an N -body problem using classical mechanics. MD can be used to describe the motion of individual molecules in models of solids, liquids, and gases. It is based on an idea that a system's behavior can be computed if the following conditions for the system's components are satisfied: the initial conditions and the forces of interaction are described.

It is also important to note that the MD simulation is as successful as the interatomic potential used to describe the model of interactions. Depending on the type of system being solved, different types of potentials are applied. One of the simplest examples is the Lennard-Jones potential [14] which is a two-body interatomic potential. It has worked well in various areas of MD simulations but has not performed as well in the case of metals. Pair-potentials were

shown that they cannot accurately describe the elastic properties of metals [2]. Therefore, many-body potentials were introduced such as the embedded-atom method [2], Sutton-Chan [15], and Finnis-Sinclair [16],[17] potentials. EAM potentials have been shown to work well with metals and allowed for the inclusion of fractures, surfaces, impurities, and alloying additions. For these reasons and future applications, EAM potentials have been chosen for this study.

Furthermore, it must be stated that when using numerical computations for solving the behavior of any system, an attainable solution depends on the processing power of the available computers. MD simulations become very expensive computationally when the motions of a large number of atoms, in the order of 10,000 atoms or more, need to be solved. To overcome this problem, multiscale (MS) methods have been introduced.

There have been a number of methods developed for linking both the macro and micro scale and creating an MS method capable of explaining the behavior of materials at both scales. The pretext behind MS methods is that there is a need of a sufficient level of resolution in the immediate vicinity of some areas such as embedded particles, voids, or other defects of interest in order to capture the local atomic rearrangements. While far away from the defect, where the interatomic effects are considered negligible, continuum mechanics should suffice. MS methods seem promising in the area of calculating mechanical properties of nanocomposites. One such application could be the calculation of macroscopic material properties of a nanocomposite made of nanosized particles embedded in a pure metal matrix. Some of the MS methods being developed are the quasicontinuum method [18] and the heterogeneous MS method [19] which are all based on solving a macroscopic model using data obtained from microscopic solutions.

However, this study is directed towards another proposed method called the molecular element method (MEM) [20] which is based on the coupling of MD simulations with FEM. The procedure of MEM is described as follows. A specimen of interest is divided up into a number of elements. If divided up properly, then one element could be used to represent other, but not all, elements in the specimen. Hence, groups of elements are created where each group can be represented by any one of the elements inside of its group. Only one element from each group is then solved using MD simulation under some specified conditions, such as a strain. The physical parameters, such as elastic constants, are then extracted from the simulations and can be used to describe the mechanical properties of each element in its appropriate group. These physical parameters are then plugged into a finite element model of the specimen and solved under the

same conditions. Element stresses are subsequently obtained from the FEM analysis. The calculated stresses are then plugged back into the MD model and the representative elements are again computed. The difference between the physical parameters from the MD simulations and FEM analysis is calculated. If the error is too big, the procedure is repeated until the error is acceptable. From there, the material can be described as a bulk or specific points of interest can be looked at in more detail.

The first step in MEM, however, is to properly calculate the elastic constants using MD simulations. There are a number of ways available, yet not all are easily implemented in MD simulations or they cannot be used at all finite temperatures. The simplest case is to find the elastic constants at absolute zero temperature since there is no motion of the molecules and hence only potential energy is present in the system. The Bulk modulus can be easily calculated using the change in potential energy with respect to the lattice constant size as shown in Chapter 4. The shear constant can be found using a similar method, and from there, the elastic constants can be derived.

Another method is to use a function based on the positions of the particles in the system and the potential energy between them as shown in [2]. This function was said to be composed of Born terms in later studies after Born's [21] work and the same notation will be used here. In [2], an EAM potential was fitted to the properties of nickel and palladium at room temperature. However, it was not shown how it would perform at other finite temperatures because the Born terms are not directly a function of temperature. This suggests that the method is limited to the specified temperature for which the EAM potentials were designed as there are no terms to account for the momentum of the particles. Mei *et al.* [22] created a number of analytic EAM potentials for FCC metals and showed that they can be used to predict material properties such as elastic constants which were shown to be in good accordance with experimental data. However, this again was limited to room temperature only.

To fix this problem, Ray and Rahman [23] introduced a fluctuation formula composed of the Born terms, a fluctuation term based on the momentum of the molecules and a temperature correction term. A simplified version of the formula was used to calculate the elastic constants of a crystalline system in equilibrium [24]. The study was used more to show that the fluctuation formulas can be used in the calculation of elastic constants rather than being an actual application to a specific metal or material. One of the first comprehensive studies of the elastic

constants as a function of temperature was done by Wolf *et al.* [25] on a model of palladium. Two new EAM potentials were developed in that study which used third and fifth neighbour interactions. These showed a much better agreement with experimental data than nearest neighbor models used in previous studies. Another study by Cagin *et al.* [26] showed thermal and mechanical properties as a function of temperature of some FCC metals using the Sutton-Chen potential [15]. These two studies were used for comparisons in this work.

As stated at the beginning of this chapter, the purpose of this study was to calculate the elastic constants, bulk modulus, and coefficients of linear thermal expansion for six monoatomic crystals. As a first step, a full 3D MD code was developed, with the theory behind it given in Chapter 2. Next, the EAM potentials were created as described in Chapter 3. Methods of calculating the elastic constants and bulk modulus were examined and are discussed in Chapter 4. The developed MD code, MDROB, is described in Chapter 5. The results of this study which include the calculated elastic constants and bulk modulus, along with comparisons to experimental data, are presented and discussed in Chapter 6. Concluding remarks along with recommendations for future studies are given in Chapter 7.

Chapter 2: Molecular Dynamics Simulation

2.1 Introduction

Molecular dynamics simulation is a technique of calculating the motions of individual molecules in models of solids, liquids and gases using classical mechanics. The theory behind MD simulations has been around for a number of years; however, it has only become more feasible as more powerful digital computers become readily available. The motion being computed is based on the change of positions, velocities, and orientations in time. Molecular dynamics gives an idea of how molecules travel in space, turning, twisting, and interacting with one another and with the imposed boundaries. Molecular dynamics is essentially based on an idea that the behavior of a system can be computed if the initial condition of the system parts and the forces of interaction between them are known and defined.

With the help of today's digital computers, complicated systems can now be computed. However, there must be some attention paid to the fact that not all systems can be easily solved. If the components of the system act nonlinearly, the system's behavior can be unpredictable. Two factors must be true in order to successfully create and solve a system. The first one is that it must be deterministic, meaning that the system outputs are connected in one way or another to the system inputs. The second one is that it must be calculable, meaning that there exists an algorithm that can be used to find the system outputs if inputs are known. Hence, one would hope to get a system that is predictable, which consists of calculable situations where the algorithm can be numerically represented to compute the results [27].

Due to the wide application of molecular dynamics, the following description of MD simulations is with respect to solid systems only, and later on to metals in particular.

2.2 Fundamentals

In order to perform MD simulations, the system in question needs to be described. The interaction of the components in the system can be divided up into two models. One deals with the interactions between the molecules making up the system and the other deals with the interactions between the molecules and their environment.

The model consisting of the interactions between the molecules is based on the intermolecular potential energy function. The potential function specifies the geometric shapes of the individual molecules. In this study, the problem is further simplified by only considering spherically symmetric molecules. In other words, only individual atoms are present in the model since only pure metals are dealt with here.

The intermolecular potential can be written as $U(\mathbf{r}^N)$ where N is the number of atoms in the system, and $\mathbf{r}^N$ is a set of vectors used to locate the atomic centers of mass, $\mathbf{r}^N = \{\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N\}$. The set of values that define $\mathbf{r}^N$ are called the *configuration* of the system.

In order to look at a system as a whole, all the potential energies between each possible isolated pair are summed. This is called pairwise additivity and is given by

$$U(\mathbf{r}^N) = \sum_a \sum_b u(r_{ab}), \quad a < b \quad (2.1)$$

where $u(r_{ab})$ is the pair potential energy function and r_{ab} is the scalar distance between molecules a and b given by $r_{ab} = |\mathbf{r}_a - \mathbf{r}_b|$. The force vector on molecule a is given by

$$\mathbf{F}_a = - \frac{\partial U(\mathbf{r}^N)}{\partial \mathbf{r}_a} \quad (2.2)$$

where $\partial/\partial r$ denotes the derivative with respect to distance. It has been assumed that repulsive forces are positive and attractive forces are negative [27].

The second model mentioned above deals with the system-environment interactions which essentially designates the imposed boundaries. These are usually stipulated by the type of

situation being simulated. There are two main ways of applying boundary conditions to a model and are described in section 2.5.

2.3 Dynamics

A simplified description of the process of performing MD simulation is explained here using a concept of phase-space trajectories. The trajectories are computed using classical mechanics which include Newtonian and Hamiltonian dynamics. The features of the trajectories are described using classical nonlinear dynamics. Kinetic theory, statistical mechanics, and sampling theory are then used to obtain properties. The trajectories are tested by probing constraints imposed by periodic boundary conditions and then by conservation principles. Once all of the above is combined, we have the foundations of molecular dynamics simulation.

2.3.1 Newtonian Dynamics

Three main laws are extracted from Newtonian dynamics that dictate the translational motion of spherical molecules when exerted on by force F_a . Newton's first law stipulates that an object in motion will remain in motion and an object at rest will remain at rest until acted upon by an external force. This can be represented mathematically as following.

$$\frac{\partial \mathbf{r}_a}{\partial t} = \mathbf{v}_a = \text{const} \quad (2.3)$$

where $\mathbf{v}_a$ is the velocity vector of molecule a .

In this case, the object will be a molecule. The second law explicitly relates the motion of a molecule to the applied force using

$$\mathbf{F}_a = m \frac{\partial^2 \mathbf{r}_a}{\partial t^2} \quad (2.4)$$

where m is the mass of the molecule and is assumed independent of the position, velocity, and time.

The third law dictates that to every action, there is an equal and opposite reaction. In other words, if we consider two molecules, the force exerted by molecule 1 on molecule 2 must be balanced by a force exerted by 2 on 1,

$$\mathbf{F}_{total} = \mathbf{F}_1 + \mathbf{F}_2 = 0 \quad (2.5)$$

Or, rewritten as,

$$\mathbf{F}_1 = -\mathbf{F}_2 \quad (2.6)$$

All of these laws can now be combined to define kinetic energy as the amount of work required to move a molecule from rest to velocity $\mathbf{v}$.

$$E_k = \frac{1}{2} m \mathbf{v}^2 \quad (2.7)$$

2.3.2 Hamiltonian Dynamics

From Newtonian dynamics, it can be seen that Newton's second law (2.4) is not time dependant even though molecular forces and positions change with time. Therefore, there must be a function of positions and velocities that is constant with time. This function will be referred to as the Hamiltonian $\mathcal{H}$.

$$\mathcal{H}(\mathbf{r}^N, \mathbf{p}^N) = const \quad (2.8)$$

where $\mathbf{p}$ is momentum and for atom a it is defined as

$$\mathbf{p}_a = m \mathbf{v}_a \quad (2.9)$$

From the definition of an isolated system, total energy is conserved and therefore it can be applied to the Hamiltonian. Total energy is the sum of potential and kinetic energy. For N spherical molecules, the Hamiltonian can be written as follows,

$$\mathcal{H}(\mathbf{r}^N, \mathbf{p}^N) = \frac{1}{2m} \sum_a \mathbf{p}_a^2 + U(\mathbf{r}^N) = E \quad (2.10)$$

To find the Hamiltonian equations of motion, a total time derivative should be taken of (2.10). Then, by simplification and comparison, the following two equations are obtained,

$$\frac{\partial \mathcal{H}}{\partial \mathbf{p}_a} = \frac{\mathbf{p}_a}{m} = \mathbf{v}_a \quad (2.11)$$

$$\frac{\partial \mathcal{H}}{\partial \mathbf{r}_a} = - \frac{\partial U}{\partial \mathbf{r}_a} \quad (2.12)$$

Eq. (2.11) and Eq. (2.12) are called Hamiltonian equations of motion and they represent $6N$ first-order differential equations for a system composed of N spherical molecules. These equations have the same effect as Newton's $3N$ second-order equation, Eq. (2.4).

The purpose of the above descriptions is to show the difference between Newtonian and Hamiltonian dynamics. In Newtonian dynamics, motion is based on the application of forces. In Hamiltonian dynamics, on the other hand, the force term does not explicitly appear in the equations, but simply the motion of the molecules can occur as long as the Hamiltonian function is preserved [27].

2.4 Phase-Space Trajectories

The first step for conducting molecular dynamics simulation was previously stated as the computation of molecular trajectories. It was assumed that this would include the position vectors $\mathbf{r}_i$ but unfortunately this is not sufficient. Molecular momenta will now be added to create a $6N$ -dimensional hyperspace for N atoms. This space consists of $3N$ -dimensional configuration

space which includes the coordinates of each individual atom given by position vectors $\mathbf{r}_i(t)$, and a *3N-dimensional momentum space*, which includes the momentum vectors $\mathbf{p}_i(t)$ in Cartesian coordinates. Such a space is called phase-space. Note that both the positions and momentum are time dependant as both change according to the applied forces and interactions between molecules over time. Hence, the phase-space trajectories replace the name of the trajectory generation step. The trajectories are obtained by numerically solving the Newtonian or Hamiltonian equations of motion [27].

2.5 Boundary Conditions

One of the prevailing limitations in MD simulations is the number of molecules that can be used in the system. Simulations with a small number of atoms are prone to be dominated by surface effects, or in other words, the interactions of the atoms with the container. This is due to the fact that individual atoms are affected by distances which can be larger than the actual dimensions of the cube that contains the atoms.

The simplest way to create a boundary condition (BC) is to allow the atoms to move more ‘freely’ at the surfaces of the system. This implies that the atoms at and close to the surfaces would not be affected by any potentials or forces outside of the described model. The only factors keeping them from traveling too far from the model are the attractive forces of the atoms in the system. This would suggest that the system size would be able to change to accommodate the lowest energy of the system but at the same time it would distort the original parameters of the model. These types of BCs are referred to as free boundary conditions (free BCs).

The problem with free BCs arises when there is a need to simulate the bulk properties of the system with specific parameters. Free BCs introduce some aspects which are not desirable in this study. One problem is that the crystal is not bounded in any way and hence the atoms at the boundaries will experience different interactions than those inside of the crystal. If the atoms of interest are those *inside* the system, a large number of atoms would need to be used to make sure that the atoms *inside* of the system are not in any way affected by the *lack* of atoms outside the system boundaries.

If the surface effects are not the primary interest in the simulation, then periodic boundary conditions can be used [27]. The use of periodic BCs consists of surrounding the primary cell with image cells from all sides. These image cells contain the same number of image atoms and behave exactly the same as the primary cell. This gives the impression that the primary cell is periodically replicated in all directions forming a macroscopic sample of the substance of interest. Figure 2.1 helps to visualize these boundaries.

Figure 2.1: *Two-dimensional representation of periodic boundary conditions. The primary cell containing all of the crystal atoms is in the center (filled circles). Image cells with image atoms (empty circles) of the primary cell surround it from all sides filling all empty space.*

As shown in Figure 2.1, an image of the primary cell is then taken and placed around the primary cell filling all empty space. All image cells will contain the same number and arrangement of atoms as the primary cell. Each image atom in the image cells will have the same position, velocity, acceleration and any other derivative as its corresponding primary atom in the primary cell.

The image cells are all referenced to the primary cell by the use of vectors so that the position of each cell is known. Cells are separated from each other using open boundaries which allow the passage of atoms from one cell to the other. The number of atoms in each cell always remains constant because as one atom leaves the primary cell and goes to an image cell, another atom enters the primary cell from the opposite image cell. Furthermore, only the positions of the atoms in the primary cell need to be stored while the atoms in the image cells are calculated as needed. To calculate the position of an image atom i in an image cell, only the position of the

atom in the primary cell and the image cell position vector are required. The trajectories of image atoms are the exact duplicates of the corresponding atoms in the primary cell.

There are, however, some restrictions imposed on the simulations by the use of periodic boundary conditions. One of these limitations is that only short-lived and short-range phenomena can be studied. Although this is the case, it does not have any effect on the outcome of the simulation because these same limitations were already introduced by the use of a small system size ($N < 1000$) and the small integration time-step needed to solve the equations of motion. In general, it has been shown that the effects of periodic boundary conditions are usually insignificant or overshadowed by other restrictions [27].

2.6 Interatomic Potential Models

The modeling portion of molecular dynamics is composed of two parts. The first is that of developing a model, and the second is the actual simulation. The model development step includes the selection of the form of the intermolecular potential and the derivation of equations of motion. One of the simplest intermolecular potentials is the Lennard-Jones¹ potential and will be used here as an illustrative example. The following is a description of the potential model and how it is modified and simplified for use in molecular dynamics. The equations of motion are based on Newton's second law as only isolated systems will be explored.

The potential energy of a system can be found by looking at the sum of all individual potentials between all distinct pairs of atoms. This is considered a pair-wise additive potential which has the basic form as follows.

$$U(\mathbf{r}^N) = \sum_a \sum_b u(r_{ab}), \quad a < b \quad (2.13)$$

The Lennard-Jones model is given by

¹ For more information on the Lennard-Jones potential model, see Appendix A

$$u(r) = k\varepsilon \left[\left(\frac{\sigma}{r} \right)^n - \left(\frac{\sigma}{r} \right)^m \right] \quad (2.14)$$

where

$$k = \frac{n}{n-m} \left(\frac{n}{m} \right)^{m/(n-m)} \quad (2.15)$$

The purpose of the n and m variables is for the potential function to account both for the short-range repulsive and longer-range attractive dispersion forces between atoms. Subbing 6 for m and $2m$ for n [27] simplifies Eq. (2.14) to

$$u(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right] \quad (2.16)$$

The other parameters such as ε is the minimum energy of $u(r)$, and σ is the distance where $u(r)$ is zero (at equilibrium between the attractive and repulsive components). The force between two atoms can now be written as.

$$F(r) = -\frac{du(r)}{dr} = 24 \frac{\varepsilon}{r} \left[2 \left(\frac{\sigma}{r} \right)^{13} - \left(\frac{\sigma}{r} \right)^7 \right] \quad (2.17)$$

where by convention, repulsive forces are positive, and attractive forces are negative.

However, as stated before, the Lennard-Jones potential poorly describes the elastic behavior of metals. Many-body potentials such as the EAM potentials have become the standard for the study of metals and are used here in this study. A full description of the EAM potentials is covered in Chapter 3.

2.6.1 Truncated Potential

When using N atoms in a system, the number of unique pair interactions can become a problem. In such a system, there are $\frac{1}{2}N(N-1)$ interactions and this number grows with the square of the number of atoms. The solution to this problems is to come up with a certain distance beyond which atom interactions are considered negligible. This distance will be referred to as the cut-off distance, r_c , meaning that only interactions between two atoms where $r < r_c$ will be considered and all others will be neglected. The common choice for r_c is $r_c = 2.5\sigma$ beyond which the forces exerted on an atom by other atoms are minimal. The Lennard-Jones potential is then given as follows

$$u(r) = \begin{cases} 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right] & r \leq r_c \\ 0 & r > r_c \end{cases} \quad (2.18)$$

Since a truncation of the potential is used, only certain properties can be provided by the simulation. These properties include internal energy and pressure which are directly related to potential. Therefore, for long range interactions ($r > r_c$), corrections must be applied.

Truncation of the potential will also have the same truncating effect in the forces which will cause small disturbances on any pair of atoms when the distance between them crosses r_c . Therefore there will be small fluctuations visible in the total energy E , but these are considered negligible. However if there is any need to decrease these fluctuations, r_c can be simply increased at the expense of computational time [27].

2.6.2 Shifted-Force Potential

Small fluctuations can be difficult to deal with when creating a code for the simulation since they can hide small errors. Therefore, for testing purposes only, the potential between atoms can be shifted vertically in order to avoid the step changes in $u(r)$ and $F(r)$ at the cutoff distance. This is illustrated in Figure 2.2.

Figure 2.2: Comparison of Lennard-Jones potential and shifted-force potential versus the distance between a pair of atoms with a cutoff distance of $r_c = 2.5$. Note that the shifted-force potential was 'shifted' vertically upwards and the new potential is zero at $r^* = 2.5$. Units here are $r^* = r / \sigma$ and $u^* = u / \epsilon$ where $\sigma = 1$ and $\epsilon = 1$.

Hence, by shifting the potential vertically, there are no more step changes and there is a smooth transition when crossing r_c . One of the methods of applying the shifted-force is given as follows².

² For derivation of the shifted-force potential equation, see Appendix A

$$u_s(r) = \begin{cases} u(r) - u(r_c) - (r - r_c) \left(\frac{du}{dr} \right)_{r_c} & r \leq r_c \\ 0 & r > r_c \end{cases} \quad (2.19)$$

Although the shifted force will get rid of the step changes, this procedure is not recommended for obtaining properties. Since the potential and force were shifted, a correction would be required which in this case can be very cumbersome. Therefore, this procedure should be used for testing the stability of the code only.

2.7 Simulation

In most cases, the actual simulation is performed for a constant number of atoms, N , constant volume, V , and constant energy, E . The first two parameters would be set at the beginning of the simulation, and the third would be the result of them. In addition, since this study concentrates on the calculation the elastic constants at specific temperatures, isothermal simulations need to be performed. This requires that the temperature of the system needs to be monitored and adjusted as necessary. This aspect of the simulation is explained in section 2.8.

Therefore, the simulation in question is at constant volume, energy, and temperature, and contains a constant number of atoms. However, before the simulation can be performed, the volume of the system needs to be calculated for each separate temperature as the crystals are prone to thermal expansion. For this step, a separate simulation has to be performed. One way of finding the volume of the system at each temperature is to look at the internal stress in the system and modify the volume (size of the crystal) to obtain zero internal stress. This is done through a zero-pressure simulation where the volume is adjusted with every iteration as explained in Section 2.10.

2.7.1 Preliminaries

The preliminaries are composed of the system of units used for the calculations, the finite-difference algorithm to be used, and the assignment of values to the parameters that will remain constant throughout the simulation.

There are three independent thermodynamic properties used in equilibrium molecular dynamics on isolated systems. These are the number of atoms N , system volume V , and total energy E . For molecular dynamics, N is usually in the range of $100 < N < 1000$ in order to sufficiently approximate the phenomena of interest yet not too big as to render the simulation too expensive [27].

The boundary conditions applied in this study will be that for periodic boundary conditions. As far as volume shapes are concerned, there are different ones to choose from as long as they are space filling. The easiest one to deal with is a cube as it provides the simplest periodic BC transformations. The length of each side of the cube is defined by the number of atoms used, lattice configuration, and the lattice constant size.

At this time, the algorithm for solving the equations of motions can be chosen. There are a number of finite-difference methods available with Verlet's algorithm and Gear's predictor-corrector algorithm being some of the more commonly used [27]. The Gear predictor-corrector algorithm has been chosen for this study as it offers greater stability due to the nature of the corrector step that dampens instabilities introduced by the predictor. The algorithm is fully explained as follows.

2.7.2 Gear Predictor-Corrector Algorithm

As the name suggests, this algorithm is based on predicting new positions dictated by the velocity, acceleration and other derivatives of each atom. The new positions of each atom are then corrected using a defined correction term based on the interatomic force at those new positions. Gear's algorithm allows for as many time derivatives of the atomic positions as is acceptable by the available computing power. The 5th order Gear predictor-corrector algorithm

[27] has been chosen for this study. The following is a detailed description of the procedure involved in the algorithm.

The procedure begins by first predicting molecular positions at $t + \Delta t$ using all available atomic position derivatives based on the Taylor series.

$$\mathbf{r}_a(t + \Delta t) =$$

$$\mathbf{r}_a(t) + \frac{\partial \mathbf{r}_a}{\partial t}(t) \Delta t + \frac{\partial^2 \mathbf{r}_a}{\partial t^2}(t) \frac{(\Delta t)^2}{2!} + \frac{\partial^3 \mathbf{r}_a}{\partial t^3}(t) \frac{(\Delta t)^3}{3!} + \frac{\partial^4 \mathbf{r}_a}{\partial t^4}(t) \frac{(\Delta t)^4}{4!} + \frac{\partial^5 \mathbf{r}_a}{\partial t^5}(t) \frac{(\Delta t)^5}{5!} \quad (2.20)$$

The velocity vector would then be obtained by taking the first derivative of Eq. (2.20) with respect to time, the acceleration vector would be obtained by taking the second derivative of Eq. (2.20) with respect to time, and so on up to the fifth derivative.

The intermolecular force $\mathbf{F}_a$ on each molecule at time $t + \Delta t$ is calculated using the predicted positions

$$\mathbf{F}_a = - \sum_{b \neq a} \frac{\partial u(r_{ab})}{\partial r_{ab}} \hat{\mathbf{r}}_{ab} \quad (2.21)$$

where $\hat{\mathbf{r}}_{ab}$ is a unit vector along $\mathbf{r}$.

Using the forces calculated at $t + \Delta t$, the corresponding accelerations $\frac{\partial^2 \mathbf{r}_a}{\partial t^2}(t + \Delta t)$ are calculated. The difference between the predicted and evaluated accelerations is given by,

$$\Delta \ddot{\mathbf{r}}_a + [\ddot{\mathbf{r}}_a(t + \Delta t) - \ddot{\mathbf{r}}_a^P(t + \Delta t)] \quad (2.22)$$

where the first derivative is denoted by a single dot above the variable and a second derivative is denoted by two dots. Higher derivatives are denoted by roman numerals in brackets.

Using the difference term, the predicted positions and their derivatives are corrected as follows.

$$\mathbf{r}_a = \mathbf{r}_a^P + \alpha_0 \Delta R2 \quad (2.23)$$

$$\dot{\mathbf{r}}_a \Delta t = \dot{\mathbf{r}}_a^P \Delta t + \alpha_1 \Delta R2 \quad (2.24)$$

$$\frac{\ddot{\mathbf{r}}_a (\Delta t)^2}{2!} = \frac{\ddot{\mathbf{r}}_a^P (\Delta t)^2}{2!} + \alpha_2 \Delta R2 \quad (2.25)$$

$$\frac{\mathbf{r}_a^{(iii)} (\Delta t)^3}{3!} = \frac{\mathbf{r}_a^{(iii)P} (\Delta t)^3}{3!} + \alpha_3 \Delta R2 \quad (2.26)$$

$$\frac{\mathbf{r}_a^{(iv)} (\Delta t)^4}{4!} = \frac{\mathbf{r}_a^{(iv)P} (\Delta t)^4}{4!} + \alpha_4 \Delta R2 \quad (2.27)$$

$$\frac{\mathbf{r}_a^{(v)} (\Delta t)^5}{5!} = \frac{\mathbf{r}_a^{(v)P} (\Delta t)^5}{5!} + \alpha_5 \Delta R2 \quad (2.28)$$

where

$$\Delta R2 = \frac{\Delta \ddot{\mathbf{r}}_a (\Delta t)^2}{2!} \quad (2.29)$$

α_0 through α_5 for the fifth-order algorithm were calculated [28] to be as follows.

α_i	Value ³
α_0	$\frac{3}{16}$
α_1	$\frac{251}{360}$
α_2	1
α_3	$\frac{11}{18}$
α_4	$\frac{1}{6}$
α_5	$\frac{1}{60}$

Table 2.1: Values of α_i parameters in Gear's fifth-order predictor-corrector algorithm.

³ Gear's [28] original value for α_0 was changed by Haile [27] to $\alpha_0 = 3/16$ to improve results.

It must be noted that at the beginning of the simulation, only the atomic positions, velocities, and accelerations are available and all other derivatives are not yet available and are hence set to zero. However, as the simulation progresses, values will be assigned to all derivatives and all of them will be contributing to the computation of the new trajectories. Since simulations are run with a couple of thousands of iterations, this is not an issue.

2.7.3 Time-Step Size

Subsequently, an integration time step Δt is required for the algorithm. Choosing the optimum Δt can be difficult. The larger the Δt , the more phase space will be sampled. However, it must be smaller than the mean time between molecular collisions in order to avoid instability.

The optimum stable time-step can be found by running a simulation starting at a small time-step, on the order of 1 femtosecond (1 fs), to ensure stability for most systems. If the simulation progresses with no apparent problems, then the time-step can be increased and the stability rechecked. This procedure would be continued until instability would be detected, which would most likely resemble atoms getting too close or even colliding with each other. Checks in the simulation can be implemented to check for unacceptable distances. Another sign of instability would be if one or more atoms traveled too far from the system to be affected by the potential of the remaining atoms. This, however, would be harder to detect when using periodic BCs since atoms traveling beyond the boundaries of the system would simply ‘reappear’ on the opposite side of the crystal.

Even though a stable time-step size can be found using the above procedure, obtaining properties from time-averages may not produce stable data. It has been found in this study that reducing the time-step further, by approximately an order of magnitude, seemed to greatly increase the stability of the entire simulation process.

2.7.4 Initial Conditions for Atoms

There are several parameters that need to be initiated at the start of a molecular dynamics simulation. The primary ones include the initial atomic positions $\mathbf{r}_a(0)$, initial velocities $\mathbf{v}_a(0)$, and initial accelerations $\mathbf{a}_a(0)$.

As discussed before, the atomic positions are measured with respect to an inertial frame. Initial positions can be obtained either from a previous simulation, or they can be obtained from a chosen lattice configuration. There are a number of lattice configurations including simple cubic (SC), body-centered-cubic (BCC), face-centered-cubic (FCC), and hexagonal crystal.

In this study, mechanical properties of pure metals are studied. This task is further limited to pure metals having a face-centered-cubic (FCC) crystal structure such as copper, silver, gold, nickel, palladium and platinum. In an ideal FCC crystal structure, atoms are placed on each corner and on each face of the cube as shown in Figure 2.3.

Figure 2.3: *Face-centered-cubic crystal* [29].

The crystal is created by placing an atom on each lattice position and repeating the procedure to create a crystal of a desired size. The lattice constant size is the length of a side of a single FCC crystal cube.

Initial velocities again can be either taken from a previous simulation or they can be randomly assigned. In the case of molecules, the velocities of molecules are normally distributed for the entire system. A normal distribution in terms of v is defined as follows,

$$f(v; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(v-\mu)^2}{2\sigma^2}\right) \quad (2.30)$$

where σ^2 is the variance (equivalently σ is the standard deviation), and μ is the mean.

From the above equation, the Maxwell-Boltzmann distribution has been obtained for a mean of $\mu = 0$, and a variance of $\sigma^2 = \frac{k_B T}{m}$. The velocity distribution in one direction [27] is

$$f_v(v) = \sqrt{\frac{m}{2\pi k_B T}} \exp\left[\frac{-mv^2}{2k_B T}\right] \quad (2.31)$$

where m is molecular mass, k_B is Boltzmann's constant, T is temperature, and v is velocity.

Normally distributed velocities can be randomly assigned to all atoms in the system in all three directions separately. In this study, all velocities were initialized at the start of each simulation and scaled to the desired temperature.

The initial accelerations are applied by computing the forces between the atoms using the initial atomic positions, and applying Newton's second law. The Gear algorithm also uses third, fourth, and fifth order derivatives of the positions. These, however, can be tedious to calculate, and hence, for simplicity, they are initially assumed to be zero as stated before.

2.7.5 Equilibration

Once initialization has occurred, then the actual simulation can begin. This is the step where the system relaxes from the initially assigned conditions to an equilibrium state. This phase is referred to as *equilibration* and involves the use of the finite-difference algorithm to obtain the phase-space trajectories and the monitoring of the trajectories by tracking system properties such as potential and kinetic energies.

The equilibration phase can be either slow or fast depending on how far the initial conditions are from the equilibrium state. While approaching equilibrium, the averages of most properties will not have stable values, they will, however, start converging to their equilibrium

values while continuously fluctuating. Since only isolated systems are being considered in this study, the total energy of the system should be the only property converging to its equilibrium without any fluctuations.

2.7.6 Identification of Equilibrium

The ideal check if a system has reached equilibrium is to stop when entropy is at a maximum. Unfortunately entropy cannot be evaluated and hence other means of detecting equilibrium must be used. This makes the task of identifying equilibrium very difficult. The only option left is to accumulate a number of conditions and claim that these are sufficient to establish a state of equilibrium. The following are a few conditions that have been proposed.

- The total number of molecules, N , and the total energy, E , should be constant and independent of time.
- Fluctuations in the kinetic energy must be equal in magnitude to the fluctuations in the potential energy but out of phase with each other.
- Thermodynamic properties such as internal energy, temperature, and pressure, should be fluctuating around their respective stable averages. These averages should not depend on the technique used to obtain equilibrium and hence they should be reproducible from different initial positions and initial velocities.
- Small perturbations in property values should not have any effect on their stability, meaning that, when a small perturbation is introduced, the property value should be able to return to its equilibrium average.
- Dividing a system into equal parts should produce the same time averages in all parts.

However, it must be stated that even if all of these conditions are met, it does not necessarily suggest that the system is in a state of equilibrium. It merely means that when more conditions are met, there is a higher probability that the system is in equilibrium [27].

2.7.7 Production

The final step occurs once the assumed equilibrium has been reached. It is called the production phase and is not much different than equilibration. Here, the finite difference algorithm is used to follow the phase-space trajectories and the contributions to properties are accumulated. Any desired quantities can be collected for later analysis.

2.8 Temperature clamp

MD simulations at constant temperature may require a constant check of the kinetic energy of the system to ensure that the desired temperature is kept constant. This is done using a *temperature clamp* which is based on calculating the temperature of the system after each iteration and then rescaling all of the atomic velocities by the same factor so that the desired temperature is obtained. Temperature, T , of the system of atoms is found using [27],

$$T = \frac{2E_k}{3Nk_B} \quad (2.32)$$

where E_k is the kinetic energy of the system and can be obtained using Newtonian dynamics. A scaling factor (SF) is obtained by calculating the actual temperature at the specific instant and comparing it to the desired temperature. Using the same scaling factor for all atoms and directions ensures that a normal velocity distribution is kept.

$$SF = \frac{T_{desired}}{T_{actual}} \quad (2.33)$$

However, it has been found in this study that *clamping* the temperature at each iteration produces a very significant change in the atomic velocities, a change that is considered ‘too abrupt’ for the simulation. This problem becomes visible when looking at the potential and

kinetic energy as functions of iterations. If the temperature is *clamped* at every iteration, the resulting kinetic energy will be a constant value from the start until the end of the simulation. The effect is that the fluctuations in the potential energy will not be equal in magnitude and be out of phase to the fluctuations in the kinetic energy.

This problem has been fixed by *delaying* the rescaling of the resultant velocities by reducing the size of the scaling factor. The delayed scaling factor is found using

$$SF_{delayed} = \chi \sqrt{SF} \quad (2.34)$$

where χ is a reduction factor. As an example, if $\chi = 10$, it has the effect of the instantaneous temperature reaching the desired temperature after 10 iterations.

It has also been found that the reduction factor needs to be bigger for smaller time-step sizes. This is due to the fact that at a bigger time-step size, the atoms travel a greater distance during each iteration and hence need to be clamped ‘more often’.

2.9 Microscopic Stress Tensor

One of the useful properties that can be obtained directly from the phase-space trajectories is the microscopic stress tensor. It can be derived from the virial theorem [27] and is given as

$$P_{ij} = \frac{1}{V} \left(\sum_a^N \frac{p_{ai} p_{aj}}{m} - \sum_{\substack{a,b=1 \\ a < b}} \frac{\partial u_{ab}}{\partial r_{ab}} \frac{x_{abi} x_{abj}}{r_{ab}} \right) \quad (2.35)$$

where V is the volume of the system containing the N atoms, p_{ai} is the momentum of particle a in the i th direction, x_{abi} is the i th component of the relative position vector of particle a and b , and r_{ab} is the distance between particle a and b .

2.10 Zero-Pressure Simulation

As mentioned before, the volume of the system needs to be calculated for each temperature. Due to thermal expansion, a crystal will expand in size as the temperature increases. To find the equilibrium lattice constant size, the stress tensor Eq. (2.35) can be used to find the state of the crystal where it will not experience any stress, pressure, or strain, both internal and external. A crystal in equilibrium will experience zero-stress at the equilibrium crystal size. This means that as the temperature rises, the size of the crystal will need to be adjusted to create a zero-stress condition.

Only hydrostatic pressure, P , will be considered in this study as all external stresses, pressures, or strains are assumed zero or negligible. Internal pressure can be calculated using the stress tensor as given by

$$P = -\left(\frac{P_{11} + P_{22} + P_{33}}{3}\right) \quad (2.36)$$

The volume of the crystal can be found using the definition of the bulk modulus, B .

$$B = -V \frac{dP}{dV} \quad (2.37)$$

The above equation can be rearranged as follows to obtain the unit volume of a crystal.

$$\frac{1}{V} dV = -\frac{1}{B} dP \quad (2.38)$$

Integrating both sides of the equation and solving for V ,

$$\ln(V) = -\frac{1}{B} P \quad (2.39)$$

$$V = \exp\left(-\frac{P}{B}\right) \quad (2.40)$$

Volume of a crystal can be rewritten as the length of a cube side, a .

$$V = a^3 \quad (2.41)$$

Rewriting Eq. (2.40), a relationship between the length of a crystal cube side and pressure is obtained.

$$a = \exp\left(-\frac{P}{3B}\right) \quad (2.42)$$

Note that when $P = 0$, the crystal cube side $a = 1$. This equation is for a unit cube representation of the crystal. To obtain the actual lattice constant size of a crystal cube under pressure P using the equilibrium lattice constant size at absolute zero, a_0 , Eq. (2.42) is rewritten as

$$a_{actual} = a_0 \exp\left(-\frac{P}{3B}\right) \quad (2.43)$$

Using Eq. (2.43) it is now possible to modify the size of the lattice constant at each iteration to maintain a zero-pressure state. Note that P will not be zero throughout an MD simulation, but it will fluctuate around zero and hence causing a_0 to be continuously adjusted.

Just as in the case of the temperature clamp, the lattice constant size cannot be adjusted to a zero-pressure state at every iteration. Therefore, the same method is used here where the *adjustment* of the lattice constant is delayed by the χ' th root. This gives a smooth transition where the effects of χ become irrelevant when the number of iterations used is far greater than χ .

This zero-pressure, variable volume simulation is only used to find the equilibrium lattice constant size at each temperature. Once complete, constant-volume simulations are used for the remainder of the calculations.

Chapter 3: Embedded-Atom Method Potential

3.1 Introduction

As mentioned before, there is a need to properly define the potential between the atoms in the system. Two-body interatomic potentials have been initially used in studying metals but have been shown to be deficient. This is due to the fact that they cannot properly describe elastic properties of metals because they require the use of an accompanying volume-dependant energy and hence any ambiguity about the volume automatically invalidates the results.

The embedded-atom method [2] has been introduced to deal with metallic systems where fractures, surfaces, impurities, and alloying additions can be included.

The basic form of the EAM potential is

$$E_{pot} = \sum_{a=1}^N \lambda_a(\rho_{a,total}) + \frac{1}{2} \sum_{\substack{a,b=1 \\ a \neq b}}^N \phi_{ab}(r_{ab}) \quad (3.1)$$

where λ is the embedding energy, ϕ_{ab} is the pair potential between atoms a and b , and $\rho_{a,total}$ is the total electron density of atom a due to all other atoms and is given by

$$\rho_{a,total} = \sum_{\substack{b=1 \\ b \neq a}}^N \rho_b^{at}(r_{ab}) \quad (3.2)$$

where ρ^{at} is the atomic (*at*) electron density.

The electron density, $\rho_{a,total}$, is the electron density at the host atom a due to all other atoms without atom a . The embedding function is the energy required to embed an atom in an ‘electron sea’ created by the presence of all other surrounding atoms. The pair potential is included in Eq. (3.1) to account for the core-core repulsions present. It assumes an exponential form for short-range pairwise repulsions such as a fitted version of the Lennard-Jones potential.

In order to use EAM potential, the electron density function, the embedding function and the pair-potential function have to be defined either using an analytical function or numerically.

The pair potential usually takes on a form of an exponential function based on the repulsive and attractive forces between atoms. Some pair-potential functions have been simplified to only include the repulsive component [2],[30] while others include both the repulsive and attractive components [22],[25]. Also, the electron density function is usually approximated using an exponential function based on the electron density for a pair of atoms.

Once the electron density and pair-potential functions have been assumed, the embedding function can then be obtained using an approximation for the energy of a perfect crystal at absolute zero as given by Rose *et al.* [31].

$$E_{cry}(a) = -E_{coh}(1 + a^*) \exp(-a^*) \quad (3.3)$$

The above equation is given as function of the lattice constant size a where E_{cry} is the internal potential energy of the crystal, E_{coh} is the cohesive energy, and a^* is given by

$$a^* = \left(\frac{a}{a_0} - 1 \right) \left(\frac{9B\Omega}{E_{coh}} \right)^{1/2} \quad (3.4)$$

where B is the bulk modulus, Ω is the equilibrium atomic volume, and a_0 is the equilibrium lattice constant size. The cohesive energy, E_{coh} , of a solid is the energy required to break the atoms of the solid into isolated atomic species and is given as follows.

$$E_{coh} = E_{solid} - \sum_a E_a^{iso} \quad (3.5)$$

where E_{solid} is the energy of a solid, and E_a^{iso} is the energy of an isolated atom a .

The embedding function can then be found by combining Eq. (3.1) and Eq. (3.3) for one atom in a perfect crystal at absolute zero as follows.

$$\lambda(a) = -E_{coh}(1 + a^*) \exp(-a^*) - \frac{1}{2} \sum_{\substack{a,b=1 \\ a \neq b}} \phi_{ab}(r_{ab}) \quad (3.6)$$

In the above function, the embedding function can then be converted to become a function of electron density, $\rho_{a,total}$, and not lattice constant size, a . The conversion can be either done analytically or numerically depending on the complexity of the electron density and pair-potential functions.

3.2 Equations used in MD simulation in EAM form

Since EAM potentials are being used in this study, the form of some equations presented in Chapter 2 needs to be changed. Force on atom a , F_a , for the EAM potential is given by

$$\mathbf{F}_a = - \sum_{b \neq a} \left[\frac{\partial \lambda_a}{\partial \rho_{a,total}} \frac{\partial \rho_b^{at}}{\partial r_{ab}} + \frac{\partial \lambda_b}{\partial \rho_{b,total}} \frac{\partial \rho_a^{at}}{\partial r_{ab}} + \frac{\partial \phi(r_{ab})}{\partial r_{ab}} \right] \hat{\mathbf{r}}_{ab} \quad (3.7)$$

or rewritten in simplified notation using denoted derivatives $\lambda_a(\rho_a)$, $\rho_a^{at}(r_{ab})$, and $\phi_{ab}(r_{ab})$ with respect to the appropriate argument by a prime.

$$\mathbf{F}_a = - \sum_{b \neq a} \left[\lambda'_a \rho_b^{at'} + \lambda'_b \rho_a^{at'} + \phi'_{ab} \right] \hat{\mathbf{r}}_{ab} \quad (3.8)$$

In the case of monoatomic simulations, Eq. (3.8) can be further simplified and rewritten as

$$\mathbf{F}_a = - \sum_{b \neq a} \left[(\lambda'_a + \lambda'_b) \rho_b^{at'} + \phi'_{ab} \right] \hat{\mathbf{r}}_{ab} \quad (3.9)$$

The microscopic stress tensor rewritten for use with EAM potential is:

$$P_{ij} = \frac{1}{V} \left(\sum_{a=1}^N \frac{P_{ai} P_{bj}}{m_a} - \sum_{\substack{a,b=1 \\ a < b}} \left(\lambda'_a \rho_b^{at'} + \lambda'_b \rho_a^{at'} + \phi_{ab}' \right) \frac{x_{abi} x_{abj}}{r_{ab}} \right) \quad (3.10)$$

3.3 Developing EAM potentials

Mei's *et al.* [22] analytic EAM potentials have been chosen as the main form of EAM potentials for this study as they work relatively and consistently well for a variety of FCC metals including copper, silver, gold, nickel, palladium and platinum. In addition, Johnson's [30] analytic nearest-neighbor EAM model for Au and Voter and Chen's [32],[33] EAM potential for Ni have been included for comparisons. Potentials based on work by Mei *et al.*, Johnson, and by Voter and Chen will be referred to as the MEI, JON, and VC potentials respectively.

EAM potentials can be either obtained numerically [25] or analytically [30],[22]. In either case, they can be made to work relatively well for a wide range of applications. Analytical EAM potentials were used in this study. However, they were converted to a numerical form in order to ease the implementation of the potential into the MD code. This conversion allowed for both analytically and numerically created EAM potentials to be used with the created MD code.

3.3.1 Mei's *et al.* EAM potential model

Mei *et al.* [22] have developed a relatively simplified version of the potential in order to put in an analytic form. The full derivation is shown in [22] while a quick summary and clarification is given here. This potential has been derived using a notion of shells of atoms surrounding an atom of interest. For atoms in perfect lattice positions, there are m -neighbour shells with a specific number of atoms in each shell. There are 12 nearest neighbours in an FCC crystal, in other words, there are 12 atoms in the first shell. The number of atoms in each shell for the first 4 shells is given below.

Shell, m	Number of atoms
1	12
2	6
3	24
4	12

Table 3.1: *Number of atoms in each shell for a perfect FCC crystal.*

The more shells that are used, the more accurate the results will be at the expense of computational time. It has been shown in [22] that the first 3 shells give a relatively good approximation and hence only 3 shells have been used in this study.

The pair-potential function, $\phi(r_{ab})$, has been given the same form as Eq. (3.3) and is given as follows,

$$\phi(r_{ab}) = -\phi_e \left[1 + \delta \left(\frac{r_{ab}}{r_{1e}} - 1 \right) \right] \exp \left[-\gamma \left(\frac{r_{ab}}{r_{1e}} - 1 \right) \right] \quad (3.11)$$

where ϕ_e , δ , and γ are fitting parameters and r_{1e} is the nearest neighbor distance at equilibrium. All fitting parameters are given in Table C.1 in Appendix C while r_{1e} is obtained based on the FCC crystal structure as,

$$r_{1e} = \frac{a_0}{\sqrt{2}} \quad (3.12)$$

The electron density function, $\rho(r)$, has been given the following form,

$$\rho_a(r_{ab}) = f_e \sum_{l=0}^k c_l \left(\frac{r_{1e}}{r_{ab}} \right)^l \quad (3.13)$$

where f_e is the electron density for the pair of atoms a and b at equilibrium, and c_l are fitting parameters given in Table C.1.

The embedding function, $\lambda(\rho_{a,total})$, has been derived to have the following form,

$$\begin{aligned} \lambda(\rho_{a,total}) = & -E_{coh} \left[1 - \frac{\alpha}{\beta} \ln \left(\frac{\rho_{a,total}}{\rho_{total,e}} \right) \right] \left(\frac{\rho_{a,total}}{\rho_{total,e}} \right)^{\alpha/\beta} \\ & + \frac{1}{2} \phi_e \sum_m s_m \exp[-(p_m - 1)\gamma] \left[1 + (p_m - 1)\delta - p_m \frac{\delta}{\beta} \ln \left(\frac{\rho_{a,total}}{\rho_{total,e}} \right) \right] \left(\frac{\rho_{a,total}}{\rho_{total,e}} \right)^{p_m(\gamma/\beta)} \end{aligned} \quad (3.14)$$

where β is a fitting parameter given in Table C.1, $\rho_{total,e}$ is the total electron density for atom a at equilibrium, s_m is the number of atoms in each shell, and p_m for an FCC crystal is given as,

$$p_m = \sqrt{m} \quad (3.15)$$

where m is the shell number, and α is given by,

$$\alpha = \sqrt{\frac{9B_e\Omega_e}{E_{coh}}} \quad (3.16)$$

where B_e and Ω_e is the bulk modulus and atomic volume respectively at equilibrium.

Note that c_l , β , δ , γ , and ϕ_e are the only fitting parameters for each type of FCC metal and are given in Table C.1, while $\rho_{total,e}$ cancels from Eq. (3.14) with the following relationship based on the number of nearest-neighbors,

$$f_e = \frac{\rho_{total,e}}{12} \quad (3.17)$$

Since Eq. (3.13) is summed up into Eq. (3.14), the choice of f_e is arbitrary as it cancels out in Eq. (3.14) after the summation. Therefore f_e was set to 1 in this study. All other material properties required in the development of the potentials and not given in [22] are listed in Table C.2 in Appendix C.

A cut-off distance, r_c , was chosen as $r_c = 1.95r_{1e}$ in [22] where the pair-potential between a pair of atoms was small enough to be considered negligible. While Mei *et al.* has used a smoothing technique to have the pair-potential and electron density functions equal to zero at r_c ,

no such smoothing was used in this study. This is due to the fact that the smoothing of the functions distorted the potential energy at distances away from the equilibrium. The three functions were developed and plotted for Cu as an example and can be found in Appendix D.

The MEI potential was checked for a perfect FCC crystal by compressing and expanding the crystal in all three directions and calculating the potential energy of each atom as a function of the lattice crystal size. The obtained potential energy was then compared to Eq. (3.3) and plotted in Figure 3.1. Copper was used as an example as it is one of the most extensively tested materials.

Figure 3.1: Potential energy vs. lattice constant size plot for Cu using Mei's *et al.* EAM potential. Rose's *et al.* potential energy plot is included for comparison.

Note that in the above figure, potential energy based on the MEI potential is almost the same as that of Rose's *et al.*[31] energy, suggesting a very good fit. Also note that the minimum potential energy occurs at 3.617 Å as expected from Eq. (3.3).

3.3.2 Johnson's and Voter's and Chen's EAM potentials

The complete derivation of JON and VC potentials, is available in [30] and [32],[33] respectively and hence will be avoided here. The JON potential was only used for the study of Au while the VC potential was used for Ni. The reason for this is that only the respective metals worked comparably well to the MEI potentials while other metals did not perform satisfactorily.

3.4 Form of EAM potentials

All potentials created in this study were converted to a tabulated form for the use in the developed MD code. As mentioned before, this conversion allowed for the MD code to be developed so that it could use both analytically and numerically created potentials.

The tabulated form consists of three lists. The first is a list of pair-potentials as a function of distance at evenly spaced increments between defined starting and ending points. The second is a list of electron densities again as a function of distance at evenly spaced increments between defined starting and ending points. The last is a list of the embedding functions as a function of the total electron density at evenly spaced increments between defined starting and ending points.

For the MEI potentials only, the starting point for the first two lists was chosen as approximately 1.5 Å, a minimum distance between a pair of atoms. This distance was chosen arbitrarily and then tested with a variety of lattice constant sizes and metals to check that it is never passed. The ending point of the first two lists was based on a distance beyond which pair-potentials were considered negligible given by r_c .

The embedding function list limits were then obtained by first calculating the minimum and maximum of the total electron densities for various metals by compressing and expanding a perfect FCC crystal. Then, the minimum was further decreased and the maximum increased by a factor of approximately 4 and 1.5 respectively to ensure that the limits are never passed.

Chapter 4: Calculation of Elastic Constants and Bulk Modulus

4.1 Bulk modulus calculation at absolute zero

Mechanical properties of a material can be described using the elastic constants and bulk modulus. As mentioned in Chapter 1, the bulk modulus can be calculated using the change in potential energy as a function of the lattice constant size at absolute zero. The bulk modulus is given again as

$$B = -V \frac{dP}{dV} \quad (2.37)$$

where P is pressure and V is volume. Eq. (2.37) can be rewritten in terms of the change of in the potential energy in the system as a function of the lattice constant size as shown below.

$$B = \frac{a_0^2}{9\Omega} \frac{d^2U}{da^2} \quad (4.1)$$

where a is lattice constant size, a_0 is the lattice constant size at equilibrium, U is the potential energy of the system, and Ω is the volume per atom. A complete derivation of Eq. (4.1) can be found in Appendix B.

To calculate the bulk modulus using the above relation and molecular dynamics, a second order derivative of the energy as a function of the lattice constant size needs to be calculated. To do this, a set of potential energies can be calculated at different lattice constant sizes by shrinking and expanding the simulated crystal equally in all three directions. Three points are sufficient for this task where a centre point is chosen at equilibrium. The crystal can then be contracted in all three directions to obtain a second point and then expanded by the same amount from equilibrium to get a third point. Note that the amount of contraction or expansion must be small

enough to ensure elastic deformation. A 1% contraction or expansion in the dimensions of the crystal has shown to work well in this study.

However, this method is only valid at absolute zero as only potential energy is present in the system. In order to use Eq. (4.1) at a finite temperature, the potential energy must be substituted with the internal energy of the system which consists of potential and free energy. The limitation of this method lies in the problem of the calculation of the free energy of the system using MD which is not trivial. For this reason, this method is abandoned at finite temperatures other than absolute zero. This method, however, was used as a comparison to check the EAM potentials and MD code at absolute zero temperature.

4.2 Calculation of elastic constants at absolute zero

The elastic constants for a crystal at absolute zero can be found using the perfect lattice positions of the atoms. Since there is no kinetic energy present in the crystal, the elastic constants are purely a function of potential energy and are given by Ray and Rahman [23] as follows in tensor notation,

$$C_{ijkl} = \frac{1}{V} \sum_{a,b=1} \left(\frac{\partial^2 u}{\partial r_{ab}^2} - \frac{1}{r_{ab}} \frac{\partial u}{\partial r_{ab}} \right) \frac{x_{abi} x_{abj} x_{abk} x_{abl}}{r_{ab}^2} \quad (4.2)$$

where V is the volume of the system, x_{abi} is the i th component of the relative position vector of particle a and b , and r_{ab} is the distance between particle a and b .

The above equation was reformatted for use with EAM potentials by Daw and Baskes [2] and it could be written as,

$$C_{ijkl} = B1_{ijkl} + B2_{ijkl} + B3_{ijkl} \quad (4.3)$$

where $B1$ thru $B3$ are the Born terms given by

$$B1_{ijkm} = \frac{1}{V} \sum_{\substack{a,b=1 \\ a < b}} \left[\phi_{ab}'' - \frac{\phi_{ab}'}{r_{ab}} \right] \frac{x_{abi} x_{abj} x_{abk} x_{abm}}{r_{ab}^2} \quad (4.4)$$

$$B2_{ijkm} = \frac{1}{V} \sum_{\substack{a,b=1 \\ a \neq b}} \lambda_a' \left[\rho_b^{ai''} - \frac{\rho_b^{ai'}}{r_{ab}} \right] \frac{x_{abi} x_{abj} x_{abk} x_{abm}}{r_{ab}^2} \quad (4.5)$$

$$B3_{ijkm} = \frac{1}{V} \sum_{a=1} \lambda_a'' g_{aij} g_{akm} \quad (4.6)$$

where g_{aij} is given by

$$g_{aij} = \sum_{\substack{b=1 \\ b \neq a}} \frac{\rho_b^{ai'} x_{abi} x_{abj}}{r_{ab}} \quad (4.7)$$

where ϕ , ρ , and λ are the pair potential, electron density and embedding function of the EAM potential respectively.

As noted above, Eq. (4.3) thru Eq. (4.7) work at absolute zero only as they are only a function of the potential energy in the system.

4.3 Calculation of elastic constants at finite temperatures

In order to calculate the elastic constants at a finite temperature above absolute zero, temperature and momentum must be introduced into the equation. Ray and Rahman [23] derived a set of fluctuation formulas shown to be efficient by Ray *et al.* [24]. They are as follows,

$$C_{ijkm} = -\frac{V}{k_B T} \left(\langle P_{ij} P_{km} \rangle_{av} - \langle P_{ij} \rangle_{av} \langle P_{km} \rangle_{av} \right) + \frac{2Nk_B T}{V_0} (\delta_{ik} \delta_{jm} + \delta_{im} \delta_{jk}) + \langle B1_{ijkm} \rangle_{av} + \langle B2_{ijkm} \rangle_{av} + \langle B3_{ijkm} \rangle_{av} \quad (4.8)$$

where k_B is Boltzmann's constant, T is temperature, P_{ij} is the microscopic stress tensor given by Eq. (2.35), N is the number of atoms in the system, $\langle \rangle_{av}$ brackets denote a time average, and δ_{ij} is given by

$$\delta_{ij} = \begin{cases} 0, i \neq j \\ 1, i = j \end{cases} \quad (4.9)$$

The first term in Eq. (4.8) is called the fluctuation term, the second term is the temperature correction term, and the last three terms are given by Eq. (4.4) thru Eq. (4.7).

The Bulk modulus can then be found for isotropic materials using the calculated elastic constants [35] as given below.

$$B = \frac{C_{1111} + 2C_{1122}}{3} \quad (4.10)$$

4.4 Voight notation

The elastic constants in tensor notation can be simplified using the Voight notation. Suppose the following tensor is used, C_{ijkl} . The possible tensor pairs and their form in Voight notation are,

$$\begin{aligned} C_{1111} &\equiv C_{11} \\ C_{1122} &\equiv C_{12} \\ C_{1133} &\equiv C_{13} \\ C_{3223} = C_{2323} &\equiv C_{44} \\ C_{3113} = C_{1313} &\equiv C_{55} \\ C_{1221} = C_{2121} &\equiv C_{66} \end{aligned}$$

For an isotropic FCC cubic crystal there are only 3 different elastic constants [34], they are C_{11} , C_{12} , and C_{44} . Rewriting C_{ijkl} in matrix form gives

$$C_{ijkl} = \begin{bmatrix} C_{11} & C_{12} & C_{12} & 0 & 0 & 0 \\ C_{12} & C_{11} & C_{12} & 0 & 0 & 0 \\ C_{12} & C_{12} & C_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & C_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & C_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & C_{44} \end{bmatrix} \quad (4.11)$$

Chapter 5: Molecular Dynamics Simulation Code

5.1 Introduction

Due to the lack of technical support for available MD codes used for academic purposes and due to the lack of information on commercial codes tailored for this type of application, a fully functional 3D MD code called MDDROB was developed. It consisted of two programs written in ASCII C and one program written in MATLAB V7 R14. There was an additional program written in MATLAB used for MEI EAM potential generation. The source codes are listed in Appendix G.

The main MD code, MDROB, and a program used to calculate the Born terms called MDROB_BORN were written in ASCII C and compiled using Microsoft Visual C++ 2005 Express Edition. It was also successfully compiled and tested using Bloodshed Dev C++ V4.9.9.2.

The MD code can be run using a text input file or using a MATLAB control file, mdrob.m, which creates a text input file automatically. MDROB has the following features and limitations:

- Dimensions: 3D
- Crystal structure: FCC
- Number of different types of atoms: One
- Potential type: EAM
- Boundary conditions: Periodic or free applied independently in the 3 coordinate directions
- Strain: Independent application in the 3 coordinate directions
- Maximum number of atoms: 1372
- Temperature clamp: Yes, with adjustable delay
- Output files: Initial, instantaneous and final atomic positions, potential, kinetic and total energy, microscopic stress tensor, and simulation box size.
- Zero-stress simulation: Yes, with adjustable rescaling delay.

Metals for which EAM potentials have been shown to work include Cu, Ag, Au, Ni, Pd, and Pt. Although the MDROB was designed for an FCC crystal structure, it can be easily modified for use with BCC structures.

5.2 Numerical approaches

Numerical interpolation was required in order to use the created EAM input files. Linear interpolation was used to obtain values for $\phi_{ab}(r_{ab})$, $\rho_a(r_{ab})$, and $\lambda_a(\rho_{a,total})$ and their first-derivatives in Eq. (3.9). Cubic-spline interpolation was used to find the second-derivatives in Eq. (4.8). The cubic-spline interpolation was accomplished using a modified Gauss-Seidel tri-diagonal matrix solver. The Gauss-Seidel numerical solver can be used to solve a linear system of equations as shown below.

$$Ax = b \quad (5.1)$$

The numerical solver has the following form.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, 3, \dots \quad (5.2)$$

The above equation was then modified and simplified since a tri-diagonal system of equations was being solved and the following was obtained.

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(b_i - a_{i,i-1} x_{i-1}^{(k)} - a_{i,i+1} x_{i+1}^{(k-1)} \right) \quad (5.3)$$

The modified numerical solver was shown to work very well and sufficiently fast with an error of less than 0.001 %. Differentiation throughout the code was accomplished using forward and backward difference schemes.

5.3 Validation of code

The code was first tested for stability by choosing an EAM potential for Cu and ensuring that the solution did equilibrate after a number of iterations. A time-step size of 10 fs was chosen at first as the code seemed stable for various potentials. However, after further testing with MDROB_BORN, it was found that the time-step size needed to be reduced to decrease the fluctuations in the calculated Born terms. A new time-step size of 2 fs was tested and produced much smoother solutions. This time-step size was then used in all simulations in this study.

As an illustrative example of the stability of the MD code, a simulation was run using the MEI potential for Cu at 300 K with 108 atoms. The energy plots as functions of iterations are shown in Figure 5.1.

Figure 5.1: Total, potential, and kinetic energy as a function of iterations for Cu at 300K.

Note that in Figure 5.1, the potential and kinetic energies are fluctuating with the iterations. The fluctuations are similar in magnitude but opposite in direction creating a smooth total energy curve. All 3 energy plots do equilibrate and fluctuate at their own respective values suggesting that an equilibrium has been reached. However, even though the energies have equilibrated in under 5000 iterations in Figure 5.1, a much larger number of iterations are required to obtain meaningful data from the simulation. Furthermore, increasing the number of atoms also improves the smoothness of the resulting data.

The code was further validated by comparing the total and potential energy levels to another MD code called XMD developed by Rifkin et al. [36]. The same potential was used in both MD codes and the same total and potential energy was obtained in both cases.

Furthermore, a zero-stress, variable volume simulation was run at different temperatures to ensure that the volume of the system did increase with increasing temperatures. A number of simulations were also run close to 0 K in order to check that the lattice constant size (calculated from the system volume) did equilibrate to a value close to and just above the equilibrium lattice constant size at 0 K for each EAM potential.

5.4 Simulation

In order to obtain the elastic constants as a function of temperature, a number of simulations had to be performed at various temperatures. For the case of Cu, six zero-stress simulations were run at temperatures above absolute zero in order to calculate the lattice constant size at each temperature. Six more constant volume simulations were then performed using the newly calculated lattice constants. From these constant volume simulations, internal stresses and Born terms were calculated and stored in order to calculate the elastic constants. An additional calculation of the elastic constants was performed at absolute zero. In this case, there was no actual MD simulation performed as there are no movements of the atoms at absolute zero. Therefore, only the Born terms needed to be solved as the fluctuation and temperature correction terms were both equal to zero. The same method was followed for the other metals in this study. The complete description of the simulation procedures can be found in Appendix F.

Chapter 6: Results and Discussion

6.1 Simulation Parameters

In this study, 6 different mono-atomic FCC crystals were examined which include Cu, Ag, Au, Ni, Pd, and Pt. As mentioned in Chapter 3, the MEI potentials have been found to work relatively well in the calculation of elastic constants as a function of temperature for all six metals. In addition, the JON and VC potentials were used for Au and Ni respectively as shown in Table 6.1

Metal	EAM potential
Cu	MEI
Ag	MEI
Au	MEI and JON
Ni	MEI and VC
Pd	MEI
Pt	MEI

Table 6.1: Potentials used in this study.

Note that all of the MEI potentials have been developed and prepared directly from [22] for this study. The JON and VC potentials have been previously prepared by Rifkin *et al.* [36] and Mehl *et al.* [37] respectively.

For each type of metal and potential, a zero-stress simulation was run at different temperatures in order to obtain the lattice constant size at the specified temperatures. Using these calculated lattice constant sizes, the elastic constants were then calculated. The following parameters were used in the zero-stress simulations to calculate the lattice constant size at each temperature.

Number of atoms: 108

Number of iterations: 30000

Using 30000 iterations has been shown to be sufficient in calculating the lattice constant size where the first 15000 iterations were used for equilibration, and the last 15000 iterations were used for production. The box size (dimensions of the simulated crystal) was outputted every 5 iterations for the last 15000 iterations and averaged to obtain the lattice constant size. A delay of 30 iterations was used to return to zero-stress conditions as discussed in Section 2.10.

For the calculation of the elastic constants, the following parameters were used:

Number of atoms: 256

Total number of iterations: 60000

In this case, the first 30000 iterations were used for equilibration and the last 30000 iterations were used for production. The energies were outputted every 20 iterations in order to save on computational time as they were not used directly in any calculation using the fluctuation formulas. The microscopic stress tensor was outputted every single iteration in order to get the largest number of sampling points. The Born terms, which require much more computational time, were calculated every 5 iterations. It was, however, shown that increasing the number of Born term calculations to every single iteration produced insignificant changes in the overall data. A delay of 70 iterations was used to ensure that the temperature clamp was not *too abrupt* in the velocity rescaling as discussed in Section 2.8.

All simulations were run with periodic BCs in all three directions and a time-step size of 2 fs. Each simulation with 256 atoms and 60,000 iterations performed at constant volume took 3.0 hours on a computer with a single Intel Pentium 4 3.0 GHz processor. For comparison, a simulation performed with 500 atoms, 60,000 iterations and a Born term calculation every 30 iterations took 13.5 hours on the same processor.

The results for each metal are discussed individually in Section 6.2 except for Ag where the results have a very similar performance to that of Cu.

6.2 Elastic Constants and Bulk Modulus

6.2.1 Cu - Copper

The results for Cu are shown in Figure 6.1. They include plots of elastic constants and bulk modulus as functions of temperature. Both the MD results of this study and experimental data that were used for comparison are included in the elastic constant and bulk modulus plots.

Figure 6.1: Elastic constants (a) and bulk modulus (b) for Cu as a function of temperature. MD results obtained using the MEI potential are compared to experimental work by Overton et al.[5] and Chang et al.[9].

As can be seen from Figure 6.1, the calculation of the elastic constants was done successfully as the MD results are relatively close to the experimental data. A tabulated form of the MD results using the MEI potential and experimental data is given in Table 6.2 for Cu.

Temp K	C_{11}			C_{12}			C_{44}			Bulk modulus		
	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error
	GPa		%	GPa		%	GPa		%	GPa		%
0	171.05	176.20	-2.92	126.37	124.94	1.14	76.27	81.77	-6.73	141.26	142.03	-0.54
10	170.91	176.20	-3.00	126.30	124.94	1.09	76.12	81.76	-6.90	141.17	142.03	-0.61
100	167.95	174.93	-3.99	125.06	124.25	0.65	74.08	80.59	-8.08	139.35	141.15	-1.27
300	161.55	169.2	-4.52	121.82	122.0	-0.11	69.54	75.6	-8.02	135.07	137.7	-1.91
500	153.56	161.5	-4.91	118.71	118.5	0.18	64.33	70.4	-8.63	130.33	132.8	-1.86
700	146.58	153.5	-4.51	115.30	114.5	0.70	59.79	65.0	-8.01	125.73	127.5	-1.39
800	142.68	149.5	-4.56	113.32	112.5	0.73	57.24	62.3	-8.13	123.11	124.8	-1.36

Table 6.2: Results of the MD calculations of the elastic constants and bulk modulus for Cu using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Overton et al.[5] for temperatures from 0 K to 100 K, by Chang et al.[9] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.

From Table 6.2, it can be seen that the error is less than -5 % for C_{11} , 1.2 % for C_{12} , -8.7 % for C_{44} , and -2 % for B . The errors are relatively consistent or very small giving a steady variation of elastic constants with temperature. This suggests that elastic constants at any other temperature between 0 K and 800 K can be easily obtained using interpolation.

Note that the elastic constants have a relatively linear variation with temperature. This is not the case for experimental data where the elastic constants below approximately 100 K began to level off suggesting that a *plateau* is reached at 0 K. The EAM potentials are not able to account for this characteristic. This is why there is a smaller error at 0 K for which the EAM potential was fitted. The error then rises until approximately 100 K where it seems to equilibrate for the case of C_{11} , C_{44} , and B . Although the MD results do not follow the same variation with temperature below 100 K, the slopes of the plots are relatively the same as the experimental data for all other temperatures providing very good approximations to experimental values.

In order to investigate the MD results further, Figure 6.2 shows the components of the fluctuation formula for both the fluctuation and Born terms as a function of temperature.

Figure 6.2: Fluctuation (a) and Born term (b) components for Cu as a function of temperature using the MEI potential.

In Figure 6.2, it can be seen that relatively smooth lines were produced in both figures. This suggests that a sufficient amount of atoms and iterations were used in the calculation of the elastic constants. It was shown that by using 500 atoms a smoother solution can be obtained but at a much higher computational cost. However, the difference in the calculated elastic coefficients between the 256 atom and the 500 atom solution was found to be negligible. A tabulated form of the above plots can be found in Appendix E for all tested elements. In addition, the energy correction term was not plotted as it is not a function of the MD simulation output, but rather only the number of atoms used, volume of the system, element type and temperature. Hence, a perfectly linear variation of the temperature correction term is always obtained.

Furthermore, the one component that made the MEI potential work well in this study is the relatively linear and consistent variation of the Born terms as a function of temperature as shown in Figure 6.2 (b). This suggests that if only potential energy is considered, the elastic constants decrease linearly with increasing temperature as expected for most of the temperature range. It was observed that not all EAM potentials behaved in this manner. These include the JON potential for Cu prepared by Rifkin *et al.* [36] where the Born terms started increasing with increasing temperature above 300 K, and an EAM potential for Cu based on work by Mishin *et al.* [38] and prepared by Mehl *et al.* [37] where the Born terms were increasing below 300 K.

These deficiencies produced very inconsistent elastic constants for the tested range of 0 K to 800 K.

The reason for the elastic constants not being extremely close to the experimental values is because the MEI EAM potentials were fitted to experimental values of elastic constants at 300 K using a crystal with all atoms at the perfect lattice positions. This is not the case in reality where all atoms vibrate around their perfect lattice positions. This logic could only be used at absolute zero where there is no movement of the atoms, and the atoms are in fact at their perfect lattice positions. Furthermore, during the fitting process, no actual MD simulation was performed; therefore, there was no fluctuation or temperature correction term used in the calculation of the elastic constants. Fitting the potentials to the elastic constants while performing an actual MD simulation is not trivial as it is not easy to specify an ideal configuration of atomic positions that would represent an average state of a system. Furthermore, the problem would be magnified by the computational expense of the fitting. Even though this is the case here, the developed potentials do provide a relatively good approximation of the elastic constants.

Overall, the elastic constants were successfully calculated for Cu. In fact, the results are much better than a previous study done by Cagin *et al.* [26] using the Sutton-Chen [15] interatomic potential. As a comparison for Cu, the elastic constants are plotted in Figure 6.3.

Figure 6.3: Comparison of elastic constants obtained using MD for Cu using the MEI potential and the Sutton-Chen potential based on work by Cagin *et al.* [26]. The two MD results are compared to experimental work by Overton *et al.* [5] and Chang *et al.* [9].

In Figure 6.3, it can be seen that the results of this study are much closer to experimental data than that of Cagin *et al.* [26]. The same trend was found in all other elements studied here and in [26].

An additional check was performed to further validate the MD code and EAM potentials. It involved using Eq. (4.1) to calculate the bulk modulus at absolute zero. A crystal made of 256 atoms with periodic BCs was contracted and expanded by 1 % from a central equilibrium lattice constant size of 3.617 Angstroms. No actual MD simulation was performed as there is no movement of atoms at absolute zero. Only the potential energy was calculated at equilibrium, at a 1% crystal contraction, and at a 1 % crystal expansion. The energies are given in Table 6.3.

a (Å)	U (eV)
3.581	-3.4775
3.617	-3.4823
3.653	-3.4778

Table 6.3: *Potential energy of a Cu crystal as a function of lattice constant size at absolute zero.*

The three points were fitted with a second degree polynomial to obtain the following:

$$U(a) = 3.6022a^2 - 26.0626a + 43.6591 \quad (6.1)$$

where the potential energy, U , is in eV and the lattice constants size, a , is in Angstroms.

Plugging the second derivative of Eq. (6.1) into Eq. (4.1) along with the atomic volume from Table C.2, the Bulk modulus was calculated to be 142.12 GPa. This bulk modulus is in excellent agreement with the experimental value [5] of 142.03 GPa giving a 0.061 % error.

6.2.2 Ag – Silver

The results for Ag have shown similar performance to the case of Cu and hence the same discussion applies in both cases. Figure 6.4 shows the elastic constants and bulk modulus of Ag as a function of temperature along with experimental values for comparison.

Figure 6.4: Elastic constants (a) and bulk modulus (b) for Ag as a function of temperature. MD results obtained using the MEI potential are compared to experimental work by Neighbours et al.[6] and Chang et al.[9].

The results are shown in Table 6.4 in tabulated form.

Temp K	C ₁₁			C ₁₂			C ₄₄			Bulk modulus		
	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error
	GPa		%	GPa		%	GPa		%	GPa		%
0	126.62	131.49	-3.70	98.72	97.33	1.43	46.87	51.09	-8.26	108.02	108.70	-0.63
10	126.48	131.44	-3.77	98.62	97.30	1.35	46.78	51.08	-8.42	107.90	108.70	-0.73
50	125.46	131.12	-4.31	98.09	97.26	0.86	46.18	50.72	-8.95	107.22	108.50	-1.18
100	124.20	129.80	-4.31	97.43	96.66	0.80	45.46	49.82	-8.75	106.35	107.70	-1.25
300	118.72	124.00	-4.25	94.41	93.84	0.62	42.41	46.31	-8.43	102.52	103.90	-1.33
500	112.50	117.00	-3.85	91.20	90.00	1.34	38.98	42.60	-8.50	98.30	99.00	-0.71
700	106.29	110.00	-3.37	87.96	86.00	2.28	35.37	38.80	-8.85	94.07	94.00	0.08
800	103.15	106.50	-3.15	86.27	84.50	2.10	33.69	36.90	-8.69	91.90	91.80	0.11

Table 6.4: Results of the MD calculations of the elastic constants and bulk modulus for Ag using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Neighbours et al.[6] for temperatures from 0 K to 100 K, by Chang et al.[9] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.

The results for Ag are in good accordance with experimental work. Errors are less than -4.4 % for C₁₁, 2.3 % for C₁₂, -9 % for C₄₄, and -1.4 % for the bulk modulus.

6.2.3 Au – Gold

Figure 6.5 shows the elastic constants and bulk modulus of Au using two different types of EAM potentials as a function of temperature along with experimental values for comparison.

Figure 6.5: Elastic constants (a) and bulk modulus (b) for Au as a function of temperature. MD results obtained using the MEI and the JON potential are compared to experimental work by Neighbours *et al.* [6] and Chang *et al.* [9].

The results are shown in Table 6.5 and Table 6.6 in tabulated form for both EAM potentials.

Temp K	C ₁₁			C ₁₂			C ₄₄			Bulk modulus		
	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error
	GPa		%	GPa		%	GPa		%	GPa		%
0	196.49	201.63	-2.55	169.82	169.67	0.09	42.58	45.44	-6.29	178.71	180.3	-0.88
10	196.28	201.61	-2.65	169.70	169.67	0.02	42.52	45.42	-6.40	178.56	180.3	-0.97
50	194.80	201.07	-3.12	168.67	169.37	-0.41	41.93	45.10	-7.03	177.38	179.9	-1.40
100	192.79	199.40	-3.32	167.04	168.22	-0.70	41.22	44.46	-7.29	175.62	178.6	-1.67
300	183.53	192.4	-4.62	160.46	163.1	-1.60	38.10	42.2	-9.66	168.15	172.9	-2.72
500	173.11	185.0	-6.43	152.85	158.0	-3.26	34.90	39.7	-12.09	159.60	167.0	-4.43
700	162.67	178.5	-8.87	144.89	153.0	-5.30	31.72	36.9	-14.05	150.82	161.5	-6.61
800	156.84	175.5	-10.63	139.87	150.5	-7.06	30.35	35.5	-14.49	145.53	158.8	-8.38

Table 6.5: Results of the MD calculations of the elastic constants and bulk modulus for Au using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Neighbours et al.[6] for temperatures from 0 K to 100 K, by Chang et al.[9] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.

Temp K	C ₁₁			C ₁₂			C ₄₄			Bulk modulus		
	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error
	GPa		%	GPa		%	GPa		%	GPa		%
10	183.64	201.61	-8.91	155.57	169.67	-8.31	41.79	45.42	-7.99	164.93	180.3	-8.52
100	178.32	199.40	-10.57	152.43	168.22	-9.38	39.26	44.46	-11.70	161.06	178.6	-9.82
300	168.99	192.4	-12.18	146.39	163.1	-10.23	35.85	42.2	-14.99	153.92	172.9	-10.95
500	160.72	185.0	-13.13	139.87	158.0	-11.47	33.42	39.7	-15.82	146.82	167.0	-12.08
800	148.16	175.5	-15.58	131.21	150.5	-12.82	29.05	35.5	-18.16	136.86	158.8	-13.84

Table 6.6: Results of the MD calculations of the elastic constants and bulk modulus for Au using the JON potential. Experimental values of the elastic constants and bulk modulus were given by Neighbours et al.[6] for temperatures from 10 K to 100 K, by Chang et al.[9] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.

The errors for Au using MEI EAM potential are larger than in case of the previous two elements. However, they are still reasonable with errors less than -11 % for C₁₁, 7.1 % for C₁₂, -15 % for C₄₄, and -8.4 % for the bulk modulus. The errors are shown to increase with increasing temperatures but are relatively low for up to 300 K.

For the JON potential, the errors are much bigger than in the case of the MEI potential. The errors also seem to increase with increasing temperature giving better approximations at lower temperatures.

Both potentials do show a relatively linear variation of the elastic constants with temperature. However, it must be noted that the MEI potential does approximate the elastic constants much better and it would be favoured over the JON potential in studies where elastic constant calculations are of issue.

Furthermore, the bulk modulus is much better approximated using the MEI potential. The reason for the poor performance of the JON potential may be the fitting parameters used in developing the potential. However, due to the lack of information on these parameters used by Rifkin *et al.* [36], the fitting procedure could not be verified.

6.2.4 Ni – Nickel

Figure 6.6 shows the elastic constants and bulk modulus of Ni using two different types of EAM potentials as a function of temperature along with experimental values for comparison.

Figure 6.6: Elastic constants (a) and bulk modulus (b) for Ni as a function of temperature. MD results obtained using the MEI and VC potentials are compared to experimental work by Alers et al. [11].

The results are shown in Table 6.7 and Table 6.8 in tabulated form for both EAM potentials.

Temp K	C_{11}			C_{12}			C_{44}			Bulk modulus		
	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error
	GPa		%	GPa		%	GPa		%	GPa		%
0	241.73	261.2	-7.45	160.14	150.8	6.19	130.09	131.7	-1.22	187.34	187.6	-0.14
20	241.24	261.2	-7.64	159.98	150.8	6.09	129.57	131.7	-1.62	187.07	187.6	-0.28
60	239.56	260.5	-8.04	159.58	150.7	5.89	128.35	131.4	-2.32	186.24	187.3	-0.57
100	238.17	259.5	-8.22	158.90	150.7	5.44	126.92	130.4	-2.67	185.32	187.0	-0.90
200	234.16	255.3	-8.28	157.44	150.1	4.89	123.47	127.0	-2.78	183.02	185.2	-1.18
300	229.18	250.8	-8.62	156.22	150.0	4.14	120.23	123.5	-2.65	180.54	183.6	-1.67
400	226.12	245.4	-7.86	154.12	149.6	3.02	116.60	119.8	-2.67	178.12	181.5	-1.86
500	221.11	239.6	-7.72	153.07	149.0	2.73	113.31	115.9	-2.24	175.75	179.2	-1.93
600	217.48	232.2	-6.34	151.37	147.8	2.42	110.33	111.8	-1.31	173.41	175.9	-1.42
700	212.59	226.1	-5.98	149.60	147.3	1.56	106.27	107.9	-1.51	170.59	173.6	-1.73
760	210.32	223.2	-5.77	148.55	146.4	1.47	104.31	105.8	-1.41	169.14	172.0	-1.66

Table 6.7: Results of the MD calculations of the elastic constants and bulk modulus for Ni using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Alers et al.[11].

Temp K	C_{11}			C_{12}			C_{44}			Bulk modulus		
	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error
	GPa		%	GPa		%	GPa		%	GPa		%
20	242.58	261.2	-7.13	147.62	150.8	-2.11	124.63	131.7	-5.37	179.27	187.6	-4.44
60	241.34	260.5	-7.35	147.49	150.7	-2.13	123.34	131.4	-6.13	178.77	187.3	-4.55
100	240.20	259.5	-7.44	147.46	150.7	-2.15	121.92	130.4	-6.50	178.37	187.0	-4.61
200	236.54	255.3	-7.35	146.57	150.1	-2.35	118.36	127.0	-6.81	176.56	185.2	-4.67
300	232.41	250.8	-7.33	145.99	150.0	-2.68	114.55	123.5	-7.25	174.80	183.6	-4.80
400	227.88	245.4	-7.14	145.04	149.6	-3.05	110.57	119.8	-7.70	172.65	181.5	-4.87
500	223.98	239.6	-6.52	144.00	149.0	-3.35	107.25	115.9	-7.46	170.66	179.2	-4.76
600	220.19	232.2	-5.17	142.56	147.8	-3.54	103.11	111.8	-7.77	168.44	175.9	-4.24
700	214.69	226.1	-5.05	141.49	147.3	-3.95	99.26	107.9	-8.01	165.89	173.6	-4.44

Table 6.8: Results of the MD calculations of the elastic constants and bulk modulus for Ni using the VC potential. Experimental values of the elastic constants and bulk modulus were given by Alers et al.[11].

The C_{11} and C_{12} errors are shown to be slightly smaller using the VC potential than when using the MEI potential while errors for C_{44} are bigger. However, the bulk modulus calculations show that the MEI potential performs much better than the VC potential.

The reason behind the big discrepancy in the bulk modulus between the two EAM potentials is again due to the fact that the VC potential may have been fitted using material properties at approximately 300 K. However, the method of calculating the elastic constants could not be verified do to the lack of information on the subject in [32], [33] and [37].

6.2.5 Pd – Palladium

Figure 6.7 shows the elastic constants and bulk modulus of Pd as a function of temperature along with experimental values for comparison.

Figure 6.7: Elastic constants (a) and bulk modulus (b) for Pd as a function of temperature. MD results obtained using the MEI potential is compared to experimental work by Walker et al.[7] and Weinmann et al.[10].

The results are shown in Table 6.9 in tabulated form.

Temp K	C_{11}			C_{12}			C_{44}			Bulk modulus		
	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error
	GPa		%	GPa		%	GPa		%	GPa		%
0	224.52	232.3	-3.34	179.81	174.5	3.05	75.35	71.9	4.87	194.71	193.8	0.50
20	224.04	232.1	-3.47	179.46	174.5	2.84	75.05	71.8	4.52	194.32	193.7	0.32
100	221.50	229.8	-3.61	178.02	174.3	2.14	73.59	71.0	3.68	192.52	192.8	-0.15
300	214.43	222.7	-3.70	174.20	171.9	1.36	70.31	71.6	-1.74	187.61	188.8	-0.63
500	206.84	215.5	-4.00	170.07	168.0	1.25	66.95	72.1	-7.08	182.32	183.8	-0.80
800	197.70	205.8	-3.92	163.47	161.6	1.18	61.12	70.5	-13.30	174.88	176.3	-0.81

Table 6.9: Results of the MD calculations of the elastic constants and bulk modulus for Pd using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by Walker *et al.*[7] for temperatures from 0 K to 100 K, by Weinmann *et al.*[10] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.

The results are relatively good for Pd for both C_{11} and C_{12} as seen from the above figures and table. However, the C_{44} data show two different trends as a function of temperature between the MD and experimental results. As in all previous crystals, the C_{44} follows a relatively linear trend. This trend is shown through the MD results for Pd. However, the experimental data show an anomaly in the C_{44} where it actually starts to increase between 100 K and 500 K after which it starts to decrease again. This anomaly has been discussed by Rayne [39] and is believed to be due to the change with temperature of the contribution to the shear constants. EAM potentials are not capable at this point to account for this anomaly. Nevertheless, the MD results for C_{44} are still relatively close to the experimental data below 500 K. Furthermore, the error is less than 1.8 % for the 300 K case which serves as a basis for most studies. The calculated bulk modulus also shows to be in excellent accordance with experimental data with an error of less than 0.9 %.

A comparison of the elastic constants of this study was made to the elastic constants obtained using the third neighbour EAM potential developed by Wolf *et al.*[25] and are shown in Figure 6.8.

Figure 6.8: Results of the MD calculations of the elastic constants for Pd using the MEI potential and EAM based on work by Wolf *et al.*[25]. Experimental values of the elastic constants and bulk modulus were given by Walker *et al.* [7] for temperatures from 0 K to 100 K, by Weinmann *et al.* [10] for temperatures from 500 K to 800 K, and an average of the two works was used to obtain the experimental values for 300 K.

Note that the results based on work by Wolf *et al.*[25] are given for 0 K and 382.8 K cases only. Since the elastic constants obtained from MD simulations follow a relatively linear trend as functions of temperature, the two cases were joined by a straight line in the above figure.

The MD results for C_{11} and C_{12} are better using the EAM potential developed by Wolf *et al.* [22] as they are much closer to the experimental data. The C_{44} results for both EAM potentials show a linear variation with temperature, but the results based on work by Wolf *et al.* show a less than 2.1 % error for the 300 K case. In conclusion, the EAM based on work by Wolf *et al.* [25] does seem to work better than the MEI potential for Pd.

6.2.6 Pt – Platinum

The results for Pt are shown in the following table. Due to the lack of experimental data for comparison, results were only obtained for the 0 K and 300 K cases.

Temp K	C_{11}			C_{12}			C_{44}			Bulk modulus		
	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error	MD	Exp	Error
	GPa		%	GPa		%	GPa		%	GPa		%
0	325.38	358.0	-9.11	268.41	253.6	5.84	89.11	77.4	15.13	287.40	288.4	-0.35
300	313.26	346.7	-9.64	261.07	250.7	4.14	83.85	76.5	9.60	278.47	282.7	-1.50

Table 6.10: Results of the MD calculations of the elastic constants and bulk modulus for Pt using the MEI potential. Experimental values of the elastic constants and bulk modulus were given by MacFarlane et al. [8].

As mentioned above, due to the lack of experimental data on Pt, a limited analysis can be done. Overall, the bulk modulus values are in very good agreement with experimental values, while the elastic constants have errors between 5.8 % and 15.2 %. In [8], it can be seen that C_{44} does not have a linear variation as in the case of Pd and hence increased errors are expected for C_{44} .

6.3 Coefficients of Linear Thermal Expansion

The lattice constant size of Cu as a function of temperature is given in Figure 6.9.

Figure 6.9: *Lattice constant size for Cu as a function of temperature using the MEI potential.*

The lattice constant size has a relatively linear variation giving a constant thermal expansion throughout the entire temperature range. The coefficients of linear thermal expansion were calculated for all eight EAM potentials using the following equation.

$$\alpha = \frac{1}{\Delta T} \frac{\Delta L}{L_0} \quad (6.2)$$

where ΔT is the temperature rise, ΔL is the change in length, and L_0 is the initial length. In this case, the lattice constant size was used as the length. The MD results are tabulated below along with experimental values.

Element	α		Error
	MD	Exp	
	$K^{-1} \times 10^6$		%
Cu - MEI	18.3	16.5	10.9
Ag - MEI	18.6	18.9	-1.6
Au - MEI	17.0	14.2	19.7
Au - JON	17.9	14.2	26.1
Ni - MEI	15.3	13.3	15.0
Ni - VC	14.8	13.3	11.3
Pd - MEI	12.0	11.8	1.7
Pt - MEI	9.2	8.8	4.5

Table 6.11: *Coefficients of linear thermal expansion for all eight EAM potentials and their corresponding experimental values [40].*

The MD results for Ag, Pd, and Pt yield relatively good results when compared to experimental data. For the other metals, the error is in excess of 10 % with the maximum for the JON potential for Au at 26 %. The MEI potential for Au performs better than the JON potential, and the VC potential for Ni works better than the MEI potential.

Note also that the MD results yield almost linear variations of lattice constant as functions of temperature. This is not the case in reality as the thermal expansion of the metals behaves differently at lower temperatures. This is another aspect that EAM potentials are not able to account for. However, the MD results do provide a good approximation of the behavior of the metals.

Chapter 7: Conclusion

The purpose of this study was to calculate the elastic constants of monoatomic FCC crystals using molecular dynamics simulations. After an extensive review of past work in this area, it was concluded that there was a lack of information on the temperature dependence of the elastic constants using EAM potentials. Therefore, the temperature dependence of the elastic properties served as the basis for this study. In addition, the coefficients of linear thermal expansion were also calculated.

A complete 3D MD code was written for this study capable of simulating FCC crystals using EAM potentials. The main MD codes were written in ASCII C due to the speed advantage of low level languages. The initialization and post-processing was accomplished with programs written in MATLAB due to the ease of implementation and plotting capabilities of the software.

The MD simulations were performed on FCC crystals made of 108 atoms for zero-stress simulations, and 256 atoms for the constant volume simulations. Periodic BCs and a time step of 2 fs were used in all simulations producing stable solutions. The MEI potentials have been shown to work well for the six FCC crystals presented here and would be recommended in future studies. The results for Cu, Ag, and Pd were especially good with errors below 5 % for C_{11} and C_{12} and below 2 % for B . In addition, the plots of the MD and experimental elastic constants and bulk modulus had comparable slopes for all six metals. However, for the case of Pd, the EAM potential developed by Wolf *et al.*[25] did show increased accuracy and it would be recommended in future studies.

It was also found that not all EAM potentials perform as well when tested at temperatures other than what they were designed for. Such was the case for the EAM potentials for Cu prepared by Rifkin *et al.* [36] and by Mehl *et al.* [37].

Overall, the study was successful as the required elastic constants and bulk modulus were calculated within reasonable limits. The methods and results presented here should serve as a basis for future work in the study of materials as outlined in Section 7.1.

7.1 Future work

Multi-scale methods show promising potential in the study of materials, specifically nanocomposites. It would be recommended to further develop the MD code and methods presented here to work with the proposed MEM. The following is a list of suggestions for future studies in this area.

- Development of new EAM potentials, including potentials for more than one type of atom.
- Further development of the MD code:
 - Incorporate *neighbor lists* [27] into the MD code which keep track of all the nearest neighbors for each particle in the system, reducing the amount of time required for force calculations.
 - Merge the Born term calculation code with the main MD code in order to save memory requirements. Since MDROB was designed to save the atomic configuration every specified number of atoms, over 6000 files were created for every case. Due to this large number of files, extra central processing unit (CPU) time was required to manage them. When using a computer with a single CPU, approximately 50% of the CPU time was wasted on managing the files near the end of each case.
 - Utilize dynamic memory allocation in the MD code in order to increase the maximum number of atoms in the simulated model.
 - Utilize different types of atoms.
 - Add capability to read atomic configurations as input files.
 - Add capability to continue a previously started simulation.
 - Modify code for parallel-processing in order to reduce total computational time.
- Development of code to run the FEM portion of MEM using outputs obtained from MD simulations and vice versa.

References

- [1] Mai, Y and Yu, Z, Polymer Nanocomposites, Centre for Advanced Materials Technology (CAMT), University of Sydney, Australia, 2006
- [2] Daw, S. and Baskes, M. I., Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals, *Phys. Rev. B* **29**, p6443, 1984
- [3] Quimby, S. L. and Siegel, Sidney, Variation of the Elastic Constants of Crystalline Sodium with Temperature Between 80°K and 210°K, *Phys. Rev.* **54**, p293, 1938
- [4] Sutton. M. Paul, The Variation of the Elastic Constants of Crystalline Aluminum with Temperature between 63°K and 773°K, *Phys. Rev.* **91**, p816, 1953
- [5] Overton, W. C. Jr. and Gaffney, John, Temperature Variation of the Elastic Constants of Cubic Elements. I. Copper, *Phys. Rev.* **98**, p969, 1955
- [6] Neighbours, J. R. and Alers, G. A., Elastic Constants of Silver and Gold, *Phys. Rev.* **111**, p707, 1958
- [7] Walker, E., Ortelli, J., and Peter, M., Elastic constants of monocrystalline alloys of Pd-Rh and Pd-Ag between 4.2K and 300K, *Phys. Lett. A* **31**, p240, 1970
- [8] MacFarlane, R.E., Rayne, J.A, and Jones, C.K., Anomalous temperature dependence of shear modulus C_{44} for platinum, *Phys. Lett.* **18**, p91-92, 1965
- [9] Chang, Y. A., and Himmel, L., Temperature Dependence of the Elastic Constants of Cu, Ag, and Au above Room Temperature, *J. Appl. Phys.* **37**, p3567, 1966
- [10] Weinmann C., and Steinemann S., Lattice and Electronic Contributions to the Elastic Constants of Palladium, *Solid State Communications* **15**, p281, 1974
- [11] Alers, G.A., Neighbours, J.R., and Sato, H., Temperature dependent magnetic contributions to the high field elastic constants of nickel and an Fe-Ni alloy, *J. Phys. Chem. Solids* **13**, 40-55, 1960
- [12] Simmons, Gene and Wang H., Single Crystal Elastic Constants and Calculated Aggregate Properties: A Handbook, The M.I.T. Press, Cambridge, 1971
- [13] Göken, M., Kempf, M., Bordenet, M., and Vehoff, H., Nanomechanical Characterizations of Metals and Thin Films, *Surf. Interface. Anal.* **27**, p302, 1991
- [14] Lennard-Jones, J.E., Cohesion, *Proceedings of the Physical Society* **43**, p461, 1931
- [15] Sutton, A. P. and Chen, J., *Phil. Mag. Lett.* **61**, p139, 1990

- [16] Finnis, M. W. and. Sinclair, J. E, *Philos. Mag. A* **50**, p45, 1984
- [17] Ackland, G.J., Tichy, G.I, Vitek, V., and Finnis, M.W., *Phil. Mag, A* **56**, p15, 1987
(for improved version of Finnis-Sinclair potential)
- [18] Shenoy, V. B., Miller, R., Tadmor, E. B., Rodney, D., Phillips, R., Ortiz, M., An Adaptive Finite Element Approach to Atomic-Scale Mechanics – The Quasicontinuum Method, *J. Mech. Phys. Sol.* **47**, p611, 1999
- [19] Weinan E., Bjorn Engquist, and Zhongyi Huang, Heterogeneous multiscale method: A general methodology for multiscale modeling, *Phys. Rev. B* **67**, 092101, 2003
- [20] Behdinan, K., Xu, Y. G., and Fawaz, Z., Molecular Element Method For Multi-scale Modeling and Simulations of Nano/Micro-Systems, Submitted for printing
- [21] Born, Max, Dynamical Theory of Crystal Lattices, Oxford, Clarendon Press, 1962
- [22] Mei, J., Davenport, J. W. and Fernando, G. W., Analytic embedded-atom potentials for fcc metals: Application to liquid and solid copper, *Phys. Rev. B* **43**, p4653, 1991
- [23] Ray, J. R. and Rahman, A., Statistical Ensembles and Molecular Dynamics Studies of Anisotropic Solids, *J. Chem. Phys.* **80**, p4423, 1984
- [24] Ray, John R., Moody, C. Michael and Rahman, Aneesur, Molecular dynamics calculation of elastic constants for a crystalline system in equilibrium, *Phys. Rev. B* **32**, p733, 1985
- [25] Wolf, J. Ralph., Mansour, A. Khalid, and Lee W. Myung, Temperature dependence of elastic constants of embedded-atom models of palladium, *Phys. Rev. B* **46**, p8027, 1992
- [26] Cagin, T., Dereli, G., Uludogan, M., and Tomak, M., Thermal and mechanical properties of some fcc transition metals, *Phys. Rev. B* **59**, p3468, 1999
- [27] Haile, J.M., Molecular Dynamics Simulation, Elementary Methods, John Wiley & Sons, Inc., New York, 1997
- [28] Gear, C. W., Numerical Initial Value Problems in Ordinary Differential Equations, Prentice-Hall, Englewood Cliffs, NJ, Chapter 9, 1971
- [29] Watkins, S. F., Crystal Lattice Structures, Department of Chemistry, Louisiana State University, [online] Available at:
<http://www.chem.lsu.edu/htdocs/people/sfwatkins/MERLOT/lattice/04fcc.html>
- [30] Johnson, R. A., Analytical nearest-neighbor model for fcc metals, *Phys Rev B* **37**, p3924, 1987

- [31] Rose, H. James, Smith, R. John, Guinea, Francisco, and Ferrante, John, Universal features of the equation of state of metals, *Phys. Rev. B* **29**, p2963, 1983
- [32] Voter, A. F. and Chen, S. P., Accurate Interatomic Potentials for Ni, Al, and Ni₃Al, *Proc. Mater. Res. Soc. Symp.* **82**, (Pittsburgh, PA: Materials Research Society), p175, 1987
- [33] Voter, Arthur F., Intermetallic Compounds: Principles and Practice, Edited by J. H. Westbrook and R. L. Fleisher, John Wiley and Sons, Ltd., Vol. 1, p77, 1994
- [34] Hyer, Michael W., Stress Analysis of Fiber-Reinforced Composite Materials, McGraw Hill, p57, 1998
- [35] Mehl, Michael J., Pressure dependence of the elastic moduli in aluminum-rich Al-Li compounds, *Phys. Rev. B* **47**, p2493, 1993
- [36] Rifkin, J., Clapp, P., and Fernando, G., University of Connecticut, Institute of Materials Science, [online] <http://www.ims.uconn.edu/centers/simul/>, 1999
- [37] Mehl, Michael, Center for Computational Materials Science, Naval Research Laboratory, [online] <http://cst-www.nrl.navy.mil/ccm6/ap/>, 2002
- [38] Mishin, Y., Mehl, M. J., Papaconstantantopoulus, D. A., Voter, A. F., and Kress, J. D., Structural stability and lattice defects in copper: *Ab initio*, tight-binding, and embedded-atom calculations, *Phys. Rev. B* **63**, 224106, (2001)
- [39] Rayne, J. A., Elastic Constants of Palladium from 4.2 – 300°K, *Phys. Rev.* **118**, p1545, 1960
- [40] Dean, John Aurie (ed), Lange's handbook of Chemistry, 16th ed. McGraw-Hill, 2005
- [41] London, F., Properties and Application of Molecular Forces, *Zeit. Physik. Chem. B* **11**, p222, (1930)

APPENDIX A: Lennard-Jones and Shifted-Force Potential

A.1 Lennard-Jones Potential

The intermolecular potential model most commonly used in molecular dynamics simulations is one first introduced by J. E. Lennard-Jones in 1924. This model is based on the following *empirical* equation which describes the intermolecular energy between two atoms as a function of the distance between their centers.

$$u(r) = k\varepsilon \left[\left(\frac{\sigma}{r} \right)^n - \left(\frac{\sigma}{r} \right)^m \right] \quad (\text{A.1})$$

where

$$k = \frac{n}{n-m} \left(\frac{n}{m} \right)^{m/(n-m)} \quad (\text{A.2})$$

and where ε is the minimum energy, and σ is the distance to the zero in $u(r)$ which is effectively the diameter of one of the atoms.

The above function is basically the sum of a repulsive and attractive component.

$$U_R = k\varepsilon \left(\frac{\sigma}{r} \right)^n \quad \text{and} \quad U_A = -k\varepsilon \left(\frac{\sigma}{r} \right)^m \quad (\text{A.3})$$

The first component is due to the repulsive forces at close distances and the second component is due to the attractive forces at larger distances. These two functions were based on experimental results where it was shown that the repulsive and attractive forces were a function of the inverse of the power of r . It was later discussed in London's theory [41] that the leading term for dispersion varied as $1/r^6$ and hence, for the Lennard-Jones potential, it was assumed that $m = 6$. Since m deals with the attractive forces which are not as strong and sensitive to distance

changes as repulsive forces are, by convention, it was assumed that $n = 2m = 12$. Note that there was no logical reasoning behind these choices, they did, however, give close approximations. Plugging these m and n values back into the Lennard-Jones potential gives:

$$u(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right] \quad (\text{A.4})$$

The above function is also referred to as the Lennard-Jones (12-6) pair potential. This function is shown in Figure A.1 along with the repulsive and attractive components.

Figure A.1: Lennard-Jones (12,6) pair potential with the attractive and repulsive component. Units here are $r^* = r/\sigma$ and $u^* = u/\epsilon$ where $\sigma = 1$ and $\epsilon = 1$.

The intermolecular forces can be obtained simply by finding the derivative of the Lennard-Jones potential.

$$F(r) = -\frac{du(r)}{dr} = 24 \frac{\epsilon}{r} \left[2 \left(\frac{\sigma}{r} \right)^{13} - \left(\frac{\sigma}{r} \right)^7 \right] \quad (\text{A.5})$$

Figure A.2 is a plot of the potential and force function.

Figure A.2: Lennard-Jones (12,6) pair potential and pair force functions again where $\sigma = 1$ and $\epsilon = 1$.

A.2 Shifted-Force Potential

Small fluctuations in the potential that occur when the distance between atoms crosses r_c can further complicate the debugging process of a computer program. Hence, a shifted-force potential is introduced in order to create a smooth transition.

As the name suggests, the shifted-force potential is simply the force function translated upwards by a factor of ΔF and then differentiated to obtain the potential energy function. ΔF is obtained in a way such that, at r_c , there is a zero intermolecular force between the pair of atoms. The shifted-force potential is shown in Figure. 2.2 and the derivation is as follows.

The shifted-force is defined as the force function plus a magnitude of the shift factor.

$$F_s(r) = \begin{cases} -\frac{du}{dr} + \Delta F & r \leq r_c \\ 0 & r > r_c \end{cases} \quad (\text{A.6})$$

where,

$$\Delta F = -F(r_c) = \left(\frac{du}{dr} \right)_{r_c} \quad (\text{A.7})$$

Since the force function is simply the derivative of the potential function with respect to r , then the shifted-potential function can be found using the following:

$$\int_0^r du_s = - \int_{r_c}^r F_s(r) dr \quad (\text{A.8})$$

Integrating,

$$u_s(r) = u(r) - u(r_c) - (r + r_c) \left(\frac{du}{dr} \right)_{r_c} \quad (\text{A.9})$$

Rewriting the full shifted-force potential function gives

$$u_s(r) = \begin{cases} u(r) - u(r_c) - (r + r_c) \left(\frac{du}{dr} \right)_{r_c} & r \leq r_c \\ 0 & r > r_c \end{cases} \quad (\text{A.10})$$

APPENDIX B: Bulk Modulus Derivation

Bulk modulus is given by:

$$B = -\frac{dP}{dV/V} = -V \frac{dP}{dV} \quad (2.37)$$

where P is pressure, and V is volume. Since energy in a system increases as the volume decreases, the bulk modulus can be related to the energy as follows.

$$dE = -PdV \quad (B.1)$$

Rearranging Eq. (B.1) and rewriting it as a derivative of volume,

$$\frac{d^2 E}{dV^2} = -\frac{dP}{dV} \quad (B.2)$$

Plugging Eq. (B.2) into Eq. (2.37), the following is obtained:

$$B = V \frac{d^2 E}{dV^2} \quad (B.3)$$

The second derivative of energy as a function of volume can be rewritten using the following identity:

$$\frac{\partial E}{\partial V} = \frac{\partial E}{\partial a} \frac{\partial a}{\partial V} \quad (B.4)$$

Taking a second derivative of Eq. (B.4), the following is obtained.

$$\begin{aligned}
\frac{\partial^2 E}{\partial V^2} &= \frac{\partial}{\partial V} \left(\frac{\partial E}{\partial V} \right) = \frac{\partial}{\partial V} \left(\frac{\partial E}{\partial a} \frac{\partial a}{\partial V} \right) \\
&= \frac{\partial}{\partial V} \left(\frac{\partial E}{\partial a} \right) \frac{\partial a}{\partial V} + \frac{\partial E}{\partial a} \frac{\partial^2 a}{\partial V^2} = \frac{\partial}{\partial a} \left(\frac{\partial E}{\partial V} \right) \frac{\partial a}{\partial V} + \frac{\partial E}{\partial a} \frac{\partial^2 a}{\partial V^2} \\
&= \frac{\partial}{\partial a} \left(\frac{\partial E}{\partial a} \frac{\partial a}{\partial V} \right) \frac{\partial a}{\partial V} + \frac{\partial E}{\partial a} \frac{\partial^2 a}{\partial V^2} = \left(\frac{\partial^2 E}{\partial a^2} \frac{\partial a}{\partial V} + \frac{\partial E}{\partial a} \frac{\partial}{\partial a} \left(\frac{\partial a}{\partial V} \right) \right) \frac{\partial a}{\partial V} + \frac{\partial E}{\partial a} \frac{\partial^2 a}{\partial V^2} \\
&= \frac{\partial^2 E}{\partial a^2} \left(\frac{\partial a}{\partial V} \right)^2 + \frac{\partial E}{\partial a} \frac{\partial^2 a}{\partial V^2}
\end{aligned} \tag{B.5}$$

If deformations are small, then $a = a_0$, where a_0 is the lattice constant at equilibrium, where

$$\frac{dE}{da} = 0 \tag{B.6}$$

Eq. (B.5) then becomes,

$$\frac{\partial^2 E}{\partial V^2} = \frac{\partial^2 E}{\partial a^2} \left(\frac{\partial a}{\partial V} \right)^2 \tag{B.7}$$

Rewriting Eq. (B.3), the following is obtained.

$$B = V \frac{\partial^2 E}{\partial a^2} \left(\frac{\partial a}{\partial V} \right)^2 \tag{B.8}$$

Volume can be rewritten in terms of volume per atom as,

$$V = \Omega = \frac{a^3}{C} \quad (\text{B.9})$$

where C is the number of atoms in one crystal cube. The following constants are used.

Cubic Lattice	C
Face-centered-cubic (FCC)	4
Body-centered-cubic (BCC)	2

Table B.1: Constant C values for cubic lattices.

Taking the derivative of Eq. (B.9) w.r.t a , the following is obtained:

$$\frac{d\Omega}{da} = \frac{3}{C} a^2 \quad (\text{B.10})$$

Rewriting the above equation, we get:

$$\frac{da}{d\Omega} = \frac{C}{3} \frac{1}{a^2} \quad (\text{B.11})$$

Plugging Eq. (B.9) and Eq. (B.11) into Eq. (B.8), the following is obtained:

$$B = \frac{a_0^3}{C} \frac{d^2 E}{da^2} \left(\frac{C}{3} \frac{1}{a^2} \right)^2 = \frac{C}{9} \frac{a_0^3}{a_0^4} \frac{d^2 E}{da^2} = \frac{C}{9} \frac{a_0^2}{a_0^3} \frac{d^2 E}{da^2} = \frac{C}{9} \frac{a_0^2}{\Omega C} \frac{d^2 E}{da^2} = \frac{a_0^2}{9\Omega} \frac{d^2 E}{da^2} \quad (\text{B.12})$$

Note that a has been replaced with a_0 since the point of interest is at equilibrium.

Rewriting the Bulk modulus, the final equation is obtained.

$$B = \frac{a_0^2}{9\Omega} \frac{d^2 E}{da^2} \quad (\text{B.13})$$

At absolute zero, only potential energy exists, and hence Eq. (B.13) can be rewritten as:

$$B = \frac{a_0^2}{9\Omega} \frac{d^2 U}{da^2} \quad (\text{B.14})$$

where U is the potential energy in the system.

APPENDIX C: EAM Potential Fitting Parameters

Fitting parameters and material properties for EAM potentials based on work by Mei *et al.* [22]

	Cu	Ag	Au	Ni	Pd	Pt
c_0	0.14095	0.15326	0.21191	0.19513	0.14775	0.21295
c_1	-1.93222	-2.07431	-2.70735	-2.53828	-2.01097	-2.71705
c_2	9.60043	10.16256	12.39603	11.88049	9.91339	12.42006
c_3	-21.23718	-22.13975	-24.91083	-24.54730	-21.74378	-24.90569
c_4	20.18581	20.69699	21.02527	21.53030	20.47884	20.95506
c_5	-5.81656	-5.85219	-5.04138	-5.55520	-5.84105	-4.99108
β	5.85	5.96	6.67	6.41	5.91	6.69
δ	12.06	12.77	8.01	13.12	12.13	7.02
γ	6.82054	7.00515	7.37987	7.54709	6.95925	7.54886
ϕ_c	0.1217	0.0990	0.1443	0.1593	0.1457	0.3065

Table C.1: Coefficients c_i and model parameters for the six FCC crystals based on work by Mei *et al.* [22]. ϕ_c is in eV while all other parameters are dimensionless.

	Cu	Ag	Au	Ni	Pd	Pt
a_0 (Å)	3.617	4.090	4.080	3.520	3.890	3.920
B_0 (GPa)	142.03	108.7	180.3	187.6	195.5	288.4
Ω_0 (cm ³ /mol)	7.11	10.27	10.20	6.59	8.85	9.09
E_c (kJ/mol)	336	284	284	428	376	564

Table C.2: Material properties used in the creation of EAM potentials based on work by Mei *et al.* [22].

APPENDIX D: EAM Potential Function Plots for Cu

EAM potential functions for Cu based on model developed by Mei *et al.*[22]

Figure D.1: Pair-potential for Cu as a function the distance between a pair of atoms based on the MEI potential.

Figure D.2: Electron density for Cu as a function of distance between a pair of atoms based on the MEI potential.

Figure D.3: *Embedding function for Cu as a function of the total electron density of an atom based on the MEI potential.*

APPENDIX E: Tabulated Results of MD Calculations

Results of the MD calculations of the lattice constant size, average potential and total energy, and elastic constants for the six FCC crystals using the MEI potential, for Au using the JON potential, and for Ni using the VC potential. The following tables include all calculated terms making up Eq. (4.8) to give the total value of the specific elastic constant. The fluctuation term and the temperature correction term are denoted by Fluct and T_{Corr} respectively.

E.1 Cu – Copper (MEI)

T	a_0	Avg. Atomic Energy		C_{11}						
		Potential	Total	Fluct	T_{Corr}	B1	B2	B3	Total	
K	Å	eV		GPa						
0	3.6170	-3.4823	-3.4823	0	0	154.47	-33.52	50.10	171.05	
10	3.6174	-3.4811	-3.4798	-0.15	0.05	154.47	-33.53	50.08	170.91	
100	3.6228	-3.4694	-3.4565	-1.68	0.46	153.02	-33.78	49.93	167.95	
300	3.6351	-3.4433	-3.4046	-4.88	1.38	149.80	-34.35	49.59	161.55	
500	3.6482	-3.4158	-3.3511	-9.46	2.27	146.47	-34.91	49.19	153.56	
700	3.6624	-3.3864	-3.2960	-12.99	3.15	143.15	-35.48	48.75	146.58	
800	3.6699	-3.3710	-3.2676	-15.38	3.58	141.73	-35.77	48.53	142.68	

T	C_{12}						C_{44}					
	Fluct	T_{Corr}	B1	B2	B3	Total	Fluct	T_{Corr}	B1	B2	B3	Total
K	GPa						GPa					
0	0	0	90.41	-14.14	50.10	126.37	0	0	90.41	-14.14	0.00	76.27
10	0.01	0.00	90.36	-14.15	50.07	126.30	-0.12	0.02	90.36	-14.15	0.00	76.12
100	0.13	0.00	89.24	-14.23	49.91	125.06	-1.15	0.23	89.21	-14.23	0.01	74.08
300	0.10	0.00	86.60	-14.41	49.54	121.82	-3.39	0.69	86.61	-14.41	0.04	69.54
500	0.27	0.00	83.90	-14.57	49.11	118.71	-6.24	1.14	83.95	-14.58	0.06	64.33
700	0.23	0.00	81.18	-14.74	48.63	115.30	-8.36	1.57	81.23	-14.74	0.09	59.79
800	-0.05	0.00	79.81	-14.82	48.37	113.32	-9.65	1.79	79.82	-14.82	0.10	57.24

Table E.1: Results for Cu using the MEI potential.

E.2 Ag – Silver (MEI)

T	a_0	Avg. Atomic Energy		C_{11}					
		Potential	Total	Fluct	T_{Corr}	B1	B2	B3	Total
K	Å	eV		GPa					
0	4.0900	-2.9434	-2.9434	0	0	97.96	-23.19	51.85	126.63
10	4.0905	-2.9421	-2.9408	-0.11	0.03	97.95	-23.21	51.81	126.48
50	4.0933	-2.9369	-2.9304	-0.53	0.16	97.57	-23.36	51.62	125.46
100	4.0967	-2.9304	-2.9175	-1.09	0.32	97.12	-23.53	51.38	124.20
300	4.1108	-2.9041	-2.8654	-3.96	0.95	95.50	-24.23	50.45	118.72
500	4.1261	-2.8763	-2.8117	-7.23	1.57	93.67	-24.94	49.42	112.50
700	4.1426	-2.8465	-2.7560	-10.70	2.18	92.11	-25.67	48.38	106.29
800	4.1510	-2.8309	-2.7275	-12.47	2.47	91.33	-26.00	47.83	103.15

T	C_{12}						C_{44}					
	Fluct	T_{Corr}	B1	B2	B3	Total	Fluct	T_{Corr}	B1	B2	B3	Total
K	GPa						GPa					
0	0	0	56.71	-9.84	51.85	98.73	0	0	56.71	-9.84	0.00	46.87
10	-0.03	0.00	56.68	-9.85	51.81	98.62	-0.07	0.02	56.68	-9.85	0.00	46.78
50	0.02	0.00	56.36	-9.90	51.61	98.09	-0.38	0.08	56.37	-9.90	0.01	46.18
100	0.02	0.00	56.00	-9.96	51.37	97.43	-0.79	0.16	56.04	-9.96	0.01	45.46
300	-0.30	0.00	54.54	-10.21	50.39	94.41	-2.44	0.48	54.54	-10.21	0.04	42.41
500	-0.58	0.00	52.90	-10.45	49.34	91.20	-4.39	0.79	52.97	-10.46	0.07	38.98
700	-0.95	0.00	51.37	-10.69	48.23	87.96	-6.39	1.09	51.26	-10.69	0.10	35.37
800	-1.17	0.00	50.56	-10.79	47.67	86.27	-7.41	1.24	50.54	-10.79	0.12	33.69

Table E.2: Results for Ag using the MEI potential.

E.3 Au – Gold (MEI)

T	a_0	Avg. Atomic Energy		C_{11}					
		Potential	Total	Fluct	T_{corr}	B1	B2	B3	Total
K	Å	eV		GPa					
0	4.0800	-2.9435	-2.9435	0	0	59.95	9.21	127.33	196.49
10	4.0804	-2.9422	-2.9409	-0.22	0.03	60.01	9.20	127.26	196.28
50	4.0829	-2.9370	-2.9306	-0.77	0.16	59.97	8.77	126.67	194.80
100	4.0859	-2.9306	-2.9177	-1.82	0.32	60.00	8.30	125.98	192.79
300	4.0988	-2.9041	-2.8654	-6.33	0.96	59.72	6.23	122.95	183.53
500	4.1125	-2.8762	-2.8116	-12.27	1.59	59.66	4.17	119.96	173.11
700	4.1277	-2.8464	-2.7560	-17.72	2.20	59.54	1.93	116.72	162.67
800	4.1356	-2.8308	-2.7274	-21.29	2.50	59.65	0.87	115.10	156.84

T	C_{12}						C_{44}					
	Fluct	T_{corr}	B1	B2	B3	Total	Fluct	T_{corr}	B1	B2	B3	Total
K	GPa						GPa					
0	0	0	38.45	4.04	127.33	169.91	0	0	38.54	4.04	0.00	42.58
10	-0.13	0.00	38.55	4.04	127.25	169.70	-0.09	0.02	38.54	4.04	0.01	42.52
50	-0.25	0.00	38.44	3.85	126.64	168.67	-0.44	0.08	38.41	3.85	0.03	41.93
100	-0.76	0.00	38.26	3.64	125.91	167.04	-0.92	0.16	38.28	3.64	0.05	41.22
300	-2.66	0.00	37.61	2.73	122.79	160.46	-2.92	0.48	37.64	2.73	0.17	38.10
500	-5.57	0.00	36.98	1.83	119.61	152.85	-4.95	0.79	36.94	1.83	0.29	34.90
700	-8.33	0.00	36.20	0.87	116.15	144.89	-6.86	1.10	36.19	0.87	0.41	31.72
800	-10.84	0.00	35.81	0.411	114.5	139.87	-7.66	1.25	35.88	0.42	0.47	30.35

Table E.3: Results for Au using the MEI potential.

E.4 Au – Gold (JON)

T	a_0	Avg. Atomic Energy		C_{11}					
		Potential	Total	Fluct	T_{corr}	B1	B2	B3	Total
K	Å	eV		GPa					
10	4.0822	-3.9236	-3.9223	-0.707	0.033	226.23	-156.23	114.32	183.64
100	4.0880	-3.9118	-3.8988	-2.04	0.32	225.25	-158.46	113.25	178.32
300	4.1017	-3.8848	-3.8460	-7.23	0.96	222.28	-157.86	110.85	168.99
500	4.1163	-3.8568	-3.7922	-13.11	1.58	219.80	-156.01	108.45	160.72
800	4.1399	-3.8120	-3.7087	-20.84	2.49	215.12	-153.19	104.58	148.16

T	C_{12}						C_{44}					
	Fluct	T_{corr}	B1	B2	B3	Total	Fluct	T_{corr}	B1	B2	B3	Total
K	GPa						GPa					
10	-0.615	0	109.8	-67.96	114.31	155.57	-0.112	0.016	109.8	-67.96	0.00	41.79
100	-0.83	0.00	109.02	-68.94	113.18	152.43	-1.01	0.16	108.99	-68.93	0.04	39.26
300	-2.82	0.00	106.87	-68.36	110.70	146.39	-3.37	0.48	107.00	-68.39	0.13	35.85
500	-5.96	0.00	104.77	-67.08	108.14	139.87	-5.35	0.79	104.83	-67.08	0.23	33.42
800	-9.22	0.00	101.36	-65.12	104.18	131.21	-8.79	1.25	101.32	-65.09	0.37	29.05

Table E.4: Results for Au using the JON potential.

E.5 Ni – Nickel (MEI)

T	a_0	Avg. Atomic Energy		C_{11}					
		Potential	Total	Fluct	T_{Corr}	B1	B2	B3	Total
K	Å	eV		GPa					
0	3.5200	-4.4356	-4.4356	0	0	277.70	-66.02	30.05	241.73
20	3.5208	-4.4330	-4.4304	-0.31	0.10	277.30	-65.98	30.13	241.24
60	3.5228	-4.4278	-4.4201	-1.23	0.30	276.04	-65.89	30.34	239.56
100	3.5249	-4.4226	-4.4097	-1.84	0.50	274.74	-65.79	30.56	238.17
200	3.5300	-4.4095	-4.3836	-4.07	1.00	271.70	-65.57	31.09	234.16
300	3.5352	-4.3962	-4.3574	-6.93	1.50	268.37	-65.34	31.59	229.18
400	3.5406	-4.3826	-4.3309	-8.24	1.99	265.38	-65.13	32.12	226.12
500	3.5460	-4.3688	-4.3042	-11.41	2.48	262.37	-64.93	32.61	221.11
600	3.5516	-4.3547	-4.2771	-13.09	2.96	259.22	-64.72	33.11	217.48
700	3.5574	-4.3402	-4.2498	-15.72	3.43	255.80	-64.51	33.59	212.59
760	3.5609	-4.3314	-4.2332	-16.94	3.72	254.05	-64.39	33.88	210.32

T	C_{12}						C_{44}					
	Fluct	T_{Corr}	B1	B2	B3	Total	Fluct	T_{Corr}	B1	B2	B3	Total
K	GPa						GPa					
0	0	0	158.7	-28.63	30.05	160.14	0	0	158.7	-28.63	0.00	130.09
20	0.03	0.00	158.43	-28.61	30.13	159.98	-0.32	0.05	158.45	-28.61	0.00	129.57
60	0.15	0.00	157.64	-28.56	30.34	159.58	-0.89	0.15	157.64	-28.56	0.00	128.35
100	0.13	0.00	156.71	-28.50	30.56	158.90	-1.63	0.25	156.79	-28.50	0.01	126.92
200	0.14	0.00	154.61	-28.37	31.07	157.44	-3.30	0.50	154.63	-28.37	0.01	123.47
300	0.32	0.00	152.57	-28.24	31.57	156.22	-4.91	0.75	152.62	-28.25	0.02	120.23
400	-0.19	0.00	150.35	-28.11	32.07	154.12	-6.66	1.00	150.35	-28.11	0.03	116.60
500	0.17	0.00	148.3	-27.99	32.56	153.07	-8.23	1.24	148.25	-27.99	0.03	113.31
600	-0.11	0.00	146.3	-27.86	33.06	151.37	-9.51	1.48	146.18	-27.86	0.04	110.33
700	-0.15	0.00	144	-27.74	33.52	149.60	-11.85	1.72	144.09	-27.74	0.05	106.27
760	-0.37	0.00	142.8	-27.65	33.81	148.55	-12.70	1.86	142.75	-27.66	0.06	104.31

Table E.5: Results for Ni using the MEI potential.

E.6 Ni – Nickel (VC)

T	a_0	Avg. Atomic Energy		C_{11}					
		Potential	Total	Fluct	T_{Corr}	B1	B2	B3	Total
K	Å	eV		GPa					
20	3.5210	-4.4540	-4.4515	-0.362	0.101	-420.93	641.16	22.61	242.58
60	3.5230	-4.4488	-4.4411	-1.06	0.30	-420.45	639.39	23.16	241.34
100	3.5249	-4.4436	-4.4307	-1.88	0.50	-419.77	637.66	23.69	240.20
200	3.5298	-4.4304	-4.4046	-4.31	1.00	-418.61	633.47	24.99	236.54
300	3.5349	-4.4170	-4.3782	-7.00	1.50	-417.82	629.43	26.30	232.41
400	3.5400	-4.4034	-4.3517	-9.49	1.99	-417.56	625.35	27.59	227.88
500	3.5453	-4.3895	-4.3249	-11.90	2.48	-417.00	621.51	28.89	223.98
600	3.5507	-4.3754	-4.2979	-13.83	2.96	-416.67	617.55	30.18	220.19
700	3.5564	-4.3610	-4.2706	-17.11	3.44	-416.57	613.44	31.49	214.69

T	C_{12}						C_{44}					
	Fluct	T_{Corr}	B1	B2	B3	Total	Fluct	T_{Corr}	B1	B2	B3	Total
K	GPa						GPa					
20	0.053	0	-105.82	230.78	22.60	147.62	-0.36	0.05	-105.87	230.81	0.00	124.63
60	0.12	0.00	-105.99	230.19	23.16	147.49	-1.04	0.15	-105.97	230.20	0.00	123.34
100	0.13	0.00	-106.02	229.67	23.68	147.46	-1.81	0.25	-106.17	229.64	0.00	121.92
200	-0.07	0.00	-106.59	228.24	24.99	146.57	-3.74	0.50	-106.64	228.23	0.00	118.36
300	-0.04	0.00	-107.24	226.98	26.29	145.99	-5.81	0.75	-107.35	226.95	0.00	114.55
400	-0.14	0.00	-108.10	225.69	27.58	145.04	-8.01	1.00	-108.13	225.71	0.00	110.57
500	-0.33	0.00	-109.05	224.50	28.88	144.00	-9.42	1.24	-109.07	224.50	0.01	107.25
600	-0.87	0.00	-110.03	223.31	30.17	142.56	-11.86	1.48	-109.88	223.36	0.01	103.11
700	-1.08	0.00	-110.91	222.00	31.48	141.49	-13.52	1.72	-110.96	222.00	0.01	99.26

Table E.6: Results for Ni using the VC potential.

E.7 Pd – Palladium (MEI)

T	a_0	Avg. Atomic Energy		C ₁₁					
		Potential	Total	Fluct	T _{corr}	B1	B2	B3	Total
K	Å	eV		GPa					
0	3.8900	-3.8969	-3.8969	0	0	153.12	-33.06	104.46	224.53
20	3.8908	-3.8943	-3.8917	-0.23	0.08	153.03	-33.15	104.31	224.04
100	3.8943	-3.8839	-3.8710	-1.37	0.37	152.41	-33.58	103.66	221.50
300	3.9031	-3.8581	-3.8193	-5.04	1.11	150.89	-34.61	102.08	214.43
500	3.9123	-3.8313	-3.7667	-8.94	1.84	149.18	-35.66	100.42	206.84
800	3.9272	-3.7888	-3.6855	-12.92	2.92	147.08	-37.26	97.88	197.70

T	C ₁₂						C ₄₄					
	Fluct	T _{corr}	B1	B2	B3	Total	Fluct	T _{corr}	B1	B2	B3	Total
K	GPa						GPa					
0	0	0	89.34	-13.99	104.46	179.81	0	0	89.34	-13.99	0.00	75.53
20	-0.04	0.00	89.22	-14.02	104.30	179.46	-0.19	0.04	89.22	-14.02	0.00	75.05
100	-0.09	0.00	88.66	-14.19	103.64	178.02	-1.02	0.19	88.59	-14.19	0.02	73.59
300	-0.39	0.00	87.18	-14.58	102.00	174.20	-2.95	0.56	87.22	-14.58	0.06	70.31
500	-0.88	0.00	85.60	-14.96	100.30	170.07	-4.73	0.92	85.62	-14.97	0.10	66.95
800	-1.83	0.00	83.20	-15.54	97.64	163.47	-8.11	1.46	83.15	-15.54	0.16	61.12

Table E.7: Results for Pd using the MEI potential.

E.8 Pt – Platinum (MEI)

T	a_0	Avg. Atomic Energy		C ₁₁					
		Potential	Total	Fluct	T _{corr}	B1	B2	B3	Total
K	Å	eV		GPa					
0	3.9200	-5.8458	-5.8458	0	0	98.83	47.25	179.30	325.38
300	3.9308	-5.8067	-5.7680	-6.39	1.09	98.55	43.71	176.31	313.26

T	C ₁₂						C ₄₄					
	Fluct	T _{corr}	B1	B2	B3	Total	Fluct	T _{corr}	B1	B2	B3	Total
K	GPa						GPa					
0	0	0	68.36	20.75	179.30	268.41	0	0	68.36	20.75	0.00	89.11
300	-1.53	0.00	67.30	19.14	176.18	261.07	-3.37	0.55	67.40	19.14	0.14	83.85

Table E.8: Results for Pt using the MEI potential.

APPENDIX F: Simulation Procedures

For any type of simulation, all necessary files are put into the same folder including all compiled codes, MATLAB files, and EAM files. A typical case would include the following files.

- MDROB.exe - Compiled ASCII C code from file MDROB.c
- MDROB_BORN.exe - Compiled ASCII C code from file MDROB_BORN.c
- mdrob.m - MATLAB main control file
- mdrob_disp_born.m - MATLAB function for mdrob.m
- eam_cu_mei.txt - EAM potential file (MEI EAM for Cu used as an example)

Additional files may include:

- input.txt - Input file for running MDROB.exe or MDROB_BORN.exe manually
- eam_cu_bulk.txt - Input file containing the bulk modulus values at various temperatures used for running multiple cases

F.1 Zero-stress simulation

The simulation procedures are given in flow charts with the shape coding for the various items given in Figure F.1. The simulation procedure for a zero-stress case used to calculate the equilibrium lattice constant size at a specified temperature is outlined in Figure F.2.

Figure F.1: Shape identification for flow charts.

Figure F.2: Zero-pressure, variable volume simulation flow chart.

In Figure F.2, The USER STEP consists of the input of case data into the control file mdrob.m. Case data includes EAM potential file name, mass of atom, initial lattice constant size, initialization and clamp temperatures, time-step size, size of crystal, total number of iterations, bulk modulus, and the number of iterations per energy, stress, and lattice constant output.

Additional controls include the number of iterations to be used in the production stage of the simulation, and the size of the delay for the temperature clamp and volume rescaling.

The actual simulation is composed of two steps. The first step consists of running the main MD code MDROB.exe which produces output files containing system and atomic energies, stresses, and lattice constant sizes as functions of iterations.

The second simulation step is the post-processing step where the energy, stress, and lattice constant size files are read into the main control file mdrob.m and analyzed. The three output files are averaged over the specified number of iterations in order to obtain the equilibrium energies, stresses, and lattice constant size. All three variables can be plotted to ensure that equilibrium has been reached and that the various stresses do equilibrate around zero.

F.2 Constant volume simulation

Once the lattice constant size for a specified temperature has been calculated, then a simulation using a constant volume (based on the lattice constant size) can be performed in order to calculate the elastic constants. This simulation procedure is outlined in Figure F.3.

Figure F.3: Constant volume simulation flow chart.

The simulation procedure begins by the first specifying the case data which is the same as in the case of the zero-stress simulation except for the following three differences.

- The initial lattice constant size is now substituted with an equilibrium lattice constant size obtained from the zero-stress simulation.
- The bulk modulus is no longer required and a value of zero is entered.
- The number of iterations at which the atomic positions will begin to be stored is entered along with the number of iterations per output.

In this case, the actual simulation is composed of three steps. The first step is similar to step one in the zero-stress simulation. However, here the file containing the lattice constant size is no longer outputted. Instead, files containing the atomic positions at specified iterations are created.

Step two consists of running MDROB_BORN.exe which calculates the Born terms using the stored files containing the atomic positions. Note that this file also requires the EAM potential file and the input file. At the end of this step, a new file containing the calculated Born terms is created.

The last step is again the post-processing step where the energy, stress, born term files are read into the control file mdrob.m and analyzed. The atomic energies and system stresses can be plotted to ensure equilibrium has been reached. All components of the elastic constants are then calculated and displayed. At the end of the simulation, a file containing all of the compiled and calculated data is created called out_results.txt. The file contains results in the same format as Table E.1 through Table E.8 where the first column contains the temperature in units of Kelvin, the second column contains the average atomic potential energy in units of eV, and so on.

Other plots that can be created include lattice constant size as function of iterations, potential energy as a function of lattice constant size, and a velocity distribution plot.

APPENDIX G: Source Code

This section contains all source codes used in this study. They include two source codes written in ASCII C used to create MDROB.exe and MDROB_BORN.exe. The control file, mdrob.m, function, mdrob_disp_born.m, and MEI EAM potential generation file for Cu written in MATLAB R14 are also included. All of the codes listed here include comments explaining the various parts of each code.

G.1 Source code for MDROB.c

```
// MDROB - Molecular Dynamics Simulation Code
// Developed by: Robert Szydlowski
// Last update: May 17, 2006
//
// ASSUMPTIONS:
// Number of entries in EAM data for PAIR POTENTIAL and DENS POTENTIAL is identical

#include <stdio.h>
#include <stdafx.h>          // Code won't compile without this file when compiling using MS
                             // VisualC 6.0
#include <stdlib.h>
#include <math.h>
#include <string.h>          // Used in strcmp(), strcat(), and strcpy()
#include <time.h>            // Used for random number generation, seed is set according to time
                             // at execution

#define MAX_PTS 1372        // Max number of points (currently set to 7x7x7 cell of 4 atoms =
                             // 1372)
#define MAX_STR 20          // Maximum number of characters in string
#define MAX_EAM 3000        // Max number of EAM file points per function

void read_input(char* argv[], int input1[], double input2[], char filename_eam[], char
filename_1[], char filename_2[], char filename_3[], char filename_4[], char filename_5a[], char
filename_5b[], int &test_break, char filename_6[]);
void read_const(double input3[]);
void create_crystal(int box_x, int box_y, int box_z, double particle_list[][3]);
void calc_vel(double vel[][3], int num_pts, double mass, double kB, double itemp, double pi,
double amu);
double integ_vel_dist(double limit_high, double mass, double kB, double itemp);
void read_eam(double eam[][6], char filename_eam[], int &eunit_type);
void calc_md(int input1[], double input2[], double particle_list[][3], double particle_vel[][3],
double eam[][6], int num_pts, int eunit_type, double mass_atom, double kB, char filename_1[],
char filename_4[], char filename_5a[], char filename_5b[], char filename_6[]);
void force_calc(double particle_force[][3], double particle_list[][3], int num_pts, double
eam[][6], int eunit_type, double perio_array[], double p_pot_array[], int test_ssave);
void scale_vel(double particle_vel[][3], double clamp, int num_pts, double mass_atom, double kB,
int delay_clamp);
void calc_pot(double particle_list[][3], int num_pts, double eam[][6], double particle_pot[],
double &total_pot, double perio_array[], int eunit_type);
void calc_stress(double particle_vel[][3], int num_pts, double mass_atom, double p_pot_array[],
double pxx[]);

int main(int argc, char* argv[])
{
    int input1[14];
    int box_x, box_y, box_z;
```

```

int perio_x, perio_y, perio_z;
int iter, iter_out;
double input2[9], input3[6];
double mass, a0, itemp, clamp, dtime; // Variables used in input2
double strain_x, strain_y, strain_z; // Variables used in input2
double kB, pi, amu; // Variables used in input3
int xsave_start, xsave; // Variables used in input1
int ssave, bsave;
double bulk;
int delay_clamp, delay_box;

int i; // Counter
int num_pts; // Number of atoms being simulated
char filename_eam[MAX_STR]; // EAM filename
int eunit_type; // Energy units used in EAM file
double mass_atom; // [kg] Mass of 1 atom

char filename_1[MAX_STR]; // Filename
char filename_2[MAX_STR]; // Filename of final positions and velocities file
char filename_3[MAX_STR]; // Filename of initial positions and velocities file
(at perfect lattice positions)
char filename_4[MAX_STR];
char filename_5a[MAX_STR];
char filename_5b[MAX_STR];
char filename_6[MAX_STR];

double particle_list[MAX_PTS][3];
double particle_vel[MAX_PTS][3];
double eam[MAX_EAM][6];

int test_break = 0;

FILE *outfile_2;
FILE *outfile_3;

printf("### MDROB ###\nCRYSTAL PARAMETERS\n");

if (argc == 1)
{
    printf("***** ERROR - No specified input file *****\n");
    return 0;
}
else if (argc > 2)
{
    printf("***** ERROR - Too many input parameters *****\n");
    printf(" format: mdrob [inputfile]\n");
    return 0;
}

// Chaning seed for random number generation according to the time at execution
srand( (unsigned int) time( NULL ) );

// Reading in input variables
read_input(argv, input1, input2, filename_eam, filename_1, filename_2, filename_3,
filename_4, filename_5a, filename_5b, test_break, filename_6);

if (test_break == 1) // Checking for errors in input parameters
    return 0;

box_x = input1[0];
box_y = input1[1];
box_z = input1[2];
perio_x = input1[3];
perio_y = input1[4];
perio_z = input1[5];
iter = input1[6];
iter_out = input1[7];
xsave_start = input1[8];
xsave = input1[9];
ssave = input1[10];
bsave = input1[11];

```

```

delay_clamp = input1[12];
delay_box   = input1[13];

mass        = input2[0];
a0          = input2[1];
itemp       = input2[2];
clamp       = input2[3];
dtime       = input2[4];
strain_x    = input2[5];
strain_y    = input2[6];
strain_z    = input2[7];
bulk        = input2[8];

// Printing input parameters
printf(" Atomic mass = %.3f\n", mass);
printf(" a0 = %.3f [Angstroms]\n", a0);
printf(" box = %i %i %i\n", box_x, box_y, box_z);
printf("SIMULATION PARAMETERS\n");
printf(" itemp = %.1f [K]\n", itemp);
if (clamp == 0)
    printf(" clamp = NO TEMPERATURE CLAMP USED\n");
else
    printf(" clamp = %.1f [K]\n", clamp);
printf(" dtime = %.1e [s]\n", dtime);
printf(" iter = %i\n", iter);
printf(" iter_out = %i\n", iter_out);
printf(" strain = %7.4f %7.4f %7.4f\n", strain_x, strain_y, strain_z);
printf(" perio = %i %i %i\n", perio_x, perio_y, perio_z);
printf(" xsave = %i %i\n", xsave_start, xsave);
printf(" ssave = %i\n", ssave);
printf(" bulk = %.2f [GPa]\n", bulk);
printf(" bsave = %i\n", bsave);
printf(" delay_clamp = %i\n", delay_clamp);
printf(" delay_box = %i\n", delay_box);

printf("EAM filename: %s\n", filename_eam);
printf("\n Output data files\n");
printf("Iterations and potential energy: %s\n", filename_1);
printf("Initial particle positions and velocities: %s\n", filename_3);
printf("Final particle positions and velocities: %s\n", filename_2);
printf("Iterations and stress tensor components: %s\n", filename_4);
printf("Instantaneous particle positions: %s %s\n", filename_5a, filename_5b);
printf("Iterations and box size: %s\n", filename_6);
printf("\n");

// Reading in simulation constants
read_const(input3);
kB = input3[0];
pi = input3[1];
amu = input3[2];

// Calculating mass of 1 atom
mass_atom = mass*amu;           // [grams] Atom weight
mass_atom = mass_atom/1000;     // [kg] Atom weight
//printf("mass_atom = %e [kg]\n", mass_atom);

// -----
//                                     INITIALIZATION
// -----

// Creating crystal array
create_crystal(box_x, box_y, box_z, particle_list);
num_pts = box_x*box_y*box_z*4;
printf("----- Crystal created ----- Number of atoms = %i\n", num_pts);

// Scaling crystal to initial lattice size and applying strains
for(i = 0; i < num_pts; i++)
{
    particle_list[i][0] = particle_list[i][0]*a0*strain_x;
    particle_list[i][1] = particle_list[i][1]*a0*strain_y;
    particle_list[i][2] = particle_list[i][2]*a0*strain_z;
}

```

```

    }

    // Kinetic energy - Applying scaled velocities to atoms
    calc_vel(particle_vel, num_pts, mass, kB, itemp, pi, amu);

    // Writing to file the initial positions and velocities
    outfile_3 = fopen(filename_3, "w"); // Opening file for writing
    fprintf(outfile_3, "atom      x      y      z      Vx      Vy      Vz
Magnitude\n");
    for(i = 0; i < num_pts; i++)
    {
        fprintf(outfile_3, "%4i %8.5f %8.5f %8.5f", i+1, particle_list[i][0],
particle_list[i][1], particle_list[i][2]);
        fprintf(outfile_3, " %9.3f %9.3f %9.3f %8.3f\n", particle_vel[i][0],
particle_vel[i][1], particle_vel[i][2], sqrt(particle_vel[i][0]*particle_vel[i][0] +
particle_vel[i][1]*particle_vel[i][1] + particle_vel[i][2]*particle_vel[i][2]));
    }
    fclose(outfile_3); // Closing file

    // Read EAM potential file
    eunit_type = 0; // Initially setting to zero
    read_eam(eam, filename_eam, eunit_type);

    // -----
    //                               MOLECULAR DYNAMICS SIMULATION
    // -----

    calc_md(input1, input2, particle_list, particle_vel, eam, num_pts, eunit_type, mass_atom,
kB, filename_1, filename_4, filename_5a, filename_5b, filename_6);

    // -----
    //                               POST-PROCESSING
    // -----

    // Writing to file the final positions and velocities
    outfile_2 = fopen(filename_2, "w"); // Opening file for writing
    fprintf(outfile_2, "atom      x      y      z      Vx      Vy      Vz
Magnitude\n");
    for(i = 0; i < num_pts; i++)
    {
        fprintf(outfile_2, "%4i %8.5f %8.5f %8.5f", i+1, particle_list[i][0],
particle_list[i][1], particle_list[i][2]);
        fprintf(outfile_2, " %9.3f %9.3f %9.3f %8.3f\n", particle_vel[i][0],
particle_vel[i][1], particle_vel[i][2], sqrt(particle_vel[i][0]*particle_vel[i][0] +
particle_vel[i][1]*particle_vel[i][1] + particle_vel[i][2]*particle_vel[i][2]));
    }
    fclose(outfile_2); // Closing file

    // -----
    // Ending program
    printf("\n----- Simulation FINISHED ----- \n");
    return 0;
}

void read_input(char* argv[], int input1[], double input2[], char filename_eam[], char
filename_1[], char filename_2[], char filename_3[], char filename_4[], char filename_5a[], char
filename_5b[], int &test_break, char filename_6[])
{
    // Reading in input variables

    int box_x, box_y, box_z;
    int perio_x, perio_y, perio_z;
    int iter, iter_out;
    double mass, a0, itemp, clamp, dtime;
    double strain_x, strain_y, strain_z;
    char string[20];
    char string2[20];
    double num1, num2, num3;
    int xsave_start, xsave;
    int ssave;
    double bulk; // Only used in zero-pressure simulation (variable volume)

```

```

int bsave; // Only used in zero-pressure simulation (variable volume)
int delay_clamp, delay_box;

// -----
// Setting inputs that are not always used
bulk = 0;
bsave = 0;
strcpy(filename_6, "none");
delay_clamp = 33; // Number of iterations, set to 33 iterations as
default
delay_box = 33; // Number of iterations, set to 33 iterations as
default

// -----
FILE *infile_input;

infile_input = fopen(argv[1], "r"); // Opening file for reading

if (NULL == infile_input) // Checking if file exists
{
    printf("***** ERROR - Cannot open file '%s' for reading, file does not exist.
*****\n", argv[1]);
    test_break = 1;
    return;
}
else
{
    fscanf(infile_input, "%s", string); // Read in first string in line

    while (feof(infile_input) == 0)
    {
        if (string[0] != '#')
        {
            if (strcmp(string, "iter") == 0) // Number of iterations
            {
                fscanf(infile_input, "%lf", &num1);
                iter = (int)num1;
            }
            else if (strcmp(string, "iter_out") == 0) // Output to be
written to file every 'iter_out' iterations
            {
                fscanf(infile_input, "%lf", &num1);
                iter_out = (int)num1;
            }
            else if (strcmp(string, "mass") == 0) // Atomic mass of atom
            {
                fscanf(infile_input, "%lf", &num1);
                mass = num1;
            }
            else if (strcmp(string, "lattice") == 0) // Lattice constant
size
            {
                fscanf(infile_input, "%lf", &num1);
                a0 = num1;
            }
            else if (strcmp(string, "itemp") == 0) // [K] Initialization
temperature
            {
                fscanf(infile_input, "%lf", &num1);
                itemp = num1;
            }
            else if (strcmp(string, "clamp") == 0) // [K] Clamp
temperature
            {
                fscanf(infile_input, "%lf", &num1);
                clamp = num1;
            }
            else if (strcmp(string, "dtime") == 0) // [s] Time-step size
            {
                fscanf(infile_input, "%lf", &num1);
                dtime = num1;
            }
        }
    }
}

```



```

    }
    else if (strcmp(string, "box") == 0)
    {
        fscanf(infile_input, "%lf %lf %lf", &num1, &num2, &num3);
        box_x = (int)num1;
        box_y = (int)num2;
        box_z = (int)num3;
    }
    else if (strcmp(string, "perio") == 0)
    {
        fscanf(infile_input, "%lf %lf %lf", &num1, &num2, &num3);
        perio_x = (int)num1;
        perio_y = (int)num2;
        perio_z = (int)num3;
    }
    else if (strcmp(string, "strain") == 0)
    {
        fscanf(infile_input, "%lf %lf %lf", &num1, &num2, &num3);
        strain_x = num1;
        strain_y = num2;
        strain_z = num3;
    }
    else if (strcmp(string, "xsave") == 0)
    {
        fscanf(infile_input, "%lf %lf %s %s", &num1, &num2, string,
string2);

        xsave_start = (int)num1;
        xsave = (int)num2;
        strcpy(filename_5a, string);
        strcpy(filename_5b, string2);
    }
    else if (strcmp(string, "file_eam") == 0) // EAM potential file
name
    {
        fscanf(infile_input, "%s", string);
        strcpy(filename_eam, string);
    }
    else if (strcmp(string, "file_iter") == 0) // Iterations and
potential energy output file name
    {
        fscanf(infile_input, "%s", string);
        strcpy(filename_1, string);
    }
    else if (strcmp(string, "file_part_fin") == 0) // Final
particle positions and velocities
    {
        fscanf(infile_input, "%s", string);
        strcpy(filename_2, string);
    }
    else if (strcmp(string, "file_part_ini") == 0) // Final
particle positions and velocities
    {
        fscanf(infile_input, "%s", string);
        strcpy(filename_3, string);
    }
    else if (strcmp(string, "ssave") == 0)
    {
        fscanf(infile_input, "%lf %s", &num1, string);
        ssave = (int)num1;
        strcpy(filename_4, string);
    }
    else if (strcmp(string, "bulk") == 0)
    {
        fscanf(infile_input, "%lf", &num1);
        bulk = num1;
    }
    else if (strcmp(string, "bsave") == 0)
    {
        fscanf(infile_input, "%lf %s", &num1, string);
        bsave = (int)num1;
        strcpy(filename_6, string);
    }

```

```

    }
    else if (strcmp(string, "delay_clamp") == 0)
    {
        fscanf(infile_input, "%lf", &num1);
        delay_clamp = (int)num1;
    }
    else if (strcmp(string, "delay_box") == 0)
    {
        fscanf(infile_input, "%lf", &num1);
        delay_box = (int)num1;
    }
    else
    {
        printf("***** ERROR - Unknown command in input file
*****\n");
        test_break = 1;
        return;
    }
}

fscanf(infile_input, "%s", string);
}

fclose(infile_input); // Closing file

//Putting all input data into two arrays
input1[0] = box_x;
input1[1] = box_y;
input1[2] = box_z;
input1[3] = perio_x;
input1[4] = perio_y;
input1[5] = perio_z;
input1[6] = iter;
input1[7] = iter_out;
input1[8] = xsave_start;
input1[9] = xsave;
input1[10] = ssave;
input1[11] = bsave;
input1[12] = delay_clamp;
input1[13] = delay_box;

input2[0] = mass;
input2[1] = a0;
input2[2] = itemp;
input2[3] = clamp;
input2[4] = dtime;
input2[5] = strain_x;
input2[6] = strain_y;
input2[7] = strain_z;
input2[8] = bulk;
}
void read_const(double input3[])
{
    // Reading in simulation constants

    double kB = 1.3806505e-23;    // [J/K] Boltzmann's constant
    double pi = 3.14159;
    double amu = 1.66054e-24;    // [grams] Atomic mass unit

    input3[0] = kB;
    input3[1] = pi;
    input3[2] = amu;
}

void create_crystal(int box_x, int box_y, int box_z, double particle_list[][3])
{
    // Creating crystal array

    int n, i, j, k, x;

```

```

// FCC crystal
static double latt_pts[4][3] = {
    { 0.25, 0.25, 0.25},
    { 0.25, 0.75, 0.75},
    { 0.75, 0.25, 0.75},
    { 0.75, 0.75, 0.25}
};

// Creating crystal
n = 0;
for(i = 0; i < box_x; i++)
{
    for(j = 0; j < box_y; j++)
    {
        for(k = 0; k < box_z; k++)
        {
            for(x = 0; x < 4; x++) // 4 points used in FCC
            {
                particle_list[n][0] = latt_pts[x][0]+i;
                particle_list[n][1] = latt_pts[x][1]+j;
                particle_list[n][2] = latt_pts[x][2]+k;
                n = n+1;
            }
        }
    }
}

void calc_vel(double vel[][3], int num_pts, double mass, double kB, double itemp, double pi,
double amu)
{
    int num_x; // Number of distinct velocities
    double area_div; // Area = (Total area under curve) / num_x
    double area_cur; // Current area being tested for
    double areaF; // Calculated area
    double x, x_inc;
    int i,j,k,ii; // Used as counters
    double vel_dist[MAX_PTS/2]; // Normally distributed distinct velocities
    double vel_max; // Max velocity used for scaling
    int rand_temp;
    int rand_num[MAX_PTS][3];
    int rand_test, rand_test2;
    double vel_temp[MAX_PTS][3];
    int test_parity; // Used for parity check, 0 for odd, 1 for even
    int parity_num = 0;
    double mass_atom;
    double vel_sum_sq = 0;
    double KE_temp;
    double T_temp;
    double factor;
    double avg_vel[3];

    // Checking if odd or even number of atoms is used
    if (floor((double)num_pts/2)*2 == num_pts) // Even number
    {
        num_x = num_pts/2;
        test_parity = 1;
    }
    if (floor((double)num_pts/2)*2 != num_pts) // Odd number
    {
        num_x = (num_pts-1)/2;
        test_parity = 0;
    }

    x_inc = 0.001;
    x = 0;
    area_div = 0.886/(double)num_x; // Area = (Total area under curve) / num_x
    area_cur = area_div; // Current area being tested for
    j = 0;

    // Creating normally distributed velocities

```

```

for(i = 0; j < num_x; i++)
{
    areaF = integ_vel_dist(x, mass, kB, itemp);
    if (areaF >= area_cur)
    {
        vel_dist[j] = x;
        area_cur = area_cur + area_div;
        j = j+1;
    }
    x = x + x_inc;
}

// Scaling velocities for maximum = 1
vel_max = vel_dist[num_x-1];
for(i = 0; i < num_x; i++)
    vel_dist[i] = vel_dist[i]/vel_max;

// Storing distinct velocities and their negative counterparts
for(i = 0; i < num_x; i++)
{
    vel[i][0] = vel_dist[i];
    vel[i][1] = vel_dist[i];
    vel[i][2] = vel_dist[i];
}

if (test_parity == 0)
{
    vel[num_x][0] = 0; vel[num_x][1] = 0; vel[num_x][2] = 0;
    parity_num = 1;
}

for(i = 0; i < num_x + parity_num; i++)
{
    vel[num_x+i+parity_num][0] = -vel_dist[i];
    vel[num_x+i+parity_num][1] = -vel_dist[i];
    vel[num_x+i+parity_num][2] = -vel_dist[i];
}

// Creating a permutation of numbers 1 thru 2*num_x+parity_num
rand_num[0][0] = 1 + rand() % (2 * num_x + parity_num);
rand_num[0][1] = 1 + rand() % (2 * num_x + parity_num);
rand_num[0][2] = 1 + rand() % (2 * num_x + parity_num);
for(ii = 0; ii < 3; ii++)
{
    j = 1;
    for(i = 1; i < 2 * num_x + parity_num; i++)
    {
        rand_test = 0;
        while(rand_test == 0)
        {
            rand_temp = 1 + rand() % (2 * num_x + parity_num);
            rand_test2 = 0;
            for(k = 0; (k < j) & (rand_test == 0); k++)
            {
                if(rand_temp == rand_num[k][ii])
                    rand_test2 = 1;
            }
            if(rand_test2 == 0)
            {
                rand_num[j][ii] = rand_temp;
                j = j+1;
                rand_test = 1;
            }
        }
    }
}

// Using permutation to randomize velocities
for(i = 0; i < 2*num_x+parity_num; i++)
{
    vel_temp[i][0] = vel[i][0];

```

```

        vel_temp[i][1] = vel[i][1];
        vel_temp[i][2] = vel[i][2];
    }
    for(i = 0; i < 2*num_x+parity_num; i++)
    {
        vel[i][0] = vel_temp[rand_num[i][0]-1][0];
        vel[i][1] = vel_temp[rand_num[i][1]-1][1];
        vel[i][2] = vel_temp[rand_num[i][2]-1][2];
    }

    // Scaling velocities for desired temperature itemp
    mass_atom = mass*amu;           // [grams] Atom weight
    mass_atom = mass_atom/1000;     // [kg] Atom weight
    for(i = 0; i < num_pts; i++)
        vel_sum_sq = vel_sum_sq + vel[i][0]*vel[i][0] + vel[i][1]*vel[i][1] +
vel[i][2]*vel[i][2];
    KE_temp = 0.5*mass_atom*vel_sum_sq;
    T_temp = 2*KE_temp/(3*num_pts*kB);
    factor = itemp/T_temp;
    for(i = 0; i < num_pts; i++)
    {
        vel[i][0] = vel[i][0]*sqrt(factor);
        vel[i][1] = vel[i][1]*sqrt(factor);
        vel[i][2] = vel[i][2]*sqrt(factor);
    }

    // Checking for set temperature
    vel_sum_sq = 0;
    for(i = 0; i < num_pts; i++)
        vel_sum_sq = vel_sum_sq + vel[i][0]*vel[i][0] + vel[i][1]*vel[i][1] +
vel[i][2]*vel[i][2];
    KE_temp = 0.5*mass_atom*vel_sum_sq;
    T_temp = 2*KE_temp/(3*num_pts*kB);
    printf("Temperature check: T = %6.2f [K]\n", T_temp);

    // Checking center of mass motion, should be zero
    avg_vel[0] = 0; avg_vel[1] = 0; avg_vel[2] = 0;
    for(i = 0; i < num_pts; i++)
    {
        avg_vel[0] = avg_vel[0] + vel[i][0];
        avg_vel[1] = avg_vel[1] + vel[i][1];
        avg_vel[2] = avg_vel[2] + vel[i][2];
    }
    avg_vel[0] = avg_vel[0]/num_pts;
    avg_vel[1] = avg_vel[1]/num_pts;
    avg_vel[2] = avg_vel[2]/num_pts;

    printf("Average velocity in each direction in [m/s]\n");
    printf(" Vx = %e, Vy = %e, Vz = %e\n", avg_vel[0], avg_vel[1], avg_vel[2]);
}

double integ_vel_dist(double limit_high, double mass, double kB, double itemp)
{
    // Function used to integrate Boltzmann's distribution function
    int num_int = 1000;           // Number of intervals used in integration
    int j;                         // Used as counter
    double interv;
    double pos_x;
    double areaI = 0;
    double F;

    interv = limit_high/num_int;
    pos_x = interv;

    for(j = 0; j < num_int; j++)
    {
        F = exp(-(2*pos_x-interv)/2*(2*pos_x-interv)/2);
        areaI = areaI + F*interv;
        pos_x = pos_x + interv;
    }
}

```

```

        return areaI;
    }

void read_eam(double eam[][6], char filename_eam[], int &eunit_type)
{
    // Reading in eam
    FILE *infile_eam;
    int c;
    int i;
    char string[20];

    // Character used for file reading
    // Used as counter

    int pair_prop1[3];
    double pair_prop2[2];
    double pair[MAX_EAM];
    int dens_prop1[2];
    double dens_prop2[2];
    double dens[MAX_EAM];
    int embed_prop1[2];
    double embed_prop2[2];
    double embed[MAX_EAM];

    // Pair potential properties (integers)
    // Pair potential properties (double)
    // Pair potential data
    // Electron density properties (integers)
    // Electron density properties (double)
    // Electron density data
    // Embedding function properties (integers)
    // Embedding function properties (double)
    // Embedding function data

    printf("----- EAM potential file -----\\n");
    infile_eam = fopen(filename_eam, "r"); // Opening file for reading

    if (NULL == infile_eam) // Checking if file exists
        printf("***** ERROR - Cannot open file '%s' for reading, file does not exist.
*****\\n", filename_eam);
    else
    {
        // -----
        // Reading File header
        printf("File header: ");
        c = getc(infile_eam);
        printf("%c", c);
        while(c != 10)
        {
            c = getc(infile_eam);
            printf("%c", c);
        }

        // -----
        // Reading file units
        printf("File units: ");
        fscanf(infile_eam, "%s %s", string, string);
        printf("%s,", string);
        if (strcmp(string, "ev") == 0 || strcmp(string, "eV") == 0 || strcmp(string, "EV")
== 0)
            eunit_type = 1; // type = 1 for eV
        else if (strcmp(string, "K") == 0)
            eunit_type = 2; // type = 2 for Kelvin
        printf(" eunit set to: %i\\n", eunit_type);
        c = getc(infile_eam); // Reading white space

        // -----
        // Reading in data parameters for POTENTIAL PAIR
        fscanf(infile_eam, "%s %s %i %i %i %i %lf %lf", string, string, &pair_prop1[0],
&pair_prop1[1], &pair_prop1[2], &pair_prop2[0], &pair_prop2[1]);
        printf("POTENTIAL PAIR: ");
        printf("%i entries,", pair_prop1[2]);
        if(pair_prop1[2] > MAX_EAM)
            printf("\\n***** ERROR - Number of entries exceeds program maximum
*****\\n");

        printf(" lim: %.2f to %.2f [Angstroms]\\n", pair_prop2[0], pair_prop2[1]);

        // -----
        // Reading in data for POTENTIAL PAIR
        for(i = 0; i < pair_prop1[2]; i++)
            fscanf(infile_eam, "%lf", &pair[i]);

        // -----
        // Reading in data parameters for ELECTRON DENSITY
    }
}

```

```

        fscanf(infile_eam, "%s %s %i %i %lf %lf", string, string, &dens_prop1[0],
&dens_prop1[1], &dens_prop2[0], &dens_prop2[1]);
        printf("ELECTRON DENSITY: ");
        printf("%i entries,", dens_prop1[1]);
        if(dens_prop1[1] > MAX_EAM)
            printf("\n***** ERROR - Number of entries exceeds program maximum
*****\n");
        printf(" lim: %.2f to %.2f [Angstroms]\n", dens_prop2[0], dens_prop2[1]);

        // -----
        // Reading in data for ELECTRON DENSITY
        for(i = 0; i < dens_prop1[1]; i++)
            fscanf(infile_eam, "%lf", &dens[i]);

        // -----
        // Reading in data parameters for EMBEDDING FUNCTION
        fscanf(infile_eam, "%s %s %i %i %lf %lf", string, string, &embed_prop1[0],
&embed_prop1[1], &embed_prop2[0], &embed_prop2[1]);
        printf("EMBEDDING FUNCTION: ");
        printf("%i entries,", embed_prop1[1]);
        if(embed_prop1[1] > MAX_EAM)
            printf("\n***** ERROR - Number of entries exceeds program maximum
*****\n");
        if (eunit_type == 1)
            printf(" lim: %.2f to %.2f\n", embed_prop2[0], embed_prop2[1]);
        else if (eunit_type == 2)
            printf(" lim: %.4e to %.4e\n", embed_prop2[0], embed_prop2[1]);

        // -----
        // Reading in data for EMBEDDING FUNCTION
        for(i = 0; i < embed_prop1[1]; i++)
            fscanf(infile_eam, "%lf", &embed[i]);

        // -----
        // Storing all data into array 'eam'
        for(i = 0; i < pair_prop1[2]; i++)
            eam[i][0] = pair[i];
        for(i = 0; i < dens_prop1[1]; i++)
            eam[i][1] = dens[i];
        for(i = 0; i < embed_prop1[1]; i++)
            eam[i][2] = embed[i];

        for(i = 0; i < 3; i++)
            eam[i][3] = (double)pair_prop1[i];
        for(i = 3; i < 5; i++)
            eam[i][3] = pair_prop2[i-3];

        for(i = 0; i < 2; i++)
            eam[i][4] = (double)dens_prop1[i];
        for(i = 2; i < 4; i++)
            eam[i][4] = dens_prop2[i-2];

        for(i = 0; i < 2; i++)
            eam[i][5] = (double)embed_prop1[i];
        for(i = 2; i < 4; i++)
            eam[i][5] = embed_prop2[i-2];

        fclose(infile_eam);          // Closing file

        printf("----- Finished reading in EAM potential file ----- \n");
    }
}

void calc_md(int input1[], double input2[], double particle_list[][3], double particle_vel[][3],
double eam[][6], int num_pts, int eunit_type, double mass_atom, double kB, char filename_1[],
char filename_4[], char filename_5a[], char filename_5b[], char filename_6[])
{
    int i, j, it;                                // Counters

    double X[MAX_PTS][3];                        // [Angstroms] Position
    double X1[MAX_PTS][3];                       // [m/s] Velocity (1st derivative)
    double X2[MAX_PTS][3];                       // [m/s^2] Acceleration (2nd derivative)

```

```

double X3[MAX_PTS][3];           // 3rd derivative
double X4[MAX_PTS][3];           // 4th derivative
double X5[MAX_PTS][3];           // 5th derivative

double particle_force[MAX_PTS][3]; // [N] Forces
double X2e;                       // [m/s^2] Evaluated accelerations
double X2delta;                   // [m/s^2] Difference between predicted and evaluated accelerations
double R2delta[MAX_PTS][3];       // Correction term

double alfa[6];                   // Gear's Predictor-Corrector parameter
double total_pot = 0;             // Total potential energy
double avg_pot;                   // Average potential energy
double particle_pot[MAX_PTS];

int iter, iter_out;
double dtime, clamp;
double dtime2, dtime3, dtime4, dtime5;

int box_x, box_y, box_z;
int perio_x, perio_y, perio_z;
double strain_x, strain_y, strain_z;
double a_inc, a_step;
double b_inc, b_step;
double c_inc, c_step;
double a0;
double perio_array[6];
int xsave_start, xsave;
int ssave;

double p_pot_array[6];            // Holds potential portion of the stress tensor
double pxx[6];                   // Stress tensor (Divide by volume)

int test_iter_out;
int test_xsave, test_ssave;
char filename_5[MAX_STR];
char it_string[MAX_STR];

double vel_sum_sq;
double avg_kin;                  // Kinetic energy

double bulk;
double a_new;
double vol;
double stress_avg;
double size_factor;
double bsave;
int test_bsave;

int delay_clamp, delay_box;
double cl;

FILE *outfile_1;                 // File pointer for iterations and potential energy output
file
FILE *outfile_4;
FILE *outfile_5;
FILE *outfile_6;

if (eunit_type == 0) // Checking for errors
{
    printf("***** ERROR - No units defined for EAM file *****\n");
    return;
}

iter          = input1[6];        // Extracting data from arrays
iter_out      = input1[7];
xsave_start   = input1[8];
xsave         = input1[9];
ssave         = input1[10];
bsave         = input1[11];
delay_clamp   = input1[12];
delay_box     = input1[13];

```



```

clamp = input2[3];
dtime = input2[4];

alfa[0] = 3.0/16.0;
alfa[1] = 251.0/360.0;
alfa[2] = 1.0;
alfa[3] = 11.0/18.0;
alfa[4] = 1.0/6.0;
alfa[5] = 1.0/60.0;

dtime2 = dtime*dtime;
dtime3 = dtime2*dtime;
dtime4 = dtime3*dtime;
dtime5 = dtime4*dtime;

// -----
box_x   = input1[0];
box_y   = input1[1];
box_z   = input1[2];
perio_x = input1[3];
perio_y = input1[4];
perio_z = input1[5];

a0      = input2[1];
strain_x = input2[5];
strain_y = input2[6];
strain_z = input2[7];

// Applying periodic boundary conditions
if (perio_x == 1)
{
    a_inc = box_x*a0*strain_x;
    a_step = a_inc;
}
else if (perio_x == 0)
{
    a_inc = 0;
    a_step = 1;
}
if (perio_y == 1)
{
    b_inc = box_y*a0*strain_y;
    b_step = b_inc;
}
else if (perio_y == 0)
{
    b_inc = 0;
    b_step = 1;
}
if (perio_z == 1)
{
    c_inc = box_z*a0*strain_z;
    c_step = c_inc;
}
else if (perio_z == 0)
{
    c_inc = 0;
    c_step = 1;
}
perio_array[0] = a_inc;
perio_array[1] = a_step;
perio_array[2] = b_inc;
perio_array[3] = b_step;
perio_array[4] = c_inc;
perio_array[5] = c_step;

// -----
bulk = input2[8];
bulk = bulk*1e9;      // Converting from [GPa] to [Pa]
a_new = a0;

```

```

// -----
printf("\n----- Iterating... -----\n");
printf(" Iter      U(tot)      U(avg)\n");
calc_pot(particle_list, num_pts, eam, particle_pot, total_pot, perio_array, eunit_type);
// Calculating initial total potential energy
avg_pot = total_pot/(double)num_pts;
// -----
// Calculating kinetic energy
vel_sum_sq = 0;
for(i = 0; i < num_pts; i++)
    vel_sum_sq = vel_sum_sq + particle_vel[i][0]*particle_vel[i][0] +
particle_vel[i][1]*particle_vel[i][1] + particle_vel[i][2]*particle_vel[i][2];
avg_kin = 0.5*mass_atom*vel_sum_sq*6.241507649E18/num_pts;
// -----
printf("      0 %11.5f %10.6f %10.6f %10.6f\n", total_pot, avg_pot, avg_kin,
avg_pot+avg_kin); // Outputting initial total potential energy

// Writing to file
outfile_1 = fopen(filename_1, "w"); // Opening file for writing
fprintf(outfile_1, " Iter      U(tot)      U(avg)\n");
fprintf(outfile_1, "%5i %11.5f %10.6f %10.6f %10.6f\n", 0, total_pot, avg_pot, avg_kin,
avg_pot+avg_kin);

outfile_4 = fopen(filename_4, "w"); // Opening file for writing

if (strcmp(filename_6, "none") == 1)
{
    outfile_6 = fopen(filename_6, "w"); // Opening file for writing
    fprintf(outfile_6, "%5i %f\n", 0, a_new);
}

// Initializing variables
for(i = 0; i < num_pts; i++)
    for(j = 0; j < 3; j++)
    {
        X[i][j] = particle_list[i][j]*1e-10; // Converting from [Angstroms] to [m]
        X1[i][j] = particle_vel[i][j];
        X2[i][j] = 0.0;
        X3[i][j] = 0.0;
        X4[i][j] = 0.0;
        X5[i][j] = 0.0;
    }

// Iterating
for(it = 0; it < iter; it++)
{
    if(floor((double)(it+1)/iter_out)*iter_out == it+1)
        test_iter_out = 1;
    else
        test_iter_out = 0;

    // -----
    if((it+1 >= xsave_start) && (floor((double)(it+1)/xsave)*xsave == it+1))
        test_xsave = 1;
    else
        test_xsave = 0;

    // -----
    if(floor((double)(it+1)/ssave)*ssave == it+1)
        test_ssave = 1;
    else
        test_ssave = 0;

    // -----
    if(floor((double)(it+1)/bsave)*bsave == it+1)
        test_bsave = 1;
    else
        test_bsave = 0;

    // -----

```

```

// Predicting new positions
for(i = 0; i < num_pts; i++)
    for(j = 0; j < 3; j++)
    {
        X[i][j] = X[i][j] + X1[i][j]*dttime + X2[i][j]*dttime2/2.0 +
X3[i][j]*dttime3/6.0 + X4[i][j]*dttime4/24.0 + X5[i][j]*dttime5/120.0;
        X1[i][j] = X1[i][j] + X2[i][j]*dttime + X3[i][j]*dttime2/2.0 +
X4[i][j]*dttime3/6.0 + X5[i][j]*dttime4/24.0;
        X2[i][j] = X2[i][j] + X3[i][j]*dttime + X4[i][j]*dttime2/2.0 +
X5[i][j]*dttime3/6.0;
        X3[i][j] = X3[i][j] + X4[i][j]*dttime + X5[i][j]*dttime2/2.0;
        X4[i][j] = X4[i][j] + X5[i][j]*dttime;
    }

// Converting position back to [Angstroms]
for(i = 0; i < num_pts; i++)
    for(j = 0; j < 3; j++)
        particle_list[i][j] = X[i][j]*1e10;

// Calculating forces at new positions
force_calc(particle_force, particle_list, num_pts, eam, eunit_type, perio_array,
p_pot_array, test_ssave);

// Calculating accelerations from forces and calculating correction term
for(i = 0; i < num_pts; i++)
    for(j = 0; j < 3; j++)
    {
        X2e = particle_force[i][j]/mass_atom;
        X2delta = X2e - X2[i][j];
        R2delta[i][j] = X2delta*dttime2/2.0;
    }

// Correcting particle positions and their derivatives
for(i = 0; i < num_pts; i++)
    for(j = 0; j < 3; j++)
    {
        X[i][j] = X[i][j] + alfa[0]*R2delta[i][j];
        X1[i][j] = X1[i][j] + alfa[1]*R2delta[i][j] /dttime;
        X2[i][j] = X2[i][j] + R2delta[i][j]* 2.0/dttime2;
        X3[i][j] = X3[i][j] + alfa[3]*R2delta[i][j]* 6.0/dttime3;
        X4[i][j] = X4[i][j] + alfa[4]*R2delta[i][j]* 24.0/dttime4;
        X5[i][j] = X5[i][j] + alfa[5]*R2delta[i][j]*120.0/dttime5;
    }

if(test_iter_out == 1)
{
    // Calculating potential energy
    for(i = 0; i < num_pts; i++)
        for(j = 0; j < 3; j++)
            particle_list[i][j] = X[i][j]*1e10;

    calc_pot(particle_list, num_pts, eam, particle_pot, total_pot,
perio_array, eunit_type);
    avg_pot = total_pot/(double)num_pts;

// -----

// Calculating kinetic energy
vel_sum_sq = 0;
for(i = 0; i < num_pts; i++)
    vel_sum_sq = vel_sum_sq + X1[i][0]*X1[i][0] + X1[i][1]*X1[i][1] +
X1[i][2]*X1[i][2];
avg_kin = 0.5*mass_atom*vel_sum_sq/num_pts*6.241507649E18; // Ek =
0.5*m*v^2

// -----

printf("%5i %11.5f %10.6f %10.6f %10.6f\n", it+1, total_pot, avg_pot,
avg_kin, avg_pot+avg_kin); // Writing to screen
fprintf(outfile_1, "%5i %11.5f %10.6f %10.6f %10.6f\n", it+1, total_pot,
avg_pot, avg_kin, avg_pot+avg_kin); // Writing to file

```

```

    }

    // -----
    // Calculating microscopic stress tensor
    calc_stress(X1, num_pts, mass_atom, p_pot_array, pxx);
    if(test_ssave == 1) // Calculate stress every iteration but save only every
ssave iterations
        fprintf(outfile_4, "%5i %14.6e %14.6e %14.6e %14.6e %14.6e %14.6e\n",
it+1, pxx[0], pxx[1], pxx[2], pxx[3], pxx[4], pxx[5]); // Writing to file

    // -----
    // Correcting velocities using temperature clamp
    if (clamp != 0)
        scale_vel(X1, clamp, num_pts, mass_atom, kB, delay_clamp);

    // -----
    if(test_xsave == 1)
    {
        itoa(it+1, it_string, 10);
        strcpy(filename_5, filename_5a);
        if (it < 9)
            strcat(filename_5, "0000");
        else if (it < 99)
            strcat(filename_5, "000");
        else if (it < 999)
            strcat(filename_5, "00");
        else if (it < 9999)
            strcat(filename_5, "0");
        strcat(filename_5, it_string);
        strcat(filename_5, ".");
        strcat(filename_5, filename_5b);

        outfile_5 = fopen(filename_5, "w"); // Opening file for writing

        for(i = 0; i < num_pts; i++)
            fprintf(outfile_5, "%4i %8.5f %8.5f %8.5f\n", i+1,
particle_list[i][0], particle_list[i][1], particle_list[i][2]);

        fclose(outfile_5); // Closing iteration and potential energy file
    }

    // -----
    // Periodic BC boundary check, if atom crosses a boundary to the 'outside' of box,
it is moved to the opposite side of the box
    for(i = 0; i < num_pts; i++)
    {
        if ((X[i][0] < 0) && (perio_x == 1))
        {
            X[i][0] = a_new*box_x*strain_x*1e-10 + X[i][0];
            printf("WARNING - Particle beyond crystal limits, < zero in x-
dir\n");
        }
        if ((X[i][1] < 0) && (perio_y == 1))
        {
            X[i][1] = a_new*box_y*strain_y*1e-10 + X[i][1];
            printf("WARNING - Particle beyond crystal limits, < zero in y-
dir\n");
        }
        if ((X[i][2] < 0) && (perio_z == 1))
        {
            X[i][2] = a_new*box_z*strain_z*1e-10 + X[i][2];
            printf("WARNING - Particle beyond crystal limits, < zero in z-
dir\n");
        }
        if ((X[i][0]*1e10 > a_new*box_x*strain_x) && (perio_x == 1))
        {
            X[i][0] = X[i][0] - a_new*box_x*strain_x*1e-10;
            printf("WARNING - Particle beyond crystal limits, > %6.2f [Angs] in
x-dir\n", a_new*box_x*strain_x);
        }
    }

```

```

        if ((X[i][1]*1e10 > a_new*box_y*strain_y) && (perio_y == 1))
        {
            X[i][1] = X[i][1] - a_new*box_y*strain_y*1e-10;
            printf("WARNING - Particle beyond crystal limits, > %6.2f [Angs] in
y-dir\n", a_new*box_y*strain_y);
        }
        if ((X[i][2]*1e10 > a_new*box_z*strain_z) && (perio_z == 1))
        {
            X[i][2] = X[i][2] - a_new*box_z*strain_z*1e-10;
            printf("WARNING - Particle beyond crystal limits, > %6.2f [Angs] in
z-dir\n", a_new*box_z*strain_z);
        }
    }

    // -----
    // Resize box volume size for zero-stress
    if (bulk != 0)
    {
        vol = a_new*a_new*a_new*box_x*box_y*box_z*1e-10*1e-10*1e-10;
            // [m^3] Volume
        stress_avg = (pxx[0]*1e-7/vol + pxx[1]*1e-7/vol + pxx[2]*1e-7/vol)/3; //
[Pa]
        size_factor = exp(stress_avg/(3*bulk));
        cl = 1/(double)delay_box;
        size_factor = pow(size_factor,cl);
        for(i = 0; i < num_pts; i++)
            for(j = 0; j < 3; j++)
                X[i][j] = X[i][j]*size_factor;
        a_new = a_new*size_factor;
        if (test_bsave == 1)
            fprintf(outfile_6, " %5i          %f\n", it+1, a_new);
    }

    // -----
    // Applying periodic boundary conditions
    if (perio_x == 1)
    {
        a_inc = box_x*a_new*strain_x;
        a_step = a_inc;
        perio_array[0] = a_inc;
        perio_array[1] = a_step;
    }
    if (perio_y == 1)
    {
        b_inc = box_y*a_new*strain_y;
        b_step = b_inc;
        perio_array[2] = b_inc;
        perio_array[3] = b_step;
    }
    if (perio_z == 1)
    {
        c_inc = box_z*a_new*strain_z;
        c_step = c_inc;
        perio_array[4] = c_inc;
        perio_array[5] = c_step;
    }
}

// -----
// Updating velocities and converting units for post-processing
for(i = 0; i < num_pts; i++)
    for(j = 0; j < 3; j++)
    {
        particle_list[i][j] = X[i][j]*1e10;
        particle_vel[i][j] = X1[i][j];
    }

printf("\nNumber of iterations performed = %i\n", it);

fclose(outfile_1);    // Closing iteration and potential energy file
fclose(outfile_4);    // Closing iteration and potential energy file

```

```

    if (strcmp(filename_6,"none") == 1)
        fclose(outfile_6);    // Closing iteration and potential energy file
}
void force_calc(double particle_force[][3], double particle_list[][3], int num_pts, double
eam[][6], int eunit_type, double perio_array[], double p_pot_array[], int test_ssave)
{
    int i, j;                // Counters
    double r[MAX_PTS];        // Distance between atoms

    int pair_prop1[3];        // Pair potential properties (integers)
    double pair_prop2[2];     // Pair potential properties (double)
    double pair[MAX_EAM];     // Pair potential data
    int dens_prop1[2];        // Electron density properties (integers)
    double dens_prop2[2];     // Electron density properties (double)
    double dens[MAX_EAM];     // Electron density data
    int embed_prop1[2];       // Embedding function properties (integers)
    double embed_prop2[2];    // Embedding function properties (double)
    double embed[MAX_EAM];    // Embedding function data

    double item;              // Item number in list
    double item_L;
    double item_H;
    double diff_factor;       // Used in numerical differentiation

    double dens_cur;          // Current electron density
    double dens_sum[MAX_PTS]; // Summed electron density
    double dens_delta;        // Difference between max and min density limits
    double dens_diff;         // Used in numerical differentiation
    double dens_cur_diff;     // Used in numerical differentiation
    double dens_temp;
    double pair_m;            // slope
    double pair_b;            // y-intercept

    double pair_cur;          // Current pair potential
    double pair_delta;        // Difference between max and min pair potential limits
    double pair_diff;         // Used in numerical differentiation
    double pair_temp;
    double pair_cur_diff;     // Used in numerical differentiation

    double embed_delta;
    double embed_temp;
    double embed_cur_i;
    double embed_cur_j;
    double embed_cur_diff;
    double embed_diff;
    double embed_m;           // slope
    double embed_b;           // y-intercept

    double force_mag;

    double a_inc, a_step, b_inc, b_step, c_inc, c_step;
    double a, b, c;

    // Defining and initializing potential portion of stress tensor
    double p11_pot = 0;    double p22_pot = 0;    double p33_pot = 0;
    double p23_pot = 0;    double p31_pot = 0;    double p12_pot = 0;

    // Extracting data from array 'eam'
    for(i = 0; i < eam[2][3]; i++)
        pair[i] = eam[i][0];
    for(i = 0; i < eam[1][4]; i++)
        dens[i] = eam[i][1];
    for(i = 0; i < eam[1][5]; i++)
        embed[i] = eam[i][2];

    for(i = 0; i < 3; i++)
        pair_prop1[i] = (int)eam[i][3];
    for(i = 3; i < 5; i++)
        pair_prop2[i-3] = eam[i][3];

    for(i = 0; i < 2; i++)

```

```

        dens_prop1[i] = (int)eam[i][4];
    for(i = 2; i < 4; i++)
        dens_prop2[i-2] = eam[i][4];

    for(i = 0; i < 2; i++)
        embed_prop1[i] = (int)eam[i][5];
    for(i = 2; i < 4; i++)
        embed_prop2[i-2] = eam[i][5];

    // Initializing particle forces to zero
    for(i = 0; i < num_pts; i++)
        for(j = 0; j < 3; j++)
            particle_force[i][j] = 0;

    // -----
    a_inc = perio_array[0];
    a_step = perio_array[1];
    b_inc = perio_array[2];
    b_step = perio_array[3];
    c_inc = perio_array[4];
    c_step = perio_array[5];

    // -----
    diff_factor = 0.5; // Used in numerical differentiation

    // -----
    // Calculating slope and y-intercept used in determining item # in list
    pair_m = (pair_prop1[2]-1)/(pair_prop2[1]-pair_prop2[0]);
    pair_b = 1 - pair_prop2[0]*(pair_prop1[2]-1)/(pair_prop2[1]-pair_prop2[0]);
    // The same slope and y-intercept will apply for electron density list
    embed_m = (embed_prop1[1]-1)/(embed_prop2[1]-embed_prop2[0]);
    embed_b = 1 - embed_prop2[0]*(embed_prop1[1]-1)/(embed_prop2[1]-embed_prop2[0]);

    // -----
    pair_delta = pair_prop2[1] - pair_prop2[0];
    dens_delta = dens_prop2[1] - dens_prop2[0]; // Should be equal to pair_delta
    embed_delta = embed_prop2[1] - embed_prop2[0];
    pair_diff = pair_delta/pair_prop1[2]*diff_factor;
    dens_diff = dens_delta/dens_prop1[1]*diff_factor; // Should be equal to pair_diff
    embed_diff = embed_delta/embed_prop1[1]*diff_factor;

    // Calculating electron density sum for each atom
    for(i = 0; i < num_pts; i++)
    {
        dens_sum[i] = 0;

        for(a = -a_inc; a <= a_inc; a = a + a_step)
        {
            for(b = -b_inc; b <= b_inc; b = b + b_step)
            {
                for(c = -c_inc; c <= c_inc; c = c + c_step)
                {
                    // Calculating the distance between atom i and every other
                    for(j = 0; j < num_pts; j++)
                        r[j] = sqrt((particle_list[i][0]-
                    (particle_list[j][0]+a))*(particle_list[i][0]-(particle_list[j][0]+a))+
                    (particle_list[i][1]-
                    (particle_list[j][1]+b))*(particle_list[i][1]-(particle_list[j][1]+b))+
                    (particle_list[i][2]-
                    (particle_list[j][2]+c))*(particle_list[i][2]-(particle_list[j][2]+c)));

                    for(j = 0; j < num_pts; j++)
                    {
                        if(r[j] == 0)
                        {
                            // DO NOTHING
                        }
                        else if(r[j] < dens_prop2[0])
                        {
                            printf("***** ERROR - Distance between atoms
is too small [1] *****\n");
                            return;
                        }
                    }
                }
            }
        }
    }

```

```

    }
    else if(r[j] > dens_prop2[1])
    {
        // DO NOTHING
    }
    else
    {
        item = pair_m*r[j] + pair_b - 1; //
        item_L = floor(item);
        item_H = ceil(item);
        dens_cur = (item-item_H)/(item_L-item_H) *
(dens[(int)item_L]-dens[(int)item_H]) + dens[(int)item_H]; // Interpolation
        dens_sum[i] = dens_sum[i] + dens_cur;
    }
    // Ending 'j' loop
    // Ending 'c' loop
    // Ending 'b' loop
    // Ending 'a' loop
}
// Ending 'i' loop

// -----
for(i = 0; i < num_pts; i++)
{
    // Calculating embedding function derivative *** FOR 'i' ATOM ***
    item = embed_m*dens_sum[i] + embed_b - 1;
    item_L = floor(item);
    item_H = ceil(item);
    embed_temp = (item-item_H)/(item_L-item_H) * (embed[(int)item_L]-
embed[(int)item_H]) + embed[(int)item_H];

    if(item < embed_prop1[1]-2)
    {
        item = embed_m*(dens_sum[i] + embed_diff) + embed_b - 1;
        item_L = floor(item);
        item_H = ceil(item);
        embed_cur_diff = (item-item_H)/(item_L-item_H) * (embed[(int)item_L]-
embed[(int)item_H]) + embed[(int)item_H];
        embed_cur_i = (embed_cur_diff - embed_temp)/embed_diff;
    }
    else
    {
        item = embed_m*(dens_sum[i] - embed_diff) + embed_b - 1;
        item_L = floor(item);
        item_H = ceil(item);
        embed_cur_diff = (item-item_H)/(item_L-item_H) * (embed[(int)item_L]-
embed[(int)item_H]) + embed[(int)item_H];
        embed_cur_i = (embed_temp - embed_cur_diff)/embed_diff;
    }

    // -----
    for(a = -a_inc; a <= a_inc; a = a + a_step)
    {
        for(b = -b_inc; b <= b_inc; b = b + b_step)
        {
            for(c = -c_inc; c <= c_inc; c = c + c_step)
            {
                // Calculating the distance between atom i and every other
atom
                for(j = 0; j < num_pts; j++)
                    r[j] = sqrt((particle_list[i][0]-
(particle_list[j][0]+a))*(particle_list[i][0]-(particle_list[j][0]+a))+
(particle_list[i][1]-
(particle_list[j][1]+b))*(particle_list[i][1]-(particle_list[j][1]+b))+
(particle_list[i][2]-
(particle_list[j][2]+c))*(particle_list[i][2]-(particle_list[j][2]+c)));

                // -----
                // Starting 'j' loop for all other variables

```



```

for(j = 0; j < num_pts; j++)
{
    if(r[j] == 0)
    {
        // DO NOTHING
    }
    else if(r[j] < pair_prop2[0])
        printf("***** ERROR - Distance between atoms
is too small [2] (NO RETURN) *****\n");
    else if(r[j] > pair_prop2[1])
    {
        // DO NOTHING
    }
    else
    {
        // -----

        // Calculating pair potential derivative
        item = pair_m*r[j] + pair_b - 1;
        item_L = floor(item);
        item_H = ceil(item);
        pair_temp = (item-item_H)/(item_L-item_H) *
(pair[(int)item_L]-pair[(int)item_H]) + pair[(int)item_H];

        if(item < pair_prop1[2]-2)
        {
            item = pair_m*(r[j] + pair_diff) +
pair_b - 1;

            item_L = floor(item);
            item_H = ceil(item);
            pair_cur_diff = (item-
item_H)/(item_L-item_H) * (pair[(int)item_L]-pair[(int)item_H]) + pair[(int)item_H];
            pair_cur = (pair_cur_diff -
pair_temp)/pair_diff;

        }
        else
        {
            item = pair_m*(r[j] - pair_diff) +
pair_b - 1;

            item_L = floor(item);
            item_H = ceil(item);
            pair_cur_diff = (item-
item_H)/(item_L-item_H) * (pair[(int)item_L]-pair[(int)item_H]) + pair[(int)item_H];
            pair_cur = (pair_temp -
pair_cur_diff)/pair_diff;

        }
        // -----

        // Calculating electron density derivative
        item = pair_m*r[j] + pair_b - 1;
        item_L = floor(item);
        item_H = ceil(item);
        dens_temp = (item-item_H)/(item_L-item_H) *
(dens[(int)item_L]-dens[(int)item_H]) + dens[(int)item_H];

        if(item < dens_prop1[1]-2)
        {
            item = pair_m*(r[j] + dens_diff) +
pair_b - 1;

            item_L = floor(item);
            item_H = ceil(item);
            dens_cur_diff = (item-
item_H)/(item_L-item_H) * (dens[(int)item_L]-dens[(int)item_H]) + dens[(int)item_H];
            dens_cur = (dens_cur_diff -
dens_temp)/dens_diff;

        }
        else
        {
            item = pair_m*(r[j] - dens_diff) +
pair_b - 1;

```

```

item_L = floor(item);
item_H = ceil(item);
dens_cur_diff = (item-
dens_cur_diff)/dens_diff;
dens_cur = (dens_temp -
}

// -----
// Calculating embedding function derivative
*** FOR 'j' ATOM ***
item = embed_m*dens_sum[j] + embed_b - 1;
item_L = floor(item);
item_H = ceil(item);
embed_temp = (item-item_H)/(item_L-item_H) *
(embed[(int)item_L]-embed[(int)item_H]) + embed[(int)item_H];

if(item < embed_prop1[1]-2)
{
item = embed_m*(dens_sum[j] +
item_L = floor(item);
item_H = ceil(item);
embed_cur_diff = (item-
embed_cur_j = (embed_cur_diff -
embed_diff) + embed_b - 1;
item_H)/(item_L-item_H) * (embed[(int)item_L]-embed[(int)item_H]) + embed[(int)item_H];
embed_temp)/embed_diff;
}
else
{
item = embed_m*(dens_sum[j] -
item_L = floor(item);
item_H = ceil(item);
embed_cur_diff = (item-
embed_cur_j = (embed_temp -
embed_diff) + embed_b - 1;
item_H)/(item_L-item_H) * (embed[(int)item_L]-embed[(int)item_H]) + embed[(int)item_H];
embed_cur_diff)/embed_diff;
}

// Calculating force on atom 'i' due to atom
'j'
force_mag = (pair_cur + (embed_cur_i +
embed_cur_j)*dens_cur); // No negative sign in front of force equation, since
// pair potential function used in EAM is always positive
// Converting units of force_mag into [J/m] = [N]
if (eunit_type == 1)
force_mag = force_mag * 1.602177e-9;
// Conversion factor for eV to Joule = 1.602177E-19
// [eV/Angstrom] * 1.602177E-19 * 1E10 = [eV/Angstrom] * 1.602177E-9 = [J/m]
else if (eunit_type == 2)
force_mag = force_mag * 1.3806505e-
13; // Conversion factor for K to Joule = 1.3806505e-23
// [K] * 1.3806505e-23[J/K] * 1e10 = [J], (kB = 1.3806505e-23[J/K])

// Calculating force component in each
direction
particle_force[i][0] = particle_force[i][0]
+ force_mag*(particle_list[j][0] + a - particle_list[i][0])/r[j]; // x-component
particle_force[i][1] = particle_force[i][1]
+ force_mag*(particle_list[j][1] + b - particle_list[i][1])/r[j]; // y-component
particle_force[i][2] = particle_force[i][2]
+ force_mag*(particle_list[j][2] + c - particle_list[i][2])/r[j]; // z-component

// Calculation used for obtaining stress
tensor

```

```

// Calculate stress every iteration
p11_pot = p11_pot +
force_mag*(particle_list[j][0] + a - particle_list[i][0])*(particle_list[j][0] + a -
particle_list[i][0])/r[j];
p22_pot = p22_pot +
force_mag*(particle_list[j][1] + b - particle_list[i][1])*(particle_list[j][1] + b -
particle_list[i][1])/r[j];
p33_pot = p33_pot +
force_mag*(particle_list[j][2] + c - particle_list[i][2])*(particle_list[j][2] + c -
particle_list[i][2])/r[j];
p23_pot = p23_pot +
force_mag*(particle_list[j][1] + b - particle_list[i][1])*(particle_list[j][2] + c -
particle_list[i][2])/r[j];
p31_pot = p31_pot +
force_mag*(particle_list[j][2] + c - particle_list[i][2])*(particle_list[j][0] + a -
particle_list[i][0])/r[j];
p12_pot = p12_pot +
force_mag*(particle_list[j][0] + a - particle_list[i][0])*(particle_list[j][1] + b -
particle_list[i][1])/r[j];
} // Ending main 'if' statement
} // Ending 'j' loop
} // Ending 'c' loop
} // Ending 'b' loop
} // Ending 'a' loop

// Storing potential portion of the stress tensor into array
p_pot_array[0] = 0.5*p11_pot*1e-10; // Converting from [N Angstrom] to [N m]
p_pot_array[1] = 0.5*p22_pot*1e-10;
p_pot_array[2] = 0.5*p33_pot*1e-10;
p_pot_array[3] = 0.5*p23_pot*1e-10;
p_pot_array[4] = 0.5*p31_pot*1e-10;
p_pot_array[5] = 0.5*p12_pot*1e-10;
}

void scale_vel(double particle_vel[][3], double clamp, int num_pts, double mass_atom, double kB,
int delay_clamp)
{
// Scalling velocities to temperature defined by 'clamp'

int i;
double vel_sum_sq = 0;
double KE_temp;
double T_temp;
double factor;
double c1;
double c2;

// Scalling velocities for desired temperatire itemp
for(i = 0; i < num_pts; i++)
vel_sum_sq = vel_sum_sq + particle_vel[i][0]*particle_vel[i][0] +
particle_vel[i][1]*particle_vel[i][1] + particle_vel[i][2]*particle_vel[i][2];
KE_temp = 0.5*mass_atom*vel_sum_sq;
T_temp = 2*KE_temp/(3*num_pts*kB);

// New factor reduction
c1 = sqrt(clamp/T_temp);
c2 = 1/(double)delay_clamp; // Reduction factor of 1/delay_clamp, meaning that the
velocity would return to the desired temperature by the specified iteration
factor = pow(c1,c2);

for(i = 0; i < num_pts; i++)
{
particle_vel[i][0] = particle_vel[i][0]*factor;
particle_vel[i][1] = particle_vel[i][1]*factor;
particle_vel[i][2] = particle_vel[i][2]*factor;
}
}

void calc_pot(double particle_list[][3], int num_pts, double eam[][6], double particle_pot[],
double &total_pot, double perio_array[], int eunit_type)

```

```

{
    // Calculating potential energy

    int i, j;                // Counters
    double r[MAX_PTS];       // Distance between atoms

    int pair_prop1[3];       // Pair potential properties (integers)
    double pair_prop2[2];    // Pair potential properties (double)
    double pair[MAX_EAM];    // Pair potential data
    int dens_prop1[2];       // Electron density properties (integers)
    double dens_prop2[2];    // Electron density properties (double)
    double dens[MAX_EAM];    // Electron density data
    int embed_prop1[2];      // Embedding function properties (integers)
    double embed_prop2[2];   // Embedding function properties (double)
    double embed[MAX_EAM];   // Embedding function data

    double item;             // Item number in list
    double item_L;
    double item_H;

    double pair_m;           // slope
    double pair_b;           // y-intercept
    double pair_sum[MAX_PTS];
    double pair_cur;
    double pair_delta;       // Difference between max and min pair potential limits

    double dens_sum[MAX_PTS];
    double dens_cur;
    double dens_delta;       // Difference between max and min electron density limits

    double embed_m;          // slope
    double embed_b;          // y-intercept
    double embed_cur;

    double a_inc, a_step;
    double b_inc, b_step;
    double c_inc, c_step;
    double a, b, c;

    // Extracting data from array 'eam'
    for(i = 0; i < eam[2][3]; i++)
        pair[i] = eam[i][0];
    for(i = 0; i < eam[1][4]; i++)
        dens[i] = eam[i][1];
    for(i = 0; i < eam[1][5]; i++)
        embed[i] = eam[i][2];

    for(i = 0; i < 3; i++)
        pair_prop1[i] = (int)eam[i][3];
    for(i = 3; i < 5; i++)
        pair_prop2[i-3] = eam[i][3];

    for(i = 0; i < 2; i++)
        dens_prop1[i] = (int)eam[i][4];
    for(i = 2; i < 4; i++)
        dens_prop2[i-2] = eam[i][4];

    for(i = 0; i < 2; i++)
        embed_prop1[i] = (int)eam[i][5];
    for(i = 2; i < 4; i++)
        embed_prop2[i-2] = eam[i][5];

    // -----
    a_inc = perio_array[0];
    a_step = perio_array[1];
    b_inc = perio_array[2];
    b_step = perio_array[3];
    c_inc = perio_array[4];
    c_step = perio_array[5];

    // -----

```

```

// Calculating slope and y-intercept used in determining item # in list
pair_m = (pair_prop1[2]-1)/(pair_prop2[1]-pair_prop2[0]);
pair_b = 1 - pair_prop2[0]*(pair_prop1[2]-1)/(pair_prop2[1]-pair_prop2[0]);
// The same slope and y-intercept will apply for electron density list
embed_m = (embed_prop1[1]-1)/(embed_prop2[1]-embed_prop2[0]);
embed_b = 1 - embed_prop2[0]*(embed_prop1[1]-1)/(embed_prop2[1]-embed_prop2[0]);

// -----
// Calculating pair potential and electron density sum for each atom
pair_delta = pair_prop2[1] - pair_prop2[0];
dens_delta = dens_prop2[1] - dens_prop2[0];
for(i = 0; i < num_pts; i++)
{
    dens_sum[i] = 0;        // Initilizing sums
    pair_sum[i] = 0;

    for(a = -a_inc; a <= a_inc; a = a + a_step)
    {
        for(b = -b_inc; b <= b_inc; b = b + b_step)
        {
            for(c = -c_inc; c <= c_inc; c = c + c_step)
            {
                // Calculating the distance between atom i and every other
                for(j = 0; j < num_pts; j++)
                {
                    r[j] = sqrt((particle_list[i][0]-
                    (particle_list[j][0]+a))*(particle_list[i][0]-(particle_list[j][0]+a))+
                    (particle_list[i][1]-
                    (particle_list[j][1]+b))*(particle_list[i][1]-(particle_list[j][1]+b))+
                    (particle_list[i][2]-
                    (particle_list[j][2]+c))*(particle_list[i][2]-(particle_list[j][2]+c)));

                    for(j = 0; j < num_pts; j++)
                    {
                        if(r[j] == 0)
                        {
                            // DO NOTHING
                        }
                        else if(r[j] < pair_prop2[0])
                        {
                            printf("***** ERROR - Distance between atoms
is too small [4] *****\n");
                            return;
                        }
                        else if(r[j] > pair_prop2[1])
                        {
                            // DO NOTHING
                        }
                        else
                        {
                            // Pair potential calculation
                            item = pair_m*r[j] + pair_b - 1;        //

                            // Interpolation
                            item_L = floor(item);
                            item_H = ceil(item);
                            pair_cur = (item-item_H)/(item_L-item_H) *
(pair[(int)item_L]-pair[(int)item_H]) + pair[(int)item_H];

                            // Electron density calculation (using same
                            dens_cur = (item-item_H)/(item_L-item_H) *
(dens[(int)item_L]-dens[(int)item_H]) + dens[(int)item_H];

                            pair_sum[i] = pair_sum[i] + pair_cur;
                            dens_sum[i] = dens_sum[i] + dens_cur;
                        }
                    }
                }
            }
        }
    }
}

```

```

// Calculating embedding function and atom potential energy
total_pot = 0;
for(i = 0; i < num_pts; i++)
{
    if(dens_sum[i] < embed_prop2[0])
        printf("***** ERROR - Electron density sum below embedding function limit.
*****\n");
    else if(dens_sum[i] > embed_prop2[1])
        printf("***** ERROR - Electron density sum above embedding function limit.
*****\n");
    else
    {
        // Embedding function calculation
        item = embed_m*dens_sum[i] + embed_b - 1;    // Calculating which item in
list
        // Interpolation
        item_L = floor(item);
        item_H = ceil(item);
        embed_cur = (item-item_H)/(item_L-item_H) * (embed[(int)item_L]-
embed[(int)item_H]) + embed[(int)item_H];

        // Potential energy calculation
        particle_pot[i] = 0.5*pair_sum[i] + embed_cur;    // ADDED 0.5 to first term
    }
    total_pot = total_pot + particle_pot[i];
}

// Converting potential energy units into eV
if (eunit_type == 1)
{
    // DO NOTHING - already in eV
}
else if (eunit_type == 2)
{
    total_pot = total_pot*8.617339E-5;
}
}

void calc_stress(double particle_vel[][3], int num_pts, double mass_atom, double p_pot_array[],
double pxx[])
{
    int i;

    double p_kin_temp;
    double p11_kin = 0;
    double p22_kin = 0;
    double p33_kin = 0;
    double p23_kin = 0;
    double p31_kin = 0;
    double p12_kin = 0;

    double p11_pot = p_pot_array[0];    // [J] = [N*m]
    double p22_pot = p_pot_array[1];
    double p33_pot = p_pot_array[2];
    double p23_pot = p_pot_array[3];
    double p31_pot = p_pot_array[4];
    double p12_pot = p_pot_array[5];

    // Calculating kinetic portion of the stress tensor
    for (i = 0; i < num_pts; i++)
    {
        p_kin_temp = particle_vel[i][0]*particle_vel[i][0];
        p11_kin = p11_kin + p_kin_temp;

        p_kin_temp = particle_vel[i][1]*particle_vel[i][1];
        p22_kin = p22_kin + p_kin_temp;

        p_kin_temp = particle_vel[i][2]*particle_vel[i][2];
        p33_kin = p33_kin + p_kin_temp;
    }
}

```

```

        p_kin_temp = particle_vel[i][1]*particle_vel[i][2];
        p23_kin = p23_kin + p_kin_temp;

        p_kin_temp = particle_vel[i][2]*particle_vel[i][0];
        p31_kin = p31_kin + p_kin_temp;

        p_kin_temp = particle_vel[i][0]*particle_vel[i][1];
        p12_kin = p12_kin + p_kin_temp;
    }
    p11_kin = p11_kin * mass_atom;          // [m/s]^2*[kg] = [N*m] = [J]
    p22_kin = p22_kin * mass_atom;
    p33_kin = p33_kin * mass_atom;
    p23_kin = p23_kin * mass_atom;
    p31_kin = p31_kin * mass_atom;
    p12_kin = p12_kin * mass_atom;

    // Calculating stress tensor
    pxx[0] = (p11_kin - p11_pot)*1e7;        // [J]*1e7 = [N*m]*1e7 = [erg]
    pxx[1] = (p22_kin - p22_pot)*1e7;
    pxx[2] = (p33_kin - p33_pot)*1e7;
    pxx[3] = (p23_kin - p23_pot)*1e7;
    pxx[4] = (p31_kin - p31_pot)*1e7;
    pxx[5] = (p12_kin - p12_pot)*1e7;

    //printf("p11_kin = %e, p11_pot = %e, pxx[0] = %e\n", p11_kin*1e7, p11_pot*1e7, pxx[0]);
}

```

G.2 Source code for MDROB_BORN.c

```
// MDROB_BORN - MD code used to calculate Born terms
// Molecular Dynamics Simulation
// Developed by: Robert Szydlowski
// Last updated: May 19, 2006

// ASSUMPTIONS:
// EAM file: Number of entries in EAM data for PAIR POTENTIAL and DENS POTENTIAL is identical

#include <stdio.h>
#include <stdafx.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>      // Used for random number generation, seed is set according to time at
                        // execution

#define MAX_PTS 1372      // Make sure MAX_PTS is an even number (for max_vel_pts)
#define MAX_STR 20        // Maximum number of characters in string
#define MAX_EAM 3000
#define MAX_FILES 1000    // Max number of files to be read

void read_input(char* argv[], int input1[], double input2[], char filename_eam[], char
filename_1[], char filename_2[], char filename_3[], char filename_4[], char filename_5a[], char
filename_5b[], int &test_break, char filename_6[]);
void read_const(double input3[]);
void read_eam(double eam[][6], char filename_eam[], int &eunit_type);
void read_pos(int input1[], int num_pts, double particle_list[][3], char filename_5a[], char
filename_5b[], int it);
void calc_B(int input1[], double input2[], int num_pts, double particle_list[][3], double
born_terms[], double eam[][6]);
void spline_interp(double x[], double y[], int num_eam_pts, double coef_array[][4]);
double spline_interp2(double item, double dens_sum, double x_embed[], double embed_coef[][4]);

int main(int argc, char* argv[])
{
    int input1[14];
    int box_x, box_y, box_z;
    int perio_x, perio_y, perio_z;
    int iter;
    int it, i;
    double input2[9], input3[6];
    double a0;
    double strain_x, strain_y, strain_z;
    double kB, pi, amu;
    int xsave_start, xsave;

    // Variables used in input2
    // Variables used in input2
    // Variables used in input3
    // Variables used in input1

    int num_pts;
    char filename_eam[MAX_STR];
    int eunit_type;

    // Number of atoms being simulated
    // EAM filename
    // Energy units used in EAM file

    char filename_1[MAX_STR];
    char filename_2[MAX_STR];
    char filename_3[MAX_STR];
    // Filename
    // Filename of final positions and velocities file
    // Filename of initial positions and velocities file
    (at perfect lattice positions)
    char filename_4[MAX_STR];
    char filename_5a[MAX_STR];
    char filename_5b[MAX_STR];
    char filename_6[MAX_STR];

    double particle_list[MAX_PTS][3];
    double eam[MAX_EAM][6];

    int test_break = 0;
    int test_xsave;
```



```

double born_terms[9];
double conv;

FILE *output_file;

printf("### MDROB ###\nCRYSTAL PARAMETERS\n");

if (argc == 1)
{
    printf("***** ERROR - No specified input file *****\n");
    return 0;
}
else if (argc > 2)
{
    printf("***** ERROR - Too many input parameters *****\n");
    printf("  format: mdrob [inputfile]\n");
    return 0;
}

// Reading in input variables
read_input(argv, input1, input2, filename_eam, filename_1, filename_2, filename_3,
filename_4, filename_5a, filename_5b, test_break, filename_6);

if (test_break == 1) // Checking for errors in input parameters
    return 0;

box_x   = input1[0];
box_y   = input1[1];
box_z   = input1[2];
perio_x = input1[3];
perio_y = input1[4];
perio_z = input1[5];
iter    = input1[6];
xsave_start = input1[8];
xsave   = input1[9];

a0 = input2[1];
strain_x = input2[5];
strain_y = input2[6];
strain_z = input2[7];

// Printing input parameters
printf(" a0 = %.3f [Angstroms]\n", a0);
printf(" box = %i %i %i\n", box_x, box_y, box_z);
printf("SIMULATION PARAMETERS\n");
printf(" iter = %i\n", iter);
printf(" strain = %7.4f %7.4f %7.4f\n", strain_x, strain_y, strain_z);
printf(" perio = %i %i %i\n", perio_x, perio_y, perio_z);
printf(" xsave = %i %i\n", xsave_start, xsave);

printf("EAM filename: %s\n", filename_eam);
printf("Instantaneous particle positions: %s %s\n", filename_5a, filename_5b);
printf("\n");

// Reading in simulation constants
read_const(input3);
kB = input3[0];
pi = input3[1];
amu = input3[2];

// Number of atoms
num_pts = box_x*box_y*box_z*4;

// Read EAM potential file
eunit_type = 0; // Initially setting to zero
read_eam(eam, filename_eam, eunit_type);

// Iterate through the saved particle position files
output_file = fopen("out_born_terms.txt", "w");
for(it = 0; it < iter; it++)

```

```

{
    // Testing if file was written for particular iteration
    if((it+1 >= xsave_start) && (floor((double)(it+1)/xsave)*xsave == it+1))
        test_xsave = 1;
    else
        test_xsave = 0;

    if(test_xsave == 1)
    {
        // Read in particle positions from file
        read_pos(input1, num_pts, particle_list, filename_5a, filename_5b, it);

        // Calculate the three Born terms for each particle list
        calc_B(input1, input2, num_pts, particle_list, born_terms, eam);

        if (eunit_type == 2) // If EAM units are not [eV], then convert to it
        {
            conv = 1.3806505e-23/1.602177e-19;
            for(i = 0; i < 9; i++)
                born_terms[i] = born_terms[i]*conv;
        }

        fprintf(output_file, "%10.4f %10.4f %10.4f %10.4f %10.4f %10.4f %10.4f %10.4f\n",
            born_terms[0], born_terms[1], born_terms[2], born_terms[3], born_terms[4],
            born_terms[5], born_terms[6], born_terms[7], born_terms[8]);
    }
    fclose(output_file);

    // -----
    // Ending program
    printf("\n----- Calculation FINISHED ----- \n");
    return 0;
}

void read_input(char* argv[], int input1[], double input2[], char filename_eam[], char
filename_1[], char filename_2[], char filename_3[], char filename_4[], char filename_5a[], char
filename_5b[], int &test_break, char filename_6[])
{
    // Reading in input variables

    int box_x, box_y, box_z;
    int perio_x, perio_y, perio_z;
    int iter, iter_out;
    double mass, a0, itemp, clamp, dtime;
    double strain_x, strain_y, strain_z;
    char string[20];
    char string2[20];
    double num1, num2, num3;
    int xsave_start, xsave;
    int ssave;
    double bulk; // Only used in zero-pressure simulation (variable volume)
    int bsave; // Only used in zero-pressure simulation (variable volume)
    int delay_clamp, delay_box;

    // -----
    // Setting inputs that are not always used
    bulk = 0;
    bsave = 0;
    strcpy(filename_6, "none");
    delay_clamp = 33; // Number of iterations, set to 33 iterations as
default
    delay_box = 33; // Number of iterations, set to 33 iterations as
default

    FILE *infile_input;

    infile_input = fopen(argv[1], "r"); // Opening file for reading

    if (NULL == infile_input) // Checking if file exists
    {

```

```

        printf("***** ERROR - Cannot open file '%s' for reading, file does not exist.
*****\n", argv[1]);
        test_break = 1;
        return;
    }
    else
    {
        fscanf(infile_input, "%s", string);          // Read in first string in line

        while (feof(infile_input) == 0)
        {
            if (string[0] != '#')
            {
                if (strcmp(string, "iter") == 0)          // Number of
iterations
                {
                    fscanf(infile_input, "%lf", &num1);
                    iter = (int)num1;
                }
                else if (strcmp(string, "iter_out") == 0)  // Output to be
written to file every 'iter_out' iterations
                {
                    fscanf(infile_input, "%lf", &num1);
                    iter_out = (int)num1;
                }
                else if (strcmp(string, "mass") == 0)      // Atomic mass of atom.
                {
                    fscanf(infile_input, "%lf", &num1);
                    mass = num1;
                }
                else if (strcmp(string, "lattice") == 0)   // Lattice constant
size
                {
                    fscanf(infile_input, "%lf", &num1);
                    a0 = num1;
                }
                else if (strcmp(string, "itemp") == 0)     // [K] Initialization
temperature
                {
                    fscanf(infile_input, "%lf", &num1);
                    itemp = num1;
                }
                else if (strcmp(string, "clamp") == 0)     // [K] Clamp
temperature
                {
                    fscanf(infile_input, "%lf", &num1);
                    clamp = num1;
                }
                else if (strcmp(string, "dtime") == 0)     // [s] Time-step size
                {
                    fscanf(infile_input, "%lf", &num1);
                    dtime = num1;
                }
                else if (strcmp(string, "box") == 0)
                {
                    fscanf(infile_input, "%lf %lf %lf", &num1, &num2, &num3);
                    box_x = (int)num1;
                    box_y = (int)num2;
                    box_z = (int)num3;
                }
                else if (strcmp(string, "perio") == 0)
                {
                    fscanf(infile_input, "%lf %lf %lf", &num1, &num2, &num3);
                    perio_x = (int)num1;
                    perio_y = (int)num2;
                    perio_z = (int)num3;
                }
                else if (strcmp(string, "strain") == 0)
                {
                    fscanf(infile_input, "%lf %lf %lf", &num1, &num2, &num3);
                    strain_x = num1;

```

```

        strain_y = num2;
        strain_z = num3;
    }
    else if (strcmp(string, "xsave") == 0)
    {
        fscanf(infile_input, "%lf %lf %s %s", &num1, &num2, string,
string2);

        xsave_start = (int)num1;
        xsave = (int)num2;
        strcpy(filename_5a, string);
        strcpy(filename_5b, string2);
    }
    else if (strcmp(string, "file_eam") == 0)    // EAM potential file
name
    {
        fscanf(infile_input, "%s", string);
        strcpy(filename_eam, string);
    }
    else if (strcmp(string, "file_iter") == 0)    // Iterations and
potential energy output file name
    {
        fscanf(infile_input, "%s", string);
        strcpy(filename_1, string);
    }
    else if (strcmp(string, "file_part_fin") == 0)    // Final
particle positions and velocities
    {
        fscanf(infile_input, "%s", string);
        strcpy(filename_2, string);
    }
    else if (strcmp(string, "file_part_ini") == 0)    // Final
particle positions and velocities
    {
        fscanf(infile_input, "%s", string);
        strcpy(filename_3, string);
    }
    else if (strcmp(string, "ssave") == 0)
    {
        fscanf(infile_input, "%lf %s", &num1, string);
        ssave = (int)num1;
        strcpy(filename_4, string);
    }
    else if (strcmp(string, "bulk") == 0)
    {
        fscanf(infile_input, "%lf", &num1);
        bulk = num1;
    }
    else if (strcmp(string, "bsave") == 0)
    {
        fscanf(infile_input, "%lf %s", &num1, string);
        bsave = (int)num1;
        strcpy(filename_6, string);
    }
    else if (strcmp(string, "delay_clamp") == 0)
    {
        fscanf(infile_input, "%lf", &num1);
        delay_clamp = (int)num1;
    }
    else if (strcmp(string, "delay_box") == 0)
    {
        fscanf(infile_input, "%lf", &num1);
        delay_box = (int)num1;
    }
    else
    {
        printf("***** ERROR - Unknown command in input file
*****\n");

        test_break = 1;
        return;
    }
}

```

```

    }

    fscanf(infile_input, "%s", string);
}

fclose(infile_input); // Closing file

//Putting all input data into two arrays
input1[0] = box_x;
input1[1] = box_y;
input1[2] = box_z;
input1[3] = perio_x;
input1[4] = perio_y;
input1[5] = perio_z;
input1[6] = iter;
input1[7] = iter_out;
input1[8] = xsave_start;
input1[9] = xsave;
input1[10] = ssave;

input2[0] = mass;
input2[1] = a0;
input2[2] = itemp;
input2[3] = clamp;
input2[4] = dtime;
input2[5] = strain_x;
input2[6] = strain_y;
input2[7] = strain_z;
}

void read_const(double input3[])
{
    // Reading in simulation constants

    double kB = 1.3806505e-23;    // [J/K] Boltzmann's constant
    double pi = 3.14159;
    double amu = 1.66054e-24;    // [grams] Atomic mass unit

    input3[0] = kB;
    input3[1] = pi;
    input3[2] = amu;
}

void read_eam(double eam[][6], char filename_eam[], int &eunit_type)
{
    // Reading in eam
    FILE *infile_eam;
    int c;                // Character used for file reading
    int i;                // Used as counter
    char string[20];

    int pair_prop1[3];    // Pair potential properties (integers)
    double pair_prop2[2]; // Pair potential properties (double)
    double pair[MAX_EAM]; // Pair potential data
    int dens_prop1[2];    // Electron density properties (integers)
    double dens_prop2[2]; // Electron density properties (double)
    double dens[MAX_EAM]; // Electron density data
    int embed_prop1[2];   // Embedding function properties (integers)
    double embed_prop2[2]; // Embedding function properties (double)
    double embed[MAX_EAM]; // Embedding function data

    printf("----- EAM potential file -----\\n");
    infile_eam = fopen(filename_eam, "r");    // Opening file for reading

    if (NULL == infile_eam)                    // Checking if file exists
        printf("***** ERROR - Cannot open file '%s' for reading, file does not exist.
*****\\n", filename_eam);
    else
    {

```

```

// -----
// Reading File header
printf("File header: ");
c = getc(infile_eam);
printf("%c", c);
while(c != 10)
{
    c = getc(infile_eam);
    printf("%c", c);
}

// -----
// Reading file units
printf("File units: ");
fscanf(infile_eam, "%s %s", string, string);
printf("%s,", string);
if (strcmp(string, "ev") == 0 || strcmp(string, "eV") == 0 || strcmp(string, "EV")
== 0)
    eunit_type = 1; // type = 1 for eV
if (strcmp(string, "K") == 0 )
    eunit_type = 2; // type = 2 for K
printf(" eunit set to: %i\n", eunit_type);
c = getc(infile_eam); // Reading white space

// -----
// Reading in data parameters for POTENTIAL PAIR
fscanf(infile_eam, "%s %s %i %i %i %lf %lf", string, string, &pair_prop1[0],
&pair_prop1[1], &pair_prop1[2], &pair_prop2[0], &pair_prop2[1]);
printf("POTENTIAL PAIR: ");
printf("%i entries,", pair_prop1[2]);
if(pair_prop1[2] > MAX_EAM)
    printf("\n***** ERROR - Number of entries exceeds program maximum
*****\n");
printf(" lim: %.2f to %.2f [Angstroms]\n", pair_prop2[0], pair_prop2[1]);

// -----
// Reading in data for POTENTIAL PAIR
for(i = 0; i < pair_prop1[2]/3; i++)
    fscanf(infile_eam, "%lf %lf %lf", &pair[i*3], &pair[i*3+1], &pair[i*3+2]);

// -----
// Reading in data parameters for ELECTRON DENSITY
fscanf(infile_eam, "%s %s %i %i %i %lf %lf", string, string, &dens_prop1[0],
&dens_prop1[1], &dens_prop2[0], &dens_prop2[1]);
printf("ELECTRON DENSITY: ");
printf("%i entries,", dens_prop1[1]);
if(dens_prop1[1] > MAX_EAM)
    printf("\n***** ERROR - Number of entries exceeds program maximum
*****\n");
printf(" lim: %.2f to %.2f [Angstroms]\n", dens_prop2[0], dens_prop2[1]);

// -----
// Reading in data for ELECTRON DENSITY
for(i = 0; i < dens_prop1[1]/3; i++)
    fscanf(infile_eam, "%lf %lf %lf", &dens[i*3], &dens[i*3+1], &dens[i*3+2]);

// -----
// Reading in data parameters for EMBEDDING FUNCTION
fscanf(infile_eam, "%s %s %i %i %i %lf %lf", string, string, &embed_prop1[0],
&embed_prop1[1], &embed_prop2[0], &embed_prop2[1]);
printf("EMBEDDING FUNCTION: ");
printf("%i entries,", embed_prop1[1]);
if(embed_prop1[1] > MAX_EAM)
    printf("\n***** ERROR - Number of entries exceeds program maximum
*****\n");
printf(" lim: %.2f to %.2f\n", embed_prop2[0], embed_prop2[1]);

// -----
// Reading in data for EMBEDDING FUNCTION
for(i = 0; i < embed_prop1[1]; i++)
    fscanf(infile_eam, "%lf", &embed[i]);

```

```

// -----
// Storing all data into array 'eam'
for(i = 0; i < pair_prop1[2]; i++)
    eam[i][0] = pair[i];
for(i = 0; i < dens_prop1[1]; i++)
    eam[i][1] = dens[i];
for(i = 0; i < embed_prop1[1]; i++)
    eam[i][2] = embed[i];

for(i = 0; i < 3; i++)
    eam[i][3] = (double)pair_prop1[i];
for(i = 3; i < 5; i++)
    eam[i][3] = pair_prop2[i-3];

for(i = 0; i < 2; i++)
    eam[i][4] = (double)dens_prop1[i];
for(i = 2; i < 4; i++)
    eam[i][4] = dens_prop2[i-2];

for(i = 0; i < 2; i++)
    eam[i][5] = (double)embed_prop1[i];
for(i = 2; i < 4; i++)
    eam[i][5] = embed_prop2[i-2];

fclose(infile_eam);          // Closing file

printf("----- Finished reading in EAM potential file -----\\n");
}

void read_pos(int input1[], int num_pts, double particle_list[][3], char filename_5a[], char
filename_5b[], int it)
{
    int i;
    int i_temp;
    char filename_5[MAX_STR];
    char it_string[MAX_STR];

    FILE *infile_5;

    // Create file name to be open
    itoa(it+1, it_string, 10);
    strcpy(filename_5, filename_5a);
    if (it < 9)
        strcat(filename_5, "0000");
    else if (it < 99)
        strcat(filename_5, "000");
    else if (it < 999)
        strcat(filename_5, "00");
    else if (it < 9999)
        strcat(filename_5, "0");
    strcat(filename_5, it_string);
    strcat(filename_5, ".");
    strcat(filename_5, filename_5b);

    printf("filename = %s\\n", filename_5);

    infile_5 = fopen(filename_5, "r");    // Opening file for reading

    for(i = 0; i < num_pts; i++)
        fscanf(infile_5, "%i %lf %lf %lf", &i_temp, &particle_list[i][0],
&particle_list[i][1], &particle_list[i][2]);

    fclose(infile_5);    // Closing iteration and potential energy file
}

void calc_B(int input1[], double input2[], int num_pts, double particle_list[][3], double
born_terms[], double eam[][6])
{
    int i, j;
    double r[MAX_PTS];          // Distance between atoms

```

```

int box_x, box_y, box_z;
int perio_x, perio_y, perio_z;
double strain_x, strain_y, strain_z;
double a0;

double a_inc, a_step, b_inc, b_step, c_inc, c_step;
double a, b, c;

double item;                // Item number in list
double item_L;
double item_H;

int pair_prop1[3];          // Pair potential properties (integers)
double pair_prop2[2];       // Pair potential properties (double)
double pair[MAX_EAM];       // Pair potential data
int dens_prop1[2];          // Electron density properties (integers)
double dens_prop2[2];       // Electron density properties (double)
double dens[MAX_EAM];       // Electron density data
int embed_prop1[2];         // Embedding function properties (integers)
double embed_prop2[2];      // Embedding function properties (double)
double embed[MAX_EAM];      // Embedding function data

double pair_m;              // slope
double pair_b;              // y-intercept
double embed_m;             // slope
double embed_b;            // y-intercept

double diff_factor;

double pair_delta;
double pair_diff;
double x_pair[MAX_EAM];
double pair_coef[MAX_EAM][4];
double pair_temp;
double pair_cur_diff;
double pair_cur_diff2;
double pair_1der;
double pair_2der;

double dens_delta;
double dens_diff;
double dens_coef[MAX_EAM][4];
double dens_cur;           // Current electron density
double dens_sum[MAX_PTS];  // Summed electron density
double dens_temp;
double dens_cur_diff;
double dens_cur_diff2;
double dens_1der;
double dens_2der;

double x_embed[MAX_EAM];   // Array used to hold 'x' values for embedding function
double embed_temp;
double embed_coef[MAX_EAM][4];
double embed_delta;
double embed_diff;
double embed_cur_diff;
double embed_cur_diff2;
double F_1der[MAX_PTS];
double F_2der[MAX_PTS];
double dens_sum_temp;

double r_temp;

double B1_1111 = 0, B2_1111 = 0, B1_2323 = 0;
double B1_1122 = 0, B2_1122 = 0, B2_2323 = 0;
double B3_1111 = 0, B3_1122 = 0, B3_2323 = 0;
double g11, g22, g23;

double const1, const2, const3, const4, const5;

```



```

// -----
box_x   = input1[0];
box_y   = input1[1];
box_z   = input1[2];
perio_x = input1[3];
perio_y = input1[4];
perio_z = input1[5];

a0      = input2[1];
strain_x = input2[5];
strain_y = input2[6];
strain_z = input2[7];

// -----
// Extracting data from array 'eam'
for(i = 0; i < eam[2][3]; i++)
    pair[i] = eam[i][0];
for(i = 0; i < eam[1][4]; i++)
    dens[i] = eam[i][1];
for(i = 0; i < eam[1][5]; i++)
    embed[i] = eam[i][2];

for(i = 0; i < 3; i++)
    pair_prop1[i] = (int)eam[i][3];
for(i = 3; i < 5; i++)
    pair_prop2[i-3] = eam[i][3];

for(i = 0; i < 2; i++)
    dens_prop1[i] = (int)eam[i][4];
for(i = 2; i < 4; i++)
    dens_prop2[i-2] = eam[i][4];

for(i = 0; i < 2; i++)
    embed_prop1[i] = (int)eam[i][5];
for(i = 2; i < 4; i++)
    embed_prop2[i-2] = eam[i][5];

// -----
// Applying periodic boundary conditions
if (perio_x == 1)
{
    a_inc = box_x*a0*strain_x;
    a_step = a_inc;
}
else if (perio_x == 0)
{
    a_inc = 0;
    a_step = 1;
}
if (perio_y == 1)
{
    b_inc = box_y*a0*strain_y;
    b_step = b_inc;
}
else if (perio_y == 0)
{
    b_inc = 0;
    b_step = 1;
}
if (perio_z == 1)
{
    c_inc = box_z*a0*strain_z;
    c_step = c_inc;
}
else if (perio_z == 0)
{
    c_inc = 0;
    c_step = 1;
}

// -----

```

```

// Calculating slope and y-intercept used in determining item # in list
pair_m = (pair_prop1[2]-1)/(pair_prop2[1]-pair_prop2[0]);
pair_b = 1 - pair_prop2[0]*(pair_prop1[2]-1)/(pair_prop2[1]-pair_prop2[0]);
// The same slope and y-intercept will apply for electron density list
embed_m = (embed_prop1[1]-1)/(embed_prop2[1]-embed_prop2[0]);
embed_b = 1 - embed_prop2[0]*(embed_prop1[1]-1)/(embed_prop2[1]-embed_prop2[0]);

// -----
// Creating array holding 'x' values for pair potential function
x_pair[0] = pair_prop2[0];
for (i = 1; i < pair_prop1[2]; i++)
    x_pair[i] = (x_pair[i-1] + (pair_prop2[1] - pair_prop2[0])/(pair_prop1[2]-1));
// Cubic spline interpolation of EAM lists
spline_interp(x_pair, pair, pair_prop1[2], pair_coef);

// -----
// Creating array holding 'x' values for electron density function
// Using x_pair since x_pair = x_dens
// Cubic spline interpolation of EAM lists

spline_interp(x_pair, dens, dens_prop1[1], dens_coef);

// -----
// Creating array holding 'x' values for embedding function
x_embed[0] = embed_prop2[0];
for (i = 1; i < embed_prop1[1]; i++)
    x_embed[i] = (x_embed[i-1] + (embed_prop2[1] - embed_prop2[0])/(embed_prop1[1]-
1));
// Cubic spline interpolation of EAM lists
spline_interp(x_embed, embed, embed_prop1[1], embed_coef);

// -----
diff_factor = 0.5; // Used in numerical differentiation, has to be <0.5
// It is the factor of distance (for pair and dens
case)
// between items in the EAM list.

pair_delta = pair_prop2[1] - pair_prop2[0]; // Distance between upper and lower
limit
pair_diff = pair_delta/pair_prop1[2]*diff_factor;
dens_delta = pair_delta; // Same as for pair potential since same limits are used.
dens_diff = pair_diff; // Same as for pair potential since same amount of items
used.

embed_delta = embed_prop2[1] - embed_prop2[0]; // Amount between upper and lower limit
embed_diff = embed_delta/embed_prop1[1]*diff_factor;

// -----
// Calculating electron density sum for each atom
for(i = 0; i < num_pts; i++)
{
    dens_sum[i] = 0;

    for(a = -a_inc; a <= a_inc; a = a + a_step)
    {
        for(b = -b_inc; b <= b_inc; b = b + b_step)
        {
            for(c = -c_inc; c <= c_inc; c = c + c_step)
            {
                // Calculating the distance between atom i and every other
atom
                for(j = 0; j < num_pts; j++)
                    r[j] = sqrt((particle_list[i][0]-
(particle_list[j][0]+a))*(particle_list[i][0]-(particle_list[j][0]+a))+(particle_list[i][1]-
(particle_list[j][1]+b))*(particle_list[i][1]-(particle_list[j][1]+b))+(particle_list[i][2]-
(particle_list[j][2]+c))*(particle_list[i][2]-(particle_list[j][2]+c)));

                for(j = 0; j < num_pts; j++)
                {
                    if(r[j] == 0)
                    {

```

```

// DO NOTHING
}
else if(r[j] < dens_prop2[0])
{
    printf("***** ERROR - Distance between atoms
    is too small [1] *****\n");
    return;
}
else if(r[j] > dens_prop2[1])
{
    // DO NOTHING
}
else
{
    item = pair_m*r[j] + pair_b - 1; //
    item_L = floor(item);
    item_H = ceil(item);
    dens_cur = (item-item_H)/(item_L-item_H) *
(dens[(int)item_L]-dens[(int)item_H]) + dens[(int)item_H]; // Interpolation
    dens_sum[i] = dens_sum[i] + dens_cur;
}
// Ending 'j' loop
// Ending 'c' loop
// Ending 'b' loop
// Ending 'a' loop

// Calculating embedding function derivatives
item = embed_m*dens_sum[i] + embed_b - 1;
embed_temp = spline_interp2(item, dens_sum[i], x_embed, embed_coef);

if (item < embed_prop1[1] - 2)
{
    // Calculating 1st derivative
    item = embed_m*(dens_sum[i] + embed_diff) + embed_b - 1;
    dens_sum_temp = dens_sum[i] + embed_diff;
    embed_cur_diff = spline_interp2(item, dens_sum_temp, x_embed, embed_coef);
    F_1der[i] = (embed_cur_diff - embed_temp)/embed_diff;

    // Calculating 2nd derivative
    item = embed_m*(dens_sum[i] + 2*embed_diff) + embed_b - 1;
    dens_sum_temp = dens_sum[i] + 2*embed_diff;
    embed_cur_diff2 = spline_interp2(item, dens_sum_temp, x_embed,
embed_coef);
    F_2der[i] = (embed_cur_diff2 - 2*embed_cur_diff +
embed_temp)/(embed_diff*embed_diff);
}
else
{
    // Calculating 1st derivative
    item = embed_m*(dens_sum[i] - embed_diff) + embed_b - 1;
    dens_sum_temp = dens_sum[i] - embed_diff;
    embed_cur_diff = spline_interp2(item, dens_sum_temp, x_embed, embed_coef);
    F_1der[i] = (embed_cur_diff - embed_temp)/embed_diff;

    // Calculating 2nd derivative
    item = embed_m*(dens_sum[i] - 2*embed_diff) + embed_b - 1;
    dens_sum_temp = dens_sum[i] - 2*embed_diff;
    embed_cur_diff2 = spline_interp2(item, dens_sum_temp, x_embed,
embed_coef);
    F_2der[i] = (embed_cur_diff2 - 2*embed_cur_diff +
embed_temp)/(embed_diff*embed_diff);
}
// Ending 'i' loop

for (i = 0; i < num_pts; i++)
{
    g11 = 0;
    g22 = 0;
    g23 = 0;

```

```

        for(a = -a_inc; a <= a_inc; a = a + a_step)
        {
            for(b = -b_inc; b <= b_inc; b = b + b_step)
            {
                for(c = -c_inc; c <= c_inc; c = c + c_step)
                {
                    // Calculating the distance between atom i and every other
                    for(j = 0; j < num_pts; j++)
                        r[j] = sqrt((particle_list[i][0]-
                    (particle_list[j][0]+a))*(particle_list[i][0]-
                    (particle_list[j][0]+a))+(particle_list[i][1]-
                    (particle_list[j][1]+b))*(particle_list[i][1]-
                    (particle_list[j][1]+b))+(particle_list[i][2]-
                    (particle_list[j][2]+c))*(particle_list[i][2]-
                    (particle_list[j][2]+c)));

                    for(j = 0; j < num_pts; j++)
                    {
                        if(r[j] == 0)
                        {
                            // DO NOTHING
                        }
                        else if(r[j] < pair_prop2[0])
                        {
                            printf("***** ERROR - Distance between atoms
is too small [1] *****\n");
                            return;
                        }
                        else if(r[j] > pair_prop2[1])
                        {
                            // DO NOTHING
                        }
                        else
                        {
                            item = pair_m*r[j] + pair_b - 1;
                            pair_temp = spline_interp2(item, r[j],
                            dens_temp = spline_interp2(item, r[j],

                            if (item < pair_prop1[2] - 2)
                            {
                                // Calculating pair potential
                                item = pair_m*(r[j] + pair_diff) +
                                r_temp = r[j] + pair_diff;
                                pair_cur_diff = spline_interp2(item,
                                pair_1der = (pair_cur_diff -

                                dens_cur_diff = spline_interp2(item,
                                dens_1der = (dens_cur_diff -

                                // Calculating pair potential 2nd
                                item = pair_m*(r[j] + 2*pair_diff) +
                                r_temp = r[j] + 2*pair_diff;
                                pair_cur_diff2 = spline_interp2(item,
                                pair_2der = (pair_cur_diff2 -

                                dens_cur_diff2 = spline_interp2(item,
                                dens_2der = (dens_cur_diff2 -

                                }
                    }
                }
            }
        }
    }
    x_pair, pair_coef);
    x_pair, dens_coef);

    derivative
    pair_b - 1;

    r_temp, x_pair, pair_coef);
    pair_temp)/pair_diff;

    r_temp, x_pair, dens_coef);
    dens_temp)/dens_diff;

    derivative
    pair_b - 1;

    r_temp, x_pair, pair_coef);
    2*pair_cur_diff + pair_temp)/(pair_diff*pair_diff);

    r_temp, x_pair, dens_coef);
    2*dens_cur_diff + dens_temp)/(dens_diff*dens_diff);
}

```

```

else
{
    // Calculating pair potential
    item = pair_m*(r[j] - pair_diff) +
    r_temp = r[j] - pair_diff;
    pair_cur_diff = spline_interp2(item,
    pair_1der = (pair_temp -
    dens_cur_diff = spline_interp2(item,
    dens_1der = (dens_temp -
    // Calculating pair potential 2nd
    item = pair_m*(r[j] - 2*pair_diff) +
    r_temp = r[j] - 2*pair_diff;
    pair_cur_diff2 = spline_interp2(item,
    pair_2der = (pair_cur_diff2 -
    dens_cur_diff2 = spline_interp2(item,
    dens_2der = (dens_cur_diff2 -
}

const1 = pair_2der - pair_1der/r[j];
const2 = dens_2der - dens_1der/r[j];
const3 = (particle_list[j][0]+a-
particle_list[i][0])*(particle_list[j][0]+a-
particle_list[i][0])/(r[j]*r[j]);
const4 = (particle_list[j][0]+a-
particle_list[i][0])*(particle_list[j][1]+b-
particle_list[i][1])/(r[j]*r[j]);
const5 = (particle_list[j][1]+b-
particle_list[i][1])*(particle_list[j][2]+c-
particle_list[i][2])/(r[j]*r[j]);

B1_1111 = B1_1111 + const1*const3*0.5;
B2_1111 = B2_1111 + F_1der[i]*const2*const3;

B1_1122 = B1_1122 + const1*const4*0.5;
B2_1122 = B2_1122 + F_1der[i]*const2*const4;

B1_2323 = B1_2323 + const1*const5*0.5;
B2_2323 = B2_2323 + F_1der[i]*const2*const5;

g11 = g11 +
dens_1der*(particle_list[j][0]+a-particle_list[i][0])*(particle_list[j][0]+a-
particle_list[i][0])/r[j];
g22 = g22 +
dens_1der*(particle_list[j][1]+b-particle_list[i][1])*(particle_list[j][1]+b-
particle_list[i][1])/r[j];
g23 = g23 +
dens_1der*(particle_list[j][1]+b-particle_list[i][1])*(particle_list[j][2]+c-
particle_list[i][2])/r[j];
}
}
}
}
// Ending 'j' loop
// Ending 'c' loop
// Ending 'b' loop
// Ending 'a' loop

B3_1111 = B3_1111 + F_2der[i]*g11*g11;

```

```

        B3_1122 = B3_1122 + F_2der[i]*g11*g22;
        B3_2323 = B3_2323 + F_2der[i]*g23*g23;
    }

    born_terms[0] = B1_1111;
    born_terms[1] = B1_1122;
    born_terms[2] = B1_2323;
    born_terms[3] = B2_1111;
    born_terms[4] = B2_1122;
    born_terms[5] = B2_2323;
    born_terms[6] = B3_1111;
    born_terms[7] = B3_1122;
    born_terms[8] = B3_2323;
}

void spline_interp(double x[], double y[], int num_eam_pts, double coef_array[][4])
{
    int i, k;          // Counters
    double B[MAX_EAM-2];
    double theta1, theta2;
    double phi[MAX_EAM][2];
    double R[MAX_EAM-2][3];
    double phi_temp;

    // -----
    // Calculating R matrix
    for (i = 0; i < num_eam_pts-2; i++)
    {
        if (i > 0)
            R[i][0] = x[i+1] - x[i];
        else
            R[i][0] = 0;

        R[i][1] = 2*(x[i+2] - x[i]);

        if (i < num_eam_pts-3)
            R[i][2] = x[i+2] - x[i+1];
        else
            R[i][2] = 0;
    }

    // -----
    // Calculating B matrix
    for (i = 0; i < num_eam_pts-2; i++)
        B[i] = 6/(x[i+2]-x[i+1])*(y[i+2]-y[i+1]) + 6/(x[i+1]-x[i])*(y[i]-y[i+1]);

    // -----
    // Gauss-Seidel method
    for (i = 0; i < num_eam_pts; i++)
        phi[i][0] = 0;          // Guess initial values of phi
    phi_temp = 0.5;
    k = 0;
    while (abs((phi_temp-phi[1][0])/phi[1][0]) > 0.00001)
    {
        for (i = 0; i < num_eam_pts-2; i++)
        {
            if (i > 0)
                theta1 = R[i][0]*phi[i][1];
            else
                theta1 = 0;

            if (i < num_eam_pts-3)
                theta2 = R[i][2]*phi[i+2][0];
            else
                theta2 = 0;

            phi[i+1][1] = (B[i] - theta1 - theta2)/R[i][1];
        }
        if (k > 0)
            phi_temp = phi[1][0];
    }
}

```

```

        for (i = 0; i < num_eam_pts-2; i++)
            phi[i+1][0] = phi[i+1][1];

        k = k++;
    }

    // -----
    for (i = 0; i < num_eam_pts-1; i++)
    {
        coef_array[i][0] = phi[i][0] / (6*(x[i+1]-x[i]));
        coef_array[i][1] = phi[i+1][0] / (6*(x[i+1]-x[i]));
        coef_array[i][2] = (y[i] / (x[i+1]-x[i]) - phi[i][0] * (x[i+1]-x[i])/6);
        coef_array[i][3] = (y[i+1] / (x[i+1]-x[i]) - phi[i+1][0] * (x[i+1]-x[i])/6);
    }
}

double spline_interp2(double item, double xi, double x_list[], double coef[][4])
{
    double item_L, item_H;
    double term1, term2, term3, term4;
    double embed_temp;

    item_L = floor(item);
    item_H = ceil(item);
    term1 = coef[(int)item_L][0]*(x_list[(int)item_H]-xi)*(x_list[(int)item_H]-
xi)*(x_list[(int)item_H]-xi);
    term2 = coef[(int)item_L][1]*(xi-x_list[(int)item_L])*(xi-x_list[(int)item_L])*(xi-
x_list[(int)item_L]);
    term3 = coef[(int)item_L][2]*(x_list[(int)item_H] - xi);
    term4 = coef[(int)item_L][3]*(xi - x_list[(int)item_L]);
    embed_temp = term1 + term2 + term3 + term4;

    return embed_temp;
}

```

G.3 Source code for mdrob.m

```
% Molecular dynamics simulation - Control file for MDROB
% Developed by: Robert Szydlowski
% Last updated: June 2, 2006

clear all; tic

run_mdrob = 1; % Execute MDROB.exe (main MD code)
calc_born = 1; % Execute MDROB_BORN.exe
calc_stress = 1; % Calculate stress
                % 0 -> Do not execution
                % 1 -> Execute

run_bulk_list = 0; % Run in multiple cases to calculate lattice constant size
run_latt_list = 0; % Run in multiple cases to calculate elastic constants

plot_iter = 1; % Plot potential energy vs. iterations
plot_stress = 1; % Plot stress tensor vs. iterations
plot_elast = 0; % Plot elastic constants vs. temperature
plot_latt = 0; % Plot potential energy vs. lattice constant
plot_histo = 0; % Plot histogram of velocity data
plot_box = 0; % Plot lattice constant vs. iterations size for zero pressure simulation
                % 0 -> No plot
                % 1 -> Create plot

% -----
% Parameter      Cu      Ag      Au      Ni      Pd      Pt      Al
% mass           63.55  107.87  196.97  58.71  106.42  195.08  26.98
% a0 [Angstroms] 3.617  4.090  4.080  3.520  3.890  3.920  4.050

% Recommended dtime for stability: 2e-15 [s]

% -----
% Simulation parameters
mass = 63.55; % [Amu] Atomic mass
lattice = 3.617; % [Angstroms] Initial lattice constant size
file_eam = 'eam_cu_Mei.txt'; % EAM File
temp_file = 'eam_cu_bulk.txt'; % File containing bulk modulus for each temperature
lattice_file = 'eam_cu_Mei_latt.txt'; % File containing equilibrium lattice constant for each temperature

itemp = 300; % [K] Initialization temperature
clamp = 300; % [K] Clamp temperature
dtime = 2e-15; % [s] (1e-15 [s] = 1 [fs] = 1 [femtosecond])
iter = 60000; % Number of iterations
iter_out = 20; % Number of iterations per energy calculation and output
xsave_start = 30000; % Number of iterations after which atomic configurations are stored
xsave = 5; % Number of iterations per atomic configuration output
ssave = 1; % Number of iterations per stress output
box_x = 4; % Size of crystal in x-direction
box_y = 4; % Size of crystal in y-direction
box_z = 4; % Size of crystal in z-direction

% Zero-pressure simulation
bulk = 0; % [GPa], If bulk = 0, then no value is assigned to bulk
bsave = 5; % Number of iterations per box size output

% Delaying factors for temperature clamp and crystal size resizing algorithms
% A bigger delay is required for smaller time-step sizes
delay_clamp = 70; % Number of iterations
delay_box = 30; % Number of iterations

frac1 = 0.7; % Fraction of iterations used for energy analysis
frac2 = 0.7; % Fraction of iterations used for stress analysis
frac3 = 0.5; % Fraction of iterations used for lattice constant analysis

perio_x = 1; % BC type in x-direction, 0 for free BC, 1 for periodic BC
```



```

perio_y      = 1;      % BC type in y-direction
perio_z      = 1;      % BC type in z-direction
strain_x     = 1.0;    % Strain in x-direction
strain_y     = 1.0;    % Strain in y-direction
strain_z     = 1.0;    % Strain in z-direction
file_iter    = 'out_energy.txt';
file_part_fin = 'out_part_fin.txt';
file_part_ini = 'out_part_ini.txt';
file_stress   = 'out_stress.txt';
file_xsave_a  = 'out_pos_';
file_xsave_b  = 'txt';
file_box      = 'out_box.txt';

% -----
% Varying parameters
dlattice = 0.000; % [Angstroms]
ddtime = 0e-15; % [s]
num_sim = 1; % Number of simulations

% #####
% NO MORE MODIFICATIONS BEYOND THIS POINT
% #####

% -----
% Program constants
kB = 1.3806505e-23; % [J/K] Boltzmann's constant

% -----
% Reading in temperature and bulk modulus list for specific metal
if run_bulk_list == 1
    fid = fopen(temp_file,'r');
    temp_bulk = fscanf(fid, '%g',[2,inf]);
    status = fclose(fid);
    temp_bulk = temp_bulk';
    size_temp_bulk = size(temp_bulk); size_temp_bulk = size_temp_bulk(1);
    num_sim = size_temp_bulk;
end

% -----
% Reading in calculated lattice constant sizes for specific potentials and temperatures
if run_latt_list == 1
    fid = fopen(lattice_file,'r');
    eq_a0 = fscanf(fid, '%g',[2,inf]);
    status = fclose(fid);
    eq_a0 = eq_a0';
    size_eq_a0 = size(eq_a0); size_eq_a0 = size_eq_a0(1);
    num_sim = size_eq_a0;
end

% -----
fprintf('\nSimulation started...\n');
for i = 1:num_sim

    if run_bulk_list == 1
        fprintf('\n -----\n');
        itemp = temp_bulk(i,1);
        clamp = itemp;
        bulk = temp_bulk(i,2);
    end

    if run_latt_list == 1
        fprintf('\n -----\n');
        itemp = eq_a0(i,1);
        clamp = itemp;
        lattice = eq_a0(i,2);
    end

    % Creating MDROB control file
    fid = fopen('input_auto.txt','w'); % Open file for writing

    fprintf(fid,'mass %3.4f\n', mass);

```

```

fprintf(fid,'lattice %.6f\n', lattice);
fprintf(fid,'itemp %.3f\n', itemp);
fprintf(fid,'clamp %.3f\n', clamp);
fprintf(fid,'dtime %3.4e\n', dtime);
fprintf(fid,'iter %3.0f\n', iter);
fprintf(fid,'iter_out %3.0f\n', iter_out);
fprintf(fid,'box %3.0f %3.0f %3.0f\n', box_x, box_y, box_z);
fprintf(fid,'perio %3.0f %3.0f %3.0f\n', perio_x, perio_y, perio_z);
fprintf(fid,'strain %3.4f %3.4f %3.4f\n', strain_x, strain_y, strain_z);
fprintf(fid,'file_eam %s\n', file_eam);
fprintf(fid,'file_iter %s\n', file_iter);
fprintf(fid,'file_part_fin %s\n', file_part_fin);
fprintf(fid,'file_part_ini %s\n', file_part_ini);
fprintf(fid,'xsave %3.0f %3.0f %s %s\n', xsave_start, xsave, file_xsave_a, file_xsave_b);
fprintf(fid,'ssave %3.0f %s\n', ssave, file_stress);

if (delay_clamp ~= 33)
    fprintf(fid,'delay_clamp %3.0f\n', delay_clamp);
end
if (delay_box ~= 33)
    fprintf(fid,'delay_box %3.0f\n', delay_box);
end

if (bulk ~= 0)
    fprintf(fid,'bulk %6.2f\n', bulk);
    fprintf(fid,'bsave %3.0f %s\n', bsave, file_box);
end

status = fclose(fid); % Close file for writing

% -----
% Running MDROB
if run_mdrob == 1
    system('del out*.'); % Clear all output file
    system('mdrob.exe input_auto.txt > out_mdrob.txt');
end

% -----
% POSTPROCESSING
% -----
% Reading in iteration and potential energy file
fid = fopen(file_iter);
discard = fgetl(fid);
energy = fscanf(fid, '%g', [5,inf]);
status = fclose(fid); % Close file for reading
energy = energy'; % Inversing the array

it(:,i) = energy(:,1);
u_part(:,i) = energy(:,3);
k_part(:,i) = energy(:,4);
t_part(:,i) = energy(:,5);

% Taking average over the last portion of data specified by 'frac1'
energy_size = size(energy);
energy_size = energy_size(1);
% Potential energy
energy_pot_part_avg = u_part(round((1-frac1)*energy_size):energy_size,i);
energy_pot(i,1) = mean(energy_pot_part_avg);
% Total energy
energy_total_avg = t_part(round((1-frac1)*energy_size):energy_size,i);
energy_tot(i,1) = mean(energy_total_avg);
% Lattice constant size
lattice_const(i,1) = lattice;
temp(i,1) = itemp;

% Printing results
fprintf('\nT = %6.2f [K], a0 (initial) = %5f [Angstrom]\n', clamp, lattice_const(i,1));
fprintf('U_pot = %6f [eV], E_total = %6f [eV]\n', energy_pot(i,1), energy_tot(i,1));

% -----
% Calculate Born terms

```

```

box = [box_x, box_y, box_z];
if calc_born == 1
    system('mdrob_born.exe input_auto.txt > out_mdrob_born.txt');

    fid = fopen('out_born_terms.txt','r'); % Open file for writing
    born = fscanf(fid, '%g',[9,inf]);
    status = fclose(fid); % Close file for writing
    born = born';

    B1 = [mean(born(:,1)) mean(born(:,2)) mean(born(:,3))];
    B2 = [mean(born(:,4)) mean(born(:,5)) mean(born(:,6))];
    B3 = [mean(born(:,7)) mean(born(:,8)) mean(born(:,9))];
    [c11, c12, c44] = mdrob_disp_born(box, lattice, B1, B2, B3);
end

% -----
% Stress tensor
fid = fopen('out_stress.txt','r'); % Open file for writing
stress = fscanf(fid, '%g',[7,inf]);
status = fclose(fid); % Close file for writing
stress = stress';
stress(:,2:7) = stress(:,2:7)*1e-7; % Converting from [erg] to [J]
stress_size = size(stress); stress_size = stress_size(1);
% -----
% Used in both stress and born calc

if calc_stress == 1
    P11 = mean(stress(round((1-frac2)*stress_size):stress_size,2));
    P22 = mean(stress(round((1-frac2)*stress_size):stress_size,3));
    P33 = mean(stress(round((1-frac2)*stress_size):stress_size,4));
    P23 = mean(stress(round((1-frac2)*stress_size):stress_size,5));
    P31 = mean(stress(round((1-frac2)*stress_size):stress_size,6));
    P12 = mean(stress(round((1-frac2)*stress_size):stress_size,7));

    fprintf('\nAverage Stress Tensor\n');
    fprintf(' P11 = %14.5e\n P22 = %14.5e\n P33 = %14.5e\n', P11, P22, P33);
    fprintf(' P23 = %14.5e\n P31 = %14.5e\n P12 = %14.5e\n', P23, P31, P12);

    % -----
    % Calculating averages
    P11P11 = mean(stress(round((1-frac2)*stress_size):stress_size,2).*...
        stress(round((1-frac2)*stress_size):stress_size,2));
    P22P22 = mean(stress(round((1-frac2)*stress_size):stress_size,3).*...
        stress(round((1-frac2)*stress_size):stress_size,3));
    P33P33 = mean(stress(round((1-frac2)*stress_size):stress_size,4).*...
        stress(round((1-frac2)*stress_size):stress_size,4));

    P11P22 = mean(stress(round((1-frac2)*stress_size):stress_size,2).*...
        stress(round((1-frac2)*stress_size):stress_size,3));
    P22P33 = mean(stress(round((1-frac2)*stress_size):stress_size,3).*...
        stress(round((1-frac2)*stress_size):stress_size,4));
    P33P11 = mean(stress(round((1-frac2)*stress_size):stress_size,4).*...
        stress(round((1-frac2)*stress_size):stress_size,2));

    P23P23 = mean(stress(round((1-frac2)*stress_size):stress_size,5).*...
        stress(round((1-frac2)*stress_size):stress_size,5));
    P31P31 = mean(stress(round((1-frac2)*stress_size):stress_size,6).*...
        stress(round((1-frac2)*stress_size):stress_size,6));
    P12P12 = mean(stress(round((1-frac2)*stress_size):stress_size,7).*...
        stress(round((1-frac2)*stress_size):stress_size,7));

    % -----
    vol = (lattice*1e-10)^3*box_x*box_y*box_z;
    coef1 = vol/kB/clamp;

    P_time_avg = (P11P11 - P11*P11)/vol^2;
    C11_fluct_1 = -P_time_avg*coef1;
    P_time_avg = (P22P22 - P22*P22)/vol^2;
    C11_fluct_2 = -P_time_avg*coef1;
    P_time_avg = (P33P33 - P33*P33)/vol^2;
    C11_fluct_3 = -P_time_avg*coef1;

```

```

C11_fluct = (C11_fluct_1 + C11_fluct_2 + C11_fluct_3)/3;

P_time_avg = (P11P22 - P11*P22)/vol^2;
C12_fluct_1 = -P_time_avg*coef1;
P_time_avg = (P22P33 - P22*P33)/vol^2;
C12_fluct_2 = -P_time_avg*coef1;
P_time_avg = (P33P11 - P33*P11)/vol^2;
C12_fluct_3 = -P_time_avg*coef1;
C12_fluct = (C12_fluct_1 + C12_fluct_2 + C12_fluct_3)/3;

P_time_avg = (P23P23 - P23*P23)/vol^2;
C44_fluct_1 = -P_time_avg*coef1;
P_time_avg = (P31P31 - P31*P31)/vol^2;
C44_fluct_2 = -P_time_avg*coef1;
P_time_avg = (P12P12 - P12*P12)/vol^2;
C44_fluct_3 = -P_time_avg*coef1;
C44_fluct = (C44_fluct_1 + C44_fluct_2 + C44_fluct_3)/3;

fprintf('\nFluctuation terms\n');
fprintf('    C11_fluct    C12_fluct    C44_fluct\n');
fprintf('%12.4f %12.4f %12.4f\n', C11_fluct_1/1e9, C12_fluct_1/1e9, C44_fluct_1/1e9);
fprintf('%12.4f %12.4f %12.4f\n', C11_fluct_2/1e9, C12_fluct_2/1e9, C44_fluct_2/1e9);
fprintf('%12.4f %12.4f %12.4f\n', C11_fluct_3/1e9, C12_fluct_3/1e9, C44_fluct_3/1e9);
end

if calc_born == 1
    vol = (lattice*1e-10)^3*box_x*box_y*box_z; % (Added in)
    C11_T_corr = 2*(4*box_x*box_y*box_z)*kB*clamp/vol^2; % [Pa]
    C12_T_corr = 2*(4*box_x*box_y*box_z)*kB*clamp/vol^0; % [Pa] Always equal to zero
    C44_T_corr = 2*(4*box_x*box_y*box_z)*kB*clamp/vol^1; % [Pa]

    C11 = C11_fluct/1e9 + C11_T_corr/1e9 + c11;
    C12 = C12_fluct/1e9 + C12_T_corr/1e9 + c12;
    C44 = C44_fluct/1e9 + C44_T_corr/1e9 + c44;

    fprintf('\n      Fluct      T_corr      Born      Total\n')
    fprintf('C11 => %7.2f + %7.2f + %7.2f = %7.2f [GPa]\n', C11_fluct/1e9, C11_T_corr/1e9,
c11, C11);
    fprintf('C12 => %7.2f + %7.2f + %7.2f = %7.2f [GPa]\n', C12_fluct/1e9, C12_T_corr/1e9,
c12, C12);
    fprintf('C44 => %7.2f + %7.2f + %7.2f = %7.2f [GPa]\n', C44_fluct/1e9, C44_T_corr/1e9,
c44, C44);

    elast_c(i,1) = C11; elast_c(i,2) = C12; elast_c(i,3) = C44;

    V = lattice^3*box_x*box_y*box_z*(1e-10)^3;
    conv1 = 1.602177e-19/V/1e9; % Use this conversion if EAM is in [eV]
    %conv1 = 1.3806505e-23/V/1e9; % Use this conversion if EAM is in [K], Not req'd

    out(i,:) = [clamp, lattice, energy_pot(i,1), energy_tot(i,1), C11_fluct/1e9,
C11_T_corr/1e9, B1(1)*conv1, B2(1)*conv1, B3(1)*conv1, C11, ...
    C12_fluct/1e9, C12_T_corr/1e9, B1(2)*conv1, B2(2)*conv1, B3(2)*conv1, C12, ...
    C44_fluct/1e9, C44_T_corr/1e9, B1(3)*conv1, B2(3)*conv1, B3(3)*conv1, C44];
end

if (bulk ~= 0) % Lattice constant size for zero pressure simulation
    fid = fopen('out_box.txt','r'); % Open file for writing
    a_size = fscanf(fid, '%g',[2,inf]);
    status = fclose(fid); % Close file for writing
    a_size = a_size';
    size_a_size = size(a_size); size_a_size(1);
    a_equilib(i,1) = mean(a_size(round((1-frac3)*size_a_size):size_a_size,2));
    fprintf('\nZero-pressure results\n');
    fprintf(' a0 = %f [Angstroms]\n', a_equilib(i,1));
end

% -----
% Open file with velocities to create histogram
fid = fopen(file_part_fin);
discard = fgetl(fid);
vel = fscanf(fid, '%g',[8,inf]);

```

```

status = fclose(fid);
vel = vel';
vel(:,1) = []; vel(:,1) = []; vel(:,1) = []; vel(:,1) = [];
if plot_histo == 1
    figure
    x = -itemp*2:20:itemp*2;
    hist(vel(:,1:3),x)
    ylabel('Distribution');
    xlabel('Velocity (m/s)');
end

% -----
% MODIFYING INPUT PARAMETERS
lattice = lattice + dlattice;
dtime = dtime + ddtime;
end

% -----
% Writing calculated lattice constant sizes for each temperature
if run_bulk_list == 1
    fid = fopen(lattice_file,'w'); % Open file for writing
    for i = 1:num_sim
        fprintf(fid,'%7.2f\t%7.4f\n', temp_bulk(i,1), a_equilib(i,1));
    end
    status = fclose(fid);
end

% -----
% Writing results to file
if (calc_stress == 1 && calc_born == 1)
    fid = fopen('out_results.txt','w'); % Open file for writing
    for i = 1:num_sim
        fprintf(fid,'%7.2f\t%9.6f\t%9.6f\t%9.6f\t', out(i,1:4));
        fprintf(fid,'%8.4f\t%8.4f\t%9.4f\t%10.4f\t%9.4f\t%9.4f\t', out(i,5:10));
        fprintf(fid,'%8.4f\t%8.4f\t%9.4f\t%10.4f\t%9.4f\t%9.4f\t', out(i,11:16));
        fprintf(fid,'%8.4f\t%8.4f\t%9.4f\t%10.4f\t%9.4f\t%9.4f\n', out(i,17:22));
    end
    status = fclose(fid);
end

% -----
% Plotting
if plot_iter == 1
    figure;
    subplot(2,1,1);
    hold on;
    plot(it, u_part);
    if (run_bulk_list ~= 1) && (run_latt_list ~= 1)
        plot(it, t_part, '--k');
        legend('Potential','Total');
        ylabel('Total and potential energy (eV)');
    end
    ylabel('Potential energy (eV)');
    hold off;
    subplot(2,1,2); plot(it, k_part);
    ylabel('Kinetic energy (eV)');
    xlabel('Iterations');
    if (run_bulk_list == 1) || (run_latt_list == 1)
        figure;
        plot(it, t_part);
        ylabel('Total energy (eV)');
        xlabel('Iterations');
    end
end
if plot_latt == 1
    figure; plot(lattice_const, energy_pot)
    xlabel('Lattice constant (Angstroms)'); ylabel('Potential energy (erg)')
end
if plot_stress == 1
    figure
    subplot(2,1,1); plot(stress(:,1), stress(:,2:4))

```

```

ylabel('Stress*Volume (J*m^3)')
legend('P11','P22','P33','Location','SouthEast');
subplot(2,1,2); plot(stress(:,1), stress(:,5:7))
ylabel('Stress*Volume (J*m^3)');
legend('P21','P23','P31','Location','SouthEast');
xlabel('Iterations')
end
if (plot_elast == 1) && (num_sim > 1)
figure
subplot(3,1,1); plot(temp, elast_c(:,1));
ylabel('c11 (GPa)')
subplot(3,1,2); plot(temp, elast_c(:,2));
ylabel('c12 (GPa)');
subplot(3,1,3); plot(temp, elast_c(:,3));
ylabel('c44 (GPa)'); xlabel('Temperature (K)');
end
if (bulk ~= 0 && plot_box == 1)
figure
subplot(2,1,1); plot(stress(:,1), stress(:,2:4))
legend('P11','P22','P33','Location','SouthEast');
ylabel('Stress*Volume (J*m^3)')
subplot(2,1,2); plot(a_size(:,1), a_size(:,2));
ylabel('Lattice constant size (Angstroms)'); xlabel('Iterations')
end

% -----
%load train;
%wavplay(y,Fs)
fprintf('\n')
toc

```

G.4 Source code for *mdrob_disp_born.m*

```
% Function used to display Born terms

function[c11, c12, c44] = mdrob_calc(box, a0, B1, B2, B3)

B1_1111 = B1(1);
B1_1122 = B1(2);
B1_2323 = B1(3);

B2_1111 = B2(1);
B2_1122 = B2(2);
B2_2323 = B2(3);

B3_1111 = B3(1);
B3_1122 = B3(2);
B3_2323 = B3(3);

% -----
num = 4*box(1)*box(2)*box(3);
V = a0^3*box(1)*box(2)*box(3)*(1e-10)^3;
fprintf('Number of atoms = %4i, V = %e [m^3]\n\n', num, V);

conv1 = 1.602177e-19/V/1e9;    % Use this conversion if EAM is in [eV]
%conv1 = 1.3806505e-23/V/1e9;    % Use this conversion if EAM is in [K], Not req'd

c11 = (B1_1111 + B2_1111 + B3_1111)*conv1;
c12 = (B1_1122 + B2_1122 + B3_1122)*conv1;
c44 = (B1_2323 + B2_2323 + B3_2323)*conv1;

fprintf('          B1          B2          B3\n');
fprintf('1111 %8.2f,  %8.2f,  %8.2f,  C11_Born = %8.2f\n', B1_1111*conv1, B2_1111*conv1,
B3_1111*conv1, c11);
fprintf('1122 %8.2f,  %8.2f,  %8.2f,  C12_Born = %8.2f\n', B1_1122*conv1, B2_1122*conv1,
B3_1122*conv1, c12);
fprintf('2323 %8.2f,  %8.2f,  %8.2f,  C44_Born = %8.2f\n', B1_2323*conv1, B2_2323*conv1,
B3_2323*conv1, c44);
```

G.5 Source code for MEI EAM potential generation

The potential generation code for Cu is shown here as an example. In order to create any other of the six FCC potentials, the fitting parameters at the beginning of the file need to be changed.

```
% MATLAB code used for MEI potential file generation
% Mei's EAM potential for Cu
clc; clear all; tic;

% #####
% NOTES: No smoothing used in EAM generation
% #####

% -----
% Control parameters
plot_func_pair = 0; % Plotting individual functions
plot_func_dens = 0;
plot_func_embed = 0;
plot_all = 0;
plot_e = 1;

file_eam = 'eam_cu_MEI.txt';

% -----
% Fitting parameters - COPPER
% -----
c0 = 0.14095; % [dimensionless]
c1 = -1.93222; % [dimensionless]
c2 = 9.60043; % [dimensionless]
c3 = -21.23718; % [dimensionless]
c4 = 20.18581; % [dimensionless]
c5 = -5.81656; % [dimensionless]

beta = 5.85; % [dimensionless]
delta = 12.06; % [dimensionless]
gamma = 6.82054; % [dimensionless]
phi_e = 0.1217; % [eV]

a0 = 3.617; % [Angstroms] Equilibrium lattice constant size at absolute zero
bulk = 142.03; % [GPa] Bulk modulus at absolute zero
Ecoh = 336; % [kJ/mol] Cohesive energy
a_vol = 7.11; % [cm^3/mol]

% -----
rle = sqrt(a0^2/2); % [Angstroms] Nearest neighbor based on the FCC lattice
rhoe = 1; % Assumed to be 1

sm = [12, 6, 24, 12]; % Number of atoms in each shell starting with the closest shell (based on
FCC lattice)
shell = 3; % Number of shells to be used in the derivation of the EAM potential

% -----
% EAM properties
rmin = 1.5; % [Angstroms]
rn_fact = 1.75; % [dimensionless]
rc_fact = 1.95; % [dimensionless]

pair_size = 1000; % Number of items in list
embed_size = 30; % Number of items in list

dens_min = 0.2; % [?] Minimum electron density
```



```

dens_max = 1.5;          % [?] Maximum electron density

% -----
rcut = rc_fact*rle;      % [Angstrom] Cut-off distance for potential
rn    = rn_fact*rle;      % [Angstrom] Distance beyond which potential is smoothed

% -----
fprintf('Minimum distance between atoms:      %.4f [Angstroms]\n', rmin);
fprintf('Distance between nearest neighbors: %.4f [Angstroms]\n', rle);
fprintf('Cut-off distance for the potential: %.4f [Angstroms]\n', rcut);

% #####
%                                     PAIR POTENTIAL
% #####
pair_step = (rcut - rmin)/(pair_size-1);

r(1,1) = rmin;
pair(1,1) = -phi_e*(1+delta*(r(1,1)/rle-1))*exp(-gamma*(r(1,1)/rle-1));
for i = 2:pair_size
    r(i,1) = r(i-1,1)+pair_step;
    pair(i,1) = -phi_e*(1+delta*(r(i,1)/rle-1))*exp(-gamma*(r(i,1)/rle-1));
end

% -----
% Smoothing function
for i = 1:pair_size
    if r(i,1) <= rn
        pair_smooth(i,1) = pair(i,1);
    elseif (r(i,1) < rcut)
        x = (r(i,1)-rn)/(rcut-rn);
        q = (1-x)^3*(1+3*x+6*x^2);
        pair_smooth(i,1) = pair(i,1)*q;
    else
        pair_smooth(i,1) = 0;
    end
end

if plot_func_pair == 1
    figure; hold on;
    plot(r, pair);
    plot(r, pair_smooth, 'xg');
    plot([rmin rcut], [0 0], '--r');
    plot([rn rn], [-1 1], '-r'); plot([rcut rcut], [-1 1], '-r'); hold off;
    title('Pair Potential Function'); legend('Normal','Smoothed');
    ylabel('Pair Potential (eV)'); xlabel('Distance between atoms, r, (Angstroms)');
    print('-djpeg','eam_fit_pair.jpg');
end
%pair = pair_smooth;

% #####
%                                     ELECTRON DENSITY
% #####
dens_step = pair_step; dens_size = pair_size;
c(1,1) = c0; c(2,1) = c1; c(3,1) = c2; c(4,1) = c3; c(5,1) = c4; c(6,1) = c5;

for i = 1:pair_size
    dens_sum_k = 0;
    for j = 1:6
        dens_sum_k = dens_sum_k + c(j,1)*(rle/r(i,1))^(j-1);
    end
    dens(i,1) = rhoe/12*dens_sum_k;
end

% -----
% Smoothing function
for i = 1:dens_size
    if r(i,1) <= rn
        dens_smooth(i,1) = dens(i,1);
    elseif (r(i,1) < rcut)
        x = (r(i,1)-rn)/(rcut-rn);
        q = (1-x)^3*(1+3*x+6*x^2);

```

```

        dens_smooth(i,1) = dens(i,1)*q;
    else
        dens_smooth(i,1) = 0;
    end
end

if plot_func_dens == 1
    figure; hold on;
    plot(r, dens);
    plot(r, dens_smooth, 'xg');
    plot([rmin rcut], [0 0], '--r');
    plot([rn rn], [-0.2 .2], '-r'); plot([rcut rcut], [-0.2 .2], '-r'); hold off;
    title('Electron Density Function'); legend('Normal','Smoothed');
    ylabel('Electron Density'); xlabel('Distance between atoms, r, (Angstroms)');
    %print('-djpeg','eam_fit_dens.jpg');
end
%dens = dens_smooth;

% #####
%                               EMBEDDING FUNCTION
% #####
Ecoh = Ecoh*1000;           % [J/mol] - Converting from [kJ] to [J]
Ecoh = Ecoh/6.02214e23;    % [J] - Converting from [per mol] to [per atom]
a_vol = a_vol/100^3;       % [m^3/mol] - Converting from [cm^3] to [m^3]
a_vol = a_vol/6.02214e23;  % [m^3] - Converting to per atom from per mol
bulk = bulk*1e9;          % [Pa] - Converting from [GPa] to [Pa]

alpha = sqrt(9*bulk*a_vol/Ecoh); % [dimensionless]

Ecoh = Ecoh*6.241507649e18; % [eV] - Converting from [J] to [eV]

fprintf('Cohesive energy = %11.4f [eV]\n', Ecoh);
fprintf('Bulk modulus    = %11.4f [GPa]\n', bulk/1e9);
fprintf('Atomic volume    = %11.4e [m^3]\n', a_vol);
fprintf('alpha            = %11.4f [dimensionless]\n', alpha);

i = 1;
for rho = dens_min:(dens_max-dens_min)/(embed_size-1):dens_max

    term1 = -Ecoh*(1-alpha/beta*reallog(rho/rhoe))*(rho/rhoe)^(alpha/beta);

    term2 = 0;
    for m = 1:shell % Number of shells to be used
        pm = sqrt(m); % For FCC lattice, pm = sqrt(m)
        t1 = sm(m)*exp(-(pm-1)*gamma);
        t2 = 1 + (pm-1)*delta - pm*delta/beta*reallog(rho/rhoe);
        t3 = (rho/rhoe)^(pm*gamma/beta);
        term2 = term2 + t1*t2*t3;
    end
    embed(i,1) = term1 + 0.5*phi_e*term2;
    dens_F(i,1) = rho;
    i = i+1;
end

if plot_func_embed == 1
    figure; plot(dens_F, embed);
    title('Embedding Function'); ylabel('Embedding Function'); xlabel('Electron density')
    %print('-djpeg','eam_fit_embed.jpg');
end

% #####
%                               PLOTTING
% #####
if plot_all == 1
    figure; subplot(3,1,1); hold on;
    plot(r, pair); plot([rmin rcut],[0 0], '-r'); hold off;
    ylabel('Pair Potential (eV)'); xlabel('Distance between atoms, r, (Angstroms)')

    subplot(3,1,2); hold on; plot(r, dens); plot([rmin rcut],[0 0], '-r'); hold off;
    ylabel('Electron Density'); xlabel('Distance between atoms, r, (Angstroms)')

```

```

        subplot(3,1,3); plot(dens_F, embed);
        ylabel('Embedding Function'); xlabel('Electron density')
    end

% #####
%                               WRITING TO FILE
% #####
fid = fopen(file_eam,'w');
fprintf(fid, '# Mei''s EAM Potential for Cu\n');
fprintf(fid, 'eunit eV\n\n');
fprintf(fid, 'POTENTIAL PAIR 1 1 %i %f %f\n', pair_size, rmin, rcut);
for i = 1:pair_size; fprintf(fid, '%16.9e\n', pair(i,1)); end
fprintf(fid, 'POTENTIAL DENS 1 %i %f %f\n', pair_size, rmin, rcut);
for i = 1:pair_size; fprintf(fid, '%16.9e\n', dens(i,1)); end
fprintf(fid, 'POTENTIAL EMBED 1 %i %11.9f %11.9f\n', embed_size, dens_F(1,1),
dens_F(embed_size,1));
for i = 1:embed_size; fprintf(fid, '%16.9e\n', embed(i,1)); end
status = fclose(fid);

% #####
%                               TESTING EAM
% #####
% Create perfect crystal
box_size = [7 7 7];
p_unscaled = eam_crystal_create(box_size);    % Creating particle list
size_p = size(p_unscaled);

% Finding atom at center of crystal
for i = 1:size_p(1)
    if (p_unscaled(i,1) == 3 && p_unscaled(i,2) == 3 && p_unscaled(i,3) == 3)
        p_num = i;
    end
end

% -----
% Iterating through different lattice constant sizes

factor = 0.976; % Starting at a 5% compression
delta_factor = ((1*(1-factor)+1)-factor)/(embed_size-1); % Delta designed for a 5% tension
delta_factor = ((2*(1-factor)+1)-factor)/(embed_size-1); % Delta designed for a 10% tension

num_it = embed_size;
h = waitbar(0,'Please wait...');    % Waitbar

fprintf('\n      a pair_sum dens_sum      F      E      Ecry      n\n');
for j = 1:num_it
    n = 0; % Number of particles in range for each atomic configuration
    % Scaling particle positions
    a(j,1) = factor*a0;
    p = p_unscaled*a(j,1);

    rb(:,1) = sqrt((p(p_num,1)-p(:,1)).^2 + (p(p_num,2)-p(:,2)).^2 + (p(p_num,3)-p(:,3)).^2);
    pair_sum = 0; dens_sum(j,1) = 0;
    for i = 1:size_p(1)
        if (rb(i,1) ~= 0 && rb(i,1) >= rmin && rb(i,1) <= rcut)
            pair_temp = interp1(r, pair, rb(i,1), 'spline');
            pair_sum = pair_sum + pair_temp;

            dens_temp = interp1(r, dens, rb(i,1), 'spline');
            dens_sum(j,1) = dens_sum(j,1) + dens_temp;

            fprintf('neighbor: %2i, pair-pot= %f\n', n, pair_temp);
            n = n+1;
        elseif (rb(i,1) ~= 0 && rb(i,1) <= rmin)
            disp('*** ERROR ***');
        end
    end
    pair_sum = pair_sum*0.5;    % Divided by 2 to avoid double counting

    % Calculation of potential energy using Rose's approximation
    alpha = sqrt(9*bulk*a_vol/(Ecoh/6.241507649e18)); % [dimensionless]

```

```

a_star = (a(j,1)/a0 - 1)*alpha;
Ecry(j,1) = -Ecoh*(1+a_star)*exp(-a_star);

F = interp1(dens_F, embed, dens_sum(j,1), 'spline');
E(j,1) = pair_sum + F;

fprintf('%6.4f %7.4f %8.4f %8.4f %8.4f %8.5f %3i\n', a(j,1), pair_sum, dens_sum(j,1), F,
E(j,1), Ecry(j,1), n);

factor = factor + delta_factor;
waitbar(j/num_it) % Counter for Waitbar
end
close(h)

% -----
if plot_e == 1
figure; hold on;
plot(a, Ecry);
plot(a, E, '--r');
%plot([a0 a0], [-3.49 -3.46], '-g');
hold off;
xlabel('Lattice constant (Angstroms)'); ylabel('Potetial energy (eV)');
legend('Rose','Mei'); grid on;
end

% -----
disp(' ')
toc

```

④ 62-87-114