AB INITIO ELASTIC AND THERMODYNAMIC PROPERTIES OF HIGH-TEMPERATURE CUBIC INTERMETALLICS AT FINITE TEMPERATURES

A Thesis

by

MICHAEL ERIC WILLIAMS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2008

Major Subject: Mechanical Engineering

AB INITIO ELASTIC AND THERMODYNAMIC PROPERTIES OF HIGH-TEMPERATURE CUBIC INTERMETALLICS AT FINITE TEMPERATURES

A Thesis

by

MICHAEL ERIC WILLIAMS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Raymundo Arróyave
Committee Members,	Tahir Cagin
	Yongmei Jin
Head of Department,	Dennis O'Neal

May 2008

Major Subject: Mechanical Engineering

ABSTRACT

Ab Initio Elastic and Thermodynamic Properties
of High-Temperature Cubic Intermetallics
at Finite Temperatures. (May 2008)
Michael Eric Williams, B.S., Brigham Young University
Chair of Advisory Committee: Dr. Raymundo Arróyave

In this work we present the development of a method for the prediciton of finite temperature elastic and thermodynamic properties of cubic, non-magnetic unary and binary metals from first principles calculations. Vibrational, electronic and anharmonic contributions to the free energy are accounted for while magnetic effects are neglected. The method involves the construction of a free energy surface in volume/temperature space through the use of quasi-harmonic lattice dynamics. Additional strain energy calculations are performed and fit to the derived thermal expansion to present the temperature dependence of single crystal elastic constants. The methods are developed within the framework of density functional theory, lattice dynamics, and finite elasticity. The model is first developed for FCC aluminum and BCC tungsten which demonstrate the validity of the model as well as some of the limitations arising from the approximations made such as the effects of intrinsic anharmonicity. The same procedure is then applied to the B2 systems NiAl, RuAl and IrAl which are considred for high temperature applications. Overall there is excellent correlation between the calculated properties and experimentally tabulated values. Dynamic methods for the prediction of temperature dependent properties are also introduced and a groundwork is laid for future development of a robust method.

To Mindy, Katelyn, Mom and Dad

ACKNOWLEDGMENTS

The calculations were performed on the CAT cluster of the Department of Chemical Engineering, Texas A&M University using the VASP ab initio code developed by the Institut für Materialphysik of the Universität Wien. The quasi-harmonic free energies were obtained through the use of the supercell method as implemented in the ATAT package, for which the authors thank Dr. Axel van de Walle. The author would also like to thank Prof. Tahir Cagin for helpful discussions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

One of the great keys in the advancement of technology is the development of materials that have the optimal combination of properties for the desired application. For example, without heat resistant ceramics the space shuttle would burn up during atmospheric re-entry and without the unique properties of semi-conductors toady's electronics wouldn't exist as we know them. Materials therefore can become either enablers or bottlenecks for scientific and technological advancement. It is the primary work of materials science and engineers to understand the properties of matter and the processes that will combine those properties into synergistic materials with the overall material properties for a given application [1]. In order to understand the phenomena that govern material behavior scientists typically develop and perform countless experiments to determine the properties of interest such as mechanical strength, stiffness, resistance to heat, conductivity of electricity and many others. With recent advances in computer hardware and software the role of computer simulations in materials science is expanding and delivering many useful insights [2]. In particular, these 'virtual experiments' can provide understanding not only of *what* happens to a material but *why* it happens [3] based on understanding of microscopic phenomena.

One area in which material properties limit the effectiveness of a system is in the field of gas turbines. Gas turbines are a primary source of power generation throughout the world and there is high demand for the development of ultra-high temperature materials so that the turbines can run more efficiently. The higher the temperature at which a turbine can operate, the greater the thermodynamic efficiency of the system will be. This higher efficiency can thus reduce the emmission of greenhouse gases and make the overall process

---

The journal model is *IEEE Transactions on Automatic Control.*

more economical. Currently the state of the art in high temperature materials in turbines are the Nickel-based supperalloys [4]. These alloys are desireable due to their high operating temperatures and other mechanical properties such as high ductility at low temperatures [5]. Recent studies have suggested that alloys including several platinum group metals (PGM) could yield operating temperatures higher than the current Ni-based alloys [5, 6, 7, 8]. For example, the melting point of RuAl is over 2300 K while maintaining good ductility and high resistance to oxidation and corrosion [8]. Similar properties have also been reported for IrAl [6]. Since these alloys are just recently being explored for applications such as structural members in turbines their properties and underlying micro-structural and atomistic mechanisms have not yet been fully characterized. The goal of this work is to develop and validate a first-principles method for the prediction of the single crystal elastic and thermodynamic properties of PGM cubic intermetallics at finite temperatures.

After a review of background information and related literature this text contains three body chapters, two of which represent original journal articles which have been or will be submitted shortly for peer review and publication. The first paper explores the development of a method for first principles prediction of finite temperature elastic constants and thermodynamic properties including considerations for intrinsic anharmonicity and tests the method for pure aluminum and tungsten. Excellent agreement is found between experimental and calculated values for aluminum and fair results for tungsten up to about 60% of the melting temperature. Effects of anharmonicity are assumed to be one of the largest sources of error for calculated properties of tungsten. The second paper included extends the established method for finite temperature predictions to the cubic intermetallics NiAl, RuAl and IrAl. The calculated property calculations compare favorably with experiment where available. Currently there is very little experimental information available for single crystal properties of RuAl and IrAl. The elastic property predictions for RuAl and IrAl are set forth as theoretical predictions awaiting experimental comparison. The third body

chapter details efforts to calculate elastic constants dynhamically from ab initio molecular dynamics and statistical fluctuation formulae. While this work hasn't yet reached publication status it is projected that this and related work will yield at least one or two additional publications. These main chapters are followed by a documentation of the computer code developed through the course of this project and finally a summary of the entire work is presented along with ideas for future work in these areas.

CHAPTER II

LITERATURE REVIEW

This chapter provides the reader with background information and a review of some of the key literature in the field of materials simulation for the prediction of elastic and thermodynamic properties. This chapter is not meant to be exhaustive on the topic but rather to provide the reader a basic framework within which the basic physical and computational principles of this work rest. We begin with a discussion of the calculation of thermodynamic properties from free energies and the idea of the thermal free energy surface. The calculation of this surface is the main task in the prediction of thermodynamic properties from first principles. The construction of this surface through density functional theory and quasi-harmonic lattice dynamics is then discussed along with basic explanations of these theories. The thermodynamic work is extended through the use of strain energy calculations to predict single crystal elastic constants of cubic systems and develop their temperature dependence. Finally a brief overview of molecular dynamics and the derivation of formulas for the calcualtion of elastic constants from statistical ensemble fluctuations is included. These topics represent the current state of the art of first principles and dynamic modeling of temperature dependent material properties.

A. Thermodynamics from Thermal Free Energy

In order to obtain the thermodynamic information about a system it is best to begin with a complete description of the free energy of the system and how it changes with temperature and volume. This is best summarized by the free energy surface in volume/temperature space as shown in (Fig. 1). This colletion of data is so useful since all thermodynamic quantities are derived from the energy via the laws of thermodynamics [9]. For example, from this surface we may find local minima along the temperature axis as $T_0 \rightarrow T_{max}$. The points along this

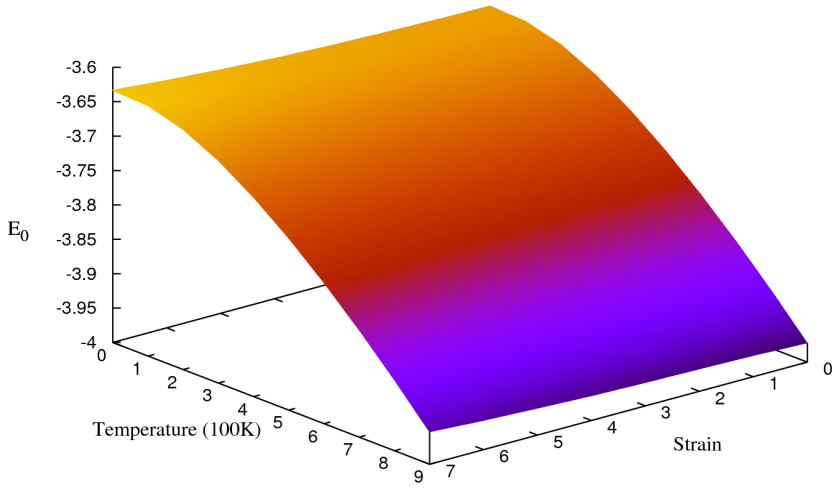


Fig. 1. The thermal free energy surface of aluminum created from seven quasi-harmonic steps.

path represent the thermodynamically stable volumes (those of minimum energy) for the structure along the potential energy curve at a given temperature and serves as a prediction of how the material would expand with increased temperature. This volume/temperature relationship provides a parameterization [10] which will be used according to classical thermodynamics [9] to calculate the temperature dependence of several properties according to established equations such as:

$$S = -\frac{\partial F(T)}{\partial T}, \tag{2.1a}$$

$$H = F(T) - T\frac{\partial F(T)}{\partial T}, \tag{2.1b}$$

$$C_p = T\frac{\partial S}{\partial T}, \tag{2.1c}$$

$$B = -V\left(\frac{\partial^2 F}{\partial V^2}\right). \tag{2.1d}$$

Which come from local slopes and curvatures of the thermal free energy surface.

Several reserach groups have been able to calculate many useful thermodynamic and thermo-mechanical properties for an assortment of materials. For example, Ackland *et al.* have demonstrated the calculation of thermal expansion for W, NiAl, and PdTi [11] with very good correlation to the experimental data. Arroyave *et al.* were also able to employ similar methods to calculate the enthalpy of formation [12] for NiAl and Ni3Al. The phase diagram of the boron nitride system has also been calculated by Hafner and co-workers [13]. It is clear that the development of the free energy surface is therefore of great use. This information can be obtained either through repetitious experimentation or through quantum mechanical calculations, the latter of which is the subject of the next two sections.

B. Quantum Mechanics and DFT

Approaches to materials simulation which are independent of experimental data and rely solely on the basic equations of quantum mechanics and other basic laws of nature are known as first-principles or *ab initio* approaches. The following section is a brief overview of density functional theory (DFT), one of the most recognized and researched first principles methods of materials modeling, and its role in solving quatum mechanical problems. The author acknowledges in advance Hafner [3, 14] and Mehl *et al.* [15] from which this overview was largely created.

A quantum mechanical understanding of a given material system begins with a many-ion, many-electron Schröedinger equation and its corresponding Hamiltonian. The basic form of the time independent Schröedinger equation is:

$$\left(\frac{-\hbar^2}{2m}\nabla^2 + V(\vec{r})\right)\psi(\vec{r},t) = e\psi(\vec{r}), \tag{2.2}$$

for which for the many-body problem [16] becomes:

$$\left[\sum_{i}^{N}\left(\frac{-\hbar^2}{2m}\nabla_i^2 + V(\vec{r_i})\right) + \sum_{i<j} U\left(r_i, r_j\right)\right]\psi(\vec{r_1}, \vec{r_2}\ldots, \vec{r_N},) = E\psi(\vec{r_1}, \vec{r_2}\ldots, \vec{r_N},). \tag{2.3}$$

For anything more than the simplest atoms and smallest systems, any type of solution involving such a Hamiltonian would be insurmountable. The key then is to simplify and make approximations when necessary to make calculations involving a many-body Hamiltonian tractable and accurate.

The first step to simplify the many-body Hamitontian is known as the Born-Oppenheimer (BO) approximation which allows for a decoupling of the electronic and ionic degress of freedom. Since electrons move much faster than ions and have significantly less mass the ions can be considered as stationary. The ion cores thus do not need to be treated quatum mechanically and their contributions to the total energy can be solved using the classical Newtonian equations of motion. A significant inclusion into the BO approximation is that of the so called adiabatic approximation which states that the classically treated ions only move on the potential energy surface of the electronic ground state [14]. These approximations have proven over time to work quite well for modeling many systems, but are incapable of explaining certain phenomena for which electron-ion interactions play significant roles such as superconductivity [3].

With the electronic and ionic degress of freedom seperated and the ionic motion being solved classically the next step in simplifying the quantum mechanical calculations is the reduction of the many-electron Hamiltonian to a physically equivalent/similar system which is easy to solve. The many electron Hamiltonian is a combination of a kintetic energy term, an electron/ion interaction term, and electron/electron interaction term. There are two principal ways of reducing the many-electron problem, Hartree-Fock and density functional theory. While the Hartree-Fock method has certain strengths it was not used in the present

study and the details of its derivation and use is not relevant to this work [17].

Density functional theory (DFT) as outlined by Hohenberg and Sham [18] states that the total energy of a many-electron system that is exposed to an external potential can be expressed as a unique functional of the electron density of the system and that there exists a minimum of the functional which corresponds to the ground state density. Kohn and Sham later developed this theory into a set of equations [19] which form the mathematical foundation for modern day DFT. If we take $n(r)$ to represent the electron density which is obtained from one electron orbitals, the total electronic energy of a system can be expressed as:

$$E[n(r)] = E_k[n(r)] + E_{e-e}[n(r)] + U_{e-i}[n(r)] + E_{XC}[n(r)], \tag{2.4}$$

where E_k represents the kinetic energy of the electrons, E_{e-e} is the electron-electron interaction energy, U_{e-i} is the energy from the electron-ion potential, and E_{XC} arises from the Pauli exclusion principle and other factors. The E_{XC} is called the exchange correlation energy and it is the source of another approximation within DFT.

The problem with the exchange-correlation energy E_{XC} is that there is no exact value or solution for it currently. It is an energy representative of several phenomena, including the need for electrons in the same quantum state to have opposite spin, any error in the kinetic energy term and other factors [15]. Since there is no way of exactly accounting for the exchange-correlation energy we must rely on further approximations in order to provide a closed solution for the total electronic Hamiltonian. The simplest and earliest solution is known as the Local Density Approximation (LDA).

The LDA assumes that the exchange-correlation energy of a given electron can be related to the exchange-correlation energy of an electron in an electron gas of the same density. The Hartree-Fock assumption referred to earlier *can* solve for exchange-correlation of this imaginary electron gas, thus providing an approximation to the actual exchange

correlation [3] energy within DFT. The use of quantum Monte Carlo techniques can then be used to further correct the assumptions [20] and find more accurate exchange-correlation energy functionals. Currently, one of the most popular parametrizations of the exhcange-correlation energy is that of Perdew and Zunger [21] based on the theory of Ceperley and Alder [22]. The LDA is overall satisfactory for predicting crystal structures and macroscopic properties. Mehl *et al.* [15] successfully used the LDA in their calculations of ground state elastic constants. There are some shortcoming of the approximation however, such as a tendency to overestimate cohesive energies resulting in smaller calculated lattice parameters than reality.

A second method for approximating the exchange-correlation energy is known as the Generalized Gradient Approximation (GGA) which was originally presented by Perdew and Wang [23]. The basic idea of the GGA is that the *gradient* of the electron density $\nabla n(r)$ and the exchange-correlation functional E_{XC} are related as opposed to just the density of an electron cloud [24]. The GGA solves many of the shortcomings of the LDA such as atomization energies in hydrocarbons, and better calculation of lattice parameters [25] in many cases. However, there have been reports of overcorrection within the GGA leading to problems such as overestimation of lattice parameters in certain semiconductors [26].

The last thing needed to solve the simplified DFT many-body Hamiltonian are the appropriate wavefunctions to describe the electron-ion interactions. As a matter of principle, in order to make calculation of the system energetics possible there should be a finite set of basis wavefunctions; linear combinations of which would be able to yeild any necessary wavefunctions. The choice of a basis set of wavefunctions is of vital importance since it will limit both the potential accuracy and computational efficiency in the simulation program [3]. One of the most popular approaches for describing the wavefunctions, especially for large solid systems is the plane-wave/pseudopotential approach, which is the method used in the calculations of the present work.

A pseudopotential is an approximation which essentially ignores the rapidly oscillating nature of wavefuctions of electrons within the ionic core [27]. Fig. 2 demonstrates the basic concept of a pseudopotential. The pseudopotential is constructed to exactly follow the true potential of the system outside of the core radius r_c, while within the r_c the pseudopotential simply ignores the many oscillations of the true wavefunction. This simplification of the wavefunction within the core dramatically simplifies the set of basis wavefunctions needed and therefore makes the ensuing calculations possible. The use of the pseudopotential approach is physically a good approximation since it is the valence electrons of atoms that have the predominant impact on how the material behaves.

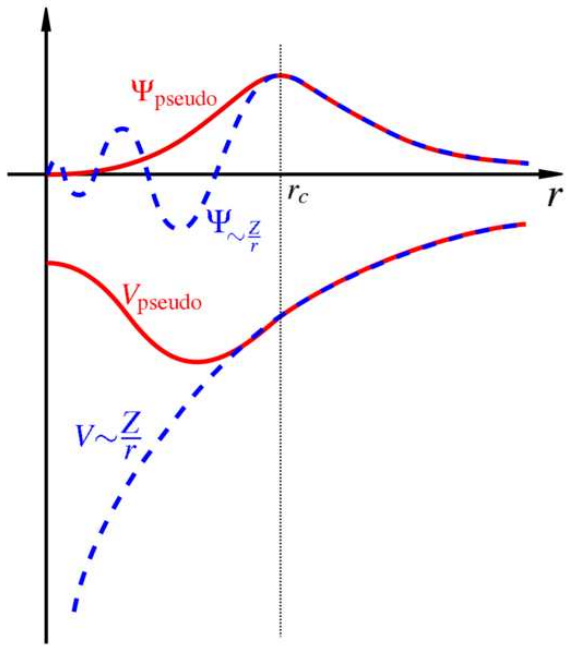


Fig. 2. An illustration of the pseudopotential approximation [28].

For the purposes of the present work the DFT calculations were performed on a widely used massively parallel electronic structure code known as the Vienna Ab-initio Simulation Package (VASP) [29, 30, 31, 32, 33], which will be assumed as sort of a DFT "black box"

taking certain inputs such as crystal structure and atom types and yielding energies and positions of the system under the imposed conditions.

While DFT does allow us to solve many problems in quantum chemistry and solid state physics it is limited to ground state properties corresponding to a tempearture of absolute zero. The theory does not account for thermal contributions to the free energy and the associated vibrations of atoms around their lattice sites. It assumes that the ions are fixed within an electron gas. In order to account for finite temperature effects and the addition of thermal energy into the system the dynamic nature of the system must be accounted for including all the possible degrees of freedom (configurational, vibrational, electronic and magnetic). The systems studied here are ordered crystals for which magnetism is a very minor issue, therefore we neglect both configurational and magentic degrees of freedom and choose to focus on vibrational and electronic contributions to the free energy.

C. Quasi-harmonic Lattice Dynamics

In order to account for the thermal/vibrational contributions to the free energy and derive temperature dependent properties from *ab initio* calculations we must make use of lattice-dynamics [34]. Based on the interaction energies between the atoms and a few statistical arguments it is possible to approximate the behavior of atoms as they vibrate around their given lattice sites [2]. Fig. 3 will serve as a guide throughout this section.

Fig. 3 represents the interatomic potential in an arbitrary crystalline solid. From thermodynamics we know that a given system naturally tends to a state of minimum energy and therefore the atoms will settle upon a given interatomic spacing [1] as shown in the figure. This interatomic potential is key to the understanding of material properties such as thermal expansion, stiffness etc. The thermal expansion is due to the asymmetry of the curve and the stiffness is related to its curvature.

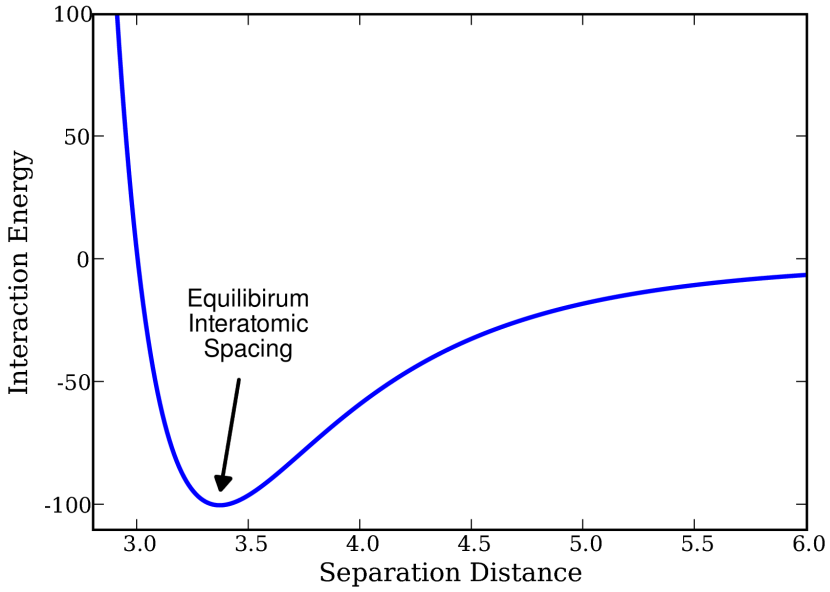


Fig. 3. Potential energy curve for a representative material.

The solution of the temperature dependent properties is really a two part process. The first step is to obtain an understanding of how the atoms vibrate around their lattice sites, the energetics involved and the corresponding modes of vibration within the solid. The second step is to calculate the same vibrational properties at several points of Fig. 3 which represent different temperatures. While this process has been explained by many researchers the author acknowledges in particular the explanation of van de Walle which gives a coherent and simple explanation of the following concepts. The next several paragraphs serve as a summary of one of his papers [2].

We begin by discussing how to calculate the dynamic properties of a given interatomic potential at the equilibrium spacing point at which point the extension to several interatomic radii is elementary. Dynamic properties are found through lattice dynamics calculations where the atoms are treated as point masses and the interatomic forces as springs connecting

the masses. In essence the crystal structure is treated as a 3D matrix of masses and springs which are free to vibrate around their lattice sites and exert forces on each other. The classical equations of motion are then used to predict the behavior of the spring-mass system as it oscillates about equilibrium.

1. Harmonic Approximation

For a single atom vibrating about its lattice site we begin with an anharmonic potential (as seen in Fig. 3 of the form:

$$U = \frac{1}{2}kx^2 + a_3x^3 + a_4x^4 \cdots \tag{2.5}$$

where k represents the classical spring constant, a_i are higher order constants and x is the distance of the atom from its equilibrium position at a given instant. Since lattice dynamics calculations are not reasonable with a potiential of infinite terms we choose to truncate it after the x^2 term. This reduces the potential to a harmonic oscillator which is very easy to work with throughout lattice dynamics and herein arises one source of future error: anharmonic effects on the thermal free energy and derived properties.

If we expand the single atom harmonic potential and add the kintetic energy of the atoms, the total Hamiltonian of the system within the harmonic approximation is given as:

$$H = \frac{1}{2}\sum_i M_i \left[\dot{u}(i)\right]^2 + \frac{1}{2}\sum_{i,j} u^T(i)\Phi(i,j)u(j) \tag{2.6}$$

where i and j represent individual particles in the system and M, $u, \dot{u}$ are the mass, position and velocity of the given particles. The $\Phi(i,j)$ represent the force constant tensors, a set of 3 x 3 matrices that correlate a displacement of an atom j to the corresponding force exerted on atom i:

$$\Phi_{\alpha\beta}(i,j) = \frac{\partial^2 E}{\partial u_\alpha(i)\partial u_\beta(j)} \mid_{u(l)=0\forall l} . \tag{2.7}$$

Each pair of atoms in the system have a complete force constant tensor which describes

their unique interaction. We then compile the $\Phi(i,j)$ into the so called dynamical matrix which is single matrix which represents all the possible pairwise interatomic forces, which are also scaled in proportion to the masses of the respective ions:

$$D = \begin{pmatrix} \frac{\Phi(1,1)}{\sqrt{M_1 M_1}} & \cdots & \frac{\Phi(1,N)}{\sqrt{M_1 M_N}} \\ \vdots & \ddots & \vdots \\ \frac{\Phi(N,1)}{\sqrt{M_N M_1}} & \cdots & \frac{\Phi(N,N)}{\sqrt{M_N M_N}} \end{pmatrix}. \tag{2.8}$$

D has eigenvalues: λ_m which are proportional to the frequencies of the normal modes of oscilation in the system:

$$\nu_m = \frac{1}{2\pi}\sqrt{\lambda_m}, \tag{2.9}$$

which can then be translated to the phonon DOS by:

$$g(\nu) = \frac{1}{N}\sum_{m=1}^{3N}\delta(\nu - \nu_m). \tag{2.10}$$

The phonon DOS is then related to thermodynamics through statistical mechanics:

$$F = -k_B T \ln Z \tag{2.11}$$

and when the appropriate form of the partition function Z is used we get the temperature dependent free energy of the system:

$$F(T) = E_0 + k_B T \sum_m \ln\left[2\sinh\left(\frac{h\nu_m}{2k_B T}\right)\right]. \tag{2.12}$$

(2.12) provides the essential link between lattice dynamics and thermodynamics. Based on the temperature dependent free energy we can then extract various quantities such as entropy and enthalpy and explore phase transition and other phenomena.

2. Quasi-harmonic Corrections

While much valuable information can be obtained through the harmonic approximation and associated lattice dynamics, the model is incomplete. The harmonic approximation does not account for thermal expansion with increasing temperature. The phenomenon of thermal expansion is rooted in the asymmetry of the interatomic potential energy curve [1] for a given material. Being symmetric, a harmonic potential excludes valuable information about the solid at elevated temperatures. Of particular interest is the softening which occurs as the atoms grow farther apart and the effect of this on mechanical properties such as stiffness and yield strenght. In order to account for thermal expansion we make use of the quasi-harmonic approximation [10].

The quasi-harmonic approximation is simply the extension of the harmonic approximation to several points on the potential energy curve [34] as shown in Fig. 4. These points are represented by the same lattice as the ground state already calculated, but with slightly scaled volumes, to simulate thermal expansion with temperature. Since they are based off of a relaxed ground state, any internal degrees of freedom for these expanded volumes are also relaxed. For example, a typical quasi-harmonic calculation will scale the volume up to 4% including steps of 1,2,3 and 4 %. If we perform quasi-harmonic lattice dynamics calculations at each of those points the potential energy curve would be constructed as show in the figure. The free energies from lattice dynamics at each quasi-harmonic step can then be combined to construct a free energy surface in Volume/Temperature space [35] from which the thermodynamic properties can be extracted as shown in Fig. 1.

This free energy surface is the key to the calculation of temperature dependent properties. The local minima, slopes and curvatures of the surface provide the necessary data for extracting many useful thermodynamic and thermo-mechanical properties.

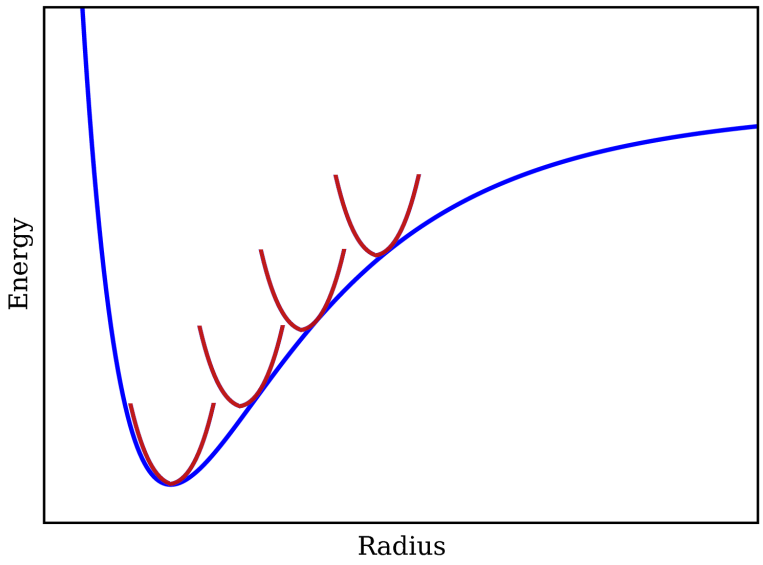


Fig. 4. The quasi-harmonic approximation illustrated on the potential energy curve of Fig. 3.

3. Anharmonic Contributions to the Free Energy

Both the harmonic and quasi-harmonic approximation are unable to completely model the free energy of a system since they are approximations and admittedly exclude certain phenomena. One of the primary excluded factors are some anharmonic contributions to the free energy due to the truncation of (2.5) after the quadratic term [36]. While some anharmonic effects are accounted for within the quasi-harmonic theory due to the volume dependence of phonon modes [10], certain intrinsic anharmonic effects are neglected which become increasingly relevant at elevated temperatures [37]. In this section we examine two approaches for including anharmonicity corrections into first principles calculations from quasi-harmonic lattice dynamics.

a. Phenomenological Approach of Wallace

Wallace [36] developed a lattice dynamics approach for calculating the intrinsic anharmonic free energy of a solid:

$$F_A = A_2 T^2 + A_0 + A_{-2} T^{-2} + L. \tag{2.13}$$

where the A coefficients are dependent on the configuration of the lattice only. The problem with this formula is that there is currently no way of easily calculating any of the A coefficents except for the A_2 which is estimated by a fit to empirical data to be:

$$A_2 = \frac{3k_B}{\Theta} \left(0.0078 \left\langle \gamma \right\rangle - 0.0154\right). \tag{2.14}$$

Another difficulty with this method is that it breaks down at low temperatures [37]. If we look at the specific heat at constant pressure C_p and include the anharmonic free energy according to this method we find that $C_p(T \to 0) \propto T$. From quantum mechanics we know that $C_p(T \to 0) \propto T^4$ and therefore we can say that this approach breaks down as $T \to 0$.

b. Thermodynamic Perturbation Theory of Oganov

Building on the work of Wallace, Oganov and Dorogokupets [37] have developed another formula for the inclusion of intrinsic anharmonicity. Based on thermodynamic perturbation theory, their expression for the anharmonic free energy is:

$$F_A = \frac{3k_B a}{6} \left[\left(\frac{1}{2}\theta + \frac{\theta}{\exp(\theta/T) - 1} \right)^2 + 2 \left(\frac{\theta}{T} \right)^2 \frac{\exp(\theta/T)}{(\exp(\theta/T) - 1)^2} T^2 \right] \tag{2.15}$$

where $a = \frac{1}{2} A_2$ from (2.14) and θ is the Debye temperature from the quasi-harmonic calculations. This expression has the advantage that it is valid at all temperatures and can be included into the total free energy as a simple addition term.

4. Electronic Degrees of Freedom

The quasi-harmonic approximation and anharmonic corrections are able to account for the vibrational degrees of freedom within the material. The incorporation of electronic degrees of freedom as outlined by Asta *et al.* [38] and Arroyave *et al.* [12] is a fairly straitforward addition to the total free energy of a system. Take $n(\varepsilon, V)$ as the electronic density of states at a given quasi-harmonic volume, and f as the Fermi function the electronic contribution to the free energy can be obtained by combining the following:

$$F_{el} = E_{el} - TS_{el}, \tag{2.16a}$$

$$E_{el}(V,T) = \int n(\varepsilon, V)\, f\varepsilon \, \mathrm{d}\varepsilon - \int^{\varepsilon_F} n(\varepsilon, V)\, \varepsilon \, \mathrm{d}\varepsilon, \tag{2.16b}$$

$$S_{el}(V,T) = -k_B \int n(\varepsilon, V)\, [f \ln f + (1-f)\ln(1-f)]\, \mathrm{d}\varepsilon. \tag{2.16c}$$

This term may then simply be added to the total expression for the free energy as a function of temperature [39].

D. Density Functional Theory Prediction of 0 K Elastic Constants

Extensive work on 0 K elastic constants using density functional theory has been done by Mehl *et al.* This section will provide a brief overview of one of their detailed publications on the matter [15]. The energy of an isotropic crystal under a finite strain and zero pressure is given by:

$$E(e_i) = E_0 + \frac{1}{2} V \sum_{i=1}^{6} \sum_{j=1}^{6} C_{ij} e_i e_j + O\left[e_i^3\right], \tag{2.17}$$

with E_0 representing the energy of the unstrained crystal, V the volume, e_i and e_j represent a strain tensor and C_{ij} are the elastic constants. By choosing specific strain states (e_i and e_j) for the lattice and by using DFT to calculated the energy at those strain states we can then extract the C_{ij}.

For an arbitrary crystal there are at most 21 independent C_{ij}. Symmetry arguments reduce this number to three for a cubic system; C_{11}, C_{12} and C_{44}. We must therefore choose appropriate strain tensors that will allow us to seperate and solve for these three C_{ij}. In theory we would need to perform at least three sets of DFT strain calculations, one for each C_{ij}, to solve for the three elastic constants. Since we are working with cubic systems we use a relationship between the bulk modulus and the two elastic constants to reduce computation time

$$B = \frac{1}{3}\left(C_{11} + 2C_{12}\right). \tag{2.18}$$

This is computationally advantageous since we can predict the bulk modulus from a quasi-harmonic model by obtaining a solution to the Birch equation of state:

$$\begin{aligned} E_{Birch} &= E_0 + \frac{9}{8}B_0V_0\left[\left(\frac{V_0}{V}\right)^{2/3} - 1\right]^2 \\ &\quad + \frac{9}{16}B_0V_0\left(B_0' - 4\right)\left[\left(\frac{V_0}{V}\right)^{2/3} - 1\right]^3 \\ &\quad + \sum_{n=4}^{N}\gamma_n\left[\left(\frac{V_0}{V}\right)^{2/3} - 1\right]^n. \end{aligned} \tag{2.19}$$

We choose a volume conserving orthorhomic strain on the lattice:

$$\begin{pmatrix} x & 0 & 0 \\ 0 & -x & 0 \\ 0 & 0 & \frac{x^2}{(1-x^2)} \end{pmatrix}$$

which reduces (2.17) to:

$$\Delta E(x) = V(C_{11} - C_{12})x^2 + O[x^4], \tag{2.20}$$

which we can combine with (2.18) to solve for and seperate C_{11} and C_{12}.

The calculation of C_{44} is obtained in a similar manner, but can be found independently

by applying a volume conserving monoclinic strain:

$$\begin{pmatrix} 0 & x & 0 \\ x & 0 & 0 \\ 0 & 0 & \frac{x^2}{(4-x^2)} \end{pmatrix}$$

which reduces (3.8) to:

$$\Delta E(x) = \frac{1}{2} V C_{44} x^2 + O[x^4]. \tag{2.21}$$

For each set of strain calculations we do the following:

1. Strain the primitive lattice vectors for several strain values for the defined strain tensors
2. Use the DFT software to calculate the total energy of the system for each strained structure
3. Fit the strain/energy data to (3.8) and extract the C_{ij}

Following this basic procedure, Mehl *et al.* [15] were able to accurately predict the elastic constants of several cubic and tetragonal systems with considerable correlation to experimental values at 0 K.

Ackland *et al.* were the first that we are aware of to extend this basic approach to the calculation of finite temperature elastic constants using DFT [11]. They performed some thermodynamics calculations on tungsten and at each quasi-harmonic step applied tetragonal strains to the system. They were then able to couple the thermal expansion to the data from their strain energy surface to extract the elastic constants C_{11} and C_{12} with good accuracy. They did neglect both anharmonic and electronic contributions to the free energy, but this didn't weaken the correlation between the calculated elastic constants and the experimental counterparts. These findings were mostly used to validate certain approximations in their calculations of thermodynamic properties and they didn't report any findings for the elastic constants of the other systems being studied.

While this method of calculating the ELC of a material is helpful, there are systems for which the preceeding approach would break down, particulary at high temperatures. This is largely due to the anharmonic contributions to the free energy which are either partially, or completely negelected. Another tools which can be used to calculate the elastic constants of materials, and which can include in anharmonicity exactly [40] is molecular dynamics.

E. Molecular Dynamics Prediction of Elastic Constants

1. Molecular Dynamics Background

Molecular dynamics (MD) is a classical materials modeling tool which solves the Newtonian equations of motion over several time steps to track the movement and interaction of particles (atoms or molecules) within a defined unit cell [41]. The simulation is controlled by two things, the ionic interaction potential, and the boundary conditions imposed on the system.

The ionic interactions are governed by a user defined potential energy function U, also known as a force field. One of the key challenges in MD simulations is the development of adequate force fields and the knowledge of which force field to use for a given simulation. Typically, the force fields are developed by a fitting of experimental data to an equation although sometimes quantum mechanics calculations are used to fine tune the fitting parameters. Examples of force fields that have been used in published works are the Morse potential which has been used for example to model strain fluctuations in nickel [40]:

$$\Phi_w(r) = \xi \left(e^{-2\alpha(r-r_0)} - 2e^{-\alpha(r-r_0)} \right), \tag{2.22}$$

and the Lennard-Jones potential which has been used in an argon simulation [42]:

$$U(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right]. \tag{2.23}$$

where ξ, α, r_0, ε, and σ are parameters obtained from fits to experimental values. There are

many potentials that can be used depending on the system and conditions being simulated. In choosing a potential there are several factors that must be taken into account such as the range of interaction and directionality of bonding within the material. As an example, a plot of a Lennard-Jones potential for some arbitrary constants is included as Fig. 5.

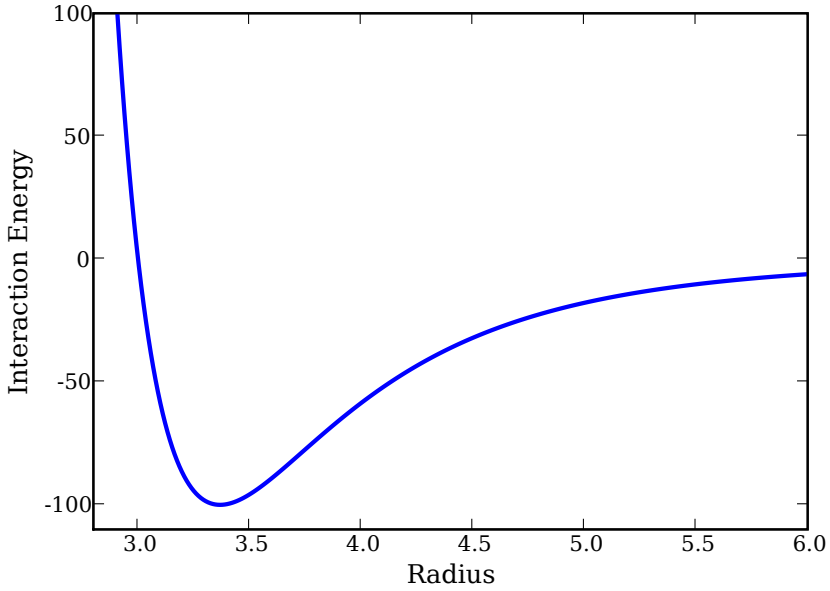


Fig. 5. Plot of a Lennard-Jones potential. The variables were set for $\sigma = 3.0$, and $\varepsilon = 100$.

The various potentials are suited for different tasks. For more complex systems such as metals with long range interactions or ceramics which are highly directional different force fields and techniques would need to be employed such as the embedded atom method [43] and long range Finnis-Sinclair potential [44] to adequately describe the system. For simplicity we restrict the current discussion to a simple pair potential.

Since MD employs Newton's equations of motion it inherently ignores any quantum mechanical effects. This is problematic for some scenarios where the quantum mechanical details are either necessary or advantageous towards solving a given problem. This exculsion

of quantum effects can also be a strength of MD simulations in other situations where this omission simplifies an overly complex or otherwise impossible to calculate system. Certain phenomena can only be described quantum mechanically and these would be impossible to determine through MD simulations. On the converse, MD calculations should serve fine for the prediction of many bulk properties such as elastic constants, provided of course that an accurate potential energy function is used. Another advantage of MD is that it can account for anharmonic effects and thermal excitations exactly [40] within temperatures where the quantum effects are small compared to those of classical mechanics.

Each molecular dynamics simulation requires that some degrees of freedom of the system be constrained and others allowed to relax over time. These constraints are the boundary conditions of the problem and define what is called an MD *ensemble* [45]. Once the constraints are applied the remaining degrees of freedom are released and the system is allowed to evolve over time according to the equations of motion, the laws of thermodynamics, and the ensemble constraints; thus simulating as realistically as possible what would happen to the system in real life. If for example, the number of atoms is to be conserved, any atom that leaves the molecular dynamics cell for any reason must be replaced. Other examples of ensemble parameters are the total energy (E), the volume of the unit cell (V), and the enthalpy (H). In the literature the ensembles are usually denoted by a collection of letters representing the properties that are to be constrained such as EVN, which is also known as the microcanonical ensemble. Here E is the energy of the system, V the volme and N the number of particles, all of which are to be kept constant at each time step of the MD run. The definition of an ensemble is critical in that it defines the 'rules' for the simulation and consequently the data that can be extracted. It was the development of new ensembles in the late 70's and early 80's that allow for the calculation of elastic constants from MD simulation [46] [47].

2. Elastic Constants from Molecular Dynamics Strain Fluctuations

In order to obtain macroscopic properties from MD we must look at the time behavior statistics of many particles over time. In traditional molecular dynamics the microcanonical or EVN ensemble [45] is used to track the movement of a given number of atoms contained in a fixed volume over time while maintaining constant system energy. Andersen [46] developed a different approach to MD which maintains constant enthalpy, pressure and number of particles, or the HPN ensemble. The key difference between EVN and HPN is that the HPN ensemble allows the size/volume of the molecular dynamics cell to change with time thus introducing a new dynamic variable.

Parinello and Rahman built on Andersen's thoery allowing both the *size* and *shape* of the cell to change [47] throughout the MD run. They took **a**, **b**, and **c** as the vectors which span the molecular dynamics cell and define $h = (\mathbf{a}, \mathbf{b}, \mathbf{c})$. The new variable h can be treated as a dynamical variable in the MD simulation [45]. The result is an HtN ensemble where t is the thermodynamic tension and H is the total enthalpy of the system. The introduction of the tension property t is what provides the link between MD and the theory of elasticity and thus the calculation of elastic constants.

The elastic constants can be found from the fluctuation of the strain matrix ϵ over an MD run for HtN and TtN ensembles. The strain matrix is defined as

$$\epsilon = \frac{1}{2}(\tilde{h}_0^{-1} G h_0^{-1} - 1) \tag{2.24}$$

where $G = \tilde{h}h$. Parrinello and Rahman showed that the fluctuations of the strain tensor in an HtN ensemble relate to the adiabatic elastic constants [48], [49] by

$$\delta\left(\epsilon_{ij}, \epsilon_{kl}\right) = \left(\frac{k_B T}{V_0}\right)\left(C^S\right)^{-1}_{ij,kl}, \tag{2.25}$$

where V_0 is the volume of the unstrained MD cell, C^S is a 9x9 matrix of the adiabatic elastic

moduli, and ϵ_{ijkl} is the strain tensor. C^S is defined as:

$$C^S_{ij,kl} = \left(\frac{\partial \sigma_{kl}}{\partial \epsilon_{ij}}\right)_S \tag{2.26}$$

with σ_{kl} representing the stress tensor. (2.25) has shown poor statistical convergence [42] and this has lead to the development of alternative fluctuation formulae using other MD ensembles.

As an alternative to using strain fluctuaions to calculate elastic constants, Ray and Rahaman [50] have developed a formalism around the *EhN* enseble which involoves tracking the fluctuations in stress over the MD run. There are two main advantages of this method over the strain fluctuation method. First is the statistical convergence of the elastic constants [45], and the second is that the calculated elastic constants are broken up into a summation of several physically significant terms [40, 51, 52],

$$\begin{aligned} C_{\alpha\beta\nu\tau} &= -\frac{V_0}{k_B T}\left(\langle \sigma^B_{\alpha\beta}\sigma^B_{\nu\tau}\rangle - \langle \sigma^B_{\alpha\beta}\rangle\langle \sigma^B_{\nu\tau}\rangle\right) \\ &\quad + \langle \Upsilon_{\alpha\beta\nu\tau}\rangle + \frac{2nk_BT}{V_0}\left(\delta_{\alpha\nu}\delta_{\beta\tau} + \delta_{\alpha\tau}\delta_{\beta\nu}\right). \end{aligned} \tag{2.27}$$

The first term is the due to the fluctuations of the microscopic stress tensor. The second term is called the Born term which involves second derivatives of the potential energy and the final term is the kinetic energy contribution which is directly related to the temperature of the system [40]. These terms all have different contributions to the total elastic constants. For example, in one of their studies, Cagin and Pettitt [40] demonstrate that for nickel the Born term overestimates the total ELC, the fluctuation term has a significantly smaller yet reducing effect and the kinetic energy term is usually quite small in comparison to the others.

The fluctuation formulas, can be implemented using the information about positions, forces, etc. of the atoms over a sufficiently long MD run to predict the desired properties. It is important to have sufficiently long MD runs so as to provide a large enough sample size to

validate the use of the statistics used in the fluctuation formulae [40]. From these methods various properties have been predicted and reported in the literature such as specific heat and thermal expansion [49]. Further work has been done to find fluctuation formulas for the microcanonical ensemble and this has proven successful in predicting the elastic constants of argon [42]. More recently, others have built on the foundation developed by Ray, Parinello, Rahman and others and formalized other approaches for finding elastic constants through the use of an alternate fluctuation formula [51, 52] for the cannonical ensemble.

While both MD and DFT provide great tools for the calculation and prediction of material properties, the ultimate benchmark of their success is their ability to be used in real-world applications. In order for simulations to provide meaningful results it must be shown that the simulations produce results that match up with reality. The only way of validating theoretical approaches and computational methods in materials simulation is how the results of the calculations ultimately correlate with experimentally obtained data.

F. Experimental Determination of Single Crystal Elastic Constants

The main purpose of this work is to develop and validate a method for the prediction of the temperature dependence of single cystal elastic constants from first principles. The validation of that method comes by comparing the results of several systems with experimentally tabulated values. This section therefore briefly describes the procedure followed for the experimental determination of elastic constants.

The elastic constants (ELC) of a solid are propotional to the square of the velocity of a wave as it propogates through the material:

$$C \propto \rho V^2. \tag{2.28}$$

In (2.28) C is an elastic constant and V is the velocity of a wave propagating through the

material. There are several different symmetry directions in a solid, each with different ELC. In order to calculate the different ELC, an appropriate wave must simply be propagated in the corresponding direction and the velocity measured. Table I is taken from [53] and demostrates the relationships between directions, wave direction, and the corresponding elastic constants.

Table I. Relationships between wave propagation modes and elastic moduli.

Velocity	Propogation direction	Particle motion direction	Relation of velocity to elastic consant
v_1	[110]	[110]	$\rho v_1^2 = \frac{1}{2}(C_{11} + C_{12}) + C_{44}$
v_2	[110]	[001]	$\rho v_2^2 = C_{44}$
v_3	[110]	[1$\bar{1}$0]	$\rho v_3^2 = (C_{11} - C_{12})/2$
v_4	[100]	[100]	$\rho v_4^2 = C_{11}$
v_5	[100]	in (100) plane	$\rho v_5^2 = C_{44}$

In order to measure the elastic constants of a pure, isotropic, homogeneous material we must begin with a single crystal. These may be grown in any one of a number of ways, but their crystallographic homogeneity is vital to the prediction of single crystal ELC. The presence of grain boundaries or defects would affect the wave velocities, cause scattering and interference effects and hence the final calculated elastic constants would not be accurate. The specimen to be tested is first analyzed for purity in order to report as close to perfect a specimen as possible [54]. Typical impurities include both substitutional ions or interstitial gasses,and their concentration must be kept extremely low in order to provide the most accurate data. Once the specimen composition and purity have been verified it is either oriented within an experimental apparatus or cut and polished to shape with respect to its crystallographic directions and then mounted accordingly. The crystallographic directions

of the specimen are found by Laue X-ray diffraction and the wave modes and corresponding ELC are measured in the appropriate directions [55]. If the specimen needs to be cut or polished great care must be taken to not initiate recrystallization or cold working in the crystal [53]. Once the specimen is shaped and aligned a thermocouple is attached [54] and the system is placed in a vacuum/furnace setup [55] so as to adjust pressure and temperature according to the nature of the experiment.

Different techniques have been used to determine the velocity of a wave through a crystal. Many works tend to use waves in the kHz range [56] while others (typically more recent) works use waves the MHz range. At lower frequency the measurments are sensitive to geometry and dislocation motion [53], so typically high frequecies are preferred. Regardless of the type of wave chosen and the experimental setup used, the basic principle is the same: propagate a wave through the crystal in a given crystal direction and then calcualte the C_{ij} according to (2.28).

The C_{ij} can be calculated at various temperature/pressure combinations and then included in experimental databases such as that of Simmons and Wang [57] which is commonly referred to in the literature.

CHAPTER III

AB INITIO THERMODYNAMIC AND ELASTIC PROPERTIES OF ALUMINUM AND TUNGSTEN AT FINITE TEMPERATURES

A. Synopsis

An ab initio method for the prediction of temperature dependent thermodynamic and elastic properties of non-magnetic metals is presented and validated for aluminum and tungsten. Through quasi-harmonic lattice dynamics and density functional theory calculations a free energy surface in temperature/volume/strain space is created, the local curvatures and slopes of which yield the properties of interest. Anharmonic contributions to the free energy are examined and their effect on final calculated properties are shown to be minimal for aluminum but very significant in tungsten at temperatures above 60% the melting point. Overall the calculated properties show good correlation with experiment and demonstrate the need for a better accounting of anharmonicity in thermal free energy calculations from DFT.

B. Introduction

Current and future technological challenges require new materials capable of meeting increasingly demanding operating conditions. Usually, materials development tends to be costly and time-consuming. Fortunately, increasingly sophisticated theoretical tools and modeling approaches, coupled to the exponential growth of computing power have led to the development of computational materials science as a field on its own [58, 59]. The use of such computational approaches have lead to an acceleration of the materials development process [60, 61], mainly through the reduction of the parameter search-space to be explored in order to arrive at an optimal materials solution.

One of the primary challenges facing the materials modeling community is the need to account for and simulate physical phenomena occurring at multiple time and length scales. On one extreme of the time/size spectrum there are atomic processes–atomic migration, for example—which occur on the order of picoseconds, while at the other end there are phenomena involving macroscopic changes in materials over the course of years–such as creep. It is therefore of fundamental interest to develop theoretically sound links between the various time and length scales involved in materials modeling.

For example, at the mesoscale, the phase-field method has proven enormously successful at describing not only the microstructural evolution of materials but also their behavior resulting from the coupling of their response to externally applied fields–elastic, chemical, and magnetic [62]. Quantitative phase-field models[63], however, can only be obtained when accurate parameters for the description of their thermodynamic, kinetic, elastic, magnetic, electric properties are available. Traditionally, such parameters have been obtained experimentally. Unfortunately, a limitation to the phase-field approach is that there are various cases in which accurate information about the properties of a given phase are unknown either due to the cost and difficulty in obtaining experimental measurement or in some cases physical impossibilities in measuring the desired properties.

Molecular dynamics is yet another useful approach to materials modeling, allowing the prediction of properties such as elastic constants [42][40] or examining phenomena involving the collective atomic behavior, such as melting[64] and diffusion processes[65]. One of the greatest challenges in molecular dynamics simulation is finding interatomic potentials that accurately describe the system. Like the input parameters for phase-field models, these potentials are also parameterized from experimental data. Again, limitations exist as to what systems can be simulated with molecular dynamics due to the lack of experimental data or adequate parametrization of the appropriate interatomic reactions.

One possible way to obtain the needed input parameters for a given model is to sup-

plement the empirical databases with quantum mechanical calculations of the electronic structure of materials. These first-principles methods—not completely parameter-free as they in turn are developed based on a number of approximations—provide the means to investigate the interactions between atoms and molecules. Knowledge of these interactions in turn allow a better understanding of the underlying physical basis for the relationship between atomic features and macroscopic behavior[3]. The information resulting from these calculations can in turn be used to develop more quantitative models of physical phenomena occurring at the meso/macroscale.

The purpose of our work is to develop a model to predict the thermodynamic and thermo-mechanical properties of materials at finite temperature based solely on first principles calculations. These properties can either be used in different modeling techniques as input parameters or provide guidance to others in narrowing the search domain for high temperature materials, for example.

Considerable work on finite temperature free energies and thermodynamics have been done by many groups [38, 2]. Mehl *et al.* have done extensive work on the calculation of single crystal elastic constants at 0 K [15]. They have examined several pure and binary intermetallic systems with either cubic or tetragonal symmetry in the unit cell with considerable correlation to experimentally obtained values. In their work on the thermodynamics of tungsten, Ackland *et al.*[11] also examined the temperature dependence of the single crystal elastic constants with marked success but did not extend the work to other elements or compounds.

In the current work we outline a method for determining several thermodynamic and thermo-mechanical properties of two pure metals, aluminum and tungsten and compare the calculated results with experimental data in order to validate the method as well as demonstrate some of its limitations. We use density functional theory (DFT) [19] and quasi-harmonic lattice dynamics[2] to calculate free energy surfaces that include electronic,

vibrational, and anharmonic contributions. From this free energy data we are able to extract temperature dependent thermodynamic data such as enthalpy, entropy, Debye temperature, and the Grunesien constant for each system. We also implement quasi-harmonic theory to obtain temperature/volume correlations which we couple with 0 K elastic constants to obtain the finite temperature elastic constants.

C. Methodology

1. Free Energy Calculations at 0 K

The ground state energies and electronic structures were calculated using density functional theory [19](DFT), within both the local-density [21] (LDA) and generalized-gradient [24][66] (GGA) approximations. The Vienna Ab-Initio Simulation Package (VASP) was used [32, 33] to perform the DFT calculations using projector augmented-wave (PAW) pseudopotentials [67, 29]. The electronic configurations used were $3s^2 3p^1$ for aluminum and $6s^2 5d^4$ for tungsten.

The structures were initially optimized by performing a relaxation calculation wherein all degrees of freedom were allowed to relax using the first order Methfessel-Paxton smearing method [68]. A second self-consistent static calculation was performed on the relaxed structure using the tetrahedron smearing method with Blöchl corrections [67]. These static calculations were performed to a precision of six significant figures with an energy cutoff of 350 eV and $1x10^4$ k-points per reciprocal atom. The energy cutoff and k-point density ensured excellent convergence in the total energies calculated.

2. Contributions to the Free Energy

In order to extend the use of DFT to the prediction of finite temperature thermodynamic and thermo-mechanical properties an accurate expression for the total free energy of the

system throughout the temperature range of interest is necessary. In principle, all thermally excited degrees of freedom—vibrational, electronic, magnetic—, as well as their contributions to the temperature-dependent free energy must be taken into account. However, the systems examined in this work warrant the neglect of magnetic DOF, focusing instead on electronic and vibrational DOF. Below, a brief overview of the contributions of the latter to the total free energy is presented. Detailed descriptions of the methodology to calculate these contributions are available elsewhere [38, 2, 39].

a. Vibrational Contributions

There are two primary ways to calculate vibrational properties from first principles; linear response theory (LRT)[69] and the supercell (SC) approach [70]. In the SC method, atoms are perturbed from their equilibrium positions and the resulting restoring forces are used to calculate the force constants, which in turn correspond to second derivatives of the crystal potential with respect to atomic displacements[2]. LRT, on the other hand, is a more accurate approach as it calculates the force constants from second derivatives of the electronic crystal energies[69]. While accurate, this last approach is computationally expensive and cannot be easily applied to any ab initio code. Moreover, it has been shown that, at least for simple systems [12], the SC method yields results equivalent to LRT. In both methods, the force constants are then used to construct the dynamical matrix—essentially the force constant matrix normalized by the mass of the interacting atoms—, whose eigenvalues correspond to the frequencies of the normal (harmonic) modes of oscillation of the crystal [2].

In this work, the SC calculations are performed using the ATAT software package [71, 72, 73]. The software assists with the harmonic and quasi-harmonic lattice dynamics calculations by creating the necessary supercell perturbations based on the underlying primitive cells. First-principle calculations using the VASP code were then used to calculate the interatomic forces. Force-constants and the corresponding dynamical matrix are then obtained

by relating the resulting forces to atomic displacements. In the SC method, the range of the force-constants considered depends on the size of the supercell used. In this particular work, the supercells used were comprised of 32 atoms and the force constants were calculated over a range of approximately half the total supercell, in order to reduce periodicity artifacts. The selected supercell sizes allow the inclusion of up to 3rd nearest neighbor interactions.

From the phonon DOS and usual statistical mechanics formulas, it is possible to calculate the vibrational contributions to the free energy of the crystal, as a function of temperature [2]:

$$F_{vib}(T) = k_B T \int_0^\infty \ln\left[2\, sinh\left(\frac{h\nu}{2k_B T}\right)\right] g(\nu)\, d\nu \tag{3.1}$$

where $g(\nu)$ represents the phonon density of states, h is Planck's constant and k_B is Boltzmann's constant. At constant volume, the harmonic approximation is accurate, especially at low temperatures. As the temperature increases, under constant pressure conditions, the contributions due to thermal expansion must be taken into account, as larger interatomic spacings in general lead to a softening of the structure, increasing its entropy [2]. Harmonic potentials, being symmetric, cannot account for thermal expansion and further corrections must thus be applied.

In order to account for volume expansion, the *quasi-harmonic approximation* can be used. This simple correction consists of performing harmonic calculations at different volumes. These calculations yield a free energy surface in volume/temperature space, the locus of the free energy minima as a function of temperature then yields the volume thermal expansion. Accurate prediction of this relationship is critical for application of the quasi-harmonic approach to predict temperature-dependent properties. Unless otherwise noted, in this work we examined 7 volume expansions spanning -2% to +4% at 1% increments of the 0 K equilibrium volume.

b. Electronic Contributions

The electronic contributions to the free energy ($F_{el}(V) = E_{el}(V) - TS_{el}(V)$) are calculated—within the one-electron approximation [38]—through the integration of the electronic density of states, resulting from the self-consistent calculation of the electronic structure at each of the volumes considered in the quasi-harmonic approximation, according to[39]:

$$E_{el}(V,T) = \int n(\varepsilon,V)\, f\varepsilon d\varepsilon - \int^{\varepsilon_F} n(\varepsilon,V)\, \varepsilon d\varepsilon \tag{3.2}$$

$$S_{el}(V,T) = -k_B \int n(\varepsilon,V)\left[f \ln f + (1-f)\ln(1-f)\right] d\varepsilon \tag{3.3}$$

where $n(\varepsilon,V)$ is the electronic DOS and f is the Fermi function. The electrochemical potential is calculated self-consistently to ensure that at each temperature the total number of electrons is conserved. Such self-consistent calculations are computationally expensive. To reduce computational costs, the exact electronic free energy is calculated only at discrete temperature intervals and then is fitted to a quadratic expression, as suggested by Asta *et al.* [38]. It is important to note that this method does exclude electron-phonon interactions, as well as direct changes in the electronic structure due to increased temperatures.

c. Anharmonic Corrections

One shortcoming of quasi-harmonic lattice dynamics is that it neglects anharmonic effects, which result from non-vanishing third and higher-order derivatives of the crystal potential with respect to atomic displacements [36]. Within the context of phonon dynamics, such higher-order contributions result from phonon-phonon interactions, and normally occur at temperatures higher than two-thirds of the crystal's melting point[36]. In order to obtain a more accurate representation of the total free energy it is possible, in principle, to incorporate anharmonic contributions to the temperature-dependent free energy of the crystal in a direct

fashion [13].

Based on higher-than-second-order expansions of the crystal potential, Wallace[36] was able to develop an exact expression for the anharmonic free energy:

$$F_{anhar} = A_2 T^2 + A_0 + A_{-2} T^{-2} + L \tag{3.4}$$

with:

$$A_2 = \frac{3k_B}{\Theta}(0.0078 \langle\gamma\rangle - 0.0154) \tag{3.5}$$

where γ represents the Grüneisen parameter and the coefficients are based on a fit to empirical data. Unfortunately, the last three terms of (3.4) cannot be easily determined, even empirically. A crude approximation, based more on ignorance rather than knowledge, is just to ignore them and consider only anharmonic corrections quadratic in temperature. Yet another limitation of an application of (3.4) is the fact it is valid only in the classical limit and breaks down as the temperature approaches 0 K. For example, if we include this expression for anharmonic free energy into an expression for C_p we find that $C_p(T \to 0) \propto T$, but we know that in the quantum limit $C_p(T \to 0) \propto T^4$[37]. In principle, this would not constitute a serious problem since anharmonic effects are only important at high temperature. However, this problem must be resolved if anharmonic contributions are to be added to the vibrational+electronic free energies from 0K on.

Recently, Oganov[37] was able to extend the approach of Wallace and develop an expression which is valid over all temperatures. Using thermodynamic perturbation theory he obtained an expression for the anharmonic free energy as a function of temperature:

$$\frac{F_{anh}}{3nk_B} = \frac{a}{6}\left[\begin{array}{c}\left(\frac{1}{2}\theta + \frac{\theta}{\exp(\theta/T)-1}\right)^2 \\ +2\left(\frac{\theta}{T}\right)^2 \frac{\exp(\theta/T)}{(\exp(\theta/T)-1)^2} T^2\end{array}\right] \tag{3.6}$$

where a is proportional to A_2 of Wallace by a factor of 1/2 and θ corresponds to the high-temperature Harmonic Debye temperature, defined as $\theta = \frac{\hbar}{k_B}\left(\frac{5}{3}\langle\omega^2\rangle\right)^{1/2}$ [36, 13]. In this work, (3.6) is calculated at each volume considered in the quasi-harmonic correction, with the parameter a calculated according to (3.5), with Θ and γ calculated directly from first-principles.

d. Total Free Energy and Finite Temperature Thermodynamics

The total temperature-dependent free energy is simply calculated by adding the various energy terms. This results in a free energy surface $F\left(V,T\right)$ [39]:

$$F(V,T) = E_{0K}(V) + F_{vib}(V,T) + F_{el}(V,T) + F_{anhar}(V,T) \tag{3.7}$$

where E_{0K} is the cold curve energy as a function of volume from the quasi-harmonic DFT calculations, F_{vib} and F_{el} are the vibrational and electronic contributions to the free energy calculated over the specified temperature range at each volume of the quasi-harmonic approach and F_{anhar} is the anharmonic free energy considering the correction by Oganov [37]. The zero-pressure free energy, and thermal expansion coefficient—is simply calculated by identifying the locus of the minima of this surface as a function of temperature. The temperature dependent values for enthalpy, entropy and specific heat can be obtained from partial derivatives to the thermal free energy according to classical thermodynamics [74]. The Bulk modulus is in turn calculated by fitting, at each temperature, the volume-dependent free energy to an equation of state [75].

3. Finite-Temperature Elastic Constants

Mehl *et al*. have done extensive work on the calculation of 0 K elastic constants using DFT. In this work, their procedure [15] is applied and then is coupled with the current approach of

finite temperature thermodynamics in order to calculate the temperature dependence of the elastic constants [11]. The elastic constants,C_{ij}, relate changes in the energy of the crystal with respect to finite strains:

$$E\left(e_i\right) = E_0 + \frac{1}{2}V\sum C_{ij}e_ie_j + O\left[e_i^3\right] \tag{3.8}$$

For an arbitrary system there are at most 21 independent C_{ij}. Symmetry arguments reduce this to 3 independent elastic constants in the case of a cubic lattice such as for Al and W; C_{11}, C_{12} and C_{44}. In order to isolate a particular C_{ij}, the lattice must be strained and the resulting changes in energies must be calculated. The energy of a crystal is much more sensitive to volume changes than to changes in strains, and thus, high accuracy in the calculation of elastic constants requires volume-conserving strains. For cubic crystals, Mehl *et al.*[15] propose the use of volume conserving orthorhombic strains:

$$\begin{pmatrix} x & 0 & 0 \\ 0 & -x & 0 \\ 0 & 0 & \frac{x^2}{(1-x^2)} \end{pmatrix} \tag{3.9}$$

which reduces (3.8) to:

$$\Delta E(x) = V(C_{11} - C_{12})x^2 + O[x^4] \tag{3.10}$$

The calculated Bulk modulus and the expression

$$B = (C_{11} + 2C_{12})/3 \tag{3.11}$$

are then used to separate C_{11} and C_{12}. In like, volume-conserving monoclinic strain can be used to find C_{44}:

$$\begin{pmatrix} 0 & x & 0 \\ x & 0 & 0 \\ 0 & 0 & \frac{x^2}{4-x^2} \end{pmatrix} \tag{3.12}$$

and (3.8) becomes:

$$\Delta E(x) = \frac{1}{2} V C_{44} x^2 + O[x^4]. \tag{3.13}$$

By applying these strains (e_i) to the crystal lattice and then using VASP to calculate the energy of the system, the C_{ij} can be determined. In order to extend this procedure through finite temperatures, it is necessary to follow the following steps:

1. Strain the lattice several times at each quasi-harmonic volume; in this work we chose 5 volume conserving strains from 0-4% in 1% intervals.

2. Calculate F(V, e) for each volume with DFT.

3. Extract the C_{ij} as a function of volume.

4. Fit the $C_{ij}(V)$ data to the thermal expansion data obtained from the quasi-harmonic free energy surface.

In essence, a free energy surface as a function of volume and strain is created. From this, one can build an elastic constant surface in volume-temperature space. The volume expansion data from the thermodynamic modeling serves as a parameterized curve along this elastic constant surface from which the C_{ij} can then be interpolated, as a function of temperature.

Table II. Experimental [76] and calculated lattice parameter for aluminum and tungsten. Both the LDA and GGA predictions are shown. Values are in Å.

System		Calculated	Experiment
Al	GGA	4.046	4.050
Al	LDA	3.983	—
W	GGA	3.172	3.165
W	LDA	3.126	—

D. Calculated Properties and Discussion

1. Properties at 0 K

Table II enumerates the calculated lattice parameters for both the LDA and GGA along with experimentally obtained values. The calculated lattice parameter within the GGA is within 0.1% of the experimental value for aluminum and 0.2% higher for tungsten. For both systems the LDA underestimates the lattice parameter, by 1.7 % in the case of aluminum and 1.2 % for tungsten. Vibrational properties such as the Debye frequency and Debye temperature are tabulated in Table III while the bulk modulus, elastic constants and their corresponding slope at room temperature for aluminum and tungsten are displayed in Table IV. At approximately 50% of the melting point the calculated bulk modulus is approximately 8 GPa lower than the experimental values for aluminum. Since we use a relationship involving C_{11} , C_{12}, and the bulk modulus to separate and solve for the elastic constants in the modified strain energy equation this offset in the calculated bulk modulus propagates through the calculation of the elastic constants yielding a similarly 8 GPa low calculated C_{11} and C_{12}.

Table III. Calculated vibrational properties of aluminum and tungsten. ν_D represents the Debye frequency, Θ_n is the Debye temperature with respect to the nth moment of the phonon DOS, γ_m is the Grunesien constant at temperature m. Θ_{-2} is also known as the Debye-Waller temperature.

System		ν_D	Θ_0	Θ_1	Θ_2	Θ_{-2}	γ_0	γ_{298}
Al	Calc	7.71	370	375	381	369	2.31	2.39
	Expt				394[‡]		2.1[†]	
W	Calc	6.14	295	292	290	321	1.84	1.86
	Expt		325[§]		310[‡]			

[†] Data from Gersten & Smith [77]
[‡] Data from [78]
[§] Data from [79]

Table IV. Calculated elastic properties of aluminum and tungsten at 0K and room temperature.

System		K			C_{11}			C_{12}			C_{44}		
		0 K	298 K	dB/dT[†]	0 K	298 K	dB/dT[†]	0 K	298 K	dB/dT[†]	0 K	298 K	dB/dT[†]
Al	Calc	71	66	-2.7	111	105	-3.2	65	62	-1.0	34	32	-1.2
	Expt[‡]	79	76	-0.7	107	106	-3.2	61	60	-1.0	28	28	-1.0
W	Calc	307	303	-2.0	516	512	-2.3	203	201	-1.1	135	134	-0.8
	Expt[§]	314	311	-3.1	533	523	-6.3	205	205	-1.6	163	161	-0.9

[†] At 298 K, units are GPa/K x10^-2
[‡] 0K data from Simmons [57], all other data from Gerlich [53]
[§] Data from Featherston [54]

2. Thermodynamic Properties at Finite Temperatures

a. Aluminum

Accurate prediction of thermal expansion data calculated by quasi-harmonic lattice dynamics is critical in the characterization of temperature dependent properties from first principles. Fig. 6 demonstrates the calculated thermal expansion properties to over 95% of the way to melting. For aluminum the anharmonic and electronic corrections to the free energy have small effects on the thermal expansion data. At high temperatures the electronic contributions indicate greater thermal expansion but the anharmonic corrections actually

negate the electronic effects and even decrease the value predicted by the quasi-harmonic approximation. These details about contributions to the thermal free energy provide better fitting parameters to interpolate temperature dependent properties.

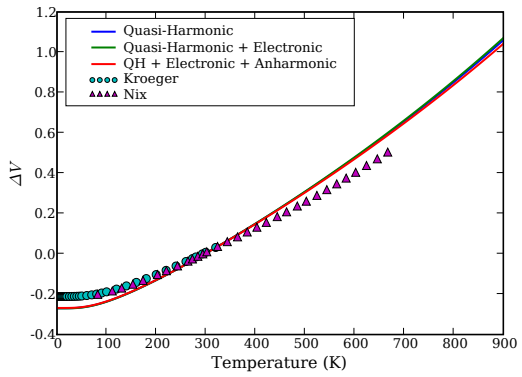
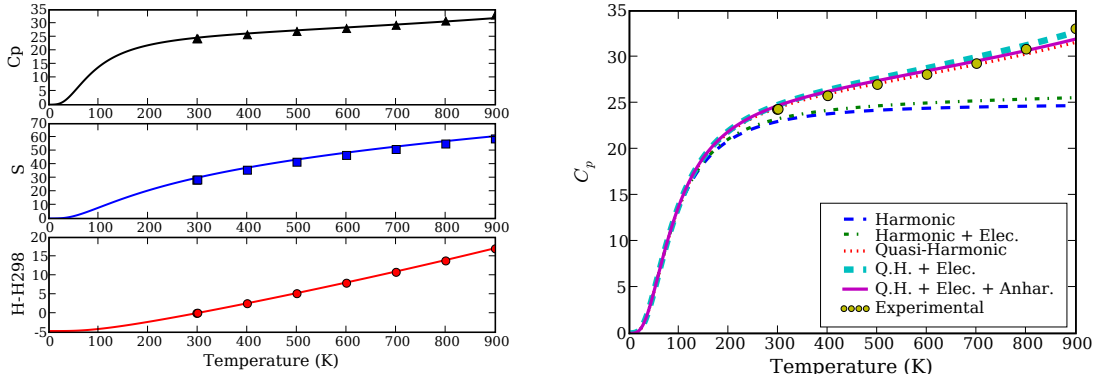


Fig. 6. Thermal expansion data for aluminum. The experimental data are from Kroeger [80] and Nix [81].

Since aluminum has a melting point of approximately 933 K, the finite temperature properties are examined up to 900 K. Temperature/property relationships become more erratic very near the melting point, making it difficult to predict accurate data within the assumptions used. As can be seen in Fig. 7 the calculated temperature dependence of the enthalpy, entropy and specific heat are predicted very close to experimentally tabulated values. Fig. 7(b) demonstrates the effect of electronic and anharmonic contributions on the prediction of the specific heat. The electronic contributions raise the specific heat and tend to overestimate it while the anharmonic contributions effectively balance out this contribution, providing very accurate data throughout the temperature range. As the temperatures get within 90% of the melting point we expect anharmonic terms to take a more dominant role and result in a breakdown in validity of the anharmonic assumptions used in the model. Throughout the thermodynamic analysis aluminum proves to be a very well behaved system within the approximations made.



(a) Calculated and experimental[82] enthalpy, entropy and specific heat.

(b) Contributions considered for C_p.

Fig. 7. Enthalpy, entropy and specific heat of aluminum. The electronic contributions and anharmonic correction are extremely important in accurately predicting the specific heat.

b. Tungsten

While aluminum is well behaved within the approximations outlined above and the calculations provide excellent agreement with experimental values, there are limitations of the current approach. An analysis of tungsten, especially at high temperatures indicates some of these limitations and indicates areas of future study. The volume thermal expansion data of tungsten is shown in Fig. 8. The calculated and experimental results agree well up to approximately 2500K. At this point the experimental data takes a sharp turn upwards, a phenomenon not captured by the calculations. At this point the anharmonic corrections decrease the accuracy of the model calculations, most likely due to the neglected terms from (3.4). In their calculations for tungsten, Ackland *et al.* neglect both electronic and anharmonic corrections, thus giving them a less thorough yet conveniently more accurate volume expansion relationship [11] which ultimately yields greater correlation with experiment for elastic constant predictions. Our deviation between calculation and experiment for thermal expansion will propagate through the thermodynamic and thermo-mechanical prop-

erty calculations resulting in pronounced errors above 2500 K. The calculated properties in Fig. 9 demonstrate the temperature dependence of specific heat, entropy and enthalpy up to 3500 K. Both the entropy and enthalpy are well characterized by our method throughout the temperature range while the specific heat shows an exponential growth above 2500 K that is unaccounted for in the calculations.

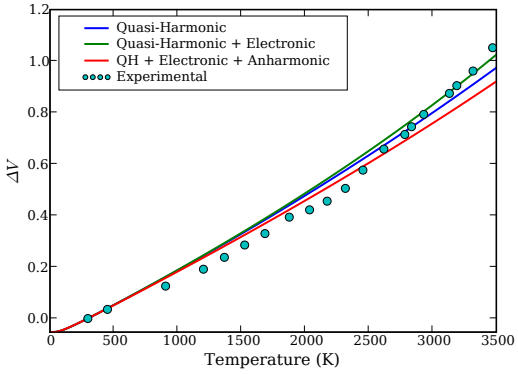
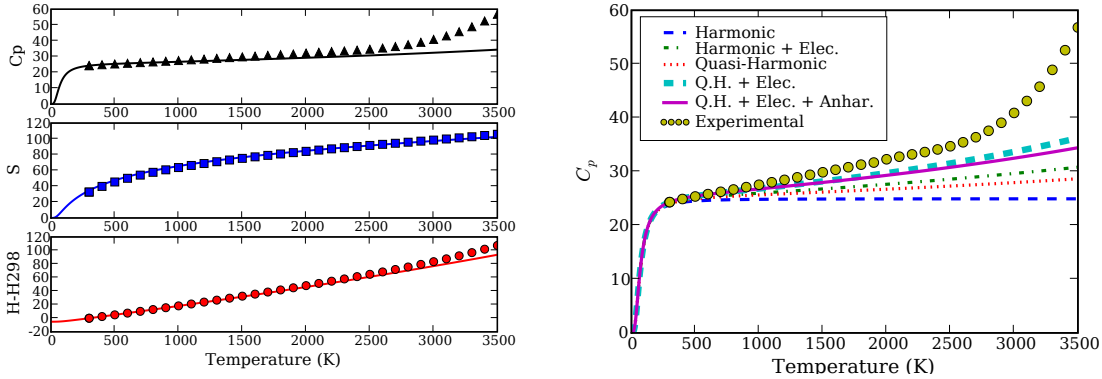


Fig. 8. Volume thermal expansion for tungsten. The two data sets were normalized relative to the room temperature volume. Experimental values are from the work of Dubrovinsky *et al*. [83].



(a) Calculated and experimental [82] enthalpy, entropy and specific heat of tungsten.

(b) Contributions considered for C_p of tungsten.

Fig. 9. Thermodynamic property calculations for tungsten.

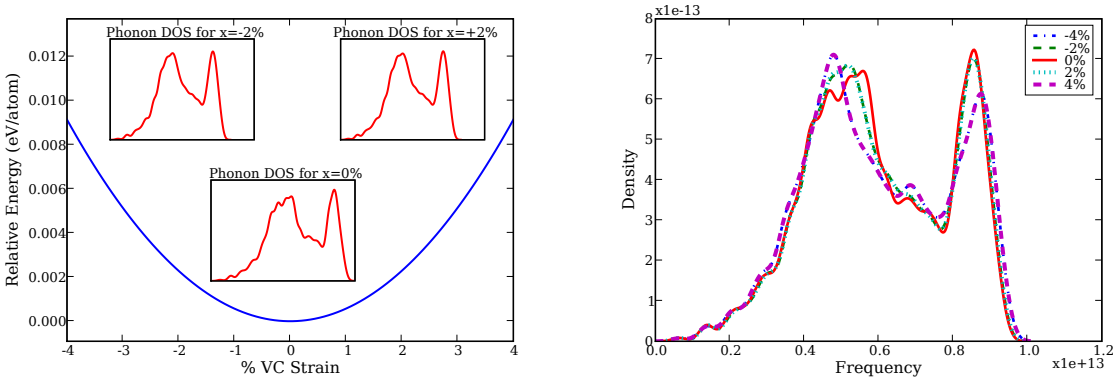
3. Elastic Constants at Finite Temperatures

a. Considerations in the Calculation of Elastic Constants from Free Energies

This section discusses some computational questions and our findings in an attempt to thoroughly validate several choices and simplifications made in our calculations. In particular we explain some symmetry effects, look at phonon DOS at the various volume conserving strains, and explain the use of a correction to the calculated bulk modulus which allows for better comparison between the calculated properties of aluminum and the corresponding experimental values.

Initially we applied symmetric strains about the equilibrium volume for use in the calculation of elastic constants. The phonon DOS for symmetric strains and their relative impact on the unstrained DOS is demonstrated in Fig. 10. In Fig. 10(a) we can see that performing a =+2% strain on the lattice had the same resulting DOS and cold curve energy as that of a -2% strain. This is a logical consequence of the combination of symmetries involved: when applying an orthorhombic strain to a cubic system, the resulting system is identical, just rotated differently in space. By examining the various phonon DOS as shown in Fig. 10(b) we can make inferences about the effect these volume conserving strains will have on the total free energy of the system. Also, by plotting the DOS on the same axes we are able to see that they are identical at symmetric strains. Due to these symmetry results we feel justified in only performing positive strain calculations in our calculation of the elastic constants, thus decreasing the required computational time.

In addition to the symmetry of the strains applied in (3.8) we examined the temperature dependent free energy at each of these strains. Upon inspection of Fig. 10(b) we can see three distinct phonon DOS, at 0, 2 and 4% strain of the lattice. The DOS have some distinctions but the frequencies at which the peaks occur and the overall area under the curve remain similar to each other. Upon investigation we discovered that the impact of these distortions



(a) Symmetric strains and their effect on E_0 (b) Phonon DOS for symmetric strains

Fig. 10. Effect of volume conserving strains on the phonon DOS of FCC Al.

on the total free energy of the system is minimal in comparison with the volume expansion effects that are accounted for with quasi-harmonic lattice dynamics. Fig. 11 displays the free energies of structures at several volumes and several strains at each volume. A similar analysis was done by Ackland *et al.* [11], from which we can infer that the impact of the phonon contributions to the free energy due to a given volume-conserving strain are quite small relative to the corresponding volume effects. Since the calculation of free energy is related to an integration of the phonon DOS, and the DOS of the volume conserving strains have the similar features mentioned this makes sense. This is a very useful result because it allows us to neglect (as a minor approximation) computationally costly lattice dynamics calculations at each of the volume conserving strains. Instead of performing lattice dynamic supercell calculations at each strain we simply calculate the ground state energy of a single primitive cell and use this cold curve data coupled with the thermal expansion data to extract the elastic constants. In the case of our aluminum calculations with five volumes and five strains at each volume this saves us from making an additional 25 supercell calculations which would effectively quintuple the computational time needed for DFT calculations on the parallel computing cluster. Naturally these additional calculations could be performed

in the hopes of attaining greater accuracy, but with volume effects dominating so clearly, the return would be minimal.

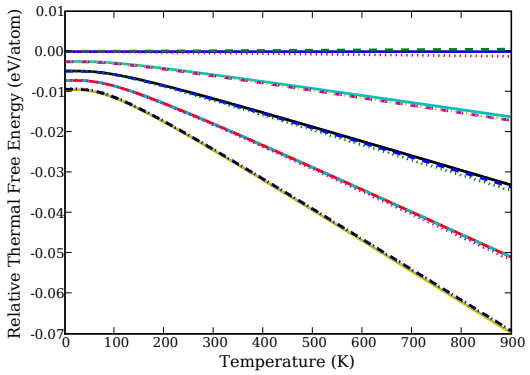


Fig. 11. Relative free energies for volume conserving strains. The data for each volume of the quasi-harmonic approximation are included. The solid lines represent the normalized volumes (in relation to the ground state) from the quasi-harmonic lattice dynamics and the dashed lines the free energy at corresponding volume conserving strains for elastic constant calculations. By neglecting thermal free energy calculations at each strain and following just the volume effects we drastically cut computation time by performing simple primitive cell energy calculations in lieu of supercell lattice dynamics.

In the case of systems with cubic symmetry we use (3.10) and (3.11) to solve for C_{11} and C_{12}. Doing so depends on the calculated bulk modulus obtained from fitting an equation of state to the lattice dynamics calculations. Previously we demonstrated an 8 GPa difference between the calculated bulk modulus and experimental results. If this difference is ignored, an error will propagate through the calculation of the elastic constants. As a rough correction we examine the effect of adding an 8 GPa error term to the bulk modulus as depicted in Fig. 12. Ultimately we would like to find a different expression for separating C_{11} and C_{12} and make the calculations independent of the fit of the bulk modulus. While this technique requires an experimental data point, it does provide for better comparison with experiment of the final calculated elastic constants. This error term does not change the slopes of the

calculated elastic constants which represents softening of a material with temperature and can be useful. Throughout the remainder of this paper all results for aluminum will reflect this 8 GPa shift while for tungsten no such correction has been made. We hope to eliminate the need for this error term soon with additional strain energy calculations.

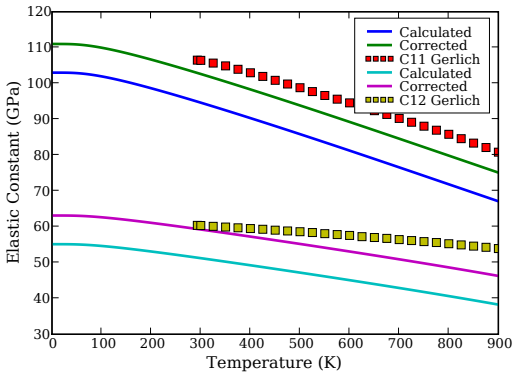


Fig. 12. Effect of shifting the bulk modulus to better match experimental values on the calculated elastic constants.

Since most experimental data for the elastic constants is currently obtained via ultrasonic measurements which yield the adiabatic elastic constants it is useful for us to convert our calculated elastic constants (which are inherently isothermal) to adiabatic for comparison [84, 85, 86]. This conversion is a function of the thermal expansion data and the isothermal elastic constants as given by:

$$C_{ij}^{S} = C_{ij}^{T} + \frac{V\lambda i\lambda_j T}{C_v}, \tag{3.14}$$

for C_{11} and C_{12} with:

$$\lambda_i = \lambda_j = \alpha\left(C_{11}^{T} + C_{12}^{T}\right) \tag{3.15}$$

and $C_{44}^{S} = C_{44}^{T}$ due to cubic symmetry. The result of this transformation along with the shift in the bulk modulus is plotted in Fig. 13.

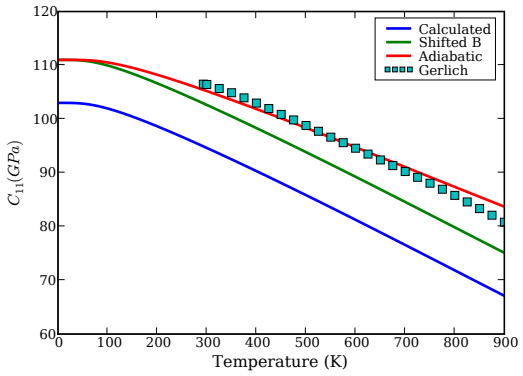


Fig. 13. Effects of the bulk modulus corrections and isothermal to adiabatic transform on the calculated elastic constants. Experimental data is that of Gerlich. [53]

b. Calculated Elastic Constants of Aluminum and Tungsten

It can be seen in Fig. 14 that there are various experimental data sets for the elastic constants of aluminum over finite temperatures with considerable variations in magnitudes and trends. All three data sets were obtained using the composite oscillator method with the distinction that the data of Gerlich was found using higher frequency waves which reduces specimen alignment errors in the experiments. A fourth data set (not shown) of Kamm and Alers [87] represents the elastic constants at low temperatures and matches up well with those of Gerlich and Tallon in that temperature range. Of particular note is the C_{12} data of Tallon which show a slight increase with temperature. While we are unsure as to why Tallon's results appear the way they do, we see nothing else in the literature to explain this upward or even a level trend for C_{12}. When looking at the disparity of the several data sets our calculated values fit well within the experimental envelope of these three data sets and agree particularly well with the data of Gerlich *et al*. which we will choose for comparison for the remainder of this work.

Fig. 15 depicts the final calculated elastic constants of aluminum with all corrections discussed and compare the results with experimental data. Calculated values for aluminum

including all correction factors agree extremely well with experiment. Considering the variation among experimental data sets we are very pleased with the first principles predictions. For aluminum the quasi-harmonic approximation and corresponding free energy calculations prove to be extremely adequate for predicting the temperature dependent elastic constants.

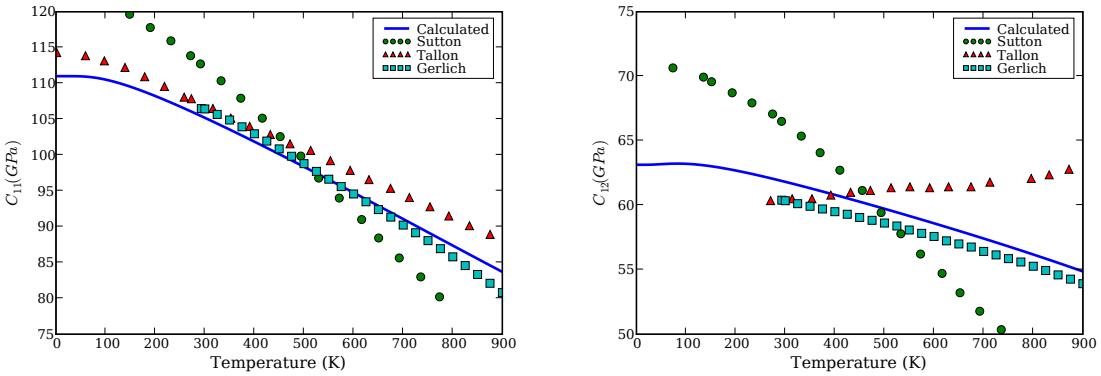


Fig. 14. Comparison of calculated elastic constants with experimental data.

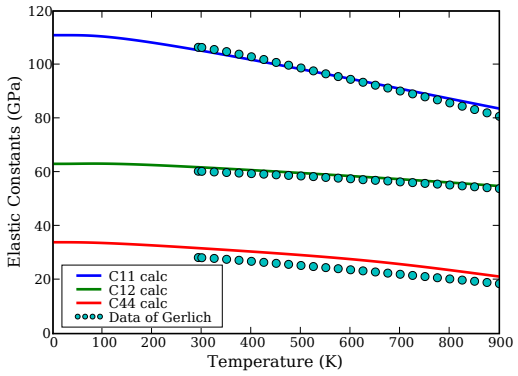


Fig. 15. Final results for calculated elastic constants of aluminum compared with the data of Gerlich.

For tungsten we obtain fair results, but not nearly as good as for aluminum. Tungsten is a highly anharmonic system at high temperatures [88] and therefore our approximations will limit our ability to accurately describe high temperature properties. In fact, the anharmonic

corrections we implement tend to have an adverse effect on property calculations as will be demonstrated. It was mentioned previously that we currently have no way of accounting for anything but the first term in (3.4). While this was sufficient for aluminum, these missing terms have significant impact on the anharmonic contributions to the thermal free energy of tungsten and propagate through the thermodynamic and thermo-mechanical property calculations.

Experiments show a strong softening of the bulk modulus of tungsten above 2000 K (Fig. 16), a phenomenon not captured by the DFT results. At approximately this same temperature the calculated C_{11} becomes greater than the experimental comparison and C_{12} deviates more from the almost constant experimental value.

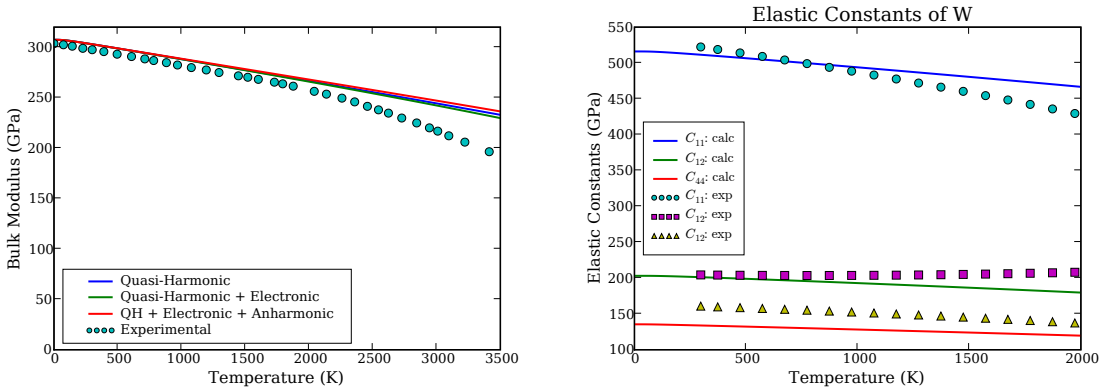


Fig. 16. Calculated bulk modulus and elastic constants for tungsten.

E. Summary and Conclusions

We have developed a method for the calculation of thermodynamic and thermo-mechanical properties of pure, cubic, non-magnetic, materials through first principles calculations and shown its validity and limitations for aluminum and tungsten. Using density functional theory and quasi-harmonic lattice dynamics we constructed thermal free energy surfaces and extracted thermodynamic and vibrational quantities such as entropy, enthalpy, specific heat,

Grunesien constant and Debye temperature. Overall, the method presented demonstrates good agreement with experimental values. We are able to predict the temperature dependence of the elastic constants of aluminum within a few percent up to 900 K and those of tungsten within 10% for temperatures under 2000 K. The tungsten calculations demonstrate some of the limitations of this model, especially the effect of anharmonic contributions to the free energy and the propagation of these effects into derived properties. We also examine several factors that contribute to the thermal free energy and demonstrate that volume expansion effects have the predominant role in the temperature dependence of the elastic constants when compared to vibrational or thermal-electric contributions. We also predicted volume thermal expansion and coupled it with cold curve strain energy calculations in order to extract the thermal behavior of the isotropic single crystal elastic constants.

Our key findings and conclusions can be summarized as follows:

1. The calculated thermodynamic and thermo-mechanical properties for aluminum agree very well with experiment throughout the solid phase.

2. The same calculated properties for tungsten are only reasonable up to approximately 60-70% of the melting point.

3. Volume expansion effects are the overwhelming predominant factor in the softening of elastic constants with increasing temperature

4. Anharmonic contributions to the free energy as demonstrated with the tungsten calculations are most likely the greatest cause of poor correlation between calculation and experiment for that system.

5. The poor prediction of the temperature/volume relationship in tungsten propagated through the mechanical property calculations, making our predictions of C_{11}, C_{12}, C_{44}, and the bulk modulus for tungsten only reasonable up to approximately 2000 K.

6. The anharmonic correction factor of Oganov was shown to improve experimental correlation of calculated properties for aluminum and degrade the same for tungsten at high temperatures. We expect that the degradation has to do with the neglected terms in (3.4)

As an additional consequence of this work, an open source, Python-based suite of computational utilities to automate the preparation, cluster submission and management, and post processing of these calculations has been created and will be available in the public domain shortly.

CHAPTER IV

AB INITIO THERMODYNAMIC AND ELASTIC PROPERTIES OF B2 NIAL, RUAL AND IRAL AT FINITE TEMPERATURES

A. Introduction and Motivation

On many occasions, lack of materials with optimal sets of attributes constitute limiting factors in many applications where all the other components of superior technology are already in place. In particular materials that will withstand higher and higher temperatures are in great demand. Recently, there has been interest in alloys containing platinum group metals [89, 90, 91] (PGM) as potential higher temperature alternatives to nickel based superalloys [4]. These new alloys could be used in bond coats of thermal barrier coatings or as structural materials in turbine systems. With new materials capable of withstanding higher temperatures the thermodynamic efficiency of power generation and propulsion systems could be increased, resulting in lower-cost and less environmental impact. Of the several potential high-temperature intermetallic,s NiAl has been well characterized over the past few decades while other potentially higher temperature alternatives have only recently begun to be investigated [92]. In this work we examine the characterization of the thermodynamic and mechanical properties of NiAl, RuAl, IrAl from finite temperature first principles calculations. It is hoped that the present work will provide a fundamental theoretical understanding and assist experimentalists in the search for new high-temperature materials.

The role of first principles approaches and quantum mechanical calculations in materials science is to help us understand the fundamental, microscopic basis for macroscopic behavior. This understanding may then lead to predictions of related phenomena which rely on similar atomic-level interactions [3]. Some of the properties of interest in materials

science include structural and mechanical properties and how they evolve with changes in temperature. For example, a correct understanding of the thermo-elastic behavior of an alloy is vitally important to anticipate the structural reliability of bond-coats or hardening precipitates based on the intermetallics mentioned above.

The purpose of this work is to present the temperature dependence of several thermodynamic and elastic properties of B2 NiAl, RuAl and IrAl which have been calculated from first principles. In the process, the same properties have been calculated for the pure consituent systems *fcc* Al, Ni, Ir and *hcp* Ru to provide validation for the calculations. Previous work on finite temperature thermodynamics has been done by several groups including predictions of properties in the Al-Sc system [38], copper [93], W, NiAl, and PdTi [11] and others [39]. Overall, these groups have found very good correlation with experiment and have been able to develop systematic methodologies and tools for such property predictions. Mehl *et al.* have laid much fundamental groundwork in the field of ab initio prediction of elastic constants at 0 K [15]. Later, this work was linked with finite temperature thermodynamics [11]. Our work extends what these and other groups have done [2, 13, 70] and applies it to materials, such as B2 RuAl and IrAl that still have not been fully characterized.

In this paper we briefly review the methods for the calculation of thermodynamic and thermo-mechanical properties from first-principles and present calculated finite temperature thermodynamic properties and elastic constants for NiAl, RuAl, IrAl as well as their elemental constituents. The calculated properties agree well with experiment where available including phonon density of states, thermodynamic quantities, and elastic constants. The first-principles calculations were done within the framework of density functional theory and quasi-harmonic lattice dynamics to calculate the ground state energy and vibrational contributions to the total free energy, respectively. The electronic degrees of freedom and intrinsic anharmonicity are accounted for while magnetic and configurational contributions to the free energy are neglected. These approximations and tools provide property calcu-

lations that in general agree very well with experimentally obtained values when available. The calculated temperature dependence of the coefficient of thermal expansion for the B2 phases demonstrate a similar slope to experimentally obtained values yet the first-principles predictions tend to overestimate its magnitude. We find that the generalized gradient (GGA) and local density (LDA) approximations provide bounding predictions for the temperature dependence of the elastic constants.

B. Methodology

The thermodynamic properties considered in this work are the enthalpy, entropy, specific heat at constant pressure, and the linear coefficient of thermal expansion. Each of these quantities can be derived from local slopes and curvatures of the free energy surface in volume/temperature space:

$$F(V,T) = E_{0K}(V) + F_{vib}(V,T) + F_{el}(V,T) + F_{anh}(V,T). \tag{4.1}$$

Here E_{0K} is the ground state electronic energy, and F_{vib}, F_{el}, and F_{anh}, represent vibrational, electronic, and anharmonic contributions to the free energy, respectively. Since all terms in (4.1) are a function of volume—and temperature—we make use of the quasi-harmonic approximation[2] and thus account for thermal expansion in the material [10, 35].

The ground state energy was calculated using density functional theory(DFT) [19] within both the local-density [21] (LDA) and generalized-gradient [24, 66] (GGA) approximations as implemented in the Vienna Ab-Initio Simulation Package (VASP)[32, 33]. The calculations were performed using projector augmented-wave pseudopotentials [67, 29]. All calculations were done with a cutoff energy of 350 eV and the convergence criteria was set to a maximum difference of 1E-6 eV. In all calculations the k-point mesh was set to a density of 10,000 k-points per reciprocal atom. Initially the lattice parameters of the

ground structures were optimized allowing all degrees of freedom to relax using the first order Methfessel-Paxton smearing technique [68] and then a final self consistent calculation was performed using the tetrahedron smearing method including Blöchl corrections [67]. The atomic configurations used were $3s^2 2p^1$ for Al, $[Ar]$ for Ni, $5s^1 4d^7$ for Ru, and $6s^1 5d^8$ for Ir.

The vibrational degrees of freedom are accounted for through the force constant (or supercell) method [70] as implemented in the ATAT software package [71, 72, 73]. Which calculates the vibrational thermal free energy through:

$$F_{vib}(T) = k_B T \int_0^\infty \ln \left[2\, sinh \left(\frac{h\nu}{2k_B T} \right) \right] g(\nu)\, d\nu. \tag{4.2}$$

The electronic contributions to the free energy are calculated self-consistently using the one-electron approximation as outlined by Asta [38]. In this method the electronic density of states is integrated at each quasi-harmonic step according to:

$$F_{el}(V,T) = \int n(\varepsilon, V)\, f \varepsilon d\varepsilon - \int^{\varepsilon_F} n(\varepsilon, V)\, \varepsilon d\varepsilon. \tag{4.3}$$

This contribution can then be included as a simple additive term to the total free energy of the system [39].

Since the quasi-harmonic approximation is unable to account for intrinsic anharmonic contributions to the free energy we implement the findings of Oganov & Dorogokupets [37]. They express the anharmonic contributions to the free energy as

$$\frac{F_{anh}}{3nk_B} = \frac{a}{6} \left[\begin{array}{c} \left(\frac{1}{2}\theta + \frac{\theta}{\exp(\theta/T)-1} \right)^2 \\ +2 \left(\frac{\theta}{T} \right)^2 \frac{\exp(\theta/T)}{(\exp(\theta/T)-1)^2} T^2 \end{array} \right] \tag{4.4}$$

where a is found according to Wallace [36]. Wallace's coefficient is a function of both the Grüneisen constant and Debye temperature which are also calculated through first-principles

methods in this work, without resorting to any experimental data.

$$a = \frac{3k_B}{2\Theta}(0.0078\,\langle\gamma\rangle - 0.0154). \tag{4.5}$$

The calculation of the temperature dependence of the single crystal elastic constants C_{ij} were performed by building on the work of Mehl [15] and Ackland [11]. At each volume considered within the quasi-harmonic approximation, the lattice was strained and the isothermal elastic constants extracted according to the strain energy equation

$$E\left(e_i\right) = E_0 + \frac{1}{2}V\sum C_{ij}e_ie_j + O\left[e_i^3\right]. \tag{4.6}$$

These strain values were then fit with a parameterized curve from the volume expansion data and the corresponding temperature dependence of the elastic constants was obtained. These values represent the isothermal elastic constants and since the majority of the literature reports the adiabatic constants the calculated values are then converted to their adiabatic counterparts through the use of the standard thermodynamic relation:

$$C_{ij}^S = C_{ij}^T + \frac{V\lambda i\lambda_j T}{C_v}, \tag{4.7}$$

with:

$$\lambda_i = \lambda_j = \alpha\left(C_{11}^T + C_{12}^T\right) \tag{4.8}$$

due to symmetry. These values can then be compared with experimentally obtained values obtained through adiabatic techniques, such as inelastic neutron scattering, raman spectroscopy and so forth.

C. Calculated Properties of the Constituent Elements

The ground state lattice parameter, bulk modulus and elastic constants of pure Al, Ni, Ir and Ru are found in Table V for both the GGA and LDA. As expected, the LDA tends to

over-bind and thus underestimate the lattice parameter [3] of all the systems. This over-binding also results in the LDA predicting higher elastic constants. From (4.6) the elastic constants are seen to be proportional to the curvature of the strain energy relationship shown in Fig. 17. The strain energy of the LDA demonstrates greater curvature than that of the GGA, resulting in the higher ground state elastic constants.

Table V. Ground state mechanical properties for Al, Ni, Ir and Ru.

System	a ()†			K (GPa)			C_{11} (GPa)			C_{12} (GPa)			C_{44} (GPa)		
	GGA	LDA	expt.	GGA	LDA	expt.	GGA	LDA	expt.	GGA	LDA	expt.	GGA	LDA	expt.
Al	4.05	3.98	4.05	72.2	81.9	79.4	103.0	118.6	114.3	55.1	61.9	61.9	34.0	38.5	31.62
Ni	3.52	3.42	3.52	196.8	252.7	187.6	244.4	313.4	261.2	169.9	219.1	150.8 x	104.7	132.3	131.7
Ir	3.88	3.82	3.84	341.0	402.2	354.7‡	564.0	657.3	580.0‡	228.1	273.0	242.0‡	243.1	285.8	256.0‡
Ru	2.73	1	2.71§	–	–	315.2§	–	–	576.3§	–	–	187.2§	–	–	–

Experimental bulk modulus and elastic constant data from Simmons & Wang [57]

† Experimental lattice parameters from Gersten & Smith [77]

‡ Data corresponds to 300 K

§ Data corresponds to 4 K

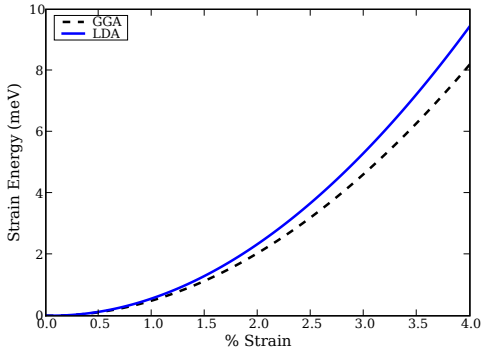


Fig. 17. Difference in ground state energy as a function of volume conserving strain for pure Ni. The difference in curvature between the LDA and GGA data is one of the key reasons why elastic constants calculated with the LDA are higher than those calculated with the GGA.

Vibrational contributions to the free energy play a predominant role in the tempera-

ture dependent property calculations. Since these degrees of freedom are accounted for through phonon behavior as calculated through lattice dynamics [34] a brief validation of the phonon dispersion and density of state proves insightful. Fig. 18 represent the calculated phonon density of states (DOS) for Ni. The DOS shows excellent agreement between the calculations and experimental results. In the case of Ni, the supercell was constructed of 40 atoms and the force constants were evaluated through the third nearest neighbors. Since the agreement with experiment is good the choice of supercell size is justified and the vibrational contributions to the free energy are assumed to be adequately described within the approximations made.

While the phonon DOS gives a point by point evaluation of phonon properties, in calculating thermodynamic quantities what is most important is the mean behavior of the phonon DOS. An accurate phonon DOS is critical to the development of the thermal free energy surface since (4.2) relies on numerical integration of this DOS ($g(\nu)$). The integration has a tendency to 'smooth' out small discrepancies as long as the overall trends and general location of the peaks within the DOS are adequately accounted for.

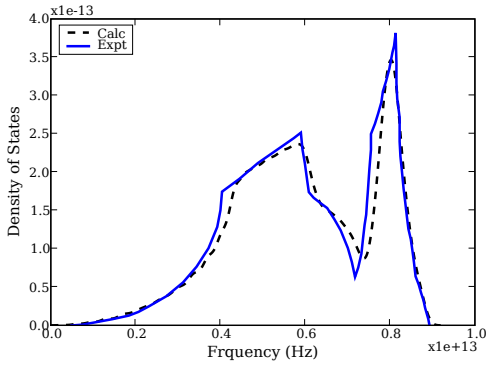
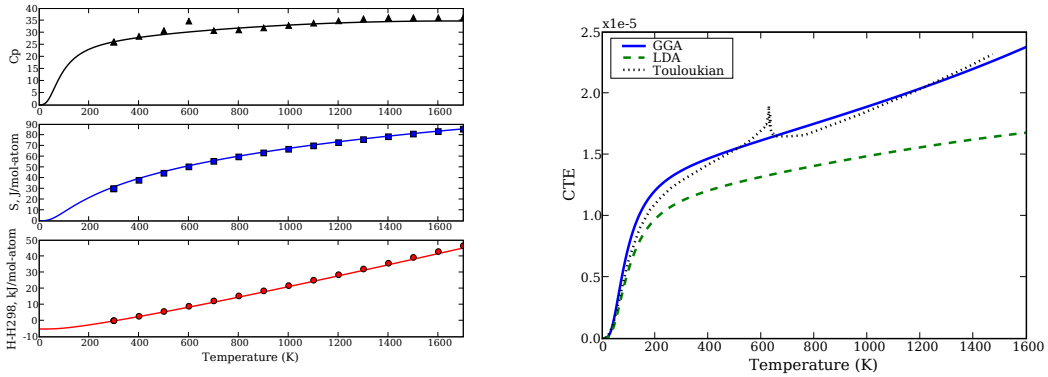


Fig. 18. Phonon density of states for Ni. Calculation of the vibrational contributions to the free energy depend on an accurate representation of the phonon density of states.

With the vibrational properties verified we look at the results from the combination of the ground state, vibrational, electronic, and anharmonic contributions to the free en-

ergy from which we extract valuable thermodynamic information about the system. Based on classical thermodynamics we can take local slopes and curvatures of the free energy surface [9] to find the heat capacity at constant pressure, entropy, relative enthalpy, and coefficient of thermal expansion for each system being investigated. Fig. 19, demonstrates these calculated thermodynamic quantities for Ni. Similar plots for the other elements have been included in the supplemental materials section of this chapter.



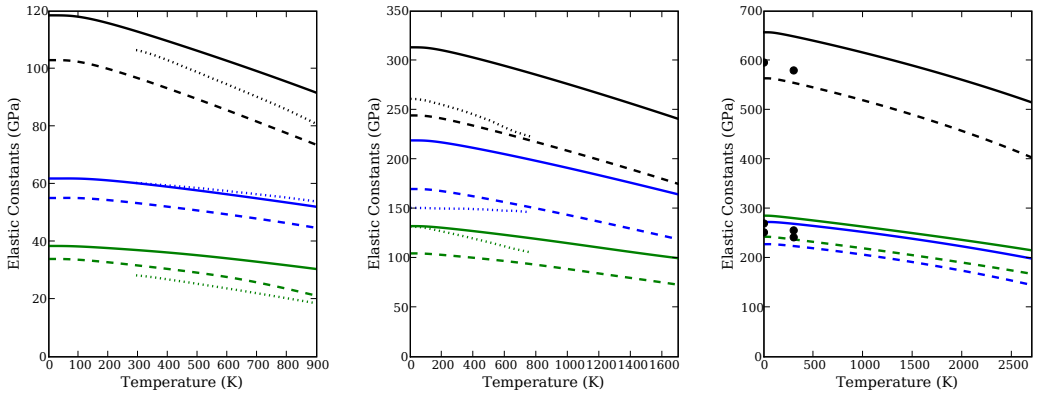
(a) Cp, S, H (b) CTE

Fig. 19. Key thermodynamic properties of pure Ni. On the left are the specific heat, entropy, and relative enthalpy of Ni from 0-1700 K with experimental data taken from the compilation of Barin [82]. On the right is the calculated CTE for Ni within both the GGA and LDA. The LDA significantly understimates the CTE at increasingly high temperatures while the GGA yields excellent experimental correlation with the data of Kollie [94].

The temperature dependent C_p, S, and $H - H_{298}$ show excellent correlation with experimentally tabulated values. For many systems the coefficient of thermal expansion (CTE) is more difficult to calculate accurately since it requires the calculation of a second order numerical derivative. For Ni the predicted CTE is in almost exact agreement with the tabulated values, except for the characteristic spike [95] due to the magnetic order/disorder transition in both the CTE and C_p around 600K which is not reflected in the calculations.

Since this anomaly is due to magnetic effects near the Curie temperature [96] and magnetic degrees of freedom have been neglected in the present calculations we would not expect to see such a phenomenon.

By coupling the volume thermal expansion behavior parameterized from local minima on the thermal free energy surface with strain energy calculations at each quasi-harmonic step we extract the temperature dependence of the single crystal elastic constants C_{ij}. Our calculated predictions for Al, Ni, and Ir are displayed in Fig. 20 along with experimental comparison. The majority of experimentally obtained C_{ij} for these three systems tend to



(a) Al (b) Ni (c) Ir

Fig. 20. Calculated temperature dependence of the elastic constants of the Al, Ni, and Ir. Dotted lines represent experimental work, solid lines, predictions due to the LDA and dashed lines those found from the GGA calculations. We have yet to implement the calculation of elastic constants in non-cubic systems and therefore the elastic constants of pure Ru (which has an *hcp* structure) were not calculated. For Ir, the values for C_{44} are higher than those for C_{12}, this is unique among systems studied in this work. The experimental results are taken from the work of Gerlich [53], Simmons and Wang [57], and Macfarlane [97] for Al, Ni, and Ir respectively.

lie within the limits set by the GGA and LDA calculations. The C_{12} of Al is predicted remarkably well by the LDA while C_{44} is better represented by the GGA calculations and

C_{11} is close to the mean of the two approximations. Different trends are seen in Ni and for Ir there is not much available data for comparison. It seems that at this time there is no clear answer as to wether the GGA or LDA should be used in such calculations. It is noteworthy that the slopes of the plots in Fig. 20 are practically identical between the LDA and GGA in all cases. The largest deviation from this behavior is in Al at very high temperatures where the GGA drops off slightly. This seems to indicate that the softening behavior of a given C_{ij} due to thermal expansion is adequately described by either approximation.

D. B2 Phases

With the model demonstrated for these simple systems, the thermodynamic and thermo-mechanical properties of NiAl, RuAl, and IrAl have been calculated in the same fashion as for their unary constituents. Of these three systems NiAl is the most thoroughly characterized to date. RuAl [8, 98] and IrAl [6] have gained exposure in recent years due to their potential as high temperature materials but a comprehensive study of their vibrational, thermodynamic and mechanical properties has yet to be reported. In this section we present our predictions for the same thermodynamic and mechanical properties of NiAl as were calculated for the simple systems. We compare these results with experiment when available and note some of the strengths and weaknesses of the model. We then present our prediction for the same properties for RuAl and IrAl and leave them as targets for further theoretical study and experimental validation.

In the implementation of the supercell (SC) method for the calculation of force constants and vibrational properties we tested several ranges over which the force constants and subsequent properties should be calculated. In Table VI we list the calculated force constants for first and second nearest A-B bonds. RuAl is shown to have the strongest interaction between dissimilar ions in the closest A-B pairs and also the only positive force constant for

the second nearest A-B neighbor. All the second nearest neighbor interactions are extremely week in comparison with their first nearest neighbor counterparts. The calculated force constants seem to indicate that these three systems are dominated by short range interactions, suggesting that the supercell sizes chosen are adequate for predicting with high degree of accuracy the properties of these intermetallics.

Table VI. Force constants for the first two nearest AB neighbors for B2 phases. Units are in $eV/$Å.

System	$1^{st}NN$	$2^{nd}NN$
NiAl	2.09	-7.6E-4
RuAl	3.07	1.0E-3
IrAl	2.51	-5.7E-2

The Debye temperatures, frequencies, and Grüneisen constants have been calculated according to lattice dynamics and the results are summarized in Table VII. The specific heat

Table VII. Calculated vibrational properties of aluminum and tungsten within the GGA. ν_D represents the Debye frequency, Θ_n is the Debye temperature with respect to the nth moment of the phonon DOS, γ_m is the Grunesien constant at temperature m. Θ_{-2} is also known as the Debye-Waller temperature.

System	ν_D (THz)	Θ_0	Θ_1	Θ_2	Θ_{-2}	γ_0	γ_{298}
NiAl	8.38	402	420	438	434	2.09	2.15
RuAl	8.64	415	432	447	442	1.97	1.99
IrAl	7.96	382	413	440	326	2.21	2.24

Debye temperature (Θ_2) and the Grüneisen constant at 0 K (γ_0) are used in the calculation of (4.5) in order to approximate anharmonic contributions to the free energy. Θ_2 is also a critical parameter in the calculation of the total system entropy and specific heat. The

values of Θ_2 for NiAl and IrAl are found to be almost identical and that of RuAl is just over 2% higher than NiAl. These similarities will carry over into the prediction of total system entropy and specific heat and will be discussed shortly.

The phonon DOS is one of the key parameters in the prediction of finite temperature thermodynamics. It is through the DOS that (4.2) allows us to make a statistical connection between harmonic vibrations in the lattice and the thermal free energy. The phonon DOS of NiAl is shown in Fig. 21 and the calculated values show very good correlation with experiment. While the simulations do not capture all the jagged peaks of a given DOS, they are able to capture the overall behavior very well. As has been previously stated, (4.2) involves a numerical integration over the DOS and therefore small discrepancies between calculated and experimental values have minimal impact on the resulting property predictions. Point by point precision is not necessary as long as overall averages are similar.

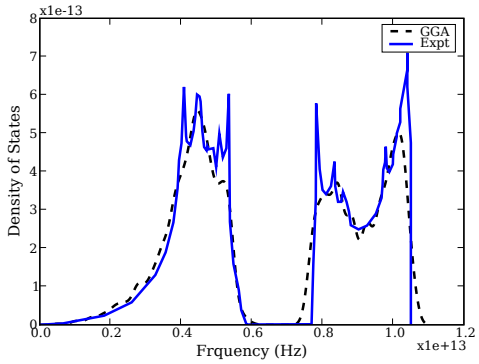


Fig. 21. Phonon density of states for NiAl. Experimental data is that of Mostoller *et al.* [99].

In Fig. 22 the LDA is shown to provide a more precise description of the CTE of NiAl than the GGA. This is noteworthy because the GGA provides the more accurate prediction of the CTE for Ni, and the LDA does better for Al (see supplemental material at the conclusion of this chapter). There is no readily apparent way of knowing *a priori* which approximation

will yield the best correlation with experiment. At cryogenic temperatures, the slope of the experimentally obtained CTE diverges significantly from the theoretical results and does not diminish as rapidly with temperature as the theory suggests. In spite of this anomaly, at normal operating temperatures the LDA provides a very reasonable prediction of how the CTE of NiAl evolves with temperature.

The LDA also provides a better description for the CTE of RuAl and IrAl and so the GGA results are not shown in Fig. 23. While the absolute magnitudes differ between theory and experimental, the calculations affirm the general slopes of the CTE at temperatures above 500 K. The experiments show that the CTE of RuAl should be higher than that of IrAl, a fact affirmed in our predictions.

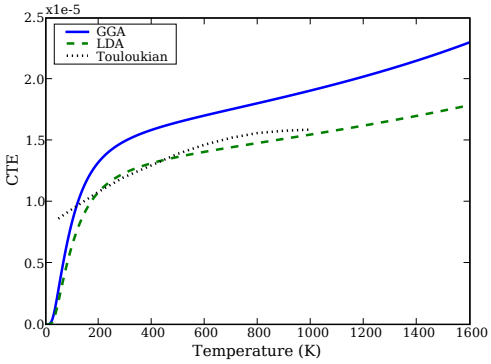


Fig. 22. Coefficient of thermal expansion for NiAl. Experimental data is that of Touloukian [100].

As mentioned previously, the specific heat and entropy are related to the phonon DOS and the Debye temperature. More specifically, the specific heat is related to a frequency weighted phonon DOS of the form $\nu^2 G(\nu)$. This weighted DOS, along with the original DOS, the entropy, and the specific heat are shown in Fig. 24. It is interesting that all three systems demonstrate a band-gap–resulting from the diatomic nature of the B2 unit cell—in the DOS between about 0.6-0.8 THz, with sharp peaks on either side. Iridium is

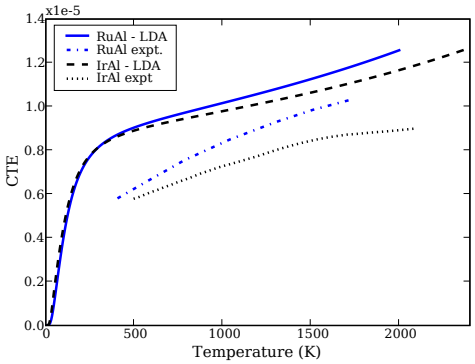
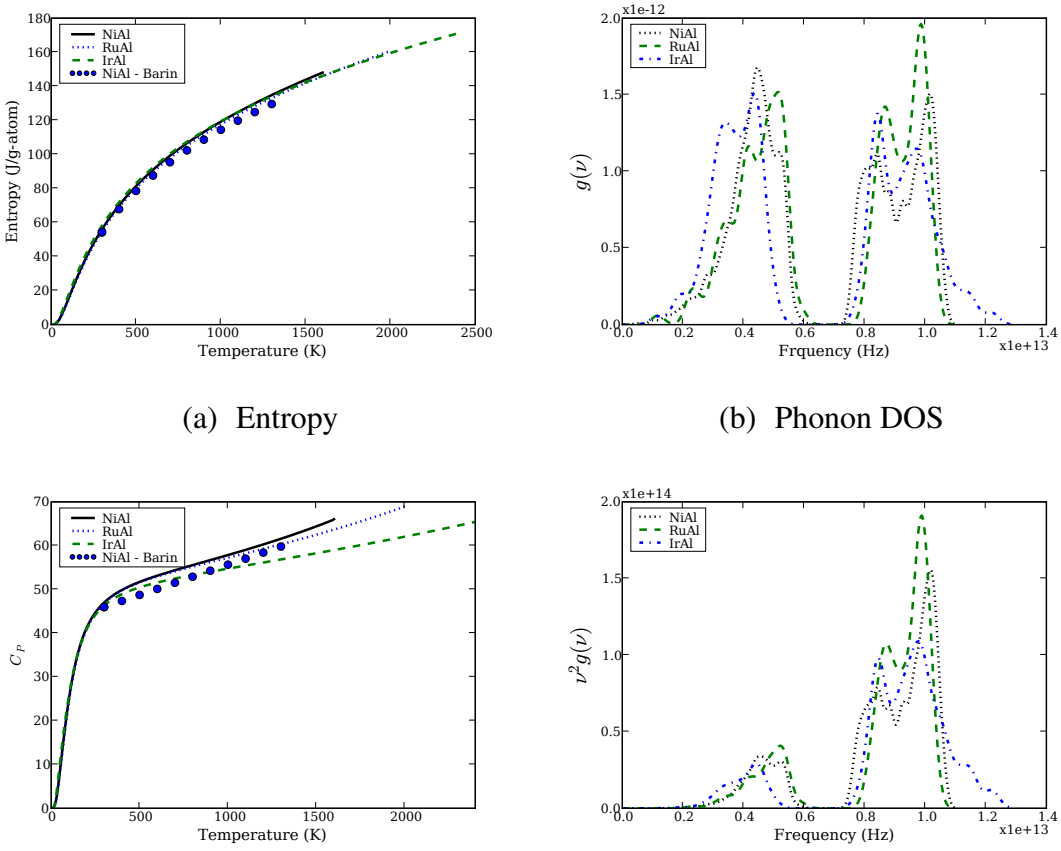


Fig. 23. Coefficient of thermal expansion for RuAl and IrAl. Experimental data for RuAl is that of Tryon *et al.* [92] and for IrAl is that of Hosoda *et al.* [89].

significantly heavier than either Ni or Ru which could be one explanation why the DOS of IrAl is weighted more to the lower frequencies than the other two. Overall, the DOS for all the systems are similar, and when included in the integral of (4.2) most of the differences in DOS will be smoothed out to yield very similar vibrational free energies. This fact, combined with the close predictions of the Debye temperatures found in Table VII result in the calculated entropy of the three systems being almost identical. The experimental values for the entropy of NiAl agree rather well with our calculations as shown in Fig. 24(a), while the corresponding experimental data sets for RuAl and IrAl are not believed to be currently available. The second moment of the phonon DOS is related to the Debye temperature derived from the heat capacity (Table VII). In Fig. 24(d), we present the second-moment density of states, $(\nu^2 g\,(\nu))$ calculated for the three B2 intermetallics considered in this work. Since Ir displays the lowest profile in the weighted density of states we would expect it to show the corresponding behavior in the specific heat, which is clearly shown as temperature increases in Fig. 24(c).

One of the important thermodynamic quantities for use in thermodynamic modeling is the enthalpy of formation. It is a measure of the enthalpy of a system minus the relative



(a) Entropy

(b) Phonon DOS

(c) Specific heat

(d) Frequency weighted DOS

Fig. 24. The connection between the thermodynamic properties such as entropy and the phonon DOS. In the upper right we see that the DOS for the various systems are very similar. This is why the calculated entropies for the systems are so similar. In the lower right we show a *frequency weighted* DOS which is used in the calculation of the specific heat as shown in the lower left.

enthalpies of its constituents. This quantity provides a measure of the system's thermodynamic stability and is very useful in the construction of phase diagrams. The total enthalpy of the systems of interest is shown in Fig. 25 and shows a similar relationship between the three systems as demonstrated for the entropy, being that they are almost equal. Again, correlation between the experiment and calculated values for NiAl is excellent. The enthalpy of formation is then plotted in Fig. 26 along with the values of Rzyman for NiAl. The ground state enthalpy of IrAl (in kJ/g-atom) is calculated to be -691 compared to that of RuAl at -681. The difference in the enthalpies of formation comes from the fact that the ground state enthalpy of pure Ir is calculated to be -846 and that of Ru is predicted to be -880, only an approximate 4% difference. This tendency of Ru to energetically prefer the pure phase more than Ir does results in a significant difference between the formation enthalpies of IrAl and RuAl.

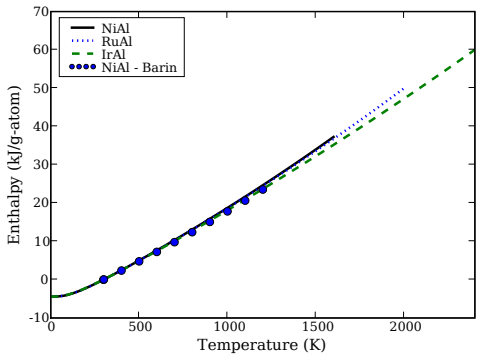


Fig. 25. Relative enthalpy for B2 systems. Experimental data is for NiAl and is taken from the compilation of Barin [82].

With the thermodynamic quantities thoroughly characterized and showing overall good agreement with experimental values we proceed with a presentation of the calculation of the elastic constants of the three B2 systems we have been examining. Fig. 27 demonstrates that the same basic patterns established for the pure elements carry over to the binary systems.

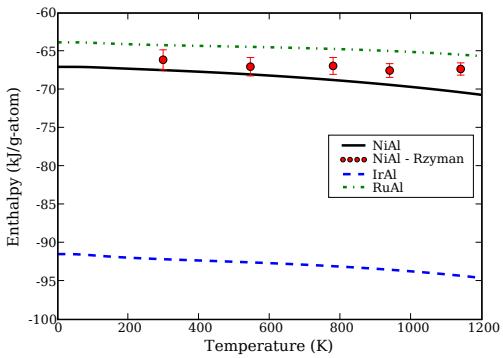


Fig. 26. Enthalpy of formation for B2 systems. The NiAl experimental data is from the work of Rzyman [101].

The LDA has a tendency to overestimate and the GGA underestimate the elastic constants, thus producing a range wherein the actual C_{ij} lie.

The calculated bulk moduli are shown in Fig. 28 and demonstrate some useful results. IrAl is shown to be the least compressible, followed by RuAl and then NiAl. This strength is one of the key reasons why these materials are being considered for many applications. They all show similar softening with temperature although RuAl seems to drop off the least. The results for the bulk modulus are used in the calculation of the elastic constants in order to separate C_{11} and C_{12} so the validity of those results depends on a correct prediction of the bulk modulus.

Finally, the predicted temperature dependence of the single crystal elastic constants of RuAl and IrAl are presented in Fig. 29. These represent the aggregate of all the methods and assumptions used in this work. We expect these predictions to be accurate at least at low and intermediate temperatures but are unsure about how the anharmonic corrections of Oganov and Wallace will hold at very high temperatures. If reality demonstrates significant departures from the assumptions used then dynamic—possibly ab initio MD—rather than static tools would be best suited to analyze and understand the behavior of the elastic constants

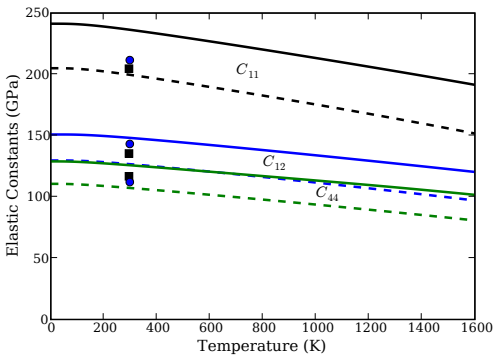


Fig. 27. Elastic constants for NiAl. The LDA (solid lines) and GGA (dotted lines) form upper and lower limits respectively for the prediction of elastic constants. The squares represent experimental data of Davenport *et al.* [102], while circles are the data from Simmons and Wang [57].

at those temperatures due to their ability to account for anharmonicity exactly [40]. These property predictions are presented as a theoretical baseline which we hope experimental and other modeling groups will take into account as they attempt to further characterize these materials.

E. Summary of Results and Conclusions

In this work we have presented many results of the prediction of finite-temperature thermodynamic and mechanical properties of NiAl, RuAl, IrAl as well as the pure elements which comprise them. Vibrational contributions to the thermal free energy were accounted for through the supercell method which was able to produce accurate phonon DOS and important constants such as the Debye temperature and Grúneisen constant. Electronic degrees of freedom were accounted for from the electron density of states and an anharmonic correction to the free energy was added according to the theory of Oganov. The GGA and LDA have been implemented in the same procedures and their results comapred and constrasted

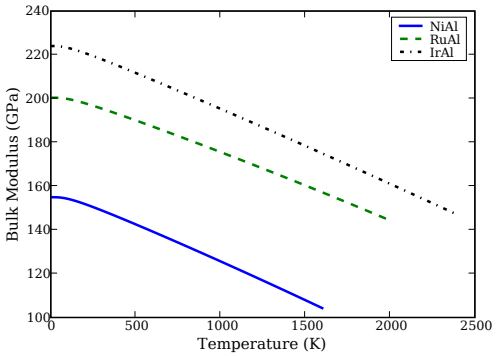
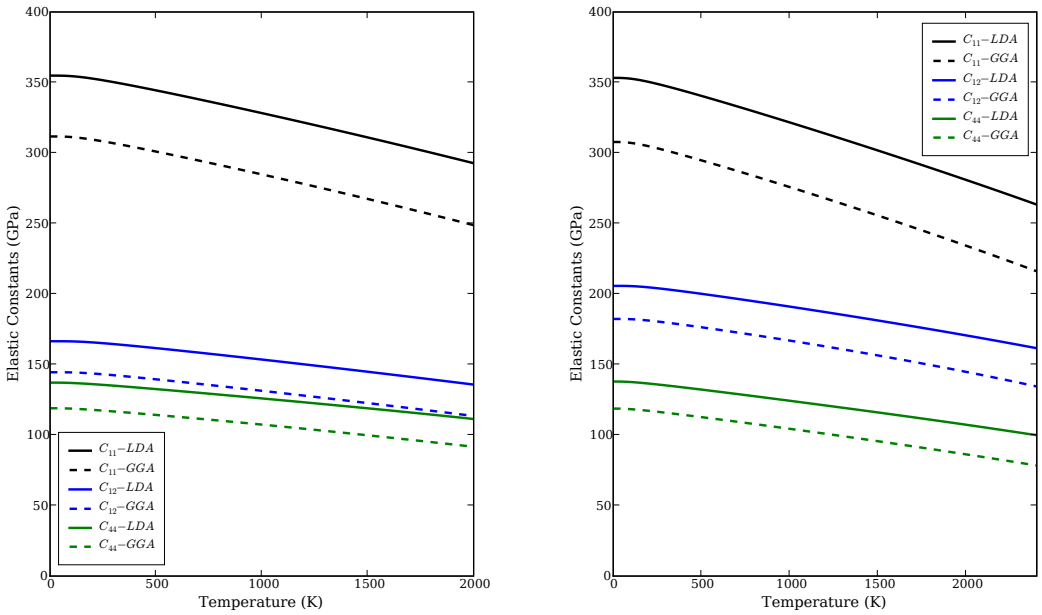


Fig. 28. Calculated bulk moduli for B2 systems within the GGA.



(a) Elastic constants if RuAl. (b) Elastic constants of IrAl

Fig. 29. ELC of RuAl and IrAl. Solid lines are from LDA calculations, dashed lines from the GGA.

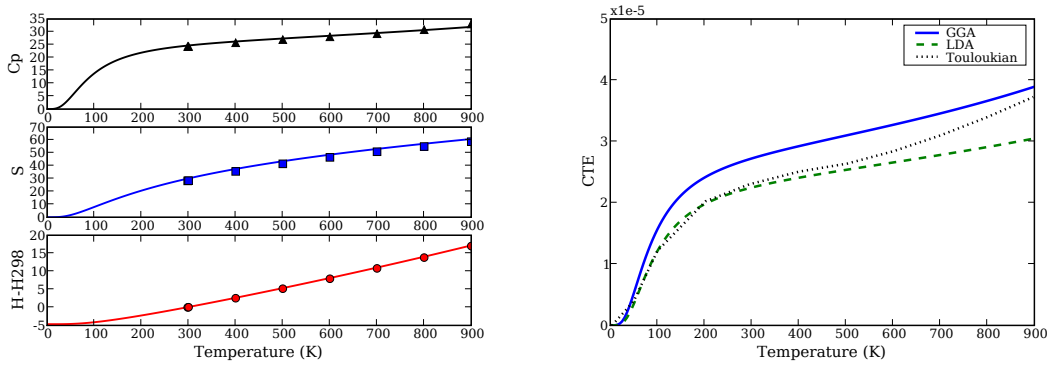
with each other showing strengths and weaknesses in each. The key findings of this work are:

- The predicted thermodynamic properties of the constituent elements agree extremely well with experiment
- The predicted elastic constants of B2 NiAl match well with the few tabulated values available
- The GGA and LDA form lower and upper bounds respectively for the prediction of elastic constants
- There is no clear 'best' approximation to the exchange correlation energy (GGA or LDA) for the calculation of all thermodynamic and thermo-mechanical properties. In future studies both should be considered
- The temperature dependence of the enthalpy and entropy of all three B2 phases are almost identical - this is due largely to the similarities in their respective Debye temperatures and phonon DOS
- We have verified the low enthalpy of formation for RuAl and IrAl and presented its temperature dependence
- The calculated single crystal elastic constants of RuAl and IrAl have been presented

F. Supplemental Materials

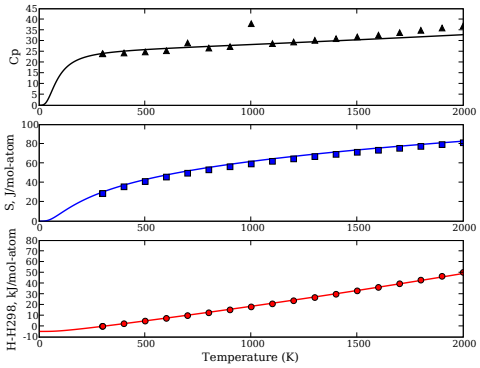
Here we include plots of thermodynamic quantities of the pure constituents that did not fit in the body of the text. Fig. 30 displays specific heat, entropy and relative enthalpy for pure aluminum. Fig. 31 shows the same data for pure ruthenium and Fig. 32 the same for pure iridium.

The CTE for Al is a bit low through most of the temperature region but follows the appropriate trends for softening with increased temperature. In the cases of Ir and Ru, the slope of the calculated CTE is significantly less than that of the experimental data throughout the entire temperature range for which data could be obtained.

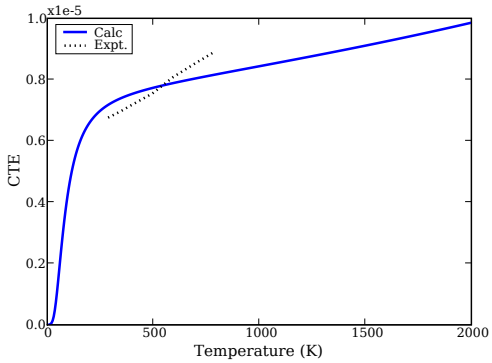


(a) Cp. S, H of Al (b) CTE of Al

Fig. 30. Thermodynamics of pure Al. Experimental data are taken from Barin [82] for the Cp, S, and H while CTE data is reported in [39].

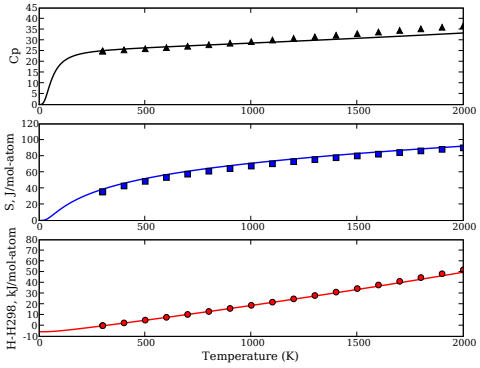


(a) Cp, S, H of Ru

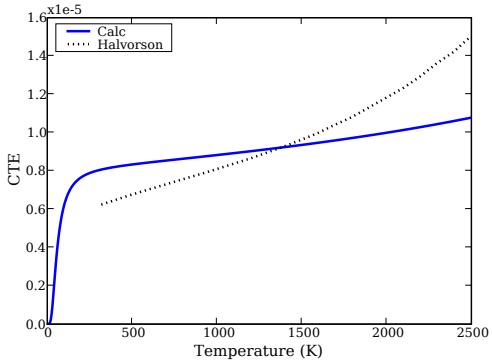


(b) CTE of Ru

Fig. 31. Thermodynamics of pure Ru. Experimental data for Cp, S, and H are from Barin [82] while the CTE experimental data is reported in [103].



(a) Cp, S, H of Ir



(b) CTE of Ir

Fig. 32. Thermodynamics of pure Ir. Experimental data for Cp, S, and H are from Barin [82] while the CTE experimental data is taken from the work of Halvorson [104].

CHAPTER V

AB INITIO MOLECULAR DYNAMICS AND ELASTIC CONSTANTS

As discussed previously, molecular dynamics has been well established as a means for the calculation of elastic properties of solids [48, 42, 40, 45]. One of the great advantages to molecular dynamics approaches to the calculation of temperature dependent properties is that it allows us to account for anharmonic effects [40] without the need of external corrections like have been employed within the DFT calculations. Traditional molecular dynamics also allows for simulations of hundred of atoms, allowing long range interactions to be studied. There are however drawbacks to traditional MD, such as the neglect of electronic effects and the need to have an adequately parametrized force field available for the simulation. For systems where force fields are not available it is impossible to implement traditional MD. Recent developments have lead to the implementation of ab initio molecular dynamics (AIMD) programs which in essence use DFT in place of interatomic force fields in a dynamic setting. At each step of the MD simulation, instead of calculating interatomic forces based on a used defined potential energy function, a quick DFT simulation is performed and the ions moved and energies updated accordingly.

The purpose of this section is to explore the possibilities of extending the calculation of elastic constants to ab initio molecular dynamics. There are several trade-offs that must be made and questions that arise from the differences between traditional and ab initio MD. This section will address several of those concerns and how they can be discussed. Currently this is a work in progress with computer code being developed to automate the AIMD simulations and perform the appropriate post-processing. We begin this section by laying the groundwork theory and discussing how this could be implemented. We then discuss some factors to be considered in developing MD simulations that will be suitable for the extraction of elastic constants. Following this we describe how to post-process the

AIMD data to get the elastic constants and display some of the challenges that remain to be overcome to make this process reliable and accurate. Finally we close with details of how to implement AIMD using VASP.

A. Theory

The determination of elastic constants from MD simulations requires the calculation of three main contributing terms, the potential energy or Born term, the kinetic energy term, and a contribution that arises from fluctuations in the microscopic stress tensor,

$$C_{ijkl} = C^B_{ijkl} + C^K_{ijkl} - C^\sigma_{ijkl}. \tag{5.1}$$

The Born term is directly related to the 0K elastic constants and requires the most care when adapting classical MD to ab initio MD techniques. The kinetic and stress contributions are fairly straitforward and will be briefly outlined first.

The kinetic term is calculated from the thermal contributions to the total system energy by

$$C^K_{ijkl} = \frac{2nk_BT}{V}\left(\delta_{ik}\delta_{ij} + \delta_{il}\delta_{kj}\right), \tag{5.2}$$

where n is the number of particles in the system and δ_{ij} are the conventional Kronecker delta. The stress fluctuation contribution is calculated by calculating ensemble averages in the fluctuations of the microscopic stress tensor,

$$C^\sigma_{ijkl} = \frac{V}{k_BT}\left(\langle\sigma_{ij}\sigma_{kl}\rangle - \langle\sigma_{ij}\rangle\langle\sigma_{kl}\rangle\right), \tag{5.3}$$

where $\langle\rangle$ denote the averages over time. This term is the reason for setting *ISIF = 2* in the INCAR file, so that it will write the stress tensor at each time step. The two terms can easily be calculated from the trajectory information that VASP writes to the *OUTCAR* and *vasprun.xml* files.

The calculation of the Born term is significantly more difficult due to the fact that it depends on second derivatives of the potential energy with respect to perturbations in various directions.

$$C^B_{ijkl} = \frac{1}{4}\left(\hat{C}^B_{ijkl} + \hat{C}^B_{jikl} + \hat{C}^B_{ijlk} + \hat{C}^B_{jilk}\right), \tag{5.4}$$

with

$$\hat{C}^B_{ijkl} = \frac{1}{V}\sum_{m=1}^{N}\sum_{n=1}^{N}\langle r^m_j r^n_l \frac{\partial^2 U}{\partial r^n_k \partial r^m_i}\rangle + \delta_{ik}\langle\sigma^v_{jl}\rangle, \tag{5.5}$$

where σ^v is the symmetric virial tensor,

$$\sigma^v_{jl} = \frac{-1}{2V}\sum_{m=1}^{N}\left[r^m_i \frac{\partial U}{\partial r^m_j} + r^m_j \frac{\partial U}{\partial r^m_i}\right]. \tag{5.6}$$

The equation (5.4) is used to symmetrize the Born term by taking the average of each possible combination of i, j, k, l.

In classical MD, the second derivative of the potential energy can be calculated analytically from the parametrized potential function and programmed into the simulation. One of the primary goals in using ab initio MD is to free ourselves from the constraint of needing such a potential energy function. The key then to the calculation of the Born term is to find a numerical approach to calculate the necessary derivatives. Originally we attempted to calculate the second derivative through finite differences in the forces and positions as reported in the MD trajectory data. This seemed like a logical choice until we realized that every time an atom changes direction, $\partial r \rightarrow 0$ and therefore a $\partial f/\partial r$ would have a zero denominator. This results in extreme numerical spikes in the data making any averaging extremely unreliable. Also since $\partial F/\partial r$ must be taken while all other atoms are stationary this approach is mathematically incorrect and another method for developing the partial derivatives numerically is necessary.

Based on the work of Yoshimoto *et al.* [105] we have developed an alternate method for the calculation of the Born term numerically. Each second derivative must be calculated

independently using finite differences. In the case of an N particle system each atom has to be moved a tiny distance in each of the x, y, z directions and the resulting changes in interatomic forces calculated in each direction for each atom. The terms $(\partial^2 U)/(\partial r_k^n \partial_i^m)$ can be summarized in a Hessian matrix which if each of the x, y, z directions is taken into account for the i, k results in a 3N x 3N matrix. Conveniently VASP has a built in routine to calculate the Hessian of a given system. Therefore to calculate the Born term using a finite difference Hessian we must extract the positions of the atoms at several time steps from the MD run, calculate the Hessian based on those positions and then calculate the Born term from the Hessian. The calculation of the Hessian involves $3m * N$ static calculations where m is the number of finite difference steps taken for each atom in each direction.

In order to limit the computational cost of calculating $3m * N$ static calculations over hundreds or thousands of timesteps it is recommended to implement a convergence algorithm such as:

1. Extract instantaneous positions of random timesteps from the MD trajectory
2. Calculate the Hessian matrix based on the instantaneous positions of one timestep
3. Calculate the Born contribution to the ELC from the positions and the Hessian
4. Check for convergence between this step and the running average
5. Repeat if convergence criteria not met

to calculate the Born term. This algorithm can also be written to use multiple threads so that several timesteps can be calculated concurrently thus reducing the real time wait for convergence on a multi-core CPU.

B. Moleculary Dynamics Methodology

The general procedure for developing an MD simulation which is suitable for elastic constant calculations from statistical fluctuation formula is outlined as follows.

1. Create the desired MD cell and relax it at 0K to get the ions in their proper positions
2. Thermalize the system by slowly bringing it up to the desired temperature
3. Equilibrating the pressure by iterating through possible lattice parameters to converge on a state of zero pressure
4. Run an extended NVE simulation
5. Extract ELC terms from trajectory data

The length of the simulation needs to be determined by how the three independent terms mentioned previously converge over time. Ray [45] and others [42] have shown that the Born and kinetic terms converge rapidly, generally within 500 to 1000 timesteps. This is advantageous since the calculation of the Born term is computationally expensive. Ray has shown that the limiting factor for how long a simulation should be run is how long it takes for the stress fluctuation term to converge, typically on the order of 25000 steps or more. Without running such test with AIMD it is impossible to say for sure how long it will take for the stress term to converge so initially we would expect similar convergence to that of the classical MD until this could be tested.

There are several factors that should be taken into account in validating the quality of an MD simulation and therefore its useability to ELC calculations. The first is that the simulation obeyed the constraints set upon it. In the case of the current problem we are looking at fluctuations during an NVE run where number of particles, volume, and energy are all conserved throughout the simulation. Number of particles and volume are easy

to maintain constant while energy is a bit more challenging. Throughout the course of a simulation, numerical errors begin to compound one upon another and produce a sort of 'drift' in the total energy of the system. In order to maintain the integrity of the simulation it is important that this drift be a little as possible. Fig. 33 shows an example of energy drift during a 5000 step NVE MD run in VASP. Since the drift is so small we can assume that

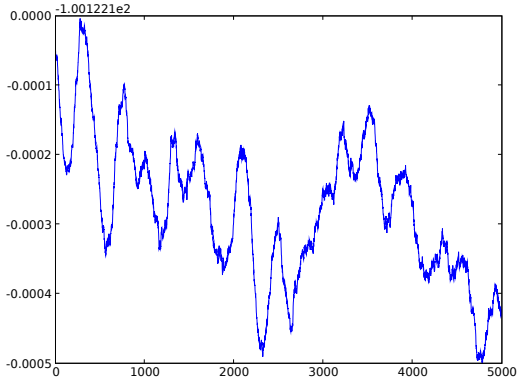
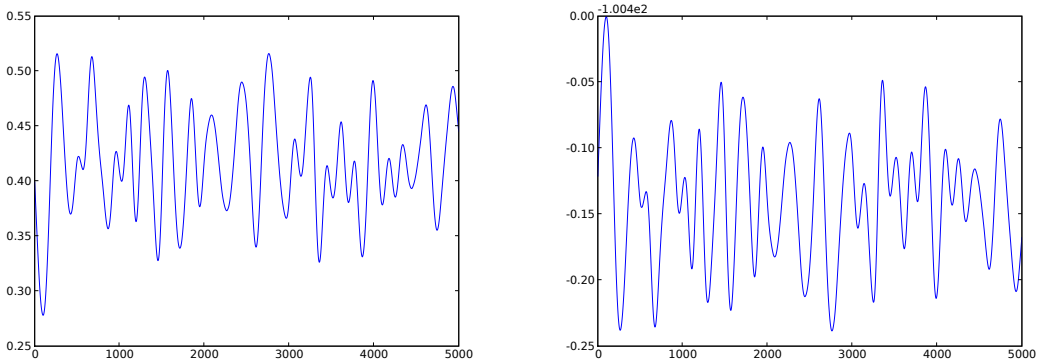


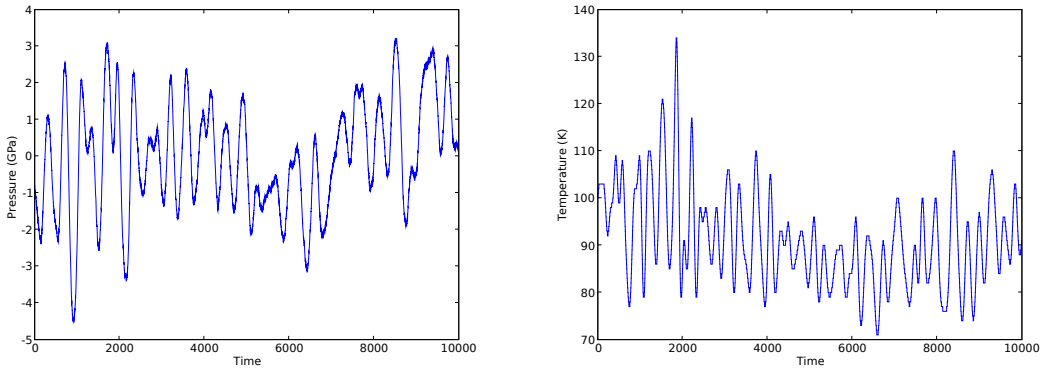
Fig. 33. Energy drift for a 5000 step AIMD run. This shows the magnitude of the fluctuations. The zero is actually at approximately -100eV meaning the drift in total energy is on the order of 5E-4 %.

the run is satisfactorily an NVE simulation. In order to maintain constant total energy, if the kinetic energy decreases, the potential energy must increase the same amount. Fig. 34 demonstrates the kinetic and potential energies over the same run. It is also important to check how the pressure and temperature fluctuate over time during the MD run as shown in Fig. 35 to ensure that the system is well behaved and truly at a state of relative equilibrium. In this case we can see that the actual mean of the temperature over time was approximately 90 K instead of the desired 100 K. Once the system is truly converged to equilibrium pressure and temperature and the energy is shown to be conserved over the MD simulation then it is time to process the trajectory data and run the calculations to extract the various elastic constants.



(a) Kinetic Energy (b) Potential Energy

Fig. 34. Kinetic and potential energy fluctuations over a 5000 step MD simulation.



(a) Pressure (b) Temperature

Fig. 35. Pressure and temperature fluctuations over a 10000 step MD simulation. The pressure plot has been normalized around the mean to show that for this run, the magnitude of the pressure fluctuations was normally less than $\pm$ 4 GPA.

C. Results to Date

So far our efforts have been focused on the calculation of the Born term according to the model discussed previously. While the process should be fairly straightforward there are several factors which influence the numerical methods used and hence the accuracy of the final product. For now we focus on the first term of (5.5) and omit the term containing the virial tensor which will have a relatively small effect on the total elastic constants.

The term we are calculating represents the static elastic constants at 0 K and theoretically should match up well with results obtained from other methods of calculating the elastic constants at the ground state. As previously mentioned, at the heart of the Born term is $\partial^2 U/\partial r_k^n \partial r_i^m$ which can be summarized in the Hessian matrix of the potential energy of the system. It is believed that since VASP has a built in function for calculating the Hessian, this term should be straightforward to calculate. The Hessian is formed by taking each atom individually and moving it a finite distance in each direction and measuring the change in the potential energy of the system. Unfortunately, the numerical methods used in calculating finite differences in the potential energy are extremely erratic and we have not yet been able to find suitable parameters for accurate Hessian matrix calculations.

The results of our most recent test are found in Table VIII. Ideally C11 should be about 110 GPa and C44 should be about 35 GPa. C12 has been omitted since in all the runs it is coming out to be effectively 0 when it should demonstrate a value of approximately 60-65 GPa. This is a cause of great concern yet it is currently unknown why this shear mode is completely absent from the Hessian calculations. The predictions of C44 are constant throughout, mostly independent of the parameters used to calculate the Hessian. Unfortunately, the value predicted is exactly twice what we would expect it to be. The σ_{C11} column is a measure of the standard deviation of C11, C22, and C33. Due to the symmetry of a cubic system these three terms should be equal and therefore it is important that the

Table VIII. Born term results.

Finite Displacements	Δr	kpoint mesh	C11	σ_{C11}	C44
2	0.015	1x1x1	348	24.55	71
4	0.015	1x1x1	168	3.34	71
2	0.015	2x2x2	175	1.04	72
2	0.015	3x3x3	387	1.74	68
2	0.010	1x1x1	382	4.06	71
2	0.030	1x1x1	223	2.05	71
2	0.050	1x1x1	249	1.67	72
4	0.010	1x1x1	308	9.7	71
4	0.050	1x1x1	263	1.06	72

calculations yield terms with a very small standard deviation. The first run yields a very high standard deviation while just about any other trial yields significantly better results. The difficulty comes in when trying to develop a systematic way of finding convergence among the terms. It is clear that increasing the number of k-points to a 2x2x2 mesh drastically reduced the variance among these three terms, but when the mesh was refined further, the variance grew. Also, both decreasing and increasing the Δr reduced the σ_{C11}.

When looking at the magnitude of C11 there are huge differences depending on the parameters used in Table VIII, again with no apparent pattern. First to note is that the calculated magnitudes all overestimate C11 drastically from about 50% over the expected value to over 250%. If the k-point mesh is increased to a 2x2x2 the value of C11 drops to about half of the value at a 1x1x1 mesh, and if the mesh is further refined to a 3x3x3, the value shoots up to more than the value at a 1x1x1 mesh. As the number of displacements and

Δr are varied C11 fluctates a lot and it is impossible to determine the optimal parameters to be used from the data available.

D. Ab initio Molecular Dynamics with VASP

Ab initio MD is a built in feature of VASP, all that is required of the user is to put the appropriate parameters in the INCAR file and the system will do the rest. The VASP manual provides examples and explanations of the various parameters and the reader is referred to that document for details. A sample INCAR file is included below and is commented in italics to show what the various keywords do.

```
SYSTEM = MD
Things you should never have to change
IALGO = 48
LREAL = A
NELMIN = 4
BMIX = 2.0
MAXMIX = 50
ISYM = 0
NBLOCK = 1
KBLOCK = 1
IBRION = 0 This is the command which tells VASP to run MD

Things you may want to change
ISIF = 2 Calculate the stress tensor - needed for ELC calcs.
LWAVE = .FALSE. Same as for static VASP
LCHARG = .FALSE.
```

```
TEBEG = 200 The temperature at the beginning of the MD run
TEEND = 200 The temperature at the end of the MD run
NSW = 50000 The number of timesteps
POTIM = 1.0 The timestep in fs
SMASS = -3 Which type of MD: -3=NVE, -1=NVT or T-scaling
```

The first section of the INCAR file contains basic recommendations from the VASP manual. The details of which will not be discussed here. The second section demonstrates the few commands that need to be changed depending on the simulation desired. For dynamic problems not requiring the instantaneous stress tensor omitting the *ISIF* flag would be recommended. The program would then revert to the default and save computation time. The *LWAVE* and *LCHARG* could be useful if repetitive calculations are needed, but in our case they are unnecessary. The last four parameters in the model *INCAR* file are self explanatory and are the ones that will get changed the most often.

In order to calculate the Hessian Matrix in VASP the IBRION, POTIM AND NFREE parameters are key. Below is a sample INCAR file for automating the calculation of the Hessian matrix.

```
SYSTEM = Hessian
NSW = 1
ISTART = 1
IBRION = 5 This is the command to calculate the Hessian
POTIM = 0.015 Δr
NFREE = 2 Number of finite displacements
LREAL = F
ISYM = 0 For some reason, the symmetry must be turned off or the
ELC calculation are not symmetric
```

This INCAR file will automatically calculate the Hessian using 2 displacements of each

atom at $\pm\Delta r$ in each direction for each atom. Once all the finite displacement calculations are done the results are stored in the OUTCAR file.

E. Future Work

The methodology for the calculation of the elastic constants from ab initio MD has been laid out and several programs have been developed to automate calculations and post-process data. At this point, the greatest challenge to solve center around the calculation of the Born term. The Hessian as given by VASP should be sufficient to calculate this term but so far we have been unable to find a systematic method for doing so accurately. Also, C12 is completely ignored, a fact which must be addressed by further looking at the Hessian and how it is calculated. An alternate method for calculating the Born term could involve the calculation of strain energies at several random snapshots of the MD simulation. This method is similar to that used for static elastic constant calculations, with the advantage of sampling the energies of the atoms away from their equilibrium positions. Preliminary tests show good convergence for this method but the magnitude of the C11-C12 modulus is overestimated. Further testing and development of this technique could make the finite displacement technique unnecessary.

Once the problems with the Born term are solved the next step would be to develop a systematic way of thermalizing and equilibrating pressure on a system and then testing for convergence of the stress fluctuation term to determine how long the simulations should be run. At that point everything should be in place to run MD simulations at several temperatures and extract the elastic constants and their temperature dependence. The theories discussed are sound, but so far the numerical methods are lacking in their ability to provide an accurate representation of the elastic behavior of a cubic system at this point.

CHAPTER VI

SOFTWARE DEVELOPED

Throughout the course of this work the author has performed countless DFT calculations on the CAT supercomputing cluster of the Department of Chemical Engineering. The repetitive and time-consuming nature of the calculations lead to the development of a set of Python based computer codes to automate this work. There are two main advantages that the automation provided by the developed software provides for the user, less manual time overseeing and managing calculations and second absolute repeatability and elimination of human error. The DFT calculations are performed in a series of steps, each subsequent step relying on the previous. Often there is a lot of time spent waiting and watching for a job to finish. These scripts allow the user to set up two input files, run a single command and walk away until the entire process is done.

A. Job Preparation and Batch Management

The `run_vasp.py` program is the main program for running the several VASP calculations needed to predict thermodynamic and thermo-mechanical properties. At the command line it can be called with one of four options:

- `-ssc`
- `-t`
- `-c11`
- `-c44`

The `-ssc` command assumes that the current working directory contains the 4 necessary VASP input files (`POSCAR, POTCAR, INCAR, KPOINTS`). The program performs an

initial relxation calculation, monitoring the progress of the job in the cluster and upon completion of the relaxation calculation it changes the INCAR parameters to reflect those of a static self-consistent calculation and re-submits the job and monitors until completion. While this is a fairly simple function in comparison to the others that will be discussed, this script performs an oft-needed function while providing automation between the relaxation and static calculations and providing for exact repeatability.

In order to calculate therodynamic and thermo-mechanical properties using VASP we must first obtain the free energy surface in volume/temperature space. Since a given VASP run only yields a single point on that surface, there is an obvious need for several DFT calculations which span both the volume and temperature dimensions. The `-t` option and its methods completely automate the DFT calculations needed to construct this surface through the supercell approach. There is a very useful tool called the *Automated Theoretic Alloy Toolkit* or ATAT which was written to partially automate the quasi-harmonic lattice dynamics calculations. ATAT has functions which create the necessary supercells and perturbations needed to calculate force constants and also post-processes the DFT calculations to obtain the phonon DOS and thermal free energy. The development a full quasi-harmonic model using ATAT is a systematic yet intricate process which requires significant user interference and attention. The purpose of the `run_vasp.py` script and the `-t` option is to automate not only each step but the entire process of the quasi-harmonic lattice dynamics. The old process of using ATAT from the user perspective was:

1. Submit an initial relaxation calculation of the basic structure in order to relax the lattice and move the ions in their ground state positions.

2. Wait for relaxtion calculation to finish.

3. Change the INCAR parameters to reflect a static self-consistent calculation (ssc) to ensure the ground state energy has been reached and then submit the job.

4. Wait for ssc calculation to finish.
5. Call the fitfc command from ATAT which creates the necessary quasi-harmonic volume directories with the necessary VASP input files.
6. Submit each volume directory calculation for a relaxation calculation (maintaining constant volume).
7. Wait for the volume relaxation calculations to finish.
8. Submit each volume directory for a ssc calculation.
9. Wait for volume SSC calculations to finish.
10. Call the fitfc command of ATAT again to create the necessary supercells for force constant calculations.
11. Submit the supercell calculations.
12. Wait for supercell calculations to finish.
13. Once the supercell calculations are done call the fitfc command one last time to perform post-processing and calculate the force constants, thermal free energy, phonon DOS etc.

At each DFT step the VASP calculations are typically done on a massively parallel system managed by some sort of batch system. Most often each job will have to be submitted to a queue, wait in queue until the requested resources are available, and then execute. This can slow down the user since even simple calculations that will only take a few minutes such as a relaxation calculation of a simple system often have to wait in queue for hours, with the operator constantly checking back.

As previously stated, the purpose of the `run_vasp.py -t` command is to automate the entire process outlined above so the user can simply enter one command and wait for the entire series of jobs to run automatically. In order to run the program the user must supply a vasp.in file as outlined below and optionally may provide an ELCparams.in file. The ELCparams.in file is a text file containing any changes to the default parameters (which will be outlined shortly). The vasp.in file is processed by the script `ezvasp` which is included as part of ATAT. It contains the INCAR parameters for a simple relaxation run and a modified version of the POSCAR section as shown below. The `ezvasp` command creates the four VASP input files (INCAR, POSCAR, KPOINTS, POTCAR) based on the vasp.in file. Included below is a sample `vasp.in` file which can be processed with the `ezvasp` command from the ATAT package.

```
[INCAR]
SYSTEM = FCC-Al
NEDOS = 1000
NELMIN = 8
ENCUT = 350
EDIFF = 1e-6
ISTART = 0
IBRION = 2
ISIF = 3
PREC = Accurate
NSW = 50
KSCHEME = Monkhorst-Pack
KPPRA = 10000
DOGGA
```

```
[POSCAR]
FCC-Al
1.0000000000
0.0000000000 2.0125000000 2.0125000000
2.0125000000 0.0000000000 2.0125000000
2.0125000000 2.0125000000 0.0000000000
D
0.0000000000 0.0000000000 0.0000000000 Al
```

There are two main sections of the `vasp.in` file, the INCAR and POSCAR sections which essentially are split up and formatted correctly to make the corresponding VASP input files. The INCAR section also has a few lines (in this case the last three) which are not VASP input parameters but rather serve as the necessary inputs for `ezvasp` to create the KPOINTS file and select the appropriate pseudopotential. Essentially they specify the type and density of the k-point mesh for Brioullioun zone sampling and the `DOGGA` command indicates that the GGA potentials should be used rather than the LDA. For further details on these parameters the reader is referred to the ATAT manual [73].

The ELCparams.in file is optional and gives the user to change any of the input parameters for the `run_vasp` script. If the file is absent all default values are used, while if it is present only the parameters which the user wishes to change from the defaults must be included, the others are automatically set to the default values. A listing of possible parameters with their defaults is given in Table IX.

With the vasp.in and optionally the ELCparams.in files in place the user must simply enter the command `run_vasp.py -t` at which point the user may walk away. The entire process to obtain a quasi-harmonic model of the system is automated and free of any need for user interaction. The cluster calculations are prepared, submitted and monitored automatically as shown in Fig. 36.

Table IX. Possible parameters and their default values for ELCparams.in.

Parameter	Default Value	Explanation
numberofposvolumes	5	The number of positive volumes to be included in the quasi-harmonic approximation (including volume 0)
numberofnegvolumes	3	The number of negative volumes to be included in the quasi-harmonic approximation (including volume -0)
maxnegvolume	-0.02	The maximum negative volume to be considered in the QH model
maxposvolume	0.04	The maximum positive volume to be considered in the QH model
minVCstrain	0	The minimum volume conserving strain to be used in the calculation of the ELC
maxVCstrain	0.04	The maximim volume conserving strain to be used in the calculation of the ELC
numVCstrains	5	The total number of volume conserving strains to be considered for ELC calculations
er	8.0	The er value to be used in the fitfc command (determines the size of the supercell to be used)
dr	0.05	The dr value to be used in the fitfc command
fr	4.0	The fr value to be used in the fitfc command
maxtemp	2000	The maximum temperature to consider when post-processing the data
mintemp	1.00E-005	The minimum temperature to consider when post-processing the data (a value of 0 will be over-ridden by this value since it leads to numerical error)
dTemp	1	The temperature step size for post processing
pertnodes	4	The number of supercomputer nodes to be used for a supercell calculation
PertPollTime	300	How often (in seconds) the queue should be checked for completed jobs during supercell calculations
relaxnodes	1	The number of supercomputer nodes to be used for relaxation and ssc calculations
RelaxPollTime	5	How often (in seconds) the queue should be checked for completed jobs during relaxation or ssc calculations
QueueToUse	MX1	The name of the cluster queue to be used
NumAtomsinSC	32	The number of atoms to have in the supercell (interacts with the er command above) âŁ“ this feature is still in testing
StrainThermo	False	Should the thermal free energy be calculated for each of the volume conserving strains of an ELC calculation

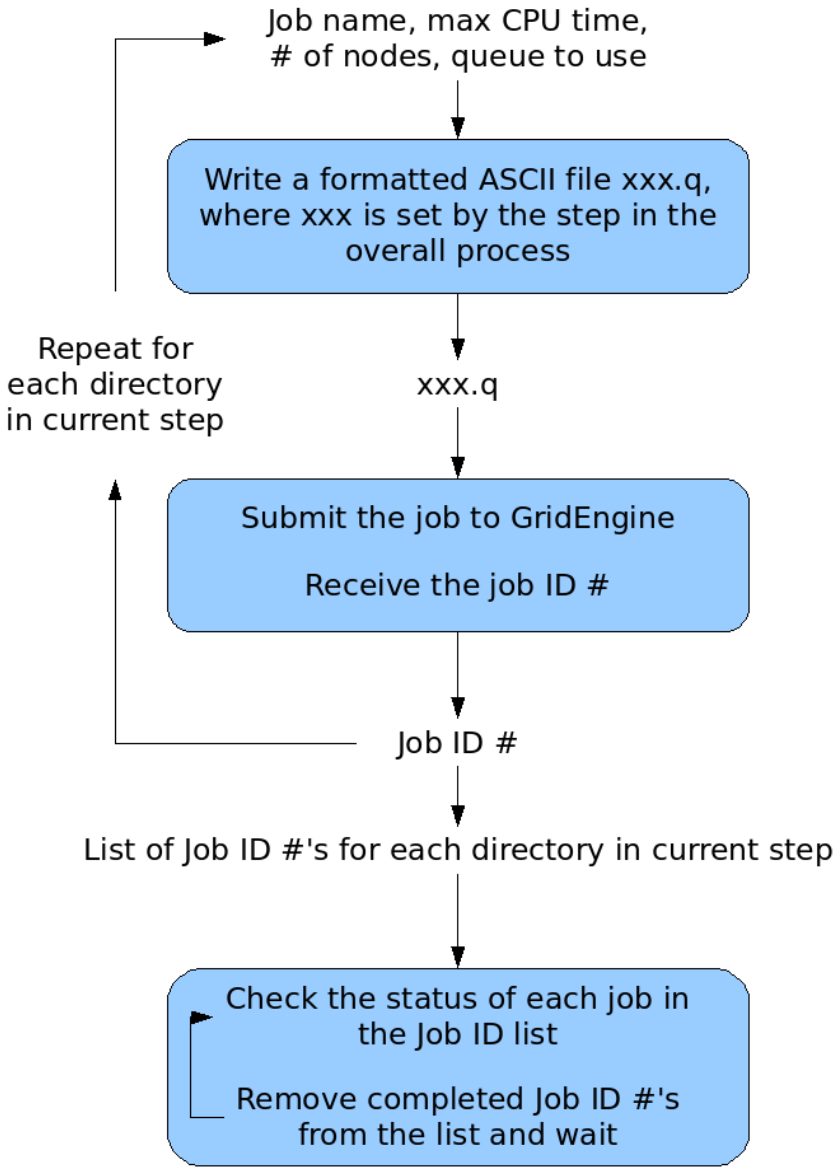


Fig. 36. Procedure for submission and monitoring jobs automatically.

One important feature of the `run_vasp.py` script is automatic error checking and the ability to resume work at the same step an error occurs. Often there are errors in the cluster computer or the user will kill jobs for some reason and one or more of the calculations anywhere in the process is terminated. Should this happen, the user must simply re-invoke the `run_vasp.py -t` command and the program will pick up at the step the error occured. As an example, the master node of the cluster could crash once two of seven supercell calculations were done and the other five were still in queue. Upon calling the `run_vasp.py -t` command the script would sense that all previous steps had been successfully completed, and that two of the supercell jobs were done. It would then re-submit the remaining five jobs and the user would be right back on track automatically. The flow of this automatic error checking is shown in Fig. 37.

Another key advantage of automating this process is that it eliminates human error when repeated situations are needed. When the process is done manually, there exists the potential of the user inadvertently entering the wrong parameters at a given step of the process. For example, the `-er` parameter for the `fitfc` command determines the size of the supercell to be used. In a manual process, this or any other parameter could accidentally be entered the incorrectly and the error go unnoticed until much later after hours of time has been wasted.

The `-c11` option for the `run_vasp.py` command performs a similar function to that of ther `-t` option but instead handles the pre-processing and job management necessary for the calculation of the finite temperature $C_{11} - C_{12}$ for a cubic system. The first step of the `-c11` option is to ensure that a full quasi-harmonic model is complete according to the parameters in the `ELCparams.in` file. If the thermodynamic model is not present or any part of it is missing the thermodynamic routine will be called first. Once this has been done the script performs the following sequence of tasks.

- Within each volume directory construct several strain directories

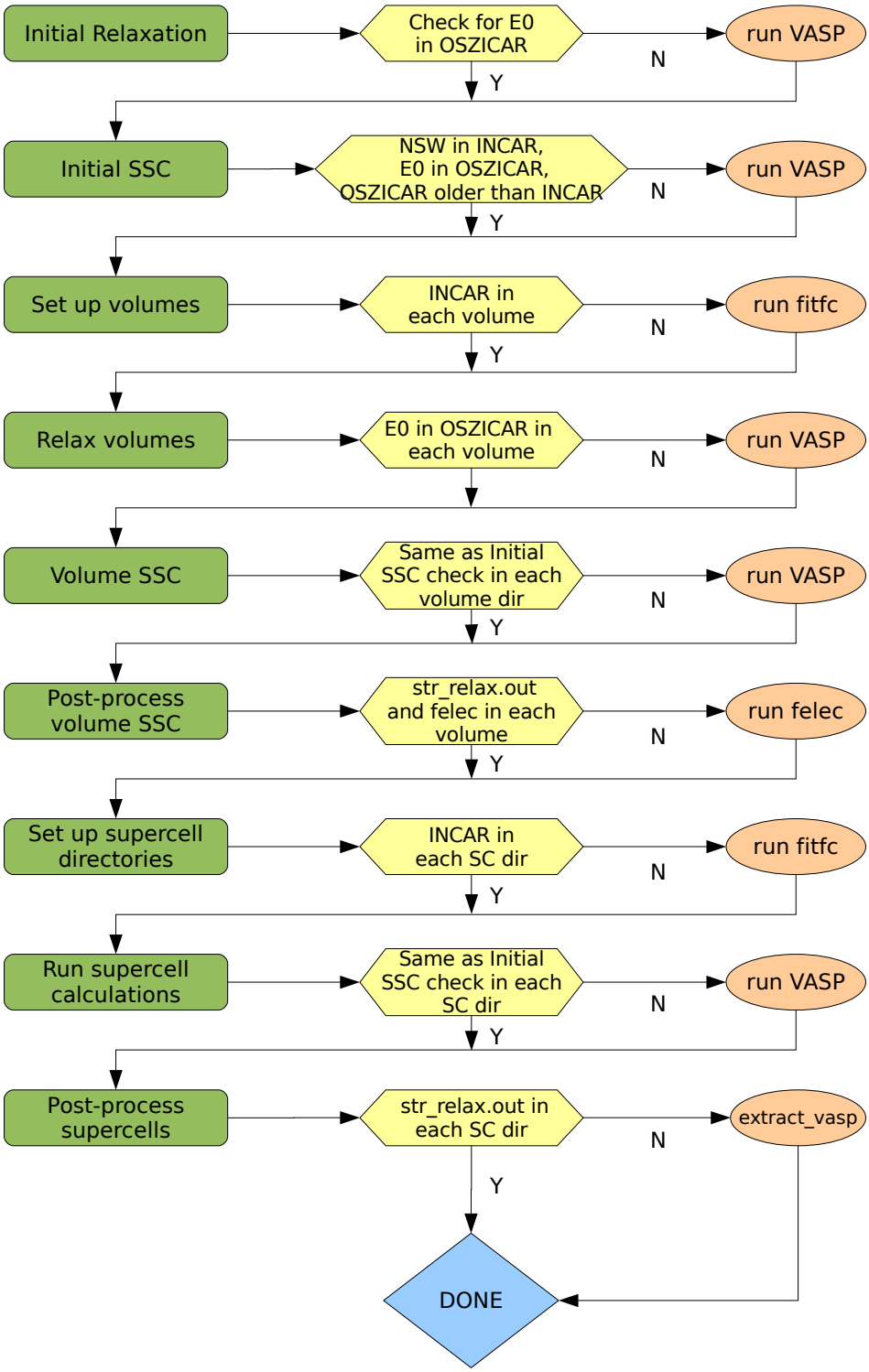


Fig. 37. Logical flow of run_vasp.py -t.

- In each strain directory change the POSCAR file to represent an appropriately strained lattice
- Copy the other necessary VASP input files into each strain directory
- Create a queue file for each calculation and submit each job to the cluster
- Watch the cluster for job completion

These tasks are significantly easier than those needed for the thermodynamic model but there are a lot of them. In a default run of `run_vasp.py -c11` there are seven volume directories with 5 strains in each volume for a total of 35 strain calculations. Without the script the user would have to manually set up each strain directory and scale the lattice vectors 28 times. The `-c44` option provides the same functionality as the `-c11` flag but creates different directories with a different lattice strain that is needed for the calculation of C_{44}. The overall flow of the various `run_vasp.py` options is depicted in Fig. 38.

B. Post Processing

Three main programs were developed for the extraction of thermodynamic properties and elastic constants from DFT calculations, `Thermodynamics.py`, `ELC.py`, and `C44.py`. `Thermodynamics.py` was written completely by Dr. Raymundo Arroyave and the other two were written by the author of the current work. Each depends on a complete set of calculations as output by the corresponding option of the `run_vasp.py` program. The function of these programs is to collect the necessary data from the DFT calculations and perform the mathematical manipulations following established formulas to calculate the properties of interest. All three programs rely on the same `ELCparams.in` file that was used with the `run_vasp.py` command ans whose parameters are outlined in the previous section. These parameters are used to direct the calculations as far as what the temperature

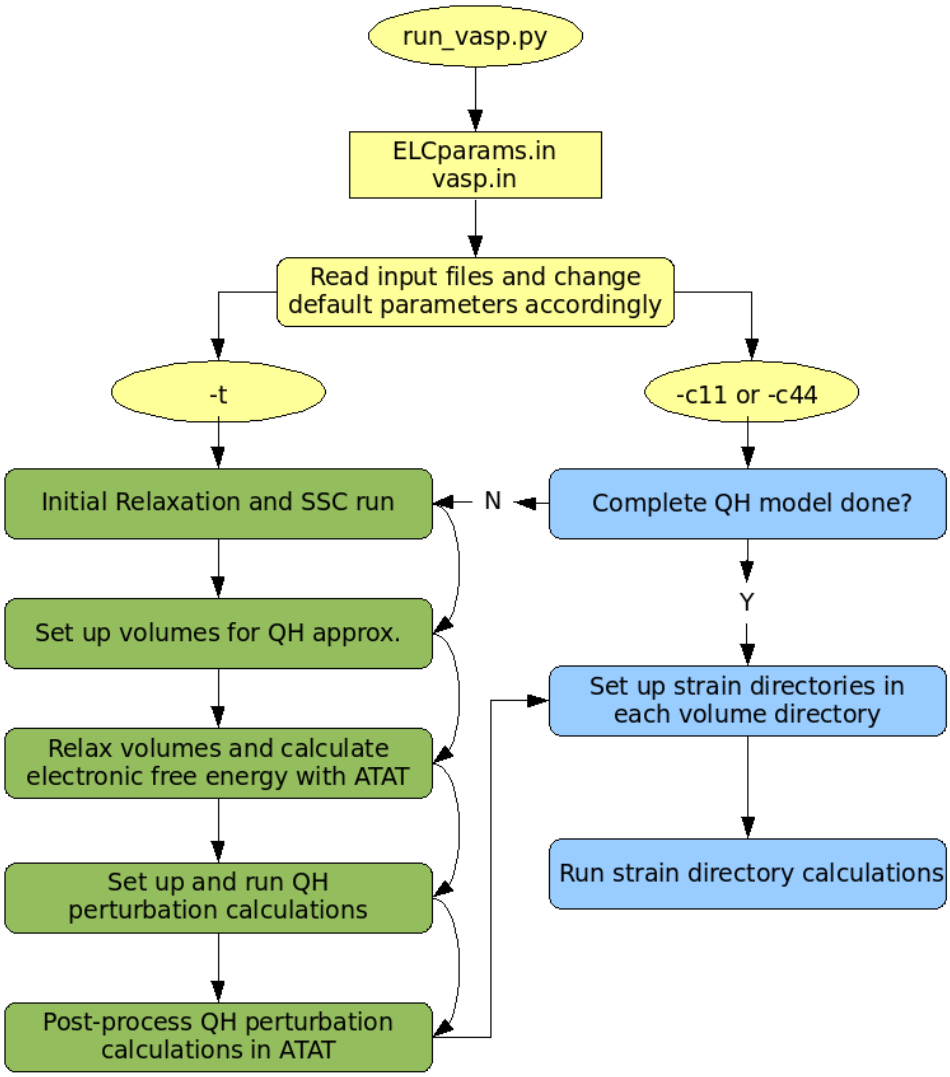


Fig. 38. Graphical representation of the run_vasp.py options.

step should be, what volume and strain directories should be included etc. Provided the user has a complete set of calculations in the current working directory the appropriate command must simply be called, the program will run and eventually the corresponding output files will be written. Most output files contain two or more columns of data with the first always representing the temperature and the other colums representing property values at that temperature. For example, the file `Entropy.dat` contains six columns. The first column is temperature and the others represent the total system entropy under various approximations (the last column includes all assumptions including electronic and anharmonic corrections to the free energy and is the one typically used). From these files the various plots can be made and relationships between properties established. For details on the exact workings of the code the reader is referred to the formulas cited throughout this work and the complete listing of documented source code which is contained in the appendix.

CHAPTER VII

SUMMARY

The objective of this work was to develop a procedure to calculate several themodynamic and thermo-mechanical properties of NiAl, RuAl and IrAl from first principles calculations. After a review of the basic theories of density functional theory, lattice dynamics, the harmonic and quasi-harmonic approximations and classical molecular dynamics we presented two journal articles detailing the details of our work.

In the first paper we presented a method for the prediction of thermodynamic and elastic properties of pure cubic metals at finite temperatures. The method is based on density functional theory and quasi-harmonic lattice dynamics to develop a free energy surface in volume/temperature space. From this surface we are able to extract thermal expansion and associated thermodynamic properties such as enthalpy, entropy and specific heat based on local slopes of the free energy surface. Effects from vibrational and electronic degrees of freedom were accounted for and an estimation of intrinsic anharmonicity was included as an addition to the free energy. Single crystal elastic constants were calculated through strain energy relations and fit to thermal expansion data to obtain their temperature dependence. We presented calculated results for aluminum and tungsten and showed the correlation with experiment to be good. The approximations made break down for tungsten at high temperatures and therefore our calculations are not reliable over approximately 60% of the melting point.

In the second paper we showed that the method developed for aluminum and tungsten applies to the B2 cubic phases of NiAl, RuAl and IrAl as well and that the same properties could be predicted for these binary intermetallics as for their constituents. In the prediction of these properties we followed the exact same procedures as for the unary systems and proved that the method is easily extended to these binary phases. The experimental data

for RuAl and IrAl is sparse but where available there was reasonable correlation between the calculations and experiments. Future work in this area should include the extension of the elastic constant calculations to systems with tetragonal and hexagonal systems. Also, magnetic degress of freedom and their effect on the total free energy of the system could be accounted for. Further study needs to be done to quantify and accurately parametrize intrinsic anharmonicity at high temperatures.

We also examined the possibility of implementing ab initio molecular dynamics for the prediction of elastic constants. This is still an emerging field with much work remaining. So far we have established basic procedures and algorithms for the numerical calculation of the Born term. Many things need to be investigates such as the best way to thermalize and equlibriate the pressure of an AIMD cell as well as to test how long it takes for the stress fluctuation term to converge. The limited results we have produces thus far have been able to produce ground state elastic constants of the right order of magnitude but are still very far from accurate. There is currently no clear choice of optimal parameters to use in the calculation of the Born term from finite displacements and this is an issue which must be resolved before this work can continue. It is hoped that with further study and discussion new methods will be implemented to overcome these difficulties. It is also of great importance to discover why the Born term as calculated from the Hessian matrix yields effectively 0 for C12. This is currently a mystery needing analysis and testing.

Finally, we presented an explanation of the various computer codes that were developed for the preparation, management and processing of the various DFT calculations. We used Python to automate the tasks wherever possible and provide a transparent user interface between VASP, ATAT, GridEngine, and our post-processing routines. The development of these codes is one of the highlights of this work since it allows complete repeateability with extremely little user interaction. Currently there are plans to release this code into the public domain for use and collaboration with other research groups. Hopefully more functionality

will be added and perhaps even a graphical user interface for preparing and monitoring the jobs as well as vizulization of results.

The project objectives have been met satisfactorily by developing a method for the prediction of elastic constants from first principles calculations. This is a small but significant step in both the advancement of multi-scale modeling of materials and the development of the next generation of ultra-high temperature materials.

REFERENCES

[1] W. D. Callister, *Materials Science and Engineering: An Introduction*, 7th ed. New York: Wiley & Sons, 2007.

[2] A. van de Walle and G. Ceder, "The effect of lattice vibrations on substitutional alloy thermodynamics," *Rev. Mod. Phys.*, vol. 74, no. 1, pp. 11–45, Jan 2002.

[3] J. Hafner, "Atomic-scale computational materials science," *Acta Materialia*, vol. 48, no. 1, pp. 71–92, 1 Jan. 2000.

[4] D. Furrer and H. Fecht, "Ni-based superalloys for turbine discs," *JOM Journal of the Minerals, Metals and Materials Society*, vol. 51, no. 1, Jan. 1999.

[5] G. B. Fairbank, C. J. Humphreys, A. Kelly, and C. N. Jones, "Ultra-high temperature intermetallics for the third millennium," *Intermetallics*, vol. 8, pp. 1091–1100, 2000.

[6] Y. Yamabe-Mitarai, H. Aoki, P. Hill, and H. Harada, "High-temperature mechanical properties of Ir-Al alloys," *Scripta Materialia*, vol. 48, pp. 565–570, 2002.

[7] Y. Yamabe-Mitarai, Y. Gu, C. Huang, R. Völkl, and H. Harada, "Platinum-group-getal-based Intermetallics as high-temperature structural materials," *JOM*, vol. 56, pp. 34–39, Sep. 2004.

[8] F. Mücklich and N. Ilić, "RuAl and its alloys. Part I. Structure, physical properties, microstructure and processing," *Intermetallics*, vol. 13, pp. 5–21, Jul. 2004.

[9] R. DeHoff, *Thermodynamics in Materials Science*, 2nd ed. Boca Raton, FL: Taylor & Francis Group, 2006.

[10] A. A. Quong and A. Y. Liu, "First-principles calculations of the thermal expansion of metals," *Phys. Rev. B*, vol. 56, no. 13, pp. 7767–7770, Oct 1997.

[11] G. J. Ackland, X. Huang, and K. M. Rabe, "First-principles thermodynamics of transition metals: W, NiAl, and PdTi," *Phys. Rev. B*, vol. 68, no. 21, p. 214104, Dec 2003.

[12] R. Arroyave, D. Shin, and Z. K. Liu, "Ab initio thermodynamic properties of stoichiometric phases in the Ni-Al system," *Acta Mater.*, vol. 53, pp. 1809–1819, 19 2005.

[13] G. Kern, G. Kresse, and J. Hafner, "Ab initio calculation of the lattice dynamics and phase diagram of boron nitride," *Phys. Rev. B*, vol. 59, no. 13, pp. 8551–8559, Apr 1999.

[14] J. Hafner, "Ab-Initio Simulations in Materials Science," [Online]. Available: http://cms.mpi.univie.ac.at/vasp-workshop/slides/comput_mat.pdf, 6 Mar. 2008.

[15] M. Mehl, B. Klein, and D. Papaconstantopoulos, "First-principles calculation of elastic properties of metals," [Online]. Available: http://cst-www.nrl.navy.mil/papers/metcij.ps, 1994.

[16] K. Capelle, "A bird's-eye view of density-functional theory," 2002. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/0211443

[17] J. C. Slater, "A simplification of the Hartree-Fock method," *Phys. Rev.*, vol. 81, no. 3, pp. 385–390, Feb 1951.

[18] P. Hohenberg and W. Kohn, "Inhomogeneous electron gas," *Phys. Rev.*, vol. 136, no. 3B, pp. B864–B871, Nov 1964.

[19] W. Kohn and L. J. Sham, "Self-consistent equations including exchange and correlation effects," *Phys. Rev.*, vol. 140, no. 4A, pp. A1133–A1138, Nov 1965.

[20] P. H. Acioli, "Review of quantum Monte Carlo methods and their applications," *Journal of Molecular Structure: THEOCHEM*, vol. 394, pp. 75–85, 1996.

[21] J. P. Perdew and A. Zunger, "Self-interaction correction to density-functional approximations for many-electron systems," *Phys. Rev. B*, vol. 23, no. 10, pp. 5048–5079, May 1981.

[22] D. M. Ceperley and B. J. Alder, "Ground state of the electron gas by a stochastic method," *Phys. Rev. Lett.*, vol. 45, no. 7, pp. 566–569, Aug 1980.

[23] J. P. Perdew and Y. Wang, "Accurate and simple analytic representation of the electron-gas correlation energy," *Phys. Rev. B*, vol. 45, no. 23, pp. 13 244–13 249, Jun 1992.

[24] J. P. Perdew, K. Burke, and M. Ernzerhof, "Generalized gradient approximation made simple," *Phys. Rev. Lett.*, vol. 77, no. 18, pp. 3865–3868, Oct 1996.

[25] J. Perdew, J. Chevary, S. Vosko, K. Jackson, M. Pederson, D. Singh, and C. Fiolhais, "Atoms, molecules, solids, and surfaces: Applications of the generalized gradient approximation for exchange and correlation," *Phys. Rev. B*, vol. 46, pp. 6671–6687, 1992.

[26] C. Filippi, D. J. Singh, and C. J. Umrigar, "All-electron local-density and generalized-gradient calculations of the structural properties of semiconductors," *Phys. Rev. B*, vol. 50, no. 20, pp. 14 947–14 951, Nov 1994.

[27] S. Clark, "Complex Structures In Tetrahedrally Bonded Semiconductors," Ph.D. dissertation, University of Edinburgh, 1994.

[28] Wolfram Quester Source, "Wikipedia - Pseudopotential", [Online]. Available: http://en.wikipedia.org/wiki/pseudopotential, 6 Mar. 2008.

[29] G. Kresse and D. Joubert, "From ultrasoft pseudopotentials to the projector augmented-wave method," *Phys. Rev. B*, vol. 59, no. 3, pp. 1758–1775, Jan 1999.

[30] G. Kresse and J. Hafner, "Ab initio molecular dynamics for liquid metals," *Phys. Rev. B*, vol. 47, pp. 558–561, 1993.

[31] ——, "Ab initio molecular-dynamics simulation of the liquid-metal-amorphous-semiconductor transition in germanium," *Phys. Rev. B*, vol. 49, pp. 14 251–14 269, 1994.

[32] G. Kresse and J. Furthmüler, "Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set," *Comput. Mat. Sci.*, vol. 6, pp. 15–50, 1996.

[33] G. Kresse and J. Furthmüller, "Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set," *Phys. Rev. B*, vol. 54, pp. 11 169–11 186, 1996.

[34] M. T. Dove, *Introduction to Lattice Dynamics*. New York: Cambridge University Press, 1993.

[35] M. Ohno, A. Kozlov, R. Arroyave, Z. K. Liu, and R. Schmid-Fetzer, "Thermodynamic modeling of the Ca-Sn system based on finite temperature quantities from first-principles and experiment," *Acta Materialia*, vol. 54, p. 4939, 14 Jun. 2006.

[36] D. C. Wallace, *Thermodynamics of Crystals*. Mineola, NY: Dover Publications Inc., 1972.

[37] A. R. Oganov and P. I. Dorogokupets, "Intrinsic anharmonicity in equations of state and thermodynamics of solids," *J. Phys.: Condens. Matter*, vol. 16, no. 8, pp. 1351–1360, 2004.

[38] M. Asta and V. Ozoliņš, "Structural, vibrational, and thermodynamic properties of al-sc alloys and intermetallic compounds," *Phys. Rev. B*, vol. 64, no. 9, p. 094104, Aug 2001.

[39] Y. Wang, Z. K. Liu, and L. Q. Chen, "Thermodynamic properties of Al, Ni, NiAl, and Ni3Al from first-principles calculations," *Acta Mater.*, vol. 52, pp. 2665–2671, 12 2004.

[40] T. Cagin and B. M. Pettitt, "Elastic constants of nickel: Variations with respect to temperature and pressure," *Phys. Rev. B*, vol. 39, no. 17, pp. 12 484 – 12 491, 15 1989.

[41] D. C. Rapaport, *The Art of Moleculary Dynamics Simulation*. Cambridge, UK: Cambridge University Press, 1995.

[42] T. Cagin and J. R. Ray, "Third-order elastic constants from molecular dynamics: Theory and an example calculation," *Phys. Rev. B*, vol. 38, no. 12, pp. 7940 – 7946, 15 1988.

[43] M. S. Daw and M. I. Baskes, "Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals," *Phys. Rev. B*, vol. 29, no. 12, pp. 6443–6453, Jun 1984.

[44] A. P. Sutton and J. Chen, "Long-range Finnis-Sinclair potentials," *Philosophical Magazine Letters*, vol. 61, no. 3, p. 139, 1990.

[45] Ray J. R., "Elastic constants and statistical ensembles in molecular dynamics," *Comput. Phys. Rep.*, vol. 8, no. 3, pp. 109–151, 1988.

[46] H. C. Andersen, "Molecular dynamics simulations at constant pressure and/or temperature," *J. Chem. Phys.*, vol. 72, no. 4, p. 2384, 15 Feb. 1980.

[47] M. Parrinello and A. Rahman, "Crystal structure and pair potentials: A molecular-dynamics study," *Phys. Rev. Lett.*, vol. 45, no. 14, pp. 1196–1199, Oct 1980.

[48] ——, "Strain fluctuations and elastic constants," *The Journal of Chemical Physics*, vol. 76, no. 5, pp. 2662–2666, 1982.

[49] Ray J. R., "Fluctuations and thermodynamic properties of anisotropic solids," *Journal of Applied Physics*, vol. 53, no. 9, pp. 6441–6443, 1982.

[50] J. R. Ray and A. Rahman, "Statistical ensembles and molecular dynamics studies of anisotropic solids," *Journal of Chemical Physics*, vol. 80, p. 4423, May 1984.

[51] J. F. Lutsko, "Generalized expressions for the calculation of elastic constants by computer simulation," *Journal of Applied Physics*, vol. 65, no. 8, pp. 2991–2997, 1989. [Online]. Available: http://link.aip.org/link/?JAP/65/2991/1

[52] Z. Zhou and B. Joós, "Fluctuation formulas for the elastic constants of an arbitrary system," *Phys. Rev. B*, vol. 66, no. 5, p. 054101, Aug 2002.

[53] D. Gerlich and E. Fisher, "The high temperature elastic moduli of aluminum," *J. Phys. Chem. Solids*, vol. 30, pp. 1197–1205, 1969.

[54] F. H. Featherston and J. R. Neighbours, "Elastic constants of tantalum, tungsten, and molybdenum," *Phys. Rev.*, vol. 130, no. 4, pp. 1324–1333, May 1963.

[55] J. Tallon and A. Wolfenden, "Temperature dependence of the elastic constants of aluminum," *J. Phys. Chem. Solids*, vol. 40, pp. 831–837, 2 1979.

[56] P. M. Sutton, "The variation of the elastic constants of crystalline aluminum with temperature between 63°k and 773°k," *Phys. Rev.*, vol. 91, no. 4, pp. 816–821, Aug 1953.

[57] G. Simmons and H. Wang, *Single Crystal Elastic Constants and Calculated Aggregate Properties*, 2nd ed. Cambridge, MA: M.I.T Press, 1971.

[58] D. Raabe, *Computational Materials Science*. Weinheim, Germany: Wiley-VCH, 1998.

[59] K. Ohno, K. Esfarjani, and Y. Kawazoe, *Computational Materials Science: From Ab Initio to Monte Carlo Methods*. Berlin, Germany: Springer-Verlag, 1999.

[60] G. B. Olson, "Computational design of hierarchically structured materials," *Science*, pp. 1237–1241, 1997.

[61] G. Ghosh and G. B. Olson, "Integrated design of Nb-based superalloys: Ab initio calculations, computational thermodynamics and kinetics, and experimental results," *Acta Mater.*, vol. 55, pp. 3281–3303, 2007.

[62] L. Chen, "Phase-field models for microsutrure evolution," *Annual Rev. Mater. Res.*, vol. 32, pp. 113–140, 2002.

[63] J. Z. Zhu, T. Wang, A. J. Ardell, S. H. Zhou, Z. K. Liu, and L. Q. Chen, "Three-dimensional phase-field simulations of coarsening kinetics of Gamma' particles in binary Ni-Al alloys," *Acta Mater.*, vol. 52, pp. 2837–2845, 2004.

[64] Z. H. Jin, P. Gumbsch, K. Lu, and E. Ma, "Melting mechanisms at the limit of superheating," *Phys. Rev. Lett.*, vol. 87, p. 055703, Jul 2001.

[65] N. Sandberg, B. Magyari-Köpe, and T. R. Mattsson, "Self-diffusion rates in Al from combined first-principles and model-potential calculations," *Phys. Rev. Lett.*, vol. 89, no. 6, p. 065901, Jul 2002.

[66] J. P. Perdew, K. Burke, and M. Ernzerhof, "Erratum: Generalized gradient approximation made simple," *Phys. Rev. Lett.*, vol. 78, p. 1396, 1997.

[67] P. E. Blöchl, "Projector augmented-wave method," *Phys. Rev. B*, vol. 50, no. 24, pp. 17 953–17 979, Dec 1994.

[68] M. Methfessel and A. T. Paxton, "High-precision sampling for Brillouin-zone integration in metals," *Phys. Rev. B*, vol. 40, no. 6, pp. 3616–3621, Aug 1989.

[69] X. Gonze, "First-principles responses of solids to atomic displacements and homogeneous electric fields: Implementation of a conjugate-gradient algorithm," *Phys. Rev. B*, vol. 55, no. 16, pp. 10 337–10 354, Apr 1997.

[70] S. Wei and M. Y. Chou, "Ab initio calculation of force constants and full phonon dispersions," *Phys. Rev. Lett.*, vol. 69, no. 19, pp. 2799–2802, Nov 1992.

[71] A. van de Walle and G. Ceder, "Automating first-principles phase diagram calculations," *J. Phase Equilib.*, vol. 23, pp. 348–359, 2002.

[72] A. van de Walle and M. Asta, "Self-driven lattice-model Monte Carlo simulations of alloy thermodynamic properties and phase diagrams," *Modelling. Simul. Mater. Sci. Eng.*, vol. 10, pp. 521–538, 2002.

[73] A. van de Walle, M. Asta, and G. Ceder, "The alloy theoretic automated toolkit: A user guide," *CALPHAD Journal*, vol. 26, p. 539, 2002.

[74] H. B. Callen, *Thermodynamics and An Introduction to Thermostatistics*, 2nd ed. New York: Wiley, 1985.

[75] R. Arroyave and Z. K. Liu, "Intermetallics in the Mg-Ca-Sn ternary system: Structural, vibrational and thermodynamic properties from first principles," *Phys. Rev. B*, vol. 74, p. 174118, 2006.

[76] M. J. Winter, "WebElements Periodic Table," [Online]. Available: http://www.webelements.com, 6 Mar. 2008.

[77] J. L. Gersten and F. W. Smith, *The Physics and Chemistry of Materials*. New York: John Wiley & Sons Inc., 2001.

[78] "Periodic Table," [Online]. Available: http://www.infoplease.com/periodictable.php, 6 Mar. 2008.

[79] Q. Chen and B. Sundman, "Calculation of debye temperature for crystalline structures - a case study on Ti, Zr and Hf," *Acta Mater.*, vol. 49, pp. 947–961, 2001.

[80] F. R. Kroeger and C. A. Swenson, "Absolute linear thermal-expansion measurements on copper and aluminum from 5 to 320 K," *J. Appl. Phys.*, vol. 48, no. 3, pp. 853–864, 1977.

[81] F. C. Nix and D. MacNair, "The Thermal Expansion of Pure Metals: Copper, Gold, Aluminum, Nickel and Iron," *Phys. Rev.*, vol. 60, pp. 597–605, 15 1941.

[82] I. Barin, *Thermochemical Data of Pure Substances*. Germany: VCH, 1989.

[83] L. Dubrovinsky and S. Saxena, "Thermal expansion of periclase (MgO) and tungsten (W) to melting temperatures," *Phys. Chem. Miner.*, vol. 24, pp. 547–550, 1997.

[84] W. P. Mason, *Piezoelectric Crystals and Their Application to Ultrasonics*. NY: D. Van Nostrand Company, Inc., 1950.

[85] J. R. Ray and M. C. Moody, "Calculation of elastic constants using isothermal molecular dynamics," *Physical Review B*, vol. 33, no. 2, pp. 895–899, 15 Jan. 1986.

[86] P. Brüesch, *Phonons, Theory and Experiments*. NY: Springer-Verlag, 1982.

[87] G. N. Kamm and G. A. Alers, "Low-temperature elastic moduli of aluminum," *J. Appl. Phys.*, vol. 35, no. 2, pp. 327–330, 1964.

[88] A. F. Guillermet and G. Grimvall, "Analysis of thermodynamic properties of molybdenum and tungsten at high temperatures," *Phys. Rev. B*, vol. 44, no. 9, pp. 4332–4340, Sep 1991.

[89] H. Hosoda, S. Miyazaki, and S. Hanada, "Potential of IrAl base alloys as ultrahigh-temperature smart coatings," *Intermetallics*, vol. 8, pp. 1081–1090, 2000.

[90] R. L. Fleischer and D. W. McKee, "Mechanical and oxidation properties of AlRu-based high-temperature alloys," *Metallurgical and Materials Transactions A*, vol. 24, no. 3, pp. 759–763, Mar. 1993.

[91] G. B. Fairbank, C. J. Humphreys, A. Kelly, and C. N. Jones, "Ultra-high temperature intermetallics for the third millennium," *Intermetallics*, vol. 8, pp. 1091–1100, Sep. 200.

[92] B. Tryon, T. M. Pollock, M. F. X. Gigliotti, and K. Hemker, "Thermal expansion behavior of ruthenium aluminides," *Scripta Materialia*, vol. 50, pp. 845–848, 2004.

[93] S. Narasimhan and S. de Gironcoli, "Ab initio calculation of the thermal properties of Cu: Performance of the LDA and GGA," *Phys. Rev. B*, vol. 65, no. 6, p. 064302, Jan 2002.

[94] T. G. Kollie, "Measurement of the thermal-expansion coefficient of nickel from 300 to 1000 K and determination of the power-law constants near the Curie temperature," *Phys. Rev. B*, vol. 16, no. 11, pp. 4872–4881, Dec 1977.

[95] F. Söffge, E. Steichele, and K. Stierstadt, "Thermal expansion anomaly of nickel near the Curie point," *Physica Status Solidi (a)*, vol. 42, no. 2, pp. 621–627, 1977.

[96] T. A. Faisst, "Determination of the critical exponent of the linear thermal expansion coefficient of nickel by neutron diffraction," *Journal of Physics: Condensed Matter*,

vol. 1, no. 33, pp. 5805–5810, 1989.

[97] R. E. MacFarlane and J. A. Rayne, "Temperature dependence of the elastic moduli of iridium," *Physics Letters*, vol. 20, no. 3, pp. 234–235, 15 Feb. 1966.

[98] P. Gargano, H. Mosca, G. Bozzolo, and R. D. Noebe, "Atomistic modeling of RuAl and (RuNi)Al alloys," *Scripta Materialia*, vol. 48, pp. 695–700, 2002.

[99] M. Mostoller, R. M. Nicklow, D. M. Zehner, S.-C. Lui, J. M. Mundenar, and E. W. Plummer, "Bulk and surface vibrational modes in NiAl," *Phys. Rev. B*, vol. 40, no. 5, pp. 2856–2872, Aug 1989.

[100] Y. S. Touloukian, R. K. Kirby, R. E. Taylor, and P. D. Desai, *Thermopysical Properteis of Matter Thermal Expansion*. New York: Plenum Press, 1975.

[101] K. Rzyman and Z. Moser, "Calorimetric studies of the enthalpies of formation of Al3Ni2, AlNi and AlNi3," *Progress in Materials Science*, vol. 49, pp. 581–606, 2004.

[102] T. Davenport, L. Zhou, and J. Trivisonno, "Ultrasonic and atomic force studies of the martensitic transformation induced by temperature and uniaxial stress in nial alloys," *Phys. Rev. B*, vol. 59, no. 5, pp. 3421–3426, Feb 1999.

[103] N. Yaozhuang, X. Youqing, P. Hongjian, and L. Xiaobo, "Ab initio thermodynamics of metals: Pt and Ru," *Physica B: Condensed Matter*, vol. 395, pp. 121–125, 31 May 2007.

[104] J. J. Halvorson and R. T. Wimber, "Thermal expansion of iridium at high temperatures," *Journal of Applied Physics*, vol. 43, no. 6, pp. 2519–2522, 1972.

[105] K. Yoshimoto, G. J. Papakonstantopoulos, J. F. Lutsko, and J. J. de Pablo, "Statistical calculation of elastic moduli for atomistic models," *Physical Review B (Condensed Matter and Materials Physics)*, vol. 71, no. 18, p. 184108, 2005.

APPENDIX A

SOURCE CODE

Thermodynamics.py

```
#! /usr/bin/env python
"""
Methods to obtain thermodynamics properties
"""

__author__ = "Raymundo Arroyave (raymundo@fastmail.fm)"
__version__ = "0.2$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams & Raymundo Arroyave"
__license__ = "Python"

from numpy import zeros, array, min
import CollectData
from CollectData import *
from EOS import FitEOS, GetExtrapolatedEnergies, FitLinear
from Fvib import *
from scipy import *
from NumericalMethods import *
import cPickle
from fileutils import *
import time
import sys
import pylab
import os
import Constants
import sys

class ThermodynamicFunctions:

    def __init__(self,Temperatures,FreeEnergy,Units="eV/atom"):
        self.Temperatures = array(Temperatures)
        if Units=="J/mol":
            self.FreeEnergy = array(FreeEnergy)*1000.
        else:
            self.FreeEnergy = array(FreeEnergy)
        self.Entropy = array(self.CalculateEntropy())
```

```
        self.HeatCapacity = array(self.CalculateHeatCapacity())
        self.Enthalpy = array(self.CalculateEnthalpy())
        self.RelativeEnthalpy = array(self.CalculateRelativeEnthalpy())
        if Units=="J/mol":
            self.Enthalpy = self.Enthalpy/1000.
            self.RelativeEnthalpy = self.RelativeEnthalpy/1000.


        self.NormalizedperAtom = False
        self.NumberofAtoms = 1.
        self.Units = Units
        self.CalculateDebyeTemperature(5.,300.)
        self.CalculatePropertiesat298K()

    def CalculateEntropy(self):
        S = -fprime(self.Temperatures, self.FreeEnergy)
        S = S-S[0]
        return S

    def CalculateEnthalpy(self):
        return  self.FreeEnergy+self.Temperatures*self.Entropy

    def CalculateRelativeEnthalpy(self):
        tck = interpolate.splrep(self.Temperatures, self.Enthalpy,k=3)
        return self.Enthalpy-interpolate.splev(298.15,tck)

    def CalculateHeatCapacity(self):
        return fprime(self.Temperatures, self.Entropy)*self.Temperatures

    def CalculateDebyeTemperature(self,Tmin,Tmax):
        Temp=linspace(Tmin,Tmax,30)
        self.DebyeTemperature = zeros((len(Temp),2),dtype=float)
        self.DebyeTemperature[:,0] = Temp
        tck=interpolate.splrep(self.Temperatures, self.HeatCapacity,s=0.0,k=3)
        Cpj = interpolate.splev(Temp,tck)
        for i in range(len(Cpj)):
            self.DebyeTemperature[i,1] = get_DebyeTemperature(Temp[i],
                                                              Cpj[i],300)
        tck=interpolate.splrep(self.DebyeTemperature[:,0],
                               self.DebyeTemperature[:,1],k=3,s=0.1)
        self.DebyeTemperature[:,1] = \
            interpolate.splev(self.DebyeTemperature[:,0],tck)

    def CalculatePropertiesat298K(self):
        tck=interpolate.splrep(self.Temperatures, self.Entropy,k=3)
```

```
        self.Entropy_at_298K = interpolate.splev(298.15,tck)
        tck=interpolate.splrep(self.Temperatures,self.HeatCapacity,k=3)
        self.HeatCapacity_at_298K = interpolate.splev(298.15,tck)

    def NormalizeperAtom(self,NumberofAtoms):
        if self.NormalizedperAtom == False:
            self.NormalizedperAtom = True
            self.NumberofAtoms = NumberofAtoms
            self.FreeEnergy = self.FreeEnergy/NumberofAtoms
            self.Entropy = self.Entropy/NumberofAtoms
            self.Enthalpy = self.Enthalpy/NumberofAtoms
            self.RelativeEnthalpy = self.RelativeEnthalpy/NumberofAtoms
            self.HeatCapacity = self.HeatCapacity/NumberofAtoms
            self.Entropy_at_298K = self.Entropy_at_298K/NumberofAtoms
            self.HeatCapacity_at_298K = \
                self.HeatCapacity_at_298K/NumberofAtoms

    def eV2Joules(self):
        if self.Units=="eV/atom":
            self.Units = "J/mol"
            #Note: Enthalpies will be in kJ/mol while entropies and
            #specific heats will be in J/mol/K
            ConversionFactor = 1.60217733e-19*6.0221367e23
            self.FreeEnergy = self.FreeEnergy*ConversionFactor/1000.
            self.Entropy = self.Entropy*ConversionFactor
            self.Entropy_at_298K = self.Entropy_at_298K*ConversionFactor
            self.Enthalpy = self.Enthalpy*ConversionFactor/1000.
            self.RelativeEnthalpy = \
                self.RelativeEnthalpy*ConversionFactor/1000.
            self.HeatCapacity = self.HeatCapacity*ConversionFactor
            self.HeatCapacity_at_298K = self.HeatCapacity_at_298K * \
                                        ConversionFactor

    def Joules2eV(self):
        if self.Units=="J/mol":
            self.Units = "eV/atom"
            ConversionFactor = 1./(1.60217733e-19*6.0221367e23)
            self.FreeEnergy = self.FreeEnergy*ConversionFactor*1000.
            self.Entropy = self.Entropy*ConversionFactor
            self.Entropy_at_298K = self.Entropy_at_298K*ConversionFactor
            self.Enthalpy = self.Enthalpy*ConversionFactor*1000.
            self.RelativeEnthalpy = \
                self.RelativeEnthalpy*ConversionFactor*1000.
            self.HeatCapacity = self.HeatCapacity*ConversionFactor
            self.HeatCapacity_at_298K = self.HeatCapacity_at_298K * \
```

ConversionFactor

```
class BulkThermodynamicProperties:

    def __init__(self,Volumes, Temperatures, Energies, VibFreeEnergies,
                 VTheta, DebyeMomentData,Units="eV/atom",
                 Extrapolation="Poly",
                 Anharmonicity=False,**kwargs):
        self.Temperatures = array(Temperatures)
        self.Volumes = array(Volumes)
        self.Energies = array(Energies)
        self.VibFreeEnergies = array(VibFreeEnergies)
        if 'EleFreeEnergies' in kwargs:
            self.EleFreeEnergies = array(kwargs['EleFreeEnergies'])
        self.VTheta = VTheta
        self.DebyeMomentData = DebyeMomentData
        self.Extrapolation = Extrapolation
        # Extrapolates to other volumes using a polynomial linear function or
        # interpolates using a spline. Note that splines cannot be used for
        # extrapolations.
        self.ExtrapolationFactor = 0.1
        self.NormalizedperAtom  = False
        self.NumberofAtoms = CountAtoms()
        self.Units = Units
        self.FittingEOS = "Linear"
        if Anharmonicity == True:
            self.Anharmonicity=True
            if 'AnharmonicityCorrection' in kwargs:
                if kwargs['AnharmonicityCorrection']=="Wu":
                    self.AnharmonicityCorrection="Wu"
                    if 'AnharmonicityFactor' in kwargs:
                        self.AnharmonicityFactor=kwargs['AnharmonicityFactor']
                    else:
                        print 'No Anharmonicity Factor!!!--using Wallace\
                         Approach Instead'
                        self.AnharmonicityCorrection="Wallace"
                elif kwargs['AnharmonicityCorrection']=="Wallace":
                    self.AnharmonicityCorrection="Wallace"
                    self.AVibFreeEnergies=zeros(shape(self.VibFreeEnergies))
                    self.vdos_all=kwargs['vdos_all']
                    self.CalculateAnharmonicFreeEnergy()
            else:
                print "No Anharmonicity Method Provided---Ignoring\
                 Anharmonicity"
                self.Anharmonicity=False
```

```
        else:
            self.Anharmonicity=False

        self.Get_0K_Properties()

        if 'EleFreeEnergies' in kwargs:
            self.Get_FiniteTemperatureProperties(EleFreeEnergies =
                                                 self.EleFreeEnergies)
        else:
            self.Get_FiniteTemperatureProperties()
        self.Fitting_chi_squared = \
            self.Fitting_chi_squared/float(len(self.Temperatures))
        print 'Average Fitting Correlation: %f12.6 '%\
            (1.-abs(self.Fitting_chi_squared))
        self.CalculateGruneissenParameter()
        self.CalculateVibrationalParametersat0K()
        self.CalculatePropertiesat298K()

    def Get_0K_Properties(self):
        energies=array(self.Energies).T+self.VibFreeEnergies[0,:]
        (FittingResults,params) = \
            FitEOS(self.Volumes,energies.T,EOS=self.FittingEOS)
        self.FittingParameters = params
        self.V0 = FittingResults["volume"]
        self.E0 = FittingResults["energy"]
        self.B0 = FittingResults["bulk"]
        self.dB0 = FittingResults["dB"]
        self.Fitting_chi_squared = FittingResults["chi-squared"]
        return

    def Get_FiniteTemperatureProperties(self,**kwargs):
        self.VT    = array(zeros((len(self.Temperatures),1)))
        self.FT    = array(zeros((len(self.Temperatures),1)))
        #Quasi-harmonic Free Energy
        self.HFT = array(zeros((len(self.Temperatures),1)))
        # Harmonic Free Energy (evaluated at V0)
        self.BT    = array(zeros((len(self.Temperatures),1)))
        self.dBT= array(zeros((len(self.Temperatures),1)))
        self.aT= array(zeros((len(self.Temperatures),1)))
        if self.Extrapolation=="Spline":
            self.ExtrapolationFactor=0.

        Vmin=(1.-self.ExtrapolationFactor)*min(self.Volumes)
        Vmax=(1.+self.ExtrapolationFactor)*max(self.Volumes)
```

```
V=linspace(Vmin,Vmax,10)

(FittingResults,params) = \
    FitEOS(self.Volumes,self.Energies,EOS=self.FittingEOS)
ExtrapolatedEnergy = \
    GetExtrapolatedEnergies(params, V,EOS=self.FittingEOS)

if self.Anharmonicity==True:
    if self.AnharmonicityCorrection == "Wallace":
        self.VibFreeEnergies = \
            self.VibFreeEnergies+self.AVibFreeEnergies

V=array([V])
for i in range(len(self.Temperatures)):

    if self.Anharmonicity==True:
        if self.AnharmonicityCorrection=="Wu":
            self.ApplyAnharmonicCorrection(i)

    if 'EleFreeEnergies' in kwargs:
        FreeEnergy=self.VibFreeEnergies[i,:]+self.EleFreeEnergies[i,:]
    else:
        FreeEnergy=self.VibFreeEnergies[i,:]

    if self.Extrapolation=="Poly":
        polycoeffs=polyfit(self.Volumes.T[0],FreeEnergy,1)
        ExtrapolatedFreeEnergy=polyval(polycoeffs,V)
    else:
        tck=interpolate.splrep(self.Volumes.T[0],FreeEnergy)
        ExtrapolatedFreeEnergy=interpolate.splev(V[0],tck)
        ExtrapolatedFreeEnergy=array([ExtrapolatedFreeEnergy])

    ExtrapolatedFreeEnergy = ExtrapolatedFreeEnergy+ExtrapolatedEnergy
    ExtrapolatedFreeEnergy = array(ExtrapolatedFreeEnergy)
    (FittingResults,params) = \
        FitEOS(V.T,ExtrapolatedFreeEnergy.T,EOS=self.FittingEOS)
    self.VT[i] = FittingResults["volume"]  # Volume vs T
    self.FT[i] = FittingResults["energy"]  # Free Energy vs T
    self.BT[i] = FittingResults["bulk"]    # Bulk vs T
    self.dBT[i]    =FittingResults["dB"]       # dB vs T
    self.Fitting_chi_squared = \
        self.Fitting_chi_squared+FittingResults["chi-squared"]

    tck=interpolate.splrep(V[0],ExtrapolatedFreeEnergy[0])
```

```
            self.HFT[i]     =interpolate.splev(self.V0,tck)

            if i == 0:     # Linear Thermal Expansion Coefficient
                self.aT[i]=0
            else:
                self.aT[i] = ( self.VT[i] - self.VT[i-1])/\
                    (self.Temperatures[i] -
                     self.Temperatures[i-1])/self.VT[i]/3.;
        self.Thermodynamics = \
            ThermodynamicFunctions(self.Temperatures,self.FT)
        return

    def ApplyAnharmonicCorrection(self,i):
        volumes = self.Volumes.T[0]
        energies = self.Energies.T[0]
        VibFreeEnergies = self.VibFreeEnergies[i,:]
        QuasiHarmonicFreeEnergy = energies+VibFreeEnergies
        vib_polycoeffs = polyfit(self.Volumes.T[0],
                                 self.VibFreeEnergies[i,:],1)
        (FittingResults,params) = \
            FitEOS(volumes.T,QuasiHarmonicFreeEnergy,EOS=self.FittingEOS)
        vTQH = FittingResults["volume"]  # Quasi-Harmonic V(T)
        vTA  = volumes*(1-self.AnharmonicityFactor*(vTQH-self.V0)/self.V0)
        self.VibFreeEnergies[i,:]=polyval(vib_polycoeffs,vTA)

    def CalculateAnharmonicFreeEnergy(self):
        h=Constants.h
        kB=Constants.kB
        self.AVibFreeEnergies=mat(self.AVibFreeEnergies)
        (row,column)=shape(self.AVibFreeEnergies)
        logVTheta=mat(log(self.VTheta))
        GruV=-fprime(logVTheta[:,0],logVTheta[:,1])
        tck=interpolate.splrep(self.VTheta[:,0],GruV)

        for i in range(column):
            (TD2,Mom2)=CalculateDebyeTemperature(self.vdos_all[i,:,:],2)
            Debye=(h/kB)*((5./3.)*Mom2**2.)**(1./2.)
            G=interpolate.splev(self.VTheta[i,0],tck)
            a=3*kB/Debye*(0.0078*G-0.0154)

            t1=(1/2.*Debye + Debye/(exp(Debye/self.Temperatures)-1.)    )**2.
            t2=(2. * (Debye/self.Temperatures)**2. * exp(Debye/\
                self.Temperatures)/(exp(Debye/self.Temperatures)-1.)**2. )\
                *self.Temperatures
```

```
            t2[isnan(t2)>0]=0
            Fanharmonic=a/3.*(t1+t2)

            self.AVibFreeEnergies[:,i]=self.NumberofAtoms*Fanharmonic

        self.AVibFreeEnergies=array(self.AVibFreeEnergies)

    def CalculateGruneissenParameter(self):
        # First we calculate the Gruneissen parameter directly from
        # variation in Debye Temperature as a function of volume, with the
        # Debye temperature calculated directly from the phonon DOS.

        self.GruneissenDirect    =    array(zeros((len(self.Temperatures),1)))

        x = log(self.VTheta[:,0])
        y = log(self.VTheta[:,1])
        tck = interpolate.splrep(x,y,s=0,k=3)
        x2 = log(self.VT)
        x2[0] = x2[0]+1e-14
        p = polyfit(x,y,2)
        y2 = polyval(p,x2)
        self.GruneissenDirect = -fprime(x2,y2)
        self.GruneissenSlater = self.dBT*0.5-1./6.
        self.GruneissenDugdale = self.dBT*0.5-1./2.
        self.GruneissenVaschenko = self.dBT*0.5-5./6.
        tck=interpolate.splrep(self.Temperatures,self.GruneissenDirect)
        self.Gruneissen_at_0K = interpolate.splev(self.V0,tck)

    def CalculatePropertiesat298K(self):
        tck = interpolate.splrep(self.Temperatures,self.aT,k=3)
        self.aT_at_298K = interpolate.splev(298.15,tck)
        tck = interpolate.splrep(self.Temperatures,self.VT,k=3)
        self.VT_at_298K = interpolate.splev(298.15,tck)
        tck = interpolate.splrep(self.Temperatures,self.BT,k=3)
        self.BT_at_298K = interpolate.splev(298.15,tck)
        tck = interpolate.splrep(self.Temperatures,self.dBT,k=3)
        self.dBT_at_298K = interpolate.splev(298.15,tck)
        tck = interpolate.splrep(self.Temperatures,self.GruneissenDirect,k=3)
        self.Gruneissen_at_298K=interpolate.splev(298.15,tck)

    def CalculateVibrationalParametersat0K(self):
        # Obtains the Vibrational Parameters at 0K
        V0 = self.V0
        DebyeMomentData = self.DebyeMomentData
        tck=interpolate.splrep(DebyeMomentData[:,0],
```

```
                                DebyeMomentData[:,1],s=0,k=3)
        self.MaximumFrequency_at_0K = interpolate.splev(V0,tck)

        tck=interpolate.splrep(DebyeMomentData[:,0],
                                DebyeMomentData[:,2],s=0,k=3)
        self.AverageFrequency_at_0K = interpolate.splev(V0,tck)

        tck=interpolate.splrep(DebyeMomentData[:,0],
                                DebyeMomentData[:,3],s=0,k=3)
        self.DebyeFrequency_at_0K = interpolate.splev(V0,tck)

        tck=interpolate.splrep(DebyeMomentData[:,0],
                                DebyeMomentData[:,4],s=0,k=3)
        self.DebyeWallerDebyeTemperature_at_0K = interpolate.splev(V0,tck)

        tck=interpolate.splrep(DebyeMomentData[:,0],
                                DebyeMomentData[:,5],s=0,k=3)
        self.EntropyDebyeTemperature_at_0K = interpolate.splev(V0,tck)

        tck=interpolate.splrep(DebyeMomentData[:,0],
                                DebyeMomentData[:,6],s=0,k=3)
        self.AverageDebyeTemperature_at_0K = interpolate.splev(V0,tck)

        tck=interpolate.splrep(DebyeMomentData[:,0],
                                DebyeMomentData[:,7],s=0,k=3)
        self.CpDebyeTemperature_at_0K = interpolate.splev(V0,tck)

    def NormalizeperAtom(self):
        if self.NormalizedperAtom == False:
            if not self.NumberofAtoms == 1:
                NumberofAtoms = self.NumberofAtoms
                self.NormalizedperAtom  = True
                self.V0 = self.V0/NumberofAtoms
                self.E0 = self.E0/NumberofAtoms
                self.VT = self.VT/NumberofAtoms
                self.VT_at_298K = self.VT_at_298K/NumberofAtoms
                self.FT = self.FT/NumberofAtoms
                self.HFT = self.HFT/NumberofAtoms
                self.Thermodynamics.NormalizeperAtom(NumberofAtoms)
            else:
                self.NormalizedperAtom = True
    def eV2Joules(self):
        if self.Units=="eV/atom":
            self.Units      =    "J/mol"
            # Note: Enthalpies will be in kJ/mol while
```

```
        # entropies and specific heats will be in J/mol/K
        ConversionFactor    =    1.60217733e-19*6.0221367e23
        self.Thermodynamics.eV2Joules()
        self.B0             =    self.B0*160.21892
        self.BT             =    self.BT*160.21892
        self.BT_at_298K     =    self.BT_at_298K*160.21892
        self.FT             =    self.FT*ConversionFactor/1000.
        self.HFT            =    self.HFT*ConversionFactor/1000.

def Joules2eV(self):
    if self.Units=="J/mol":
        self.Units          =    "eV/atom"
        # Note: Enthalpies will be in kJ/mol while
        # entropies and specific heats will be in J/mol/K
        ConversionFactor    =    1.60217733e-19*6.0221367e23
        self.Thermodynamics.eV2Joules()
        self.B0             =    self.B0/160.21892
        self.BT             =    self.BT/160.21892
        self.BT_at_298K     =    self.BT_at_298K/160.21892
        self.FT             =    self.FT/ConversionFactor*1000.
        self.HFT            =    self.HFT/ConversionFactor*1000.

def WriteInfoFile(self,filename,approximation):
    f=open("POSCAR","r")
    SystemName=f.readline()
    f.close

    f=open(filename,'w')
    f.write('_____\n')
    system='This is the information file for system:  '+SystemName
    f.write(system)
    f.write('Calculations were done under the ' +
            approximation + ' approximation\n')
    if self.Anharmonicity==True:
        f.write('Anharmonicity was considered.                    \n')
        f.write('Method used:   %s          \n' %\
            (self.AnharmonicityCorrection))
        if self.AnharmonicityCorrection=="Wu":
            f.write('Correction Factor = %12.6f\n' %\
                (self.AnharmonicityFactor))
    f.write('_____\n')
    f.write('Units used: '+self.Units+'\n')
    if self.Units=="J/mol":
        BulkUnits="GPa"
        EnthalpyUnits="kJ/mol"
```

```
        EntropyUnits="J/mol/K"
    else:
        BulkUnits="eV/A^2"
        EnthalpyUnits="eV/atom"
        EntropyUnits="eV/atom/K"

    if self.NormalizedperAtom==True:
        npa="True"
    else:
        npa="False"
    f.write('The quantities have been normalized: '+npa+'\n')
    f.write\
       ('==================Properties at 0K=========================\n')
    f.write\
       ('Equilibrium volume at 0K              : %12.6f A^3 \n' %\
        (self.V0))
    f.write('Total Energy at 0K                   : %12.6e eV  \n' %\
        (self.E0))
    f.write('Bulk Modulus at 0K                   : %12.6e %s  \n' %\
        (self.B0, BulkUnits))
    f.write('Pressure Derivative of Bulk Modulus  : %12.6f     \n' %\
        (self.dB0))
    f.write\
       ('Vibrational Properties at 0K============================\n')
    f.write\
        ('Gruneissen Parameter                    : %12.6f     \n' %\
            (self.Gruneissen_at_0K))
    f.write\
        ('Debye Frequency at 0K                   : %12.6e Hz  \n' %\
            (self.DebyeFrequency_at_0K))
    f.write\
        ('Entropy Debye Temperature (0th Moment)  : %12.6f K   \n' %\
            (self.EntropyDebyeTemperature_at_0K))
    f.write\
        ('Average Debye Temperature (1st Moment)  : %12.6f K   \n' %\
            (self.AverageDebyeTemperature_at_0K))
    f.write\
        ('Cp Debye Temperature      (2nd Moment)  : %12.6f K   \n' %\
            (self.CpDebyeTemperature_at_0K))
    f.write\
        ('Debye-Waller Debye Temperature (-2nd Moment) : %12.6f K   \n' %\
            (self.DebyeWallerDebyeTemperature_at_0K))
    f.write\
        ('=========================================================\n')
    f.write\
```

```
            ('Properties at 298K==========================================\n')
        f.write\
            ('Equilibrium volume at 298K               : %12.6f A^3 \n' %\
                (self.VT_at_298K))
        f.write('Thermal Expansion Coefficient at 298K : %12.6e 1/K \n' %\
            (self.aT_at_298K))
        f.write('Bulk Modulus at 298K                  : %12.6e %s  \n' %\
            (self.BT_at_298K, BulkUnits))
        f.write('Pressure Derivative of Bulk Modulus   : %12.6f      \n' %\
            (self.dBT_at_298K))
        f.write('Gruneissen Parameter  at 298K:        : %12.6f      \n' %\
            (self.Gruneissen_at_298K))
        f.write('Entropy at 298K                       : %12.6f %s   \n' %\
            (self.Thermodynamics.Entropy_at_298K,EntropyUnits))
        f.write('Heat Capacity at 298K                 : %12.6f %s   \n' %\
            (self.Thermodynamics.HeatCapacity_at_298K,EntropyUnits))
        f.write('==========================================================\n')
        f.close()

class FullThermodynamics:

    def __init__(self,Anharmonicity=False,**kwargs):

        if Anharmonicity==True:
            if 'AnharmonicityFactor' in kwargs:
                self.CalculateThermodynamics(AnharmonicCorrection = "Wu",
                        AnharmonicityFactor = kwargs['AnharmonicityFactor'])
            else:
                self.CalculateThermodynamics(AnharmonicCorrection="Wallace")
        else:
                self.CalculateThermodynamics()
        self.Anharmonicity    =    Anharmonicity
        self.NormalizedperAtom    =    False
        self.Units        =    "eV/atom"

    def CalculateThermodynamics(self,**kwargs):
        print 'Collecting data....'
        CollectedData    =    CollectData.CollectData(what='vol')
        temperatures    =    CollectedData["temperatures"]
        volumes        =    CollectedData["volumes"]
        energies    =    CollectedData["energies"]
        fvib        =    CollectedData["fvib"]
        felec        =    CollectedData["felec"]
        VTheta        =    CollectedData["VTheta"]
        DebyeMomentData =    CollectedData["DebyeMomentData"]
```

```
vdos_all = CollectedData["vdos"]

del CollectedData

print 'Calculating quasi-harmonic approximation.....'
self.QH = BulkThermodynamicProperties(volumes, temperatures,
                                      energies, fvib,
                                      VTheta,DebyeMomentData)
print 'Calculating harmonic approximation.....'
self.H = ThermodynamicFunctions(temperatures, self.QH.HFT)
print 'Calculating quasi-harmonic +electronic approximation.....'
self.QHEL = BulkThermodynamicProperties(volumes, temperatures,
                                        energies, fvib, VTheta,
                                        DebyeMomentData,
                                        EleFreeEnergies=felec)
print 'Calculating harmonic + electronic approximation.....'
self.HEL = ThermodynamicFunctions(temperatures, self.QHEL.HFT)

if 'AnharmonicCorrection' in kwargs:
    print 'Calculating quasi-harmonic + electronic + anharmonicity'\
        +' approximation.....'
    if not 'AnharmonicityFactor' in kwargs:
        self.QHELAN = \
            BulkThermodynamicProperties(volumes,
                                        temperatures,
                                        energies,
                                        fvib,
                                        VTheta,
                                        DebyeMomentData,
                                        EleFreeEnergies=felec,
                                        Anharmonicity=True,
                                    AnharmonicityCorrection="Wallace",
                                        vdos_all=vdos_all)
    else:

        self.QHELAN = BulkThermodynamicProperties(volumes,
                    temperatures,
                    energies,
                    fvib,
                    VTheta,
                    DebyeMomentData,
                    EleFreeEnergies=felec,
                    Anharmonicity=True,
                    AnharmonicityCorrection="Wu",
                    AnharmonicityFactor=kwargs['AnharmonicityFactor'])
```

```
def eV2Joules(self):
    if self.Units=="eV/atom":
        self.Units="J/mol"
        self.QH.eV2Joules()
        self.QHEL.eV2Joules()
        self.H.eV2Joules()
        self.HEL.eV2Joules()
        if self.Anharmonicity==True:
            self.QHELAN.eV2Joules()
        print 'Convertion to Joules done.'

def Joules2eV(self):
    if self.Units=="J/mol":
        self.Units="eV/atom"
        self.QH.Joules2eV()
        self.QHEL.Joules2eV()
        self.H.Joules2eV()
        self.HEL.Joules2eV()
        if self.Anharmonicity==True:
            self.QHELAN.Joules2eV()
        print 'Convertion to eV done.'

def NormalizeperAtom(self):
    if self.NormalizedperAtom==False :
        self.NormalizedperAtom=True
        NumberofAtoms=CountAtoms()
        self.QH.NormalizeperAtom()
        self.H.NormalizeperAtom(NumberofAtoms)
        self.QHEL.NormalizeperAtom()
        self.HEL.NormalizeperAtom(NumberofAtoms)
        if self.Anharmonicity==True:
            self.QHELAN.NormalizeperAtom()
        print 'Results Normalized per Number of Atoms.'

def WritetoFile(self):

    print 'Writing data to file...'
    whatlist = ["Enthalpy", "RelativeEnthalpy", "Entropy",
                "HeatCapacity", "FreeEnergy"]
    if not self.Anharmonicity==True:
        XX=mat(zeros((len(self.H.Temperatures),5)))
    else:
        XX=mat(zeros((len(self.H.Temperatures),6)))
    XX[:,0]    =    array(self.H.Temperatures)
```

```
        for i in range(len(whatlist)):
            XX[:,1]    =    getattr(self.H, whatlist[i])
            XX[:,2]=    getattr(self.HEL, whatlist[i])
            XX[:,3]=    getattr(self.QH.Thermodynamics, whatlist[i])
            XX[:,4]=    getattr(self.QHEL.Thermodynamics, whatlist[i])
            if self.Anharmonicity==True:
                XX[:,5]=    getattr(self.QHELAN.Thermodynamics, whatlist[i])
            Array2File(XX, whatlist[i]+".dat")

        whatlist2=["VT", "aT", "BT", "dBT", "GruneissenDirect"]
        namelist=["VolumeExpansion.dat", "CoefficientofThermalExpansion.dat",
                  "BulkModulus.dat", "BulkModulusPressureDerivative.dat",
                  "GruneissenConstant.dat"]
        if not self.Anharmonicity==True:
            XX=mat(zeros((len(self.H.Temperatures),3)))
        else:
            XX=mat(zeros((len(self.H.Temperatures),4)))
        XX[:,0]    =    array(self.H.Temperatures)
        for i in range(len(whatlist2)):
            XX[:,1]    =    getattr(self.QH, whatlist2[i])
            XX[:,2]    =    getattr(self.QHEL, whatlist2[i])
            if self.Anharmonicity==True:
                XX[:,3]    =    getattr(self.QHELAN, whatlist2[i])
            Array2File(XX, namelist[i])

    def Pickled(self,filename):
        """
        This method pickles the Full Thermodynamics Class itself.

        """
        file = open(filename+".p", "w")
        cPickle.dump(self,file)
        file.close()

def Runfitfc(fr,kp):

    fr=float(fr)
    kp=float(kp)
    parentdir = os.getcwd()

    print parentdir
    (dirs,dirsfloat) = GetDirectoriesList('vol')
    for directory in dirs:
        os.chdir(parentdir)
        os.chdir(directory)
```

```
            os.system('rm fc.out vdos.out')
        os.chdir(parentdir)
        fitfccommand='fitfc -f -fr=%f -kp=%f'%(fr,kp)
        os.system(fitfccommand)

        counter =0
        for directory in dirs:
            os.chdir(parentdir)
            os.chdir(directory)
            if not os.path.isfile('vdos.out'):
                counter=counter+1

        os.chdir(parentdir)

        return counter


def AdaptiveRunfitfc(fr,kp):
        fr=float(fr)
        kp=float(kp)

        frlist=linspace(fr,fr/2.,50)

        for i in range(len(frlist)):
            fr=frlist[i]
            counter=Runfitfc(fr,kp)
            if counter == 0:
                break
        return counter

if __name__ == "__main__":

        try:
            import psyco
            psyco.bind(intfr2)
            psyco.bind(get_Cp_from_DebyeTemperature)
            psyco.bind(GetVibrationalFreeEnergy)
        except ImportError:
            pass


        tic=time.time()
        if '-full' in sys.argv:
            print 'Calculating Thermodynamic Properties...'
            FT=FullThermodynamics(Anharmonicity=True)
```

```
        # Collects Data, Calculates THermodyamic Properties
        FT.eV2Joules()
        FT.NormalizeperAtom()
        FT.Pickled("FT") # Pickles the Full Thermodynamics Instance
        FT.WritetoFile()
        # Writes to Files .dat Contents are as follows: In files with five
        # columns ,called Enthalpies.dat, for example, the first column
        # corresponds to Temperature, then Harmonic Enthalpy, then Harmonic +
        # Electronic Enthalpy, etc.
        # In files with three columns, the first one corresponds to
        # temperature, the second one corresponds to quasi-harmonic
        # calculations and the third one corresponds to quasi-harmonic +
        #electronic calculations
        FT.QH.WriteInfoFile('Info-QH.dat','quasi-harmonic')
        FT.QHEL.WriteInfoFile('Info-QHEL.dat','quasi-harmonic + electronic')
        if FT.Anharmonicity==True:
            FT.QHELAN.WriteInfoFile('Info-QHELAN.dat',
                                    'quasi-harmonic + electronic + \
                                    anharmonicity')

        print 'Please check Info-xxx.dat files for summary of calculations'
    elif '-H' in sys.argv:
        fvib=File2Array('fvib.dat')
        H=ThermodynamicFunctions(mat(fvib[:,0]).T,mat(fvib[:,1]).T)
    elif '-profile' in sys.argv:
        import cProfile
        cProfile.run('FT=FullThermodynamics()','FTprof')
        import pstats
        p=pstats.Stats('FTprof')
        p.sort_stats('time').print_stats(15)
    elif '-fitfc' in sys.argv:
        if len(sys.argv)>2:
            fr=float(sys.argv[2])
        else:
            fr=6.0
        counter=AdaptiveRunfitfc(fr,25000.)
        print counter
    else:
        pass
    toc=time.time()
    print toc-tic, 'has elapsed'
```

ELC.py

```
#! /usr/bin/env python
```

```
"""
This is the main module for post-processing C11 and C12 elastic
constant calculations.  An object is created for each strain
directory, containing all the necessary information from that
directory.  An object is created for each volume directory, containing
all the strain directory objects along with other information.  All
volume directory objects are condensed into a single object and it is
pickled into a single object C11_C12.pkl

The data collection and pickling steps may be skipped if previously
done by using the -p option at the command line.
"""

__author__ = "Mike Williams & Raymundo Arroyave"
__version__ = "0.3$"
__date__ = "$Date: July 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams & Raymundo Arroyave"
__license__ = "Python"

import os
import sys
import fileutils
import Constants
import Fele
import numpy
import scipy
import CollectData
import Fvib
import cPickle
import pylab

class StrainDir:
    """
    This object is a data container.  We collect all necessary
    information from a given strain directory and place it in this
    object
    """
    def __init__(self, params):
        self.Strain = self.GetStrain()
        self.E0 = self.GetE0()
        self.Temperatures = \
            scipy.array(CollectData.GetTemperatures(params['mintemp'],
                                                    params['maxtemp'],
                                                    params['dTemp']))
        self.Fvib = self.GetFvib()
```

```
        self.Felec = self.GetFelec()

    def GetStrain(self):
        directory = os.getcwd().split('/')[-1]
        if directory[-2] == '-':
            strain = float(directory[-2:])*.01
        else:
            strain = float(directory[-1])*.01
        return strain

    def GetE0(self):
        data = open('OSZICAR').readlines()[-1].split()[4]
        return float(data)

    def GetFvib(self):
        parentdir = os.getcwd()
        os.chdir('../')
        data = Fvib.GetVibrationalFreeEnergy(params['mintemp'],
                                             params['maxtemp'],
                                             params['dTemp'],
                                             writeflag=True)
        os.chdir(parentdir)
        return data

    def GetFelec(self):
        parentdir = os.getcwd()
        os.chdir('../')
        data = Fele.GetElectronicFreeEnergy(params['mintemp'],
                                            params['maxtemp'],
                                            params['dTemp'],
                                            writeflag=True)
        os.chdir(parentdir)
        return data

class VolDir:
    """
    This object is another data container.
    It collects all the strain sub-directories as well as other data
    at the volume level.
    """
    def __init__(self, params):
        self.CellVolume     =   CollectData.CellVolume()
        self.Strains        =   []
        self.StrainDirnames =   CollectData.GetDirectoriesList('11str')[0]
        for strain in self.StrainDirnames:
```

```
        parentdir = os.getcwd()
        os.chdir(parentdir+'/'+strain)
        self.Strains.append(StrainDir(params))
        os.chdir(parentdir)
    self.StrainValues     =     self.GetStrainValues()
    self.C_C_C11_C12      =     self.ColdCurveC11_C12()
    self.C11_C12_Fvib, self.C11_C12_Felec, self.C11_C12_Felec_Fvib = \
        self.GetC11_C12()

def GetStrainValues(self):
    data = []
    for i in self.Strains:
        data.append(i.Strain)
    return data

def GetC11_C12(self):
    """
    This module is used if the vibrational and electric contributions to
    the thermal free energy are to be included in the calculation of the
    elastic constants.  Normally, these affects are assumed to be
    negligible and this function is not used.
    """
    FvibEnergies = []
    FelecEnergies = []
    FvibFelecEnergies = []
    for strain in self.Strains:
        FvibEnergies.append(strain.E0 + strain.Fvib[:,1])
        FelecEnergies.append(strain.E0 + strain.Felec[:,1])
        FvibFelecEnergies.append(strain.E0 + strain.Fvib[:,1] +
                                 strain.Felec[:,1])
    FvibEnergies = scipy.array(FvibEnergies).T
    FelecEnergies = scipy.array(FelecEnergies).T
    FvibFelecEnergies = scipy.array(FvibFelecEnergies).T
    C11_C12_Fvib = scipy.zeros(len(FvibEnergies))
    C11_C12_Felec = scipy.zeros(len(FelecEnergies))
    C11_C12_Felec_Fvib = scipy.zeros(len(FvibFelecEnergies))
    for i in range(len(FvibEnergies)):
        C11_C12_Fvib[i] = self.FitC11_C12(scipy.array(self.StrainValues),
                                          FvibEnergies[i])
        C11_C12_Felec[i] = self.FitC11_C12(scipy.array(self.StrainValues),
                                           FelecEnergies[i])
        C11_C12_Felec_Fvib[i] = self.FitC11_C12(scipy.array
                                                (self.StrainValues),
                                                FvibFelecEnergies[i])
    return C11_C12_Fvib, C11_C12_Felec, C11_C12_Felec_Fvib
```

```
    def ColdCurveC11_C12(self):
        energies = []
        for strain in self.Strains:
            energies.append(strain.E0)
        energies = scipy.array(energies)
        return self.FitC11_C12(scipy.array(self.StrainValues), energies)

    def FitC11_C12(self, strains, energies):
        # For extracting the elastic constants a quadratic polynomial
        # is fit to the strain/energy data
        p = scipy.polyfit(strains, energies, 2)
        answer = p[0]/self.CellVolume*160.21892
        #the 160.21892 converts the answer to GPa
        return answer

class AllData:
    def __init__(self, params):
        self.VolDirnames =  CollectData.GetDirectoriesList('vol')[0]
        self.Volumes = []
        for volume in self.VolDirnames:
            parentdir = os.getcwd()
            os.chdir(parentdir+'/'+volume)
            self.Volumes.append(VolDir(params))
            os.chdir(parentdir)

if __name__ == "__main__":
    params = fileutils.ReadInputFile('ELCparams.in')
    UsePickled = False
    if len(sys.argv) > 1:
        if sys.argv[1][1:] == 'p':
            UsePickled = True
    if UsePickled:
        print 'Using Pickled data'
        pkl_file = open('C11_C12.pkl', 'rb')
        data = cPickle.load(pkl_file)
    else:
        data = AllData(params)
        output = open('C11_C12.pkl', 'wb')
        cPickle.dump(data, output)
        output.close()
    Volumes = []
    for vol in data.Volumes:
        Volumes.append(vol.CellVolume)
    Volumes = scipy.array(Volumes)
```

```
NAtoms = CollectData.CountAtoms()
VolumeExpansion = fileutils.File2Array('VolumeExpansion.dat')
VolumeExpansion[:,1:] = VolumeExpansion[:,1:]*NAtoms

#get answer from Cold Curve data alone
#note the C11-C12 we are getting is the ISOTHERMAL Elastic Constant
ColdCurve = []
C11_C12vT = []
index = 0
for volume in data.Volumes:
    ColdCurve.append(volume.C_C_C11_C12)

#interpolate C11-C12 with cubic spline
tck = scipy.interpolate.splrep(Volumes, ColdCurve, k=3)
for row in VolumeExpansion:
    ThisC11_C12 = scipy.interpolate.splev(VolumeExpansion[index,2], tck)
    C11_C12vT.append(ThisC11_C12)
    index = index + 1
C11_C12vT = scipy.array(C11_C12vT)
Output = scipy.column_stack((VolumeExpansion[:,0], scipy.array(C11_C12vT)))
fileutils.Array2File(Output, 'C11_C12vT.dat')

#seperate C11 and C12 using the bulk modulus
B = fileutils.File2Array('BulkModulus.dat')[:,2]
C11 = (2./3.)*C11_C12vT+B
C11vT = scipy.column_stack((VolumeExpansion[:,0], C11))
C12 = C11 - C11_C12vT
C12vT = scipy.column_stack((VolumeExpansion[:,0], C12))
fileutils.Array2File(C11vT, 'C11vT.dat')
fileutils.Array2File(C12vT, 'C12vT.dat')

#Change ISOTHERMAL to ADIABATIC
Cv = fileutils.File2Array('HeatCapacity.dat')[:,2]*6.24e18/6.02e23# j/mol/k
alpha = fileutils.File2Array('CoefficientofThermalExpansion.dat')[:,3]
V = fileutils.File2Array('VolumeExpansion.dat')[:,3]
T = C11vT[:,0]
factor = 6.241506363e-3
C11 = C11 * factor
C12 = C12 * factor

lambda1 = alpha*(C11+C12)
lambda2 = alpha*(C11+C12)

C11Adiabatic = C11 + (V*lambda1*lambda1*T)/Cv
C12Adiabatic = C12 + (V*lambda2*lambda2*T)/Cv
```

```
    C11Adiabatic = C11Adiabatic/(factor)
    C12Adiabatic = C12Adiabatic/(factor)

    C11AdiabvT = scipy.column_stack((VolumeExpansion[:,0], C11Adiabatic))
    C12AdiabvT = scipy.column_stack((VolumeExpansion[:,0], C12Adiabatic))
    fileutils.Array2File(C11AdiabvT, 'C11Adiabatic.dat')
    fileutils.Array2File(C12AdiabvT, 'C12Adiabatic.dat')

    #Create C11-C12 volume temperature surface - this is for visualization
    #purposes only
    ELCsurface = []
    for volume in data.Volumes:
        ELCsurface.append(volume.C11_C12_Felec_Fvib)
    ELCsurface = scipy.array(ELCsurface).T
    fileutils.Array2File(ELCsurface[ ::100,:], 'C11_C12Surface.dat')
```

C44.py

```
#! /usr/bin/env python
"""
Contains various functions for the calculation of elastic constants
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com) &\
              Raymundo Arroyave (rarroyave at tamu dot edu"
__version__ = "0.3$"
__date__ = "$Date: July 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams & Raymundo Arroyave"
__license__ = "Python"

import os
import sys
import fileutils
import Constants
import Fele
import numpy
import scipy
import CollectData
import Fvib
import cPickle
import pylab

class StrainDir:
    def __init__(self, params):
```

```
        self.Strain       = self.GetStrain()
        self.E0           = self.GetE0()
        self.Temperatures = scipy.array(CollectData.GetTemperatures
                                        (params['mintemp'], params['maxtemp'],
                                        params['dTemp']))
        self.Fvib         = self.GetFvib()
        self.Felec        = self.GetFelec()

    def GetStrain(self):
        directory = os.getcwd().split('/')[-1]
        if directory[-2] == '-':
            strain = float(directory[-2:])*.01
        else:
            strain = float(directory[-1])*.01
        return strain

    def GetE0(self):
        data = open('OSZICAR').readlines()[-1].split()[4]
        return float(data)

    def GetFvib(self):
        parentdir = os.getcwd()
        os.chdir('../')
        data = Fvib.GetVibrationalFreeEnergy(params['mintemp'],
                                             params['maxtemp'],
                                             params['dTemp'],
                                             writeflag=True)
        os.chdir(parentdir)
        return data

    def GetFelec(self):
        parentdir = os.getcwd()
        os.chdir('../')
        data = Fele.GetElectronicFreeEnergy(params['mintemp'],
                                            params['maxtemp'],
                                            params['dTemp'],
                                            writeflag=True)
        os.chdir(parentdir)
        return data

class VolDir:
    def __init__(self, params):
        self.CellVolume = CollectData.CellVolume()
        self.Strains = []
        self.StrainDirnames = CollectData.GetDirectoriesList('44str')[0]
```

```
        for strain in self.StrainDirnames:
            parentdir = os.getcwd()
            os.chdir(parentdir+'/'+strain)
            self.Strains.append(StrainDir(params))
            os.chdir(parentdir)
        self.StrainValues = self.GetStrainValues()
        self.C_C_C44 = self.ColdCurveC44()
        self.C44_Fvib, self.C44_Felec, self.C44_Felec_Fvib = self.GetC44()

    def GetStrainValues(self):
        data = []
        for i in self.Strains:
            data.append(i.Strain)
        return data

    def GetC44(self):
        FvibEnergies = []
        FelecEnergies = []
        FvibFelecEnergies = []
        for strain in self.Strains:
            FvibEnergies.append(strain.E0 + strain.Fvib[:,1])
            FelecEnergies.append(strain.E0 + strain.Felec[:,1])
            FvibFelecEnergies.append(strain.E0 +
                                     strain.Fvib[:,1] +
                                     strain.Felec[:,1])
        FvibEnergies = scipy.array(FvibEnergies).T
        FelecEnergies = scipy.array(FelecEnergies).T
        FvibFelecEnergies = scipy.array(FvibFelecEnergies).T
        C44_Fvib = scipy.zeros(len(FvibEnergies))
        C44_Felec = scipy.zeros(len(FelecEnergies))
        C44_Felec_Fvib = scipy.zeros(len(FvibFelecEnergies))
        for i in range(len(FvibEnergies)):
            C44_Fvib[i] = self.FitC44(scipy.array(self.StrainValues),
                                      FvibEnergies[i])
            C44_Felec[i] = self.FitC44(scipy.array(self.StrainValues),
                                       FelecEnergies[i])
            C44_Felec_Fvib[i] = self.FitC44(scipy.array(self.StrainValues),
                                            FvibFelecEnergies[i])
        return C44_Fvib, C44_Felec, C44_Felec_Fvib

    def ColdCurveC44(self):
        energies = []
        for strain in self.Strains:
            energies.append(strain.E0)
        energies = scipy.array(energies)
```

```
        return self.FitC44(scipy.array(self.StrainValues), energies)

    def FitC44(self, strains, energies):
        p = scipy.polyfit(strains, energies, 2)
        answer = p[0]/self.CellVolume*160.21892*2

        return answer

class AllData:
    def __init__(self, params):
        self.VolDirnames = CollectData.GetDirectoriesList('vol')[0]
        self.Volumes = []
        for volume in self.VolDirnames:
            parentdir = os.getcwd()
            os.chdir(parentdir+'/'+volume)
            self.Volumes.append(VolDir(params))
            os.chdir(parentdir)

if __name__ == "__main__":
    params = fileutils.ReadInputFile('ELCparams.in')
    UsePickled = False
    if len(sys.argv) > 1:
        if sys.argv[1][1:] == 'p':
            UsePickled = True
    if UsePickled:
        print 'Using Pickled data'
        pkl_file = open('C44.pkl', 'rb')
        data = cPickle.load(pkl_file)
    else:
        data = AllData(params)
        output = open('C44.pkl', 'wb')
        cPickle.dump(data, output)
        output.close()

    Volumes = []
    for vol in data.Volumes:
        Volumes.append(vol.CellVolume)
    Volumes = scipy.array(Volumes)

    NAtoms = CollectData.CountAtoms()
    VolumeExpansion = fileutils.File2Array('VolumeExpansion.dat')
    VolumeExpansion[:,1:] = VolumeExpansion[:,1:]*NAtoms

    #get answer from Cold Curve data alone
    ColdCurve = []
```

```
    C44vT = []
    index = 0
    for volume in data.Volumes:
        ColdCurve.append(volume.C_C_C44)
    tck = scipy.interpolate.splrep(Volumes, ColdCurve, k=3)
    for row in VolumeExpansion:
        ThisC44 = scipy.interpolate.splev(VolumeExpansion[index,2], tck)
        C44vT.append(ThisC44)
        index = index + 1
    C44vT = scipy.array(C44vT)
    Output = scipy.column_stack((VolumeExpansion[:,0], scipy.array(C44vT)))
    fileutils.Array2File(Output, 'C44vT.dat')
    #create C11-C12 volume temperature surface
    ELCsurface = []
    for volume in data.Volumes:
        ELCsurface.append(volume.C44_Felec_Fvib)
    ELCsurface = scipy.array(ELCsurface).T
    fileutils.Array2File(ELCsurface[ ::100,:], 'C44Surface.dat')
```

CollectData.py

```
#! /usr/bin/env python
"""
Methods used in the calcultion of electronic free energies.
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com), and \
                              Raymundo Arroyave (raymundo@fastmail.fm)"
__version__ = "0.3$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams & Raymundo Arroyave"
__license__ = "Python"

import numpy
import os
import string
import Fvib
import Fele
import fileutils
import pylab
import sys
from scipy import *

def CellVolume():
    """
```

```
    Calculates the volume of a primitive unit cell

    Requires: nothing
    Returns: volume of the primitive unit cell
    """
    #read in data from CONTCAR file
    inputfile = open("CONTCAR", 'r')
    data = inputfile.readlines()
    inputfile.close()
    latticevectors = zeros((3,3))
    x = []
    x.append(data[2].split(" "))
    x.append(data[3].split(" "))
    x.append(data[4].split(" "))
    scalingfactor = 1
    #eliminate null spaces in data
    for counter in range(3):
        while '' in x[counter]:
            x[counter].remove('')
    #convert elements of lists to array of floats
    for counter in range(3):
        for element in range(len(x[counter])):
            latticevectors[counter, element] = float(x[counter][element])
    a = scalingfactor*latticevectors[0]
    b = scalingfactor*latticevectors[1]
    c = scalingfactor*latticevectors[2]
    #calculate volume with the vector triple product
    d = cross(b,c)
    volume = dot(a,d)
    return volume

def GetE0():
    data = open('OSZICAR').readlines()[-1].split()[4]
    return float(data)

def CountAtoms():
    dummy = os.popen('cat POSCAR | head -n 6 | tail -n 1')
    data = dummy.readlines()
    data = data[0].split()
    integerlist = []
    for element in data:
        integerlist.append(int(element))
    numberofatoms = sum(integerlist)
    return numberofatoms
```

```
def GetTemperatures(Tmin, Tmax, dT):
    temp = numpy.arange(Tmin, Tmax, dT)
    final = numpy.zeros((len(temp), 1))
    for element in range(len(temp)):
        final[element] = temp[element]
    return final


def GetDirectoriesList(prfx):
    """
    This function returns a SORTED list of directories with the prefix prfx.
    It assumes that the name of the directories must be prfx_Number, where
    prfx stands for vol or str and NUMBER may be any integer or real
    """

    cmd='echo `ls -l | grep "^d"| grep'+" "+prfx+" "+ \
        '| awk \'{print $8}\' | sed \'s/'+prfx+'_'+ \
        '//g\' | grep \'[0-9]\' | sort -n ``'
    dummy=os.popen(cmd)
    directories= dummy.readlines()
    directories =directories[0].split()
    DirectoriesList=['any'] # This initializes the list
    DirectoriesList.pop(0)  # Removes dummy element

    for i in numpy.arange(len(directories)):
        DirectoriesList.append(prfx+'_'+str(directories[i]))
        directories[i]=float(directories[i])
    return DirectoriesList,directories

def CollectData(what='vol'):
    #input parameters and variable initialization
    inputparams = fileutils.ReadInputFile('ELCparams.in')
    (dirs,dirsfloat) = GetDirectoriesList(what)
    if len(dirs)==0:
        sys.stderr.write('Error: No subdirectories present!!....')
        sys.exit()

    Tmin = inputparams['mintemp']
    Tmax = inputparams['maxtemp']
    dT = inputparams['dTemp']

    #ensure that parameters are the right type
    Tmin=string._float(Tmin)
    Tmax=string._float(Tmax)
```

```
dT=string._float(dT)

temperatures = GetTemperatures(Tmin, Tmax+dT, dT)
volumes = numpy.zeros((len(dirs), 1))
energies = numpy.zeros((len(dirs), 1))
fvib = numpy.zeros((len(temperatures), len(dirs)))
felec = numpy.zeros((len(temperatures), len(dirs)))
vdos_all = []
VTheta= numpy.zeros((len(dirs),2))
DebyeMomentData=numpy.zeros((len(dirs),9))
#iterate through volume directories
parentdir = os.getcwd()
counter = 0
for directory in dirs:
    print 'Processing directory = %s ....'%(directory)
    os.chdir(parentdir)
    os.chdir(directory)
    if what=='str':
        os.chdir('vol_0')

    volumes[counter][0] = CellVolume()
    energies[counter][0] = GetE0()
    fvib[:,counter] = Fvib.GetVibrationalFreeEnergy(Tmin, Tmax, dT,
                                                    writeflag=False,
                                                    plotflag=False)[:,1]
    felec[:,counter] = Fele.GetElectronicFreeEnergy(Tmin, Tmax, dT,
                                                    writeflag=False,
                                                    plotflag=False)[:,1]
    if what=='vol':
        VDOS=fileutils.File2Array('vdos.out')
        vdos_all.append(VDOS)
        ph=Fvib.PhononDOS(writeflag=True)
        VTheta[counter,0]=volumes[counter][0]
        VTheta[counter,1]=ph.TDM[1,1]
        DebyeMomentData[counter,0]=volumes[counter][0]
        DebyeMomentData[counter,1]=ph.MaximumFrequency
        DebyeMomentData[counter,2]=ph.AverageFrequency
        # Debye Frequency with respect to several moments
        DebyeMomentData[counter,3]=ph.TDM[1,1]*2.0837E10 # 0th Moment
        DebyeMomentData[counter,4]=ph.TDM[0,1] # n=-2
        DebyeMomentData[counter,5]=ph.TDM[1,1] # n= 0
        DebyeMomentData[counter,6]=ph.TDM[2,1] # n= 1
        DebyeMomentData[counter,7]=ph.TDM[3,1] # n= 2
        DebyeMomentData[counter,8]=ph.TDM[4,1] # n= 4
```

```
            counter = counter + 1

        #write output files
        volumes=mat(volumes)
        energies=mat(energies)
        temperatures=mat(temperatures)
        fvib=mat(fvib)
        felec=mat(felec)
        vdos_all=array(vdos_all)

        dirsfloat=mat(dirsfloat).T

        os.chdir(parentdir)
        fileutils.Array2File(temperatures, 'temperatures.dat')
        fileutils.Array2File(volumes, 'volumes.dat')
        fileutils.Array2File(energies, 'energy_all.dat')
        fileutils.Array2File(fvib, 'fvib_all.dat')
        fileutils.Array2File(felec, 'felec_all.dat')
        if what=='vol':
            fileutils.Array2File(VTheta,'VTheta.dat')
            fileutils.Array2File(DebyeMomentData,'DebyeMomentData.dat')

        if what=='str':
            fileutils.Array2File(felec, 'strains.dat')
            results={'temperatures':temperatures, 'volumes':volumes,
                     'energies':energies, 'fvib':fvib, 'felec':felec,
                     'strains' : dirsfloat}
        else:
            results={'temperatures':temperatures, 'volumes':volumes,
                     'energies':energies, 'fvib':fvib, 'felec':felec,
                     'VTheta': VTheta, 'DebyeMomentData':DebyeMomentData,
                     'vdos':vdos_all}

        return results

if __name__ == "__main__":
    (temperatures,volumes,energies,fvib,felec) = CollectData()
```

EOS.py

```
#! /usr/bin/env python
"""
Modules for fitting an equation of state
"""
```

```
__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.1$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

from scipy import *
import numpy
from Scientific.Functions.LeastSquares import leastSquaresFit
from fileutils import *
import pylab
import math

def FitLinear(vol, energies):
    """
    Performs a linear fit to the volume vs. energy curve
    """

    vol=mat(vol)
    energies=mat(energies)
    (rv,cv) = shape(vol)
    (re,ce)    = shape(energies)

    if cv > rv:
        vol=vol.T
    if ce > re:
        energies=energies.T

    vol=array(vol)
    energies=array(energies)

    A1 = numpy.ones((len(vol), 1))
    A2 = numpy.mat(numpy.power(vol, -1./3))
    A3 = numpy.mat(numpy.power(vol, -2./3))
    A4 = numpy.mat(vol**(-1.))
    A = numpy.concatenate((A1,A2,A3,A4), axis=1)
    x = numpy.linalg.pinv(A) * energies
    newenergies = A*x

    a = x[0, 0]
    b = x[1, 0]
    c = x[2, 0]
    d = x[3, 0]

    #extract values
```

```
    V0 = 4.*c**3 - 9.*b*c*d + \
         math.sqrt((c**2. - 3.*b*d) * (4.*c**2 - 3.*b*d)**2.)
    V0 = -V0/b**3.
    B0 = 9.*d + 5.*c*V0**(1./3.) +2.*b*V0**(2./3.)
    B0  =(2.*B0/(9*V0**2))
    BP =(54.*d +25.*c*V0**(1./3)+8.*b*V0**(2./3.))/ \
         (27.*d +15.*c*V0**(1./3)+6.*b*V0**(2./3))
    E0=a + b*V0**(-1./3.) + c*V0**(-2./3.)+ d*V0**(-1.)

    newenergies = a + b*vol**(-1./3.) + c*vol**(-2./3.)+ d*vol**(-1.)
    Correlation = stats.pearsonr(energies,newenergies)[0]
    chisquared = 1-Correlation

    FittingParameters = {'energy' : E0, 'bulk' : B0, 'volume' : V0,
                         'dB' : BP, 'chi-squared' : chisquared}
    params=(a,b,c,d)
    return FittingParameters,params

def Linear(params, V):
    return params[0] + params[1]*V**(-1./3) + params[2]*V**(-2./3) + \
         params[3]*V**(-1.)

def Birch(params, V):
    p = params
    ans = p[0]+(9./8.)*p[1]*p[2]*((p[2]/V)**(2./3.)-1)**2.+ \
         (9./16.)*p[1]*p[2]*(p[3]-4.)*((p[2]/V)**(2./3.)-1.)**3.
    return ans

def GetExtrapolatedEnergies(params,V,EOS="Birch"):
    if EOS=="Linear":
        return Linear(params,V)
    else:
        return Birch(params,V)

def Birch_Murnaghan(params, V):
    E0 = params[0]
    B0 = params[1]
    V0 = params[2]
    dB0 = params[3]
    ans = E0 + (9.*V0*B0/16.)*(((V0/V)**(2./3.)-1.)**3.* \
         dB0+((V0/V)**(2./3.)-1.)**2.*(6.-4.*(V0/V)*(2./3.)))
    return ans

def FitEOS(volumes,energies,EOS="Birch"):
```

```
    if EOS=="Linear":
        FittingParameters= FitLinear(volumes,energies)
    else:
        FittingParameters= FitBirch(volumes,energies)

    return FittingParameters

def FitBirch(volumes,energies):
    data = mat(numpy.zeros((len(energies), 2)))
    volumes=mat(volumes)
    energies=mat(energies)
    (rv,cv) = shape(volumes)
    (re,ce)     = shape(energies)

    if cv > rv:
        volumes=volumes.T
    if ce > re:
        energies=energies.T

    data[:,0]=volumes
    data[:,1]=energies
    data=numpy.array(data)
    initialvolume=numpy.sum(volumes)/len(volumes)
    initialenergy=numpy.sum(energies)/len(energies)
    initialB=0.5
    initialBP=4
    c = leastSquaresFit(Birch, (initialenergy, initialB, initialvolume,
                        initialBP), data, stopping_limit = 1e-10)
    FittingParameters = {'energy':c[0][0], 'bulk':c[0][1], 'volume':c[0][2],
                         'dB' : c[0][3], 'chi-squared' : c[1]}
    params=(c[0][0],c[0][1],c[0][2],c[0][3])

    return FittingParameters,params

if __name__ == "__main__":
    #load data
    energies = File2Array('energy_all.dat')
    data = numpy.zeros((len(energies), 2))
    volumes = File2Array('volumes.dat')
    data[:,0] =volumes
    data[:,1] = energies
    initialvolume=numpy.sum(volumes)/len(volumes)
    x = numpy.linspace(data[:,0][0], data[:,0][-1], 100)
    c = leastSquaresFit(Birch, (-3.5, 74, initialvolume, 5), data,
                                stopping_limit = 1e-10)
```

```
                                #volume parameter is super sensitive
    Birch_out = {'E0':c[0][0], 'B0':c[0][1], 'V0':c[0][2], 'dB':c[0][3]}
    fitdata_Birch = Birch(c[0], x)

    (FittingParameters,params)=FitLinear(volumes, energies)
    print 'Using Alternate EOS:'
    print FittingParameters
    NewEnergies=GetExtrapolatedEnergies(params,volumes,EOS="Linear")
    print NewEnergies
    (FittingParameters,params)=FitBirch(volumes, energies)
    print 'Using Birch EOS:'
    print FittingParameters
    NewEnergies=GetExtrapolatedEnergies(params,volumes,EOS="Birch")
    print NewEnergies

    Birch_fit = pylab.plot(x, fitdata_Birch, label = 'Birch Fit')
    Data_plot = pylab.plot(volumes,energies, 'o', label='Data')

    pylab.ylabel('E0')
    pylab.xlabel('Volume')
    pylab.title('E/V Curve Fitting')
    pylab.legend(loc='upper left')
    pylab.savefig('EVFits.eps')
    pylab.show()
```

Fele.py

```
#! /usr/bin/env python
"""
Methods used in the calcultion of electronic free energies.
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com), \
                and Raymundo Arroyave (raymundo@fastmail.fm)"
__version__ = "0.2$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams & Raymundo Arroyave"
__license__ = "Python"

import Constants
from scipy import *
from fileutils import *
import pylab
import sys
import os
```

```
from NumericalMethods import *

def FermiFunction(energy, FermiLevel, Temperature):
    """
    Calculates the Fermi Function

    Requires:  an array of Energy Values, the Fermi Level, and a temperature
    Returns:  an array with the value of the Fermi Function at the given
              temperature
    """
    kB = Constants.kB
    return 1/(exp((energy-FermiLevel)/kB/Temperature)+1)

def GetTotalElectrons(EDOS, FermiLevel, Temperature):
    """
    Calculates the total number of electrons in a system

    Requires:  an array with the electronic density of states, the Fermi Level
               and a temperature
    Returns:  a float with the total number of electrons
    """
    kB = Constants.kB
    x = EDOS[:,0]
    y = EDOS[:,1]*FermiFunction(EDOS[:,0], FermiLevel, Temperature)
    XX=Vect2Matrix(x,y)
    TotalElectrons=intfr2(XX)
    return TotalElectrons

def FindMu(EDOS, Temperature):
    """
    Finds the Fermi Level

    Requires:  an array with the electronic density of states
    Returns:  a float with the Fermi Level
    """
    FermiLevel = 0
    x = EDOS[:,0]
    y = EDOS[:,1]*FermiFunction(EDOS[:,0], FermiLevel, Temperature)
    tck = interpolate.splrep(x,y,k=3)
    if abs(interpolate.splev(0,tck))<1e-3:
        mu =0.
        return mu
    N0 = GetTotalElectrons(EDOS, FermiLevel, 1e-10)
    FermiLevelGuesses = linspace(-0.5, 0.5, 11)
    NumberofElectrons = zeros(len(FermiLevelGuesses),dtype=float)
```

```
    for i in range(len(NumberofElectrons)):
        NumberofElectrons[i] = GetTotalElectrons(EDOS, FermiLevelGuesses[i],
                                                 Temperature)
    XX=Vect2Matrix(FermiLevelGuesses,NumberofElectrons-N0)
    (mu,fx)=newton(0,XX,tol1=1e-3,nmax=20)
    return mu

def TestFunction(x,t,c,k):
    return interpolate.splev(x,(t,c,k))

def GetSingleTemperatureElectronicFreeEnergy(EDOS, Temperature):
    """
    For a given temperature, calculates the total Electronic Free Energy

    Requires:  an array with the electronic density of states and a temperature
    Returns:  a float with the electronic free energy
    """

    mu = FindMu(EDOS, Temperature)
    x=EDOS[:,0]
    y=EDOS[:,1]*FermiFunction(EDOS[:,0], mu, Temperature)
    tck=interpolate.splrep(x,y,s=0,k=3)
    DEf_mu=interpolate.splev(mu,tck)
    if abs(DEf_mu)<1e-3:
        ElectronicFreeEnergy=0
        return ElectronicFreeEnergy
    kB = Constants.kB

    ElectronicEnergy = GetElectronicEnergy(EDOS, Temperature, mu)
    ElectronicEntropy = GetElectronicEntropy(EDOS, Temperature, mu)
    ElectronicFreeEnergy = ElectronicEnergy - Temperature*ElectronicEntropy
    return ElectronicFreeEnergy

def GetElectronicEnergy(EDOS, Temperature, mu):
    """

    """
    x = EDOS[:,0]
    y = EDOS[:,1]*x*FermiFunction(x, mu, Temperature)
    z = EDOS[:,0]*EDOS[:,1]
    XX=Vect2Matrix(x,y)
    integral1 = intfr2(XX)
    XX2=Vect2Matrix(x,z)
    integral2 = intfr2(XX2,min(XX2[:,0]),mu)
    Eel = integral1 - integral2
```

```
    return Eel

def GetElectronicEntropy(EDOS, Temperature, mu):
    """
    For a given temperature, calculates the electronic entropy

    Requires:  an array with the electronic density of states, a temperature
               and the Fermi Level
    Returns:  a float with the total electronic entropy
    """
    kB = Constants.kB
    MinimumEnergy = mu - 20*kB*Temperature
    MaximumEnergy = mu + 20*kB*Temperature
    NewEnergy = linspace(MinimumEnergy, MaximumEnergy, 101)
    tck = interpolate.splrep(EDOS[:,0], EDOS[:,1],s=0,k=3)
    NewDOS = interpolate.splev(NewEnergy, tck)
    f = FermiFunction(NewEnergy, mu, Temperature)
    #DOS * calculation of entropy per state
    dummy1 = NewDOS*(f*log(f)+(1-f)*log(1-f))
    XX=Vect2Matrix(NewEnergy,dummy1)
    #Total electronic entropy
    Sel = -kB*intfr2(XX)

    return Sel

def GetElectronicFreeEnergy(Tmin, Tmax, dT, writeflag=False, plotflag=False):
    """
    Over a range of temperatures, calls other functions to get the electronic
    free energy for each temperature and then fits a curve to obtain the
    temperature dependence of the electronic free energy

    Requires:  minumum, maximum temperatures and a delta temperature
    Returns:  an array with the temperature dependence of the electronic free
              energy (harmonic approximation)
    """
    if os.path.isfile('DOS0'):
        EDOS =File2Array('DOS0')
        if size(EDOS)==0:
            sys.stderr.write('Warning, DOS0 is empty!!....\n')
            sys.exit()
    else:
        sys.stderr.write('Warning, DOS0 does not exist!!....\n')
        sys.stderr.write('Trying to generate it....\n')
        os.system('split_dos &>/dev/null')
        os.system('sleep 1')
```

```
    sys.stderr.write('split_dos command successful\n')
    EDOS = File2Array('DOS0')

    if size(EDOS)==0:
        sys.stderr.write('Error, DOS0 is empty!!....\n')
        sys.exit()

EDOS = File2Array('DOS0')
(R,C)= shape(EDOS)
tck=interpolate.splrep(EDOS[:,0],EDOS[:,1],s=0,k=3)
DEf=interpolate.splev(0,tck)

NewTemperatures = linspace(Tmin, Tmax, 11)

ElectronicFreeEnergies = zeros((len(NewTemperatures)),dtype=float)
for i in range(len(NewTemperatures)):
    ElectronicFreeEnergies[i] = \
        GetSingleTemperatureElectronicFreeEnergy(EDOS, NewTemperatures[i])
ElectronicFreeEnergies=mat(ElectronicFreeEnergies)

if C>3:
    EDOSb=zeros((len(EDOS[:,0]),2),dtype=float)
    EDOSb[:,0]=EDOS[:,0]
    EDOSb[:,1]=abs(EDOS[:,2])
    tckb=interpolate.splrep(EDOSb[:,0],EDOSb[:,1],s=0,k=3)
    ElectronicFreeEnergiesb = zeros((len(NewTemperatures)),dtype=float)
    for i in range(len(NewTemperatures)):
        ElectronicFreeEnergiesb[i] = \
         GetSingleTemperatureElectronicFreeEnergy(EDOSb,
                                                  NewTemperatures[i])
    ElectronicFreeEnergiesb = mat(ElectronicFreeEnergiesb)
    ElectronicFreeEnergies = ElectronicFreeEnergies+ElectronicFreeEnergiesb

if abs(linalg.norm(ElectronicFreeEnergies))<1e-5:
    Temperatures=arange(Tmin,Tmax+dT,dT)
    Output=zeros((len(Temperatures),2),dtype=float)
    Output[:,0]=Temperatures
    if writeflag :          # If writeflag =True
        Array2File(Output, 'felec.dat')
    return Output

a = ones((len(NewTemperatures),1),dtype=float)
b = mat(NewTemperatures**2)
c = mat(NewTemperatures**3)
fittingMatrix = mat(concatenate((a,b.T,c.T),1))
```

```
    pinvFel=linalg.pinv(fittingMatrix)

    xf=pinvFel*ElectronicFreeEnergies.T

    NewElectronicFreeEnergies=fittingMatrix*xf
    Temperatures=arange(Tmin,Tmax+dT,dT)

    a = ones((len(Temperatures),1))
    b = mat(Temperatures**2)
    c = mat(Temperatures**3)
    fittingMatrix = mat(concatenate((a,b.T,c.T),1))
    FinalElectronicFreeEnergy=fittingMatrix*xf

    if plotflag:
        pylab.plot(NewTemperatures,ElectronicFreeEnergies,NewTemperatures,
                   NewElectronicFreeEnergies,Temperatures,
                   FinalElectronicFreeEnergy)
        pylab.show()

    Output = zeros((len(Temperatures), 2))

    Temperatures=array(Temperatures)

    Output[:,0] = Temperatures
    Output[:,1] = FinalElectronicFreeEnergy.T

    if writeflag :          # If writeflag =True
        Array2File(Output, 'felec.dat')

    return Output

if __name__ == "__main__":
    if '-w' in sys.argv:
        writeflag = True
    else:
        writeflag = False

    if '-p' in sys.argv:
        plotflag = True
    else:
        plotflag = False
    Fel=GetElectronicFreeEnergy(1,100,1, writeflag=writeflag,
                                plotflag=plotflag)
```

fileutils.py

```
"""
Contains functions for reading and writing numpy arrays to and from
ascii text files
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.1$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

from os import path
from sys import exit, stderr
from scipy import io

def File2Array(filename):
    if path.isfile(filename):
        inFile = file(filename, 'r')
        output = io.read_array(inFile)
    else:
        print filename
        stderr.write('Error, file does not exist!!...exiting...\n')
        exit()
    return output

def Array2File(data, filename):
    outFile = file(filename, 'w')
    io.write_array(outFile,data,precision=12)
    outFile.close()

def ReadInputFile(filename):
    params = {}
    defaults = {'numberofposvolumes' : 5, 'numberofnegvolumes':3,
                'maxnegvolume' : -0.02, 'maxposvolume' : 0.04,
                'minVCstrain' : 0.0, 'maxVCstrain' : 0.04, 'numVCstrains' : 5,
                'er' : 8.0, 'dr': 0.05, 'maxtemp' : 900, 'mintemp': 1e-5,
                'dTemp' : 5, 'pertnodes' : 4, 'QueueToUse' : 'MXI', 'fr' : 4.0,
                'NumAtomsinSC' : 32}

    if path.isfile(filename):
        inputfile = open(filename)
        data = inputfile.readlines()
        inputfile.close()
        for line in data:
            key = line.split()[0]
```

```
            value = line.split()[2]
            if value.isdigit():
                params[key] = float(value)
            else:
                params[key] = value
    for parameter in defaults:
        if not params.has_key(parameter):
            params[parameter] = defaults[parameter]
    if params['mintemp'] == 0:
        params['mintemp'] = 1e-5

    return params
```

Fvib.py

```
#! /usr/bin/env python

__author__ = "Mike Williams (michaeleric.williams@gmail.com), \
              and Raymundo Arroyave (raymundo@fastmail.fm)"
__version__ = "0.2$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams & Raymundo Arroyave"
__license__ = "Python"

import Constants
import fileutils
from numpy import *
import sys
import pylab
import os
import CollectData
#from CollectData import *
from NumericalMethods import *

def GetVibrationalFreeEnergy(Tmin, Tmax, dT, writeflag=False, plotflag=False):
    """
    Calculates the vibrational contribution to the free energy
    To be run in each volume directory

    Requires: Numpy array with the Density of States
    (read from an input file, not passed as an argument)
    Tmin, Tmax, dT a boolean flag telling it weather to write an output file

    Returns:  Vibrational free energy as a function of temperature
    (in a 2d array)
```

```
Writes an fvib.dat file in each volume directory
"""
h = Constants.h
kB = Constants.kB

#Read vdos.out
if os.path.isfile('vdos.out'):
    data = fileutils.File2Array('vdos.out')
    if size(data)==0:
        sys.stderr.write('Warning, vdos.out is empty!!....')
        sys.exit()
else:
    #print '%s does not exist!'%(filename)
    sys.stderr.write('Warning, vdos.out does not exist!!....')
    sys.exit()

VDOS = zeros((len(data[:,0]), 2))
counter = 0
for row in range(len(data[:,0])):
    frequency = data[row,0]
    states    = data[row,1]
    if frequency >= 0:
        VDOS[counter][0] = frequency
        VDOS[counter][1] = states
    counter = counter + 1

#remove negative frequencies
VDOS = VDOS[:counter+1,:]
Nu = VDOS[:,0] #frequencies
Gv = VDOS[:,1] #number of states
deltaNu      = diff(Nu, 1, axis = 0).T
NuN = (Nu[1:,] + Nu[0:-1])*0.5
GvN = (Gv[1:,] + Gv[0:-1])*0.5
sumdegrees = sum(deltaNu * GvN)
IntegratedNumberOfAtoms = sumdegrees/3
RealNumberOfAtoms = CollectData.CountAtoms()
ImaginaryStates = (RealNumberOfAtoms - IntegratedNumberOfAtoms)*3

#create a temperature array
Temperatures = arange(Tmin, Tmax+dT, dT).T
Fvib = zeros(len(Temperatures),dtype=float)
FvibImaginary = zeros(len(Temperatures),dtype=float)
for i in range(len(Temperatures)):
    constant = (h * NuN)/(2 * kB * Temperatures[i])
```

```
        temporary = deltaNu*GvN*log(2*sinh(constant))
        #overflow
        if sum(temporary) == Inf:
            for j in range(len(constant)):
                if constant[j] > 500:
                    temporary[j] = deltaNu[j]*GvN[j]*constant[j]

        Fvib[i] = kB*Temperatures[i]*sum(temporary)
        FvibImaginary[i] = ImaginaryStates*(1-log(Temperatures[i]))\
                           * kB/2 * Temperatures[i]
    TotalFvib = Fvib + FvibImaginary

    if plotflag:
        pylab.plot(Temperatures,TotalFvib)
        pylab.show()

    Output = zeros((len(Temperatures), 2))
    Output[:,0] = Temperatures
    Output[:,1] = TotalFvib
    if writeflag :          # If writeflag =True
        fileutils.Array2File(Output, 'fvib.dat')

    return Output

def CalculateDebyeTemperature(VDOS,n):
    """
    Calculates the Debye Temperature from a given Phonon Density of States
    based on the nth Moment of said DOS.
    n=0    ->    Entropy Debye Temperature
    n=1    ->    Average ny in vdos.out
    n=2    ->    Cp Debye Temperature
    n=-2   ->    Debye-Waller Debye Temperature
    """
    h = Constants.h
    kB = Constants.kB
    n    =    float(n)

    # Identify where negative frequencies are:
    indexV=0
    for i in range(len(VDOS[:,0])):
        if VDOS[i,0]>=0:
            indexV=i
            break

    VDOS=VDOS[indexV:,:]
```

```
if n <0:
    VDOS[:,1]=VDOS[:,1]/intfr2(VDOS)
    VDOS[0,0]=1e4
    VDOS[0,1]=0
    (DT_0,M0)=      CalculateDebyeTemperature(VDOS,0)
    nu_0      =     DT_0*kB/h
    nu_min    =     nu_0*0.5
    nu_max    =     nu_0*2
    nu_i      =     linspace(nu_min,nu_max,31)
    Integralb     =     zeros((len(nu_i),2))
    Integralb[:,0]=nu_i
    a         =     zeros((len(VDOS[:,0]),2))
    a[:,0]    =     VDOS[:,0]
    a[:,1]    =     VDOS[:,0]**n
    a[:,1]    =     a[:,1]*VDOS[:,1]
    Integrala     =     intfr2(a)
    b         =     zeros((len(VDOS[:,0]),2))
    for i in range(len(nu_i)):
        b[:,0]      =     VDOS[:,0]
        b[:,1]    =     VDOS[:,0]**2
        b[:,1]    =     3.*b[:,1]*(b[:,0]**n)
        b[:,1]    =     b[:,1]/(nu_i[i]**3.)
        Integralb[i,1] =     intfr2(b)
    Integralb[:,1]    =     Integralb[:,1]-Integrala
    (nu_n,fx)     =     newton(nu_0,Integralb,tol1=1e-6)
    Moment        =     Integrala**(1/n)

elif n ==0:
    VDOS[0,0]=1e-6
    VDOS[:,1]=VDOS[:,1]/intfr2(VDOS)
    a         =     zeros((len(VDOS[:,0]),2))
    a[:,0]        =     VDOS[:,0]
    a[:,1]        =     log(VDOS[:,0])
    a[:,1]        =     a[:,1]*VDOS[:,1]
    Integrala     =     intfr2(a)
    nu_n          =     exp(1/3.+Integrala)
    Moment        =     exp(Integrala)
else:
    VDOS[0,0]=1e-6
    VDOS[:,1]=VDOS[:,1]/intfr2(VDOS)

    a         =     zeros((len(VDOS[:,0]),2))
    a[:,0]        =     VDOS[:,0]
    a[:,1]        =     VDOS[:,0]**n
```

```
        a[:,1]          =    a[:,1]*VDOS[:,1]
        Integrala       =    intfr2(a)
        nu_n            =    ((n+3.)/3.*Integrala )**(1/n)
        Moment          =    Integrala**(1/n)

    DebyeTemperature    =    h*nu_n/kB

    return (DebyeTemperature,Moment)


class PhononDOS:

    def __init__(self,writeflag=False,*VDOS):

        if len(VDOS)==0:
            if os.path.isfile('vdos.out'):
                VDOS = fileutils.File2Array('vdos.out')
                if size(VDOS)==0:
                    sys.stderr.write('Warning, vdos.out is empty!!....')
                    sys.exit()
            else:
                sys.stderr.write('Warning, vdos.out does not exist!!....')
                sys.exit()

        self.VDOS = VDOS
        self.TDM = zeros((5,3),dtype=float)
        # TDM is a Matrix with the Debye Temperatures and characteristic
        #moments for n=-2,0,1,2 and 4.
        self.CalculateTDM()
        if writeflag :          # If writeflag =True
            fileutils.Array2File(self.TDM, 'DTMnm.dat')
        self.MaximumFrequency=max(self.VDOS[:,0])
        self.AverageFrequency=self.TDM[2,2]
        self.EntropyDebyeTemperature=self.TDM[1,1]
        self.AverageDebyeTemperature=self.TDM[2,1]
        self.CpDebyeTemperature=self.TDM[3,1]
        self.DebyeWallerDebyeTemperature=self.TDM[0,1]

    def CalculateTDM(self):
        ilist=[-2,0,1,2,4]
        for i in range(len(ilist)):
            self.TDM[i,0]=ilist[i]
            (self.TDM[i,1],self.TDM[i,2]) = \
                CalculateDebyeTemperature(self.VDOS,ilist[i])
```

```
    def PlotDOS(self):
        pylab.plot(self.VDOS[:,0],VDOS[:,1])
        pylab.ylabel('Frequency, Hz')
        pylab.xlabel('DOS')
        pylab.title('Phonon DOS')
        pylab.show()

def get_Cp_from_DebyeTemperature(T,TD):
    """
    Calculates the Specific heat from a Debye solid.
    Requires: Temperature and Debye Temperature
    Output: Specific heat
    """
    kB=Constants.kB

    if TD/T>20.:
        Cp=12.*pi**4/5.*kB*(T/TD)**3.
        return Cp
    else:
        xmin=1e-6;
        xmax=TD/T;
        x=linspace(xmin,xmax,200)
        XX=zeros((len(x),2),dtype=float)
        XX[:,0]=x
        XX[:,1]=(x**4.)*exp(x)/(exp(x)-1.)**2.
        Cp=9*kB*(T/TD)**3.*intfr2(XX)
        return Cp

def get_DebyeTemperature(T,TargetCp,GuessDebyeTemperature):
    """
    Obtains the Debye Temperature of a Harmonic Solid so it
    has the target Specific heat
    """
    DT_Test=linspace(30.,GuessDebyeTemperature+400.,20)
    XX=zeros((len(DT_Test),2),dtype=float)

    XX[:,0]=DT_Test
    i=range(len(XX[:,0]))
    for i in range(len(XX[:,0])):
        XX[i,1]=get_Cp_from_DebyeTemperature(T,XX[i,0])
    XX[:,1]=XX[:,1]-TargetCp
    (DebyeTemperature,fx)=newton(GuessDebyeTemperature,XX,toll=1e-10)
    return DebyeTemperature
```

```
if __name__ == "__main__":
    if '-w' in sys.argv:
        writeflag = True
    else:
        writeflag = False

    if '-p' in sys.argv:
        plotflag = True
    else:
        plotflag = False



    if '-ph' in sys.argv:
        print "Testing Phonon Class and Methods"
        VDOS = fileutils.File2Array('vdos.out')
        (DebyeTemperature,Moment)=CalculateDebyeTemperature(VDOS,0)
        ph=PhononDOS(writeflag=True)
        print DebyeTemperature,Moment
        print ph.TDM
        if '-p' in sys.argv:
            ph.PlotDOS()
    elif '-db' in sys.argv:
        print "Testing Debye Temperature Calculations"
        Cp=get_Cp_from_DebyeTemperature(50.,300.)
        print Cp
        DebyeTemperature=get_DebyeTemperature(100, 6e-5, 600)
        print DebyeTemperature
    else:
        #read parameter file
        inputparameterfile = 'ELCparams.in'
        params = fileutils.ReadInputFile(inputparameterfile)
        maxtemp = float(params['maxtemp'])
        mintemp = float(params['mintemp'])
        dTemp = float(params['dTemp'])
        #perform operations
        FVib = GetVibrationalFreeEnergy(1, 1000, 1, writeflag = writeflag,
                                         plotflag=plotflag)
```

GetParams.py

```
"""
A method for reading an input file and preparing
a dictionary of input parameters
"""
```

```
__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.3$"
__date__ = "$Date: November 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

import os
def ReadInputFile(filename):
    params = {}
    defaults = {'numberofposvolumes':5, 'numberofnegvolumes':3,
                'maxnegvolume':-0.02, 'maxposvolume':0.04,
                'minVCstrain':0.0, 'maxVCstrain':0.04, 'numVCstrains':5,
                'er':8.0, 'dr':0.05, 'fr':4.0, 'maxtemp':2000,
                'mintemp':1e-5, 'dTemp':1, 'pertnodes':4,
                'PertPollTime':300, 'relaxnodes':1,
                'RelaxPollTime':5,'QueueToUse':'MXI', 'NumAtomsinSC':32,
                'StrainThermo' : False}

    #open input file and read in specified parameters
    if os.path.isfile(filename):
        inputfile = open(filename)
        data = inputfile.readlines()
        inputfile.close()
        for line in data:
            key = line.split()[0]
            value = line.split()[2]
            if value[0].isdigit() or value[0] == '-':
                params[key] = float(value)
            else:
                params[key] = value

    #for each parameter not in the input file use the default value
    for parameter in defaults:
        if not params.has_key(parameter):
            params[parameter] = defaults[parameter]
    if params['mintemp'] == 0:
        params['mintemp'] = 1e-5

    #the grid engine script requires integer numbers for the number of nodes
    params['pertnodes'] = int(params['pertnodes'])
    params['relaxnodes'] = int(params['relaxnodes'])
    return params
```

NumericalMethods.py

```
from scipy import *
from numpy import *

def newton(x0,data,toll=1e-10,nmax=20):
    tck=interpolate.splrep(data[:,0],data[:,1],k=3,s=0)
    err=toll+1
    nit=0
    x=float(x0)
    fx=interpolate.splev(x,tck)
    dfx=interpolate.splev(x,tck,der=1)

    while (nit<nmax) and (err>toll):
        if (dfx==0):
            err=toll*1e-10
        else:
            xn=x-fx/dfx
            err=abs(xn-x)
            x=xn
            fx=interpolate.splev(x,tck)
            dfx=interpolate.splev(x,tck,der=1)
        nit=nit+1
    fx=interpolate.splev(x,tck)
    return (xn,fx)

def intfr2(data,*args):
    if (len(args)<2):
        a=array(data[:,0]).min()
        b=array(data[:,0]).max()
    else:
        a=args[0]
        b=args[1]
    x=array(data[:,0])
    y=array(data[:,1])
    for i in range(len(x)):
        if x[0]>a:
            intmin=0
            break
        elif x[i]==a:
            intmin=i
            break
        elif ((a>x[i])and(a<x[i+1])):
            intmin=i+1
            break
    for i in range(len(x)):
            if x[i]==b:
```

```
                intmax=i
                break
            elif (b>x[i])and(b<x[i+1]):
                intmax=i
                break
        dx=diff(x[intmin:intmax+1],n=1,axis=0)
        y0=y[intmin:intmax]
        y1=y[intmin+1:intmax+1]

        s=sum(dx*0.5*(y0+y1))

        sa=0
        sb=0

        if not(intmin == 0):
            ya = y[intmin-1]+(y[intmin]-y[intmin-1])/(x[intmin]-x[intmin-1])*\
                 (a-x[intmin-1])
            sa=0.5*(y[intmin]+ya)*(x[intmin]-a)

        if not(intmax == len(x)-1):
            yb = y[intmax]+(y[intmax+1]-y[intmax])/(x[intmax+1]-x[intmax])*\
                 (b-x[intmax])
            sb=0.5*(y[intmax]+yb)*(b-x[intmax])

        s=s+sa+sb

        return s

def Vect2Matrix(*args):

    NC=len(args)
    XX = zeros((len(args[0]),1),dtype=float)

    XX[:,0]=args[0]
    for i in range(NC-1):
        XX2=zeros((len(args[0]),i+2),dtype=float)
        XX2[:,0:i+1]=XX
        XX2[:,i+1]=args[i+1]
        del XX
        XX=XX2
        #print XX
        del XX2
    #print XX
    return XX
```

```
#=============================================================
def fprime(x,y):
# Calculates the first numerical derivative
    x=array(x)
    y=array(y)

    nx=len(x)
    dx=diff(x,n=1,axis=0)
    dfp=zeros((nx,1))
    coef_ffd=ForwardFirstDifference(dx,0)
    coef_bfd=BackwardFirstDifference(dx,nx-1)
    dfp[0]  = dot(coef_ffd,y[0:3])
    dfp[-1] = dot(coef_bfd,y[-3:])
    j=arange(1,nx-1)
    dfp[j]=CentralFirstDifference(y,dx,j)
    return dfp

def ForwardFirstDifference(dx,i):
    a=(1+dx[i+1]/dx[i])*dx[i]/dx[i+1];
    b=-dx[i]/(dx[i+1]*(1+dx[i+1]/dx[i]));
    coef=mat(concatenate((-(a+b) , a ,b ),1))/dx[i]
    return coef

def BackwardFirstDifference(dx,i):
    a=-(1+dx[i-2]/dx[i-1])*dx[i-1]/dx[i-2];
    b=dx[i-1]/(dx[i-2]*(1+dx[i-2]/dx[i-1]));
    coef=mat(concatenate((b ,a ,-(a+b)),1))/dx[i-1]
    return coef

def CentralFirstDifference(y,dx,i):
    a=-dx[i]/(dx[i-1]*(dx[i]/dx[i-1]+1));
    b= dx[i-1]/(dx[i]*(dx[i]/dx[i-1]+1));
    fpc=(a*y[i-1] -(a+b)*y[i]+b*y[i+1])/dx[i-1];
    return fpc

def IntegrateSampleData(x, y, xmin, xmax):
    """
    Fits a cubic spline to a set of data and integrates over a given range

    Requires:  2 arrays, one of x and one of y values, 2 floats defining
    the range over which to integrate

    Returns:  a float with the integrated value
    """
```

```
    tck = interpolate.splrep(x,y)
    return     interpolate.splint(xmin,xmax,tck)

if __name__ == "__main__":
    x=array(linspace(0.,10.,10000))
    x=mat(x)
    x=x.T
    x=array(x)
    y=x**2-5
    data=ones((len(x),2),dtype=float)
    data=mat(data)
    data[:,0]=x
    data[:,1]=y
    s=intfr2(data,2,8)[0]
    print s
    tck=interpolate.splrep(data[:,0],data[:,1],k=3)
    fx=interpolate.splev(1.,tck)
    dfx=interpolate.splev(1.,tck,der=0)
    (xf,fx)=newton(1.1,data,toll=1e-15,nmax=30)
    print xf, fx
```

CheckJobs.py

```
"""
Methods used in the calcultion of electronic free energies.
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)
__version__ = "0.3$"
__date__ = "$Date: 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams & Raymundo Arroyave"
__license__ = "Python"

import os

def CheckRelax(dirname):
  """
  Requires:  complete path to directory where job is
  """
  done = False
  if os.path.isfile(dirname+'/OSZICAR'):
    oszicar = open(dirname+'/OSZICAR').readlines()
    for line in oszicar:
      if 'E0' in line:
        done = True
```

```
  return done


def CheckSSC(dirname):
  """
  Requires:  complete path to directory where job is
  """
  done = False
  if os.path.isfile(dirname+'/INCAR'):
    incar = open(dirname+'/INCAR').readlines()
    for line in incar:
      if 'NSW = 0' in line:
        if os.path.isfile(dirname+'/OSZICAR') \
            and (os.path.getsize(dirname+'/OSZICAR') > 0):
          oszicar = open(dirname+'/OSZICAR').readlines()
          for line2 in oszicar:
            if 'E0' in line2:
              if os.path.getmtime(dirname+'/OSZICAR') \
                  > os.path.getmtime(dirname+'/INCAR'):
                done = True
  return done

def CheckExtractVasp(dirname):
  done = False
  if os.path.isfile(dirname+'/str_relax.out'):
    done = True
  return done

def CheckFelec(dirname):
  done = False
  if os.path.isfile(dirname+'/CONTCAR'):
    if os.path.isfile(dirname+'/felec'):
      if os.path.getmtime(dirname+'/CONTCAR') < \
          os.path.getmtime(dirname+'/felec'):
        done = True
  return done

def CheckDirSetup(dirname):
  done = False
  if os.path.isdir(dirname):
    if os.path.isfile(dirname+'/INCAR'):
      done = True
  return done
```

controlvasp.py

```
"""
Methods used to control VASP calculations for the
calculation of thermodynamic and elastic properties
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.3$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

import os
import numpy
import filecmp
import sys
import shutil
import gridengine
import SuperCell
import CheckJobs
from ELCMethods import *
from VASPUtils import *
from GetParams import *
from CheckJobs import *
from ThermoMethods import *

params = ReadInputFile('ELCParams.in')

#--------------------Directory Names--------------------------------
def StrainDirList(minstrain, maxstrain, numstrains, whichELC):
  strains = numpy.linspace(minstrain*100, maxstrain*100, numstrains)
  output = []
  for i in strains:
    if whichELC == 'c44':
      output.append('44str_'+str(int(i)))
    else:
      output.append('11str_'+str(int(i)))
  return output

def VolDirList(Vmin, Vmax, n):
  Vmin = Vmin * 100
  Vmax = Vmax * 100
  vols = numpy.linspace(Vmin, Vmax, n)
  output = []
  for i in vols:
    output.append('vol_'+str(int(i)))
```

```
  return output

#----------------------Complete scripts-------------------------
def C11_C12(params, commands):
  whichELC = 'c11'
  volumedirectories = VolDirList(params['maxnegvolume'], \
                      params['maxposvolume'], \
                      params['numberofposvolumes'] + \
                      params['numberofnegvolumes']-1)
  straindirectories = StrainDirList(params['minVCstrain'], \
                      params['maxVCstrain'], params['numVCstrains'],\
                      whichELC)
  parentdir = os.getcwd()
  Thermo(params, commands)
  os.chdir(parentdir)
  SetUpStrains(volumedirectories, straindirectories,
               params['minVCstrain'], params['maxVCstrain'],
               params['numVCstrains'], whichELC)
  StrainCalcs(volumedirectories, straindirectories, params)
  if params['StrainThermo']:
    SetUpELCPerturbations(params['er'], params['dr'],
                          commands['feleccommand'],
                          commands['strainfitfccommand'],
                          volumedirectories, straindirectories,
                          params['NumAtomsinSC'])
    ELCPerturbationCalculations(params['er'], params['dr'],
                                commands['feleccommand'],
                                commands['strainfitfccommand'],
                                volumedirectories,
                                straindirectories,
                                params['pertnodes'], params)
    ELCPostProcessing(volumedirectories, straindirectories,
                      commands['coldcurvefitfc'])

def C44(params, commands):
  whichELC = 'c44'
  volumedirectories = VolDirList(params['maxnegvolume'], \
                      params['maxposvolume'], \
                      params['numberofposvolumes'] + \
                      params['numberofnegvolumes']-1)
  straindirectories = StrainDirList(params['minVCstrain'],\
                                    params['maxVCstrain'],\
                                    params['numVCstrains'], whichELC)
  Thermo(params, commands)
  SetUpStrains(volumedirectories, straindirectories,
```

```
                params['minVCstrain'], params['maxVCstrain'],
                params['numVCstrains'], whichELC)
  StrainCalcs(volumedirectories, straindirectories, params)

def SCC():
  InitialRelaxation()
  InitialSSC()

def Thermo(params, commands):
  if params['numberofnegvolumes'] == 0:
    numberofvolumes = params['numberofposvolumes']
  else:
    numberofvolumes = params['numberofposvolumes'] + \
                      params['numberofnegvolumes']-1
  volumedirectories = VolDirList(params['maxnegvolume'],
                                 params['maxposvolume'],
                                 numberofvolumes)
  InitialRelaxation(params)
  InitialSSC(params)
  SetUpVolumes(volumedirectories, commands['posvolumefitfccommand'],
               commands['negvolumefitfccommand'])
  RelaxVolumes(volumedirectories, params)
  VolumesSSC(volumedirectories, params)
  VolumesPostProcess(volumedirectories, commands['feleccommand'])
  SetUpQHPerturbations(volumedirectories, commands['posvolumefitfccommand'],
                       commands['negvolumefitfccommand'])
  pertdirs = QHPerturbationCalculations(volumedirectories,
                                        params['pertnodes'], params)
  QHExtractVasp(pertdirs)
  if not os.path.isfile('vol_0/vdos.out'):
    os.system(commands['coldcurvefitfc'])
```

ELCMethods.py

```
"""
Specific scripts for running ELC strain calculations with VASP
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.4$"
__date__ = "$Date: November 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

import os
```

```
import gridengine
import SuperCell
import CheckJobs
from VASPUtils import *

def SetUpStrains(voldirs, straindirs, minstrain, maxstrain,
                 numstrains, whichELC):
  parentdirectory = os.getcwd()
  badjobs = []
  for i in voldirs:
    for j in straindirs:
      dirname = parentdirectory+'/'+i+'/'+j
      if not CheckJobs.CheckDirSetup(dirname):
        badjobs.append(dirname)
  if len(badjobs) > 0:
    print 'Setting up strain directories'

  for directory in badjobs:
    if not os.path.isdir(directory):
      os.mkdir(directory)
    os.chdir(directory)
    if directory[-2] == '-':
      strain = float(directory[-2:])*.01
    else:
      strain = float(directory[-1])*.01
    os.system('cp ../POSCAR .')
    os.system('cp ../POTCAR .')
    os.system('cp ../INCAR .')
    ModifyInputFile('INCAR', 'ISIF', 2)
    os.system('cp ../KPOINTS .')
    StrainLattice(strain, whichELC)
    KpointsToGamma()
  os.chdir(parentdirectory)
  print 'Strain directories set up'

def StrainCalcs(voldirs, straindirs, params):
  badjobs = []
  parentdirectory = os.getcwd()
  for i in voldirs:
    for j in straindirs:
      dirname = parentdirectory+'/'+i+'/'+j
      if not CheckJobs.CheckSSC(dirname):
        badjobs.append(dirname)
  JobIDs = []
  for directory in badjobs:
```

```
      os.chdir(directory)
      queuefile = gridengine.CreateQueueFile('Strain', '00:30:00', 1,
                                             params['QueueToUse'])
      print os.getcwd()
      JobIDs.append(gridengine.SubmitJob(queuefile))
    if len(badjobs) > 0:
      print 'Performing Strain Calculations'
      gridengine.WaitForJobs(JobIDs)
    os.chdir(parentdirectory)
    print 'Strain calculations done'


def SetUpELCPerturbations(er, dr, feleccommand, strainfitfccommand, voldirs,
                          straindirs, NumAtomsinSC):
    parentdirectory = os.getcwd()
    badjobs = []
    for i in voldirs:
      for j in straindirs:
        if not os.path.isdir(parentdirectory+'/'+i+'/'+j+'/vol_0'):
          badjobs.append(parentdirectory+'/'+i+'/'+j)
    for directory in badjobs:
      print 'Setting up perturbations in %s'%(directory)
      thisstraindirectory = directory
      perturbationdir = thisstraindirectory+'/vol_0/p+'+str(dr)+\
                        '_'+str(int(er))+'_0'
      os.system('cp '+parentdirectory+'/vasp.wrap.static '+
                thisstraindirectory+'/vasp.wrap')
      os.chdir(thisstraindirectory)
      os.system('extract_vasp')
      os.system('cp '+thisstraindirectory+'/str_relax.out '
                +thisstraindirectory+'/str.out')
      os.system(feleccommand)
      os.system(strainfitfccommand)
      os.chdir(perturbationdir)
      os.system('str2ezvasp')
      os.system('ezvasp -n vasp.in')
      newer = SuperCell.VerifyNumAtoms(thisstraindirectory,
                                       perturbationdir, er, dr,
                                       NumAtomsinSC)
      perturbationdir = thisstraindirectory + \
                        '/vol_0/p+'+str(dr) + '_'+str(int(newer)) + '_0'
      os.system('str2ezvasp')
      os.system('ezvasp -n vasp.in')
    os.chdir(parentdirectory)
```

```
  print 'Perturbation directories set up'

def ELCPerturbationCalculations(er, dr, feleccommand, strainfitfccommand,
                                voldirs, straindirs, nodes, params):
  badjobs = []
  PertDirs = []
  parentdirectory = os.getcwd()
  for i in voldirs:
    thisvolumedirectory = parentdirectory+'/'+i
    for j in straindirs:
      thisstraindirectory = thisvolumedirectory+'/'+j
      PertDirs.append(SuperCell.GetPertDirList(thisstraindirectory))
  for i in PertDirs:
    if not CheckJobs.CheckSSC(i[0]):
      badjobs.append(i[0])
  JobIDs = []
  for directory in badjobs:
    os.chdir(directory)
    queuefile = gridengine.CreateQueueFile('Pert', '10:00:00', nodes,
                                           params['QueueToUse'])
    shutil.copyfile(directory+'/KPOINTS', directory+'/../KPOINTS')
    KpointsToGamma()
    JobIDs.append(gridengine.SubmitJob(queuefile))
  if len(JobIDs) > 0:
    print 'Performing perturbation calculations'
    gridengine.WaitForJobs(JobIDs)
  os.chdir(parentdirectory)

  print 'Perturbation calculations done'

def ELCPostProcessing(voldirs, straindirs, fitfccommand):
  #check if this step has been done
  #by looking for an str_relax.out file
  PertDirs = []
  badjobs = []
  parentdirectory = os.getcwd()
  for i in voldirs:
    thisvolumedirectory = parentdirectory+'/'+i
    for j in straindirs:
      thisstraindirectory = thisvolumedirectory+'/'+j
      PertDirs.append(SuperCell.GetPertDirList(thisstraindirectory))
  for i in PertDirs:
    if not CheckJobs.CheckExtractVasp(i[0]):
      badjobs.append(i[0])
  if len(badjobs) > 0:
```

```
    print 'Extracting data from perturbation calculations'
    for directory in badjobs:
      print directory
      os.chdir(directory)
      os.system('extract_vasp')

  #same routine but for final fitfc command
  badjobs = []
  for i in voldirs:
    thisvolumedirectory = parentdirectory+'/'+i
    for j in straindirs:
      PetDirs = []
      thisstraindirectory = thisvolumedirectory+'/'+j
      if not (os.path.isdir(thisstraindirectory) \
        and os.path.isfile(thisstraindirectory+'/vol_0/vdos.out')):
        badjobs.append(thisstraindirectory)
  if len(badjobs) > 0:
    print 'Running final fitfc command'
    for directory in badjobs:
      print directory
      os.chdir(directory)
      os.system(fitfccommand)
  os.chdir(parentdirectory)

  print 'Perturbation post processing done'
```

gridengine.py

```
"""
Contains various functions for used to interact with the Sun Grid Engine
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.2$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

import os
import time
import datetime
import CheckJobs

def WaitForJobs(JobIDlist, sleeptime=5):
  """
```

```
    Checks the queue status for a list of job ID numbers and waits for
    them to finish. This in essence freezes the program until the jobs are
    done.

    Requires: A list of integer Job ID's that have been assigned by the
    SGE system

    Returns: Nothing
    """

    print 'Monitoring jobs'
    while 1:
      alldone = 1
      data = os.popen('qstat -u mew2454')
      lines = data.readlines()
      data.close()
      if len(lines) == 0:
        print 'You have no jobs in queue!'
        break
      else:
        lines = lines[2:]
        runningjobs = []
        for row in lines:
          job = int(row.split()[0])
          if job in JobIDlist:
            runningjobs.append(job)
        print 'Checking for the following jobs: %s'%(runningjobs)
        for i in JobIDlist:
          if i not in runningjobs:
            index = JobIDlist.index(i)
            del JobIDlist[index]
            print 'Job %d is finished.'%(i)
      if len(JobIDlist) == 0:
        print 'All Done!'
        break
      time.sleep(sleeptime)


def CreateQueueFile(JobName, WallTime, NumberofNodes, QueueToUse):
    """
    Creates a queue file for use in the SGE

    Requires: string - the name for the job
        string - the walltime for the job in hh:mm:ss format
        integer - the number of nodes to use
```

```
    Returns: a string with name of the queuefile
    """
    lines = []
    lines.append('#!/bin/sh\n')
    lines.append('#$ -N '+JobName+'\n')
    lines.append('#$ -l h_rt='+WallTime+'\n')
    if QueueToUse == 'MXI':
      lines.append('#$ -pe mx-mpich '+str(NumberofNodes)+'\n')
      lines.append('#$ -q %s.q'%QueueToUse+'\n')
      lines.append('#$ -cwd'+'\n')
      lines.append('#$ -v MPICH_PROCESS_GROUP=no'+'\n')
      lines.append('/Users/Shared/mx/mpich-mx-1.2.7..4/bin/mpirun.ch_mx \
          --mx-kill 15 -np $NSLOTS -machinefile $TMPDIR/machines \
          /Users/mew2454/vasp/bin/vasp.mpichmx_mac_intel_ifort_mkl >\
          vasp.out'+'\n')
    else:
      lines.append('#$ -pe rshlammp* '+str(NumberofNodes)+'\n')
      lines.append('#$ -cwd'+'\n')
      lines.append('#$ -q %s.q'%QueueToUse+'\n')
      lines.append('/Users/Shared/m0u1971/bin/lam-7.1.1_underscore/bin/\
          mpirun C -ssi rpi tcp /Users/mew2454/vasp/bin/vasp.lam_g5_tcp >\
          vasp.out'+'\n')
    f = open(JobName+'.q', 'w')
    f.writelines(lines)
    f.close()
    return JobName+'.q'

def SubmitJob(queuename):
    """
    Submits job to the cluster and captures the Job ID in the queue system

    Requires: String with name of queue file to submit
    Returns: Job Id (integer)
    """
    command = 'qsub '+queuename
    j = os.popen(command)
    line = j.readline()
    line = line.split()
    JobID = int(line[2])
    return JobID

def ErrorCheck(jobtype, dirlist):
    """
    Checks a list of directories for completion of a job,
    if there was an error, it lists the directory and the step
```

```
  in the process where the error occured to a file 'pyvasp.err'
  """
  parentdir = os.getcwd()
  badjobs = []
  if jobtype == 'relax':
    checkfunc = CheckJobs.CheckRelax
  elif jobtype == 'static':
    checkfunc = CheckJobs.CheckSSC
  elif jobtype == 'pert':
    checkfunc = CheckJobs.CheckSSC

  for directory in dirlist:
    if not checkfunc(directory):
      badjobs.append(directory, jobtype, datetime.datetime.now())
  if len(badjobs) > 0:
    outputfile = open('pyvasp.err', w)
    outputfile.writelines(badjobs)
    outputfile.close()
    sys.exit()
```

run_vasp.py

```
#! /usr/bin/env python
"""
Main program for controlling VASP calculations
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.2$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

import sys
import os
from controlvasp import *
from GetParams import ReadInputFile

def ProcessArgs(args):
  #get input parameters
  process = 'none'
  if '-thermo' in args or '-t' in args:
    process = 'thermo'
  if '-c11' in args or '-C11' in args:
    process = 'C11-C12'
```

```
  if '-ssc' in args:
    process = 'ssc'
  if ('-c44' or '-C44') in args:
    process = 'C44'
  if process == 'none':
    print 'Please enter a valid process to run'
    sys.exit()
  return process

if __name__ == "__main__":
  #check for a vasp.in file
  if not os.path.isfile('vasp.in'):
    print 'No vasp.in file!'
    sys.exit()

  #select a job to run and read in parameters
  process = ProcessArgs(sys.argv)
  if os.path.isfile('ELCparams.in'):
    inputfilename = 'ELCparams.in'
  else:
    inputfilename = ''
  params = ReadInputFile(inputfilename)

  #various commands to be used later
  posvolumefitfccommand = 'fitfc -er=%f -ns=%f -ms=%f -dr=%f'%\
      (params['er'], params['numberofposvolumes'],
       params['maxposvolume'], params['dr'])
  if params['numberofnegvolumes'] > 0:
    negvolumefitfccommand = 'fitfc -er=%f -ns=%f -ms=%f -dr=%f'%\
        (params['er'], params['numberofnegvolumes'],
         params['maxnegvolume'], params['dr'])
  else:
    negvolumefitfccommand = ''
  strainfitfccommand = 'fitfc -er=%f -ns=%f -ms=%f -dr=%f -nrr'%\
      (params['er'], 1, 0, params['dr'])
  coldcurvefitfc = 'fitfc -f -fr=%f -T1=%f'%(float(params['fr']),
                                             float(params['maxtemp']))
  feleccommand = 'felec -T1=%f'%(float(params['maxtemp']))
  commands = {'posvolumefitfccommand':posvolumefitfccommand,
              'negvolumefitfccommand':negvolumefitfccommand,
              'strainfitfccommand':strainfitfccommand,
              'coldcurvefitfc':coldcurvefitfc,
              'feleccommand': feleccommand}

  #execute the appropriate script
```

```
  if process == 'C11-C12':
    C11_C12(params, commands)
  if process == 'C44':
    C44(params, commands)
  if process == 'thermo':
    Thermo(params, commands)
  if process == 'SSC':
    SSC()
```

SuperCell.py

```
"""
Collection of methods dealing with supercells
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.1$"
__date__ = "$Date: June 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

import os
import sys

def CountAtoms():
  """
  Looks in the POSCAR file to get the number of atoms in a supercell
  """
  if os.path.isfile('POSCAR'):
    dummy = os.popen('cat POSCAR | head -n 6 | tail -n 1')
    data = dummy.readlines()
    data = data[0].split()
    integerlist = []
    for element in data:
      integerlist.append(int(element))
    numberofatoms = sum(integerlist)
    return numberofatoms
  else:
    sys.exit()

def VerifyNumAtoms(straindir, pertdir, er, dr, target):
  """
  Ensures that the correct -er option was set in ATAT,
  if not, it re-runs it until it finds the correct supercell size
```

```
    Requires:  fitfc command with necessary options
               parameters dictionary used in fitfc command
               an integer of the target number of atoms
    Returns:  parameter dictionary with new er value
    """

    os.chdir(pertdir)
    NAtoms = CountAtoms()
    print NAtoms
    while NAtoms < target:
      er = er + .05
      print 'Incorrect # of atoms in supercell, attempting to fix %s'%\
          (pertdir)
      os.chdir(straindir)
      os.system('rm -r vol_*')
      fitfccommand = 'fitfc -er=%f -ns=%f -ms=%f -dr=%f -nrr'%(er, 1, 0, dr)
      os.system(fitfccommand)
      perturbationdir = GetPertDirList(straindir)[0]
      os.chdir(perturbationdir)
      os.system('str2ezvasp')
      os.system('ezvasp -n vasp.in')
      NAtoms = CountAtoms()
      print NAtoms
    return er

def GetPertDirList(straindirpath):
    pertdirlist = []
    if os.path.isdir(straindirpath+'/vol_0'):
      files = os.listdir(straindirpath+'/vol_0')
      for i in files:
        if os.path.isdir(straindirpath+'/vol_0/'+i) and i[0] =='p':
          pertdirlist.append(straindirpath+'/vol_0/'+i)
    return pertdirlist

def GetQHPertDirList(voldirs):
    pertdirlist = []
    parentdir = os.getcwd()
    for i in voldirs:
      directory = parentdir + '/' + i
      files = os.listdir(directory)
      for j in files:
        if os.path.isdir(directory+'/'+j) and j[0] =='p':
          pertdirlist.append(directory+'/'+j)
    os.chdir(parentdir)
    return pertdirlist
```

ThermoMethods.py

```
"""
Methods for running thermodynamic calculations using VASP and ATAT
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.4$"
__date__ = "$Date: November 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

import os
import shutil
import gridengine
import SuperCell
import CheckJobs
from VASPUtils import *

def InitialRelaxation(params):
  #check for a vasp.in file
  if not os.path.isfile('vasp.in'):
    print 'No vasp.in file!  Aborting.'

  #check if this step has been done before
  done = CheckJobs.CheckRelax(os.getcwd())

  #if not done already then proceed with calculations
  if not done:
    print 'Initializing'
    os.system('ezvasp -n vasp.in')
    queuefile = gridengine.CreateQueueFile('InitialRelax', '20:00:00',
                                           params['relaxnodes'],
                                           params['QueueToUse'])
    JobIDs = []
    JobIDs.append(gridengine.SubmitJob(queuefile))
    gridengine.WaitForJobs(JobIDs)
    ArchiveFiles('relax')
  print 'Initial relaxation done.'

def InitialSSC(params):
  # Check if this step has been done before:
  # If CONTCAR and POSCAR are the same and E0 is in
  # OSZICAR and OSZICAR timestamp is newer than CONTCAR
  done = CheckJobs.CheckSSC(os.getcwd())
```

```
    #if not done already then proceed with this step
    if not done:
      print 'Performing SSC on initial structure.'
      shutil.copyfile('CONTCAR', 'POSCAR')
      ModifyInputFile('INCAR','NSW', 0)
      ModifyInputFile('INCAR','ISIF', 2)
      ModifyInputFile('INCAR','ISMEAR', -5)
      ModifyInputFile('INCAR','IBRION', -1)
      JobIDs = []
      queuefile = gridengine.CreateQueueFile('InitialSSC', '20:00:00',
                                             params['relaxnodes'],
                                             params['QueueToUse'])
      JobIDs.append(gridengine.SubmitJob(queuefile))
      gridengine.WaitForJobs(JobIDs)
      ArchiveFiles('static')
    print 'Initial SSC done.'

def SetUpVolumes(volumedirectories, posfitfccommand, negfitfccommand):
    # Check if this step has been done before
    # Go into each volume directory and make sure there is an INCAR file
    parentdirectory = os.getcwd()
    badvoldirs = []
    done = False
    for i in volumedirectories:
      thisvolume = parentdirectory+'/'+i
      if not os.path.isfile(thisvolume+'/str.out'):
        badvoldirs.append(thisvolume)
    if len(badvoldirs) == 0:
      done = True

    #if not done already then proceed with this step
    if not done:
      print 'Setting up volume directories'
      ArchiveFiles('static')
      for i in ['CONTCAR', 'OSZICAR', 'OUTCAR', 'INCAR']:
        if not os.path.isfile(i+'.static'):
          shutil.copyfile(i, i+'.static')
      CreateWrapperFiles()
      os.system('cp vasp.wrap.relax vasp.wrap')
      os.system('extract_vasp')
      shutil.copyfile('str_relax.out','str.out')
      os.system(posfitfccommand)
      os.system(negfitfccommand)
      os.system('rm -r vol_-0')
```

```
  os.chdir(parentdirectory)
  print 'Volume directories set up'

def RelaxVolumes(voldirs, params):
  parentdir = os.getcwd()
  badjobs = []
  volumeJobIDs = []
  for i in voldirs:
    directory = parentdir+'/'+i
    if not CheckJobs.CheckRelax(directory):
      badjobs.append(directory)
  if len(badjobs) > 0:
    print 'Relaxing volume directories'
  for j in badjobs:
    os.chdir(j)
    os.system('str2ezvasp')
    os.system('ezvasp -n vasp.in')
    queuefile = gridengine.CreateQueueFile('Vol-Relax', '20:00:00', 4,
                                           params['QueueToUse'])
    volumeJobIDs.append(gridengine.SubmitJob(queuefile))
  if len(volumeJobIDs) > 0:
    gridengine.WaitForJobs(volumeJobIDs,
                           sleeptime=params['RelaxPollTime'])
    ArchiveFiles('relax')
  os.chdir(parentdir)
  print 'Volume directories relaxed.'

def VolumesSSC(voldirs, params):
  badjobs = []
  JobIDs = []
  parentdir = os.getcwd()
  for i in voldirs:
    directory = parentdir+'/'+i
    if not CheckJobs.CheckSSC(directory):
      badjobs.append(directory)
  if len(badjobs) > 0:
    print 'Performing SSC calculations'
  for j in badjobs:
    os.chdir(j)
    os.system('cp CONTCAR POSCAR')
    ModifyInputFile('INCAR','NSW', 0)
    ModifyInputFile('INCAR','ISIF', 2)
    ModifyInputFile('INCAR','ISMEAR', -5)
    ModifyInputFile('INCAR','IBRION', -1)
    queuefile = gridengine.CreateQueueFile('Vol-SSC', '20:00:00', 4,
```

```
                                                    params['QueueToUse'])
      JobIDs.append(gridengine.SubmitJob(queuefile))
    if len(JobIDs) > 0:
      gridengine.WaitForJobs(JobIDs, sleeptime=params['RelaxPollTime'])
      ArchiveFiles('static')
    os.chdir(parentdir)
    print 'Volume SSC calculations done'

def VolumesPostProcess(voldirs, feleccommand):
    # Check by looking for str_relax.out and felec in each volume directory
    badjobs = []
    parentdir = os.getcwd()
    for i in voldirs:
      if not (CheckJobs.CheckFelec(i) and CheckJobs.CheckExtractVasp(i)):
        badjobs.append(parentdir+'/'+i)

    #if not done already then proceed with this step
    if len(badjobs) > 0:
      print 'Post processing volume directories'
      os.system('cp vasp.wrap.static vasp.wrap')
      for i in badjobs:
        print 'Processing %s'%(i)
        os.chdir(i)
        os.system('extract_vasp')
        os.system(feleccommand)
        ArchiveFiles('static')
    os.chdir(parentdir)
    print 'Volume directories processed and felec run sucessfully!'

def SetUpQHPerturbations(volumedirectories, posfitfccommand, negfitfccommand):
    # Check if this step has been done before
    # Go into each volume directory and make sure there is an INCAR file
    parentdirectory = os.getcwd()
    pertdirs = SuperCell.GetQHPertDirList(volumedirectories)
    done = False
    if len(pertdirs) > 0:
      done = True

    if not done:
      print 'Setting up perturbation directories'
      os.system('cp vasp.wrap.static vasp.wrap')
      os.system(posfitfccommand)
      os.system(negfitfccommand)
      os.system('rm -r vol_-0')
      os.system('foreachfile wait foreachfile wait str2ezvasp')
```

```
    os.system('foreachfile wait foreachfile wait ezvasp -n vasp.in')
  os.chdir(parentdirectory)
  print 'Perturbation directories set up'

def QHPerturbationCalculations(volumedirectories, nodes, params):
  parentdirectory = os.getcwd()
  JobIDs = []
  pertdirlist = SuperCell.GetQHPertDirList(volumedirectories)
  for directory in pertdirlist:
    if not CheckJobs.CheckSSC(directory):
      print directory
      os.chdir(directory)
      queuefile = gridengine.CreateQueueFile('QH-Pert', '1000:00:00',
                                             nodes, params['QueueToUse'])
      KpointsToGamma()
      JobIDs.append(gridengine.SubmitJob(queuefile))
  if len(JobIDs) > 0:
    print 'Performing Quasi-harmonic perturbation calculations'
    gridengine.WaitForJobs(JobIDs, sleeptime=params['PertPollTime'])
  os.chdir(parentdirectory)
  gridengine.ErrorCheck('pert', pertdirlist)
  print 'Quasi-harmonic perturbation calculations done'
  return pertdirlist

def QHExtractVasp(pertdirs):
  parentdir = os.getcwd()
  for i in pertdirs:
    if not os.path.isfile(i+'/str_relax.out'):
      os.chdir(i)
      os.system('extract_vasp')
  os.chdir(parentdir)
```

VASPUtils.py

```
"""
Various utilities for modifying VASP input files for ELC calculations
"""

__author__ = "Mike Williams (michaeleric.williams@gmail.com)"
__version__ = "0.4$"
__date__ = "$Date: November 2007 $"
__copyright__ = "Copyright (c) 2007 Mike Williams"
__license__ = "Python"

import numpy
```

```
import os
import shutil

def StrainLattice(strain, whichELC):
  """
  Distorts lattice vectors in the parent directory's CONTCAR file
  and writes them out as a POSCAR file in the current directory

  Requires: float of strain value
  Returns: nothing
  """

  inputfile = open("../CONTCAR", 'r')
  data = inputfile.readlines()
  inputfile.close()
  #pull out lattice vectors and put in matrix
  latticevectors = numpy.identity(3, float)
  a = data[2].split(" ")
  b = data[3].split(" ")
  c = data[4].split(" ")
  #eliminate null spaces
  for x in [a,b,c]:
    while '' in x:
      x.remove('')
  d = a+b+c
  #convert elements of list to matrix of floats (allows matrix operations)
  index = 0
  for x in range(3):
    for y in range(3):
      latticevectors[x,y] = float(d[index])
      index = index+1
  latticevectors = numpy.mat(latticevectors)
  #distort lattice vectors with matrix operations
  I = numpy.mat(numpy.identity(3))
  e = numpy.mat(numpy.zeros((3,3)))
  if whichELC == 'c11':
    e[0,0] = strain
    e[1,1] = -strain
    e[2,2] = strain*strain/(1-strain*strain)

  elif whichELC == 'c44':   #taken from Mehl (23) & (31)
    e[0,1] = strain/2
    e[1,0] = strain/2
    e[2,2] = strain**2./(4-strain**2.)
```

```
    B = I+e
    A= latticevectors * B

    #write distorted lattice vectors out to POSCAR file
    D = numpy.array(A)
    outputfile = open("POSCAR", 'w')
    #copy first 2 lines
    outputfile.writelines(data[0])
    outputfile.writelines(data[1])
    #write new lattice vectors
    for row in D:
      for item in row:
        outputfile.write(numpy.array2string(item))
        outputfile.write('\t')
      outputfile.write('\n')
    #write remaining data
    outputfile.writelines(data[5:])
    outputfile.close()

def KpointsToGamma():
    """
    Changes a KPOINTS file from Monkhost Pack scheme to Gamma Centered scheme

    Requires: nothing
    Returns: nothing
    """
    inputfile = open('KPOINTS', 'r')
    data = inputfile.readlines()
    inputfile.close()
    outputfile = open('KPOINTS', 'w')
    data[2] = 'Gamma\n'
    outputfile.writelines(data)

def ModifyInputFile(filename, parameter, newvalue):
    """
    Takes an input file and changes one of the input parameter's value

    Requires: a string - the parameter to change
        the new value - can be a string or integer
    Returns: Nothing
    """
    f = open(filename, 'r')
    data = f.readlines()
    f.close()
    f = open(filename, 'w')
```

```
    exists = 1
    for line in range(len(data)):
      if data[line].find(parameter) == 0:
        data[line] = parameter+' = '+str(newvalue)+'\n'
        exists = 0
    f.writelines(data)
    if exists == 1:
      f.close()
      f = open(filename, 'a')
      f.writelines(parameter+' = '+str(newvalue)+'\n')

def CreateWrapperFiles():
    """
    Creates the vasp.wrap, vasp.wrap.static and vasp.wrap.relax
    files for use by fitfc
    """
    shutil.copyfile('vasp.in', 'vasp.wrap.relax')
    shutil.copyfile('vasp.in', 'vasp.wrap.static')
    for i in ['vasp.wrap.relax', 'vasp.wrap.static']:
      inputfile = open(i, 'r')
      data = inputfile.readlines()
      inputfile.close()
      badlinenumber = data.index('[POSCAR]\n')
      data = data[:badlinenumber]
      outputfile = open(i, 'w')
      outputfile.writelines(data)
      outputfile.close()
    ModifyInputFile('vasp.wrap.relax', 'ISIF', 4)
    ModifyInputFile('vasp.wrap.static', 'ISIF', 2)
    ModifyInputFile('vasp.wrap.static', 'IBRION', -1)
    ModifyInputFile('vasp.wrap.static', 'NSW', 0)

def ArchiveFiles(relaxorstatic):
    """
    Copies 4 essential files to *.static or *.relax for archiving purposes
    """
    if relaxorstatic == 'relax':
      for i in ['CONTCAR', 'OSZICAR', 'OUTCAR', 'INCAR']:
        if not os.path.isfile(i+'.relax'):
          os.system('cp %s %s.relax'%(i, i))
    else:
      for i in ['CONTCAR', 'OSZICAR', 'OUTCAR', 'INCAR']:
        if not os.path.isfile(i+'.static'):
          os.system('cp %s %s.static'%(i, i))
```

VITA

Michael Eric Williams received his Bachelor of Science degree in mechanical engineering from Brigham Young University in April 2005 and the degree of Master of Science in mechanical engineering at Texas A&M University in College Station, TX in May 2008. While at Texas A&M he performed research in the field of computational materials science, specifically the prediction of finite temperature properties from ab initio techniques. He has accepted an offer for full time employment with Schlumberger to work as a mechanical engineer after graduation.

Mr. Williams may be reached at the Schlumberger Reservior Completion Center, P.O. Box 1590, Rosharon, TX 77583-1590 or via email at michaeleric.williams@gmail.com.