Phonons, Defects and the Thermal Conductivity of ZnO

Timothy Lehner

Royal Holloway, University of London

A Dissertation submitted to the University of London for the degree of Doctor of Philosophy

August 2019

Declaration of Authorship

I, Timothy Sean Lehner, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

_____

Signature

_____

Date

Abstract

ZnO is an important semiconductor with a wide range of applications. The high thermal stability, corrosion resistance, non-toxicity and abundance, coupled with excellent charge carrier transport, make it an attractive candidate for thermoelectric applications, particularly in reducing wasted heat energy in high temperatures processes. A combination of first-principles calculations using Density Functional Theory, large-facility neutron scattering experiments and in situ characterisation experiments were used to investigate the lattice dynamics, intrinsic defect structures and thermoelectric properties of ZnO. Calculated phonon modes are in excellent agreement with those directly measured using inelastic scattering. Powder inelastic neutron scattering measurements of bulk and nano-structured ZnO reveal the presence of anharmonic, multi-phonon scattering processes. A novel model for fitting this multi-phonon density of states is presented which relies only on the size of the nanocrystals. The calculated thermal conductivity is in excellent agreement with experimental data. Finally, the intrinsic defect structure was found to be 5% oxygen vacancies with hydrogen interstitials.

Acknowledgements

To my RHUL supervisor Professor J. P. Goff, thank you for your constant support, guidance and optimism throughout the PhD. I have thoroughly enjoyed working with you, Jon, you were a constant source of reassurance, laughs and sanity. To Uthay, thank you for all the sample preparation work, discussions, and phenomenal curries. Professor K. Refson, thank you for your advice and help understanding the world of DFT.

To those at Johnson Matthey: my supervisor Dr. R. Potter, thank you for your helpful discussions and support of these studies, I hope you have a long and happy retirement. To Chris, thank you for everything you did to help this project, it was a pleasure working in the lab with you.

To the instrument scientists at ISIS: David Voneshen, Duc Le, Helen Walker and Matthias Guttmann, thank you for your support during both the experiments and analysis. My fellow neutron scattering colleagues I met at the Oxford and ISIS schools, particularly Nicolo and Anthony who helped get me through the stressful moments on the beam, thank you.

My fellow PhD colleagues (there are many and I cannot list you all), but particularly: Toby, David, Sercan, Jimmy, Alex, Rupert, Stef, Jacob, Saeed and Giriz, you were wonderful officemates and I thank you for your interesting discussions, and importantly for tea at three.

To my climbing friends, particularly Mitch and Sam, thank you for getting me through the highs and lows.

To Arianwen, thank you for everything, I would never have achieved this without you.

To Debi, Holly and Angel, thank you for everything. I would not be who I am without you all.

To Owen and James, I love you guys, thank you for always being there. Everything I achieve is, in part, thanks to you two.

Finally I cannot forget my family, who put up with my grumblings and were there when I needed them most.

This thesis was funded by an EPSRC CASE grant, and Computing resources were provided by STFC Scientific Computing Department's SCARF cluster.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Zinc Oxide (ZnO) is a non-toxic, abundant semiconductor with a wide range of applications. It is of particular interest for optoelectronic applications due to its wide band gap, which has driven much research interest [1]. Other applications of, typically polycrystalline, ZnO are diverse and include piezoelectric transducers, varistors, phosphors, transparent conducting films and even facial powders [2].

Compared to similar wide-gap semiconductors, such as GaN, ZnO possesses a number of 'fundamental advantages' whilst having a comparable band-gap and crystallographic structure [3]. Attractive properties of ZnO include its corrosion and radiation resistance, high thermal stability and a wide range of growth methods allowing low manufacturing costs [1, 3]. Furthermore ZnO is highly tunable via chemical doping [4].

It is possible to produce large, single crystals of bulk ZnO [5], and these high quality samples are readily available from commercial suppliers. The high quality samples allow detailed characterisation using inelastic neutron spectroscopy (INS) and single-crystal diffraction.

1.1 Defects in ZnO

For semiconductor applications, particularly thermoelectric modules, it is important to be able to dope the semiconductor to be p- and n- type to fabricate p-n devices. It is difficult to dope ZnO p-type [6] which has been attributed to the nature of its intrinsic defects [7]. Studies of the structure of ZnO date back to 1935 [8], however investigation of the intrinsic defect structure is not as well understood and consists mostly theoretical work.

First-principles DFT calculations show oxygen vacancies that lead to a distortion of the lattice [9], which can be seen in Fig. 1.1. These calculations suggest oxygen vacancies as they have the lowest formation energy over a wide of fermi energies. The +1 charge state is unstable [9]. Furthermore, DFT calculations suggest the presence of hydrogen interstitials in large concentrations with unexpected consequences, as it behaves solely as a donor in this case [10].

It is possible to determine the defect structure using a combination of neutron and x-ray diffraction, and diffuse scattering of neutrons as has been performed previously for $Y_2Ti_2O_7$ and $Dy_2Ti_2O_7$ [11].

1.2 Thermoelectricity

The thermoelectric effect allows the direct conversion of heat to electricity, or vice versa. The thermoelectric effect describes the set of three distinct phenomena: the Seebeck, Peltier and Thomson effects. The Seebeck effect describes the electromotive field, $\vec{\boldsymbol{E}}_{\text{emf}}$, generated across a temperature gradient, ∇T as:

$$\vec{\boldsymbol{E}}_{\text{emf}} = -S\nabla T, \tag{1.1}$$

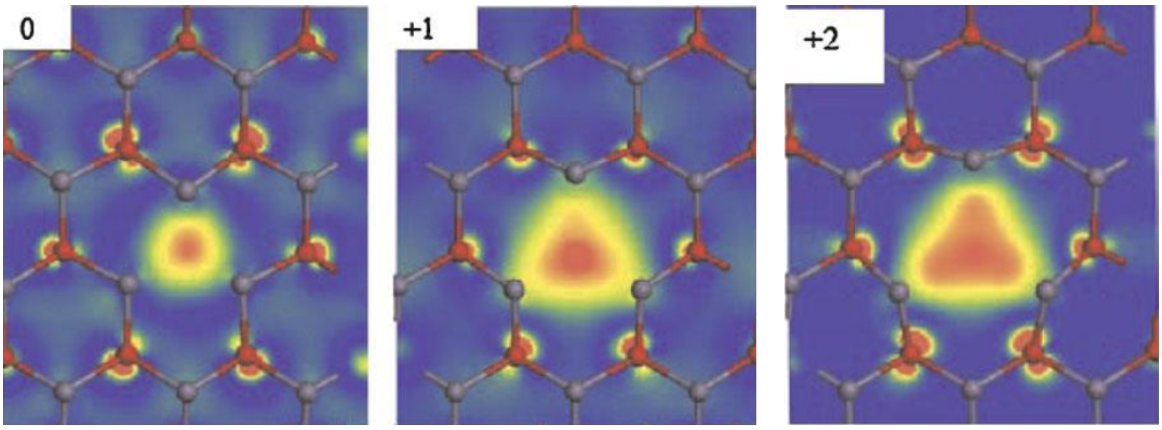


Figure 1.1: Intrinsic defects structures obtained from DFT calculations using the screened-exchange functional for an oxygen vacancy with 0, +1 and +2 charge states, reported in ZnO [9].

where S is the Seebeck coefficient. The Seebeck effect can be understood by considering the behaviour of charge carriers. In the presence of a temperature gradient, electrons in a material have a shorter mean free path in the higher temperature section as they are more energetic. These energetic electrons then diffuse to, and collect at, the cold side inducing an electric field as depicted in Fig. 1.2 [12].

Alternatively, the application of an electric field gives rise to a temperature gradient, known as the Peltier effect. A modest improvement in thermoelectric performance would allow commercially viable applications of these two effects. For example energy recovery applications, where losses due to waste heat can be salvaged such as in a car exhaust, as well as solid-state refrigeration, to cool hot spots in computer chips [13]. Global efforts to reduce carbon emissions has increased interest in methods to minimise energy waste, and $\frac{1}{6}$ of the energy used by UK industry is potentially recoverable [14].

The Thomson effect is outside the scope of this thesis.

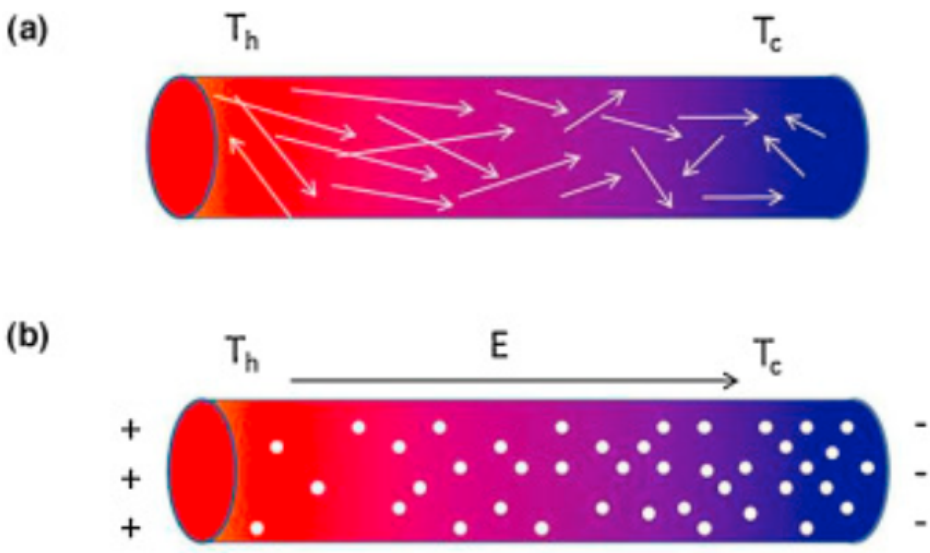


Figure 1.2: Schematic depiction of the thermoelectric effect [12]

1.2.1 Figure of merit

The efficiency of a thermoelectric is determined by the figure of merit, Z; the dimensionless quantity ZT is most often used, where:

$$ZT = \frac{\sigma S^2 T}{\kappa}, \tag{1.2}$$

S is the Seebeck coefficient, σ the electrical conductivity, κ the thermal conductivity and T the temperature. In order to create thermoelectrics with acceptable efficiencies, ZT values exceeding 2 are required [13].

From Eq. (1.2), ZT is proportional to the electrical conductivity, whilst inversely proportional to the thermal conductivity. This conflict is a difficult barrier to overcome, due to the Wiedemann–Franz law; typically materials that are good electrical conductors are also good thermal conductors, and materials that are poor thermal

conductors are also poor electrical conductors. The thermal conductivity consists of two contributions, one from the electrons κ_e and another from lattice dynamics, κ_l, such that $\kappa = \kappa_e + \kappa_l$. κ_e is strongly correlated with σ as shown in Fig. 1.3, however κ_l can be tuned independently of σ through phonon engineering.

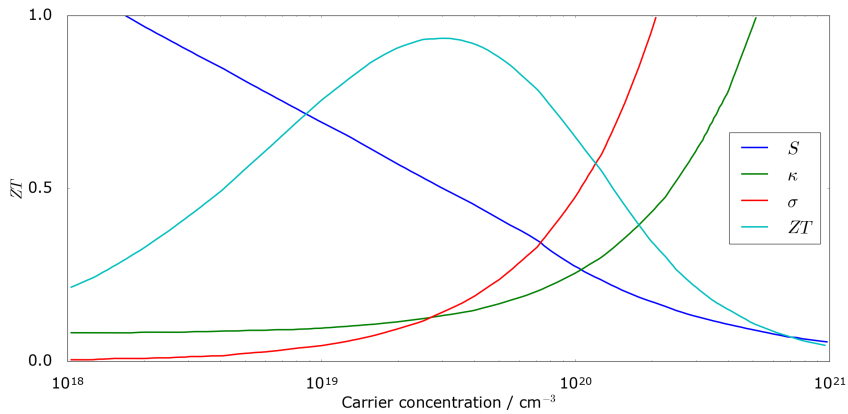


Figure 1.3: Carrier concentration dependence of: Seebeck coefficient (S, blue), thermal conductivity (κ, green) and electrical conductivity (σ, red). Optimising ZT (cyan) requires a compromise of all three parameters, S, κ and σ [15].

1.2.2 ZnO for Thermoelectric Applications

For a real-world device, both p- and n- type materials are required. For waste heat recovery applications, these materials need to be stable at high temperatures; a car exhaust operating temperature can exceed 1000 K [16].

There exist high-performance thermoelectrics such as PbTe, PbTeSe and Bi_2Te_3 with reported ZT as high as 1.75 [17]. The materials exhibit rapid degradation at higher temperatures, and require scarce or toxic materials such as Te and Pb, making them unsuitable for use in power recovery applications in vehicles.

ZnO is an attractive candidate due to the properties discussed earlier, as well as

its charge carrier transport properties, which are excellent for thermoelectric applications. Unfortunately the large thermal conductivity leads to a poor ZT value, making most practical applications unfeasible [1]. The thermal conductivity is dominated by lattice contributions; between 10 to 100 times larger than κ_e [18]. A combination of nanostructuring and Al-doping has been shown to suppress thermal conductivity by a factor 20 (see Fig. 1.4), whilst maintaining excellent electrical conductivity [19].

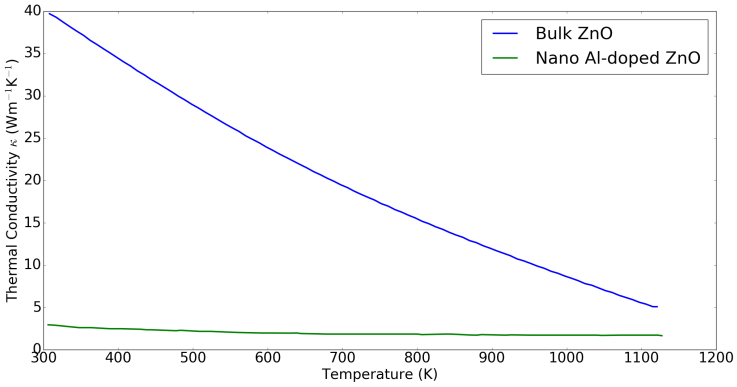


Figure 1.4: Nano-structuring + Al-doping ZnO reduces the thermal conductivity by a factor 20 at room temperature. The two samples have similar electrical conductivities due to Al-doping [19].

1.3 Synopsis

In this thesis the thermal conductivity of bulk and nanostructured ZnO is investigated using a combination of inelastic neutron scattering (INS) techniques and first-principles calculations. The intrinsic defect structure of ZnO is also studied using a combination of x-ray and neutron diffraction.

In the following chapter, the experimental techniques and background theory is described, with particular focus on time-of-flight neutron scattering. In Chapter 3,

first-principles DFT calculations of the lattice dynamics are benchmarked and validated against other calculations in the literature and, more importantly, single-crystal INS measurements of bulk ZnO performed at ISIS neutron source. Chapter 4 focuses on the effects of nanostructuring on the phonons in ZnO to determine the consequences on the thermal conductivities. In the final results chapter, Chapter 5, the intrinsic defect structure of ZnO is determined. Finally the last chapter contains the summarised main conclusions as well as a discussion on the future outlook.

Chapter 2

Experimental Techniques

In this chapter, the requisite scattering theory for this thesis is presented. The necessary language is introduced and a model describing elastic scattering from a macroscopic, ideal crystal built from considerations starting from a single atom is derived following Ref. [20] and [21]. The generalisation for inelastic scattering of this formalism is then presented without derivation, allowing calculation of $S(\vec{\boldsymbol{Q}}, \omega)$ from first principles.

Pulsed neutron spallation sources, and time-of-flight techniques for elastic diffraction and inelastic spectroscopy are discussed. Following this, the relevant technical information of the instruments used in this thesis: SXD; MARI; MERLIN; and LET, are shown.

Finally, the laser flash method and experimental apparatus used to measure the thermal conductivities of single-crystal samples are described.

2.1 Elastic Scattering

2.1.1 Fundamentals of a Scattering Experiment

Scattering of some particle such as a photon or neutron, by a sample, is characterized by both a change in momentum, $\vec{\boldsymbol{P}}$, and energy, E. A particle with incident wavevector, $\vec{\boldsymbol{k}}_i$, and angular frequency, ω_i, will have final momentum, $\vec{\boldsymbol{k}}_f$ and frequency, ω_f after a scattering event. The momentum transfer can then be expressed as:

$$\vec{\boldsymbol{P}} = \hbar(\vec{\boldsymbol{k}_f} - \vec{\boldsymbol{k}_i}) = \hbar\vec{\boldsymbol{Q}}, \tag{2.1}$$

where $\hbar$ is the reduced Planck constant and the wavevector transfer, $\vec{\boldsymbol{Q}}$, is, by convention, defined as

$$\vec{\boldsymbol{Q}} = \vec{\boldsymbol{k}}_f - \vec{\boldsymbol{k}}_i. \tag{2.2}$$

The energy transfer for neutrons is then

$$E = \frac{1}{2}mv^2 = \frac{\hbar^2}{2m}(k_f^2 - k_i^2). \tag{2.3}$$

It is helpful to start with the simple case by considering the special case of elastic scattering. In this case there is zero energy transfer, i.e. $\left|\vec{\boldsymbol{k}}_f\right| = \left|\vec{\boldsymbol{k}}_i\right| = 2\pi/\lambda$. In this case, the magnitude of Q can be found as:

$$Q = \frac{4\pi \sin\theta}{\lambda}, \tag{2.4}$$

through some simple trigonometry, as seen in Fig. 2.1.

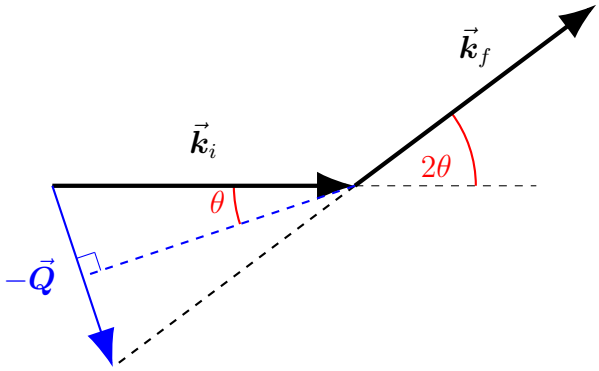


Figure 2.1: Schematic diagram for elastic scattering through an angle of 2θ. Since $\left|\vec{k}_i\right| = \left|\vec{k}_f\right|$, this forms an isosceles triangle. From this, Eq. (2.4) is found with straightforward trigonometry.

2.1.2 Scattering from a Single Atom

A steady stream of particles to be scattered, with wavelength λ travelling along $\vec{x}$, can be described as a plane wave

$$\vec{\psi} = \vec{\psi}_0 \mathrm{e}^{\mathrm{i}\vec{k}\cdot\vec{x}}, \tag{2.5}$$

with particle density $\left|\vec{\psi}\right|^2$, where $\vec{k} = \dfrac{2\pi}{\lambda}$. A fixed atom placed at the origin will then scatter incident particles along a displacement vector $\vec{r}$, The final wavefunction of the incident wave is then

$$\vec{\psi}_f = \vec{\psi}_0 f(\lambda, \theta) \frac{\mathrm{e}^{\mathrm{i}\vec{k}_f\cdot\vec{r}}}{r}, \tag{2.6}$$

where θ is the angle between $\vec{x}$ and $\vec{r}$, $f(\lambda, \theta)$ is the probability of incident wave being scattered in a certain direction, and $r = |\vec{r}|$. This is illustrated in Fig. 2.2.

The form of $f(\lambda, \theta)$ can be examined in two regimes: where λ is of similar size to the scatterer, and the case where λ is much larger. For the first case, $f(\lambda, \theta)$ will be maximum for $\theta = 0$ as all path lengths are the same, but as $\theta \to \pi$ this will decay to

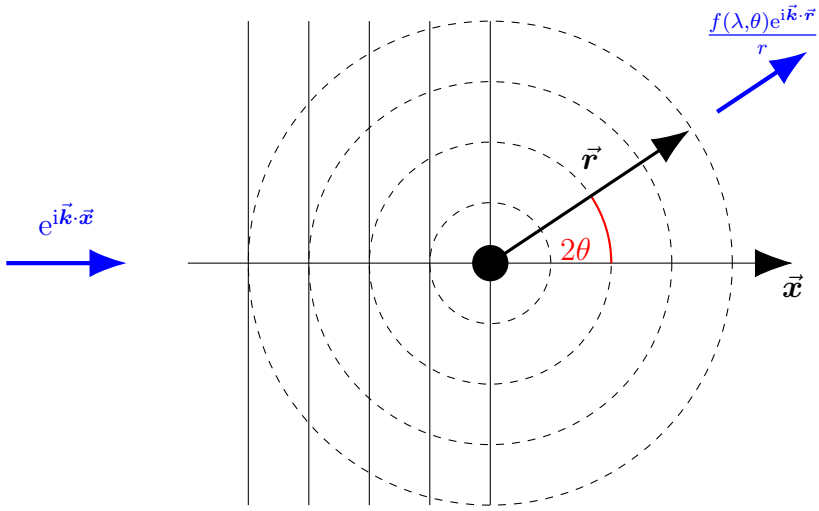


Figure 2.2: Idealised geometry of plane wave of particles scattered by a fixed atom at the origin.

a minimum, as path length differences between the front and back of the atom lead to interference. In the latter case, $f(\lambda, \theta) = b = \text{const}$; scattering from a single point is independent of θ.

This function, $f(\lambda, \theta)$, can be related to the scattering cross-section, σ. Consider a total scattering rate from an atom in all directions, R, due to an incident flux Φ. Since the scattering rate is the product of the incident flux and cross-sectional area, the cross-sectional area can be found as:

$$\sigma = \frac{R}{\Phi}. \tag{2.7}$$

From Eq. (2.6), the scattering rate can also be obtained as the integral over all possible angles:

$$R = \int_{2\theta=0}^{\pi} \int_{\phi=0}^{2\pi} |\psi_f|^2 dA = 2\pi\Phi \int_{2\theta=0}^{\pi} |f(\lambda, \theta)|^2 \sin 2\theta d2\theta, \tag{2.8}$$

where $dA = r^2 \sin 2\theta d\phi d2\theta$ is the area element of a spherical surface with radius r.

Hence, by combining Eqs. (2.7) and (2.8) shows σ is defined by $f(\lambda, \theta)$.

2.1.3 Differences in Scattering for Photons and Neutrons

The scattering theory covered so far has made little assumption about the types of particles being scattered. A range of particles can be used for scattering experiments, and in this thesis the focus will be on neutron scattering, with a brief discussion on x-ray scattering.

For neutron scattering the scattering probability is invariant with respect to θ and λ, i.e. $f(\lambda, \theta) = -b$, where b is called the scattering length, and negative by convention. This value is a constant due to the nature of interaction – the neutron interacts with the nucleus via the strong force. The nucleus, with typical radius 10^{-14} m, is point-like compared to the wavelength of neutrons used for probing atomic length scales, $\sim 10^{-10}$ m hence there is no angular dependence on the scattering.

For x-ray scattering, the photon interacts with the electrons in an atom through the electromagnetic force. The electron orbitals are of similar size to λ, hence the scattering probability $f(\lambda, \theta)$ will vary as discussed in Section 2.1.2, and also scale linearly with the number of electrons; this is called the x-ray form factor.

This discussion suggests the neutron case is simpler than that of x-rays, however this is incorrect. The neutron interaction is not well understood, and the scattering length b: varies for different isotopes or spin orientations; can be positive or negative; and appears random when compared with an atom's atomic number, whereas for x-rays it: decreases monotonically with increasing θ or decreasing λ; has the same sign for all elements; and is proportional to atomic number [20]. Finally, the neutron possesses a dipole moment, and scattering from magnetic moments can occur with a

similar angular dependence as x-rays, quantified by the magnetic form factor.

At this point it is helpful to consider the consequences of the neutron scattering length on the cross-section. The special case $f(\lambda, \theta) = -b$, allows simplification of σ obtained from Eq. (2.8) as:

$$\sigma = 4\pi |b|^2. \tag{2.9}$$

It turns out the scattering lengths of isotopes whose nuclei have non-zero spin is better described by an average value, $\langle b \rangle$, and standard deviation, Δb, as:

$$b = \langle b \rangle \pm \Delta b \rightarrow \left\langle b^2 \right\rangle = \langle b \rangle^2 + (\Delta b)^2, \tag{2.10}$$

such that the total scattering cross-section can be written as the sum of the coherent and incoherent components [20], i.e.

$$\langle \sigma_{\text{total}} \rangle = 4\pi \left\langle b^2 \right\rangle = \sigma_{\text{coh}} + \sigma_{\text{incoh}}, \tag{2.11}$$

$$\sigma_{\text{coh}} = 4\pi \langle b \rangle^2, \qquad \sigma_{\text{incoh}} = 4\pi (\Delta b)^2. \tag{2.12}$$

The incoherent cross-section gives rise to an additional flat background [22] and is typically a good indicator of hydrogen, which possesses an incoherent cross-section some 46 times larger than its coherent cross-section [23].

2.1.4 Scattering from Multiple Atoms

A real experiment will not involve scattering from a single atom, but instead a sample which can be described as an assembly of atoms. An incident beam, a complex plane wave with wavevector $\vec{\boldsymbol{k}}_i = (k, 0, 0)$, is scattered by a particular atom indexed by j at some arbitrary position $\vec{\boldsymbol{R}}_j$, as shown in Fig. 2.3. This will contribute to the total

scattered wave with some small change, $\delta\psi_f$:

$$[\delta\psi_f]_j = \psi_0 \mathrm{e}^{\mathrm{i}\vec{\boldsymbol{k}}_i \cdot R_j} f_j(\lambda, \theta) \frac{\mathrm{e}^{\mathrm{i}\vec{\boldsymbol{k}}_f \cdot (\vec{\boldsymbol{r}} - \vec{\boldsymbol{R}}_j)}}{\left|\vec{\boldsymbol{r}} - \vec{\boldsymbol{R}}_j\right|}, \tag{2.13}$$

which simplifies to Eq. (2.6) for $\vec{\boldsymbol{R}}_j = 0$.

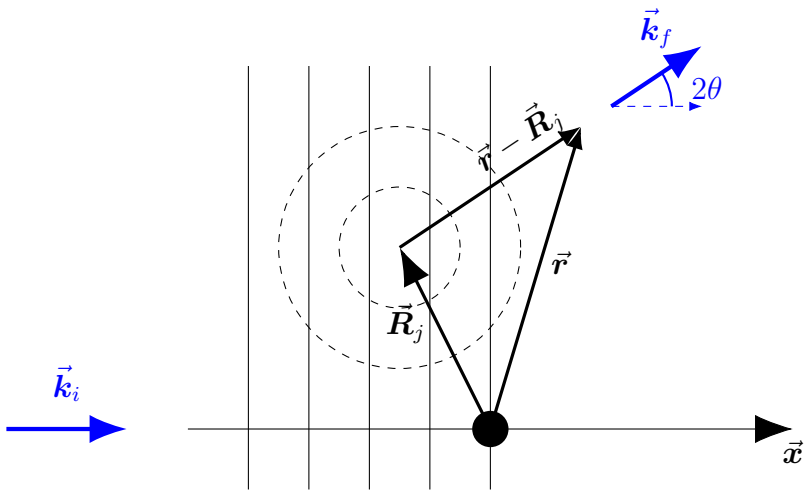


Figure 2.3: Idealised geometry of plane wave of particles scattered by an atom, j, at position $\vec{\boldsymbol{R}}_j$.

Hence for a large ensemble of atoms, the total, scattered wavefunction can be written as the sum of these contributions over the N atoms in the ensemble:

$$\psi_f = \sum_{j=1}^{N} [\delta\psi_f]_j \,. \tag{2.14}$$

Using the fact that detector distances are on a completely different length scales to inter-atomic distances, i.e. $\vec{\boldsymbol{r}} - \vec{\boldsymbol{R}}_j \simeq \vec{\boldsymbol{r}}$ allows simplifying Eq. (2.14). The probability

of observing a particle is given by the modulus square of the wavefunction, hence:

$$|\psi_f|^2 = \left| \frac{\psi_0}{r} \sum_{j=1}^{N} f_j(\lambda, Q) \mathrm{e}^{\mathrm{i}\vec{\boldsymbol{Q}}\cdot\vec{\boldsymbol{R}}_j} \right|^2, \tag{2.15}$$

where, in the case of neutron scattering, $f_j(\lambda, Q) = -b$.

2.1.5 Scattering in Macroscopic Crystals

A crystal is defined by a Bravais lattice, $\mathbf{B}$, and basis set, $\{(a_j, \vec{\boldsymbol{R}}_j)\}$ – the set of atoms of species a_j at positions $\vec{\boldsymbol{R}}_j$. The Bravais lattice can be written in terms of the three lattice vectors $\vec{\boldsymbol{a}}, \vec{\boldsymbol{b}}, \vec{\boldsymbol{c}}$ that define the parallelepiped known as the unit cell,

$$\mathbf{B} = (\vec{\boldsymbol{a}}, \vec{\boldsymbol{b}}, \vec{\boldsymbol{c}}), \tag{2.16}$$

It is convenient to express the atomic positions, $\vec{\boldsymbol{R}}_j$, as their component in terms of these lattice vectors, $(R_{j,a}\vec{\boldsymbol{a}} + R_{j,b}\vec{\boldsymbol{b}} + R_{j,c}\vec{\boldsymbol{c}})$.

With Eq. (2.15), it is possible to calculate the elastic scattering for an ensemble of atoms, however this involves summing over all atoms which, for macroscopic samples, is infeasible. The periodicity of the atomic positions allows simplification: at any position the scattering length density, $\beta(\vec{\boldsymbol{r}})$, can be expressed in terms of the unit cell, i.e.

$$\beta(\vec{\boldsymbol{r}}) \equiv \beta(\vec{\boldsymbol{r}} + n_1\vec{\boldsymbol{a}} + n_2\vec{\boldsymbol{b}} + n_3\vec{\boldsymbol{c}}), \tag{2.17}$$

where n_1, n_2, n_3 are integer. Hence Eq. (2.15) can be re-written as a sum over N_{cells}

unit cells with n atoms as:

$$|\psi_f|^2 = \left| \frac{\psi_0}{r} \sum_{k=1}^{N_{\text{cells}}} \sum_{j=1}^{n} f_j(\lambda, Q) \mathrm{e}^{\mathrm{i}\vec{\boldsymbol{Q}}\cdot(\vec{\boldsymbol{R}}_{\text{cell},k} + r_{j,a}\vec{\boldsymbol{a}} + r_{j,b}\vec{\boldsymbol{b}} + r_{j,c}\vec{\boldsymbol{c}})} \right|^2, \tag{2.18}$$

$$|\psi_f|^2 = S(\vec{\boldsymbol{Q}}) = \frac{\psi_0^2 N_{\text{cells}}^2}{r^2} \left| \sum_{j=1}^{n} f_j(\lambda, Q) \mathrm{e}^{\mathrm{i}\vec{\boldsymbol{Q}}\cdot(r_{j,a}\vec{\boldsymbol{a}} + r_{j,b}\vec{\boldsymbol{b}} + r_{j,c}\vec{\boldsymbol{c}})} \right|^2, \tag{2.19}$$

where in the second equation the sum over unit cells has been extracted as simply an additional factor N_{cells}^2. The exponential term will cancel out unless the terms over n_1, n_2 and n_3 sum constructively, or more concretely where $\vec{\boldsymbol{Q}}$ satisfies the equation [20]:

$$\vec{\boldsymbol{Q}} \cdot (n_1\vec{\boldsymbol{a}} + n_2\vec{\boldsymbol{b}} + n_3\vec{\boldsymbol{c}}) = \phi_0 + 2\pi n, \tag{2.20}$$

where n is an integer. At this point it is convenient to introduce the reciprocal lattice, a construction that generates $\vec{\boldsymbol{Q}}$ that satisfy Eq. (2.20). Consider a point defined by integer h, k and l:

$$\vec{\boldsymbol{Q}} = h\vec{\boldsymbol{a}^*} + k\vec{\boldsymbol{b}^*} + l\vec{\boldsymbol{c}^*}, \tag{2.21}$$

where $\vec{\boldsymbol{a}^*}, \vec{\boldsymbol{b}^*}, \vec{\boldsymbol{c}^*}$ are the reciprocal lattice vectors, related to $\vec{\boldsymbol{a}}, \vec{\boldsymbol{b}}, \vec{\boldsymbol{c}}$ by:

$$\vec{\boldsymbol{a}^*} = \frac{2\pi}{V}(\vec{\boldsymbol{b}} \times \vec{\boldsymbol{c}}), \qquad \vec{\boldsymbol{b}^*} = \frac{2\pi}{V}(\vec{\boldsymbol{c}} \times \vec{\boldsymbol{a}}), \qquad \vec{\boldsymbol{c}^*} = \frac{2\pi}{V}(\vec{\boldsymbol{a}} \times \vec{\boldsymbol{b}}), \tag{2.22}$$

where V is the unit cell volume, $V = \vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} \times \vec{\boldsymbol{c}}$. It is simple to see these vectors obey

$$\vec{\boldsymbol{v}}_i \cdot \vec{\boldsymbol{v}^*}_j = 2\pi\delta_{ij}, \tag{2.23}$$

where $\vec{\boldsymbol{v}}_1 = \vec{\boldsymbol{a}}, \vec{\boldsymbol{v}}_2 = \vec{\boldsymbol{b}}$ and $\vec{\boldsymbol{v}}_3 = \vec{\boldsymbol{c}}$, and similarly for the reciprocal vectors. δ_{ij} is the Kronecker delta function.

Hence, the elastic scattering for a crystalline sample is a set of sharp peaks located at well-defined positions; the reciprocal lattice. These are called Bragg peaks due to their relation to Bragg's law.

2.1.6 Bragg's Law

An alternative approach to understanding the elastic scattering in crystals is to consider the lattice as a three-dimensional diffraction grating. Planes of atoms with inter-planar spacing d lead to constructive interference when:

$$n\lambda = 2d\sin\theta. \tag{2.24}$$

This result is known as Bragg's law, and a schematic diagram which demonstrates the difference in path length is $2d\sin\theta$ can be seen in Fig. 2.4. It is helpful to relate this key equation to the scattering wavevector, $\vec{\boldsymbol{Q}}$:

$$|Q| = 2\left|\vec{\boldsymbol{k}}_i\right|\sin\theta = \frac{4\pi}{\lambda}\sin\theta = \frac{2\pi N}{d}. \tag{2.25}$$

The values h, k, l are related to these planes, referred to as the Miller indices [21].

This result can also be obtained by performing the Fourier transform of the Bravais lattice convolved with the basis set [24]. Thus it is possible to determine the static crystal structure by performing a Fourier transform on the scattered wavefunction. Unfortunately it is only possible to measure $|\psi_f|^2$, and this is referred to as the phase problem and discussed in some length in Refs. [20, 24].

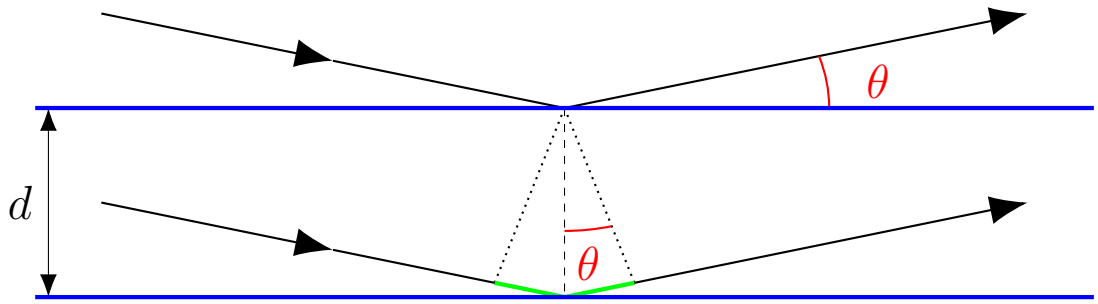


Figure 2.4: Bragg's setup showing reflections from uniformly spaced planes (blue) with inter-planar spacing d. The path difference between subsequent planes (green) is therefore $2d\sin\theta$

2.2 Inelastic Scattering

Considerations thus far have neglected to consider any time-dependence, such as the motion of atoms in solids. Nonetheless this provides an ability to achieve a considerable level of understanding of the static properties of solids [24]. Gaining any understanding of the thermodynamics of the crystal requires an analysis of the dynamics, which will be introduced in this section following Ref. [24].

The atoms in the lattice can be described as having an equilibrium position. A displacement of the j-th atom along a Cartesian direction denoted by α is given by $u_{j,\alpha}$. The energy can then be written as a Taylor expansion:

$$E = E_0 + \frac{1}{2}\sum_{\substack{j,j' \\ \alpha,\alpha'}} \frac{\partial^2 E}{\partial u_{\alpha,j}\partial u_{\alpha',j'}} u_{\alpha,j} u_{\alpha',j'} + ... \tag{2.26}$$

where the second term is the harmonic energy, and higher-order terms are neglected.

The scattering intensity obtained in Eq. (2.19) assumes scattering from pairs of atoms at the same time. A more general treatment takes into account the contributions to interference effects from components of the beam that are scattered at

different times, by including terms of the form:

$$b_j b_k \exp\{\mathrm{i}\vec{\boldsymbol{Q}} \cdot [\vec{\boldsymbol{r}}_j(t) - \vec{\boldsymbol{r}}_k(0)]\}. \tag{2.27}$$

In the case of inelastic scattering the neutron beam is subject to a change in energy, E, before and after scattering, by definition. The scattering function is therefore modified to be:

$$S(\vec{\boldsymbol{Q}}, \omega) = \sum_{i,j} b_i b_j \int \left\langle \exp\Big(\mathrm{i}\vec{\boldsymbol{Q}} \cdot [\vec{\boldsymbol{r}}_i(t) - \vec{\boldsymbol{r}}_j(0)]\Big) \right\rangle \exp(-\mathrm{i}\omega t) dt. \tag{2.28}$$

Let the instantaneous position of the i-th atom be

$$\vec{\boldsymbol{r}}_i(t) = \vec{\boldsymbol{R}}_i + \vec{\boldsymbol{u}}_i(t), \tag{2.29}$$

where $\vec{\boldsymbol{R}}_i$ is the average position and $\vec{\boldsymbol{u}}_i(t)$ the instantaneous displacement relative to the average position. For two variables whose distributions are characteristic of harmonic motion [24]:

$$\langle \exp(\mathrm{i}(X+Y)) \rangle = \exp\big(\langle (X+Y)^2 \rangle / 2\big), \tag{2.30}$$

such that

$$\begin{aligned} \left\langle \exp\Big(\mathrm{i}\vec{\boldsymbol{Q}} \cdot [\vec{\boldsymbol{u}}_i(t) - \vec{\boldsymbol{u}}_j(0)]\Big) \right\rangle = & \exp\Big(-\left\langle (\vec{\boldsymbol{Q}} \cdot \vec{\boldsymbol{u}}_i)^2/2 \right\rangle\Big) \times \exp\Big(-\left\langle (\vec{\boldsymbol{Q}} \cdot \vec{\boldsymbol{u}}_j)^2/2 \right\rangle\Big) \\ & \times \exp\Big(\left\langle [\vec{\boldsymbol{Q}} \cdot \vec{\boldsymbol{u}}_i(t)][\vec{\boldsymbol{Q}} \cdot \vec{\boldsymbol{u}}_i(0)] \right\rangle\Big), \end{aligned} \tag{2.31}$$

The first two terms are the so-called temperature factors obtained during structure

refinements for a diffraction experiment.

It can be shown that the one-phonon scattering function in the quantum mechanical limit is given by the sum over ν modes and j atoms, i.e. [24]:

$$S(\vec{\boldsymbol{Q}},\omega) = \frac{N\hbar}{2}\sum_{\nu}\frac{1}{\omega_\nu}\left|\sum_j \frac{b_j}{m_j^{1/2}}[\vec{\boldsymbol{Q}}\cdot\vec{\boldsymbol{e}}_j(\vec{\boldsymbol{k}},\nu)]\exp\Big(\mathrm{i}\vec{\boldsymbol{Q}}\cdot\vec{\boldsymbol{r}}_j\Big)T_j(\vec{\boldsymbol{Q}})\right|^2 \times ([n(\omega,T)+1]\delta(E+\hbar\omega_\nu)+n(\omega,T)\delta(E-\hbar\omega_\nu)), \tag{2.32}$$

where ω_ν is the frequency of the ν-th mode, b_j and m_j the neutron scattering length and atomic mass of the j-th atom, $\vec{\boldsymbol{e}}_j(\vec{\boldsymbol{k}},\nu)$ the eigenvector of the ν-th phonon mode at reduced wavevector $\vec{\boldsymbol{k}}$. $T_j(\vec{\boldsymbol{Q}})$ are the temperature factors seen above, and $n(\omega,T)$ the Bose factor which describes the occupation of phonon modes at the given energy, E. The Dirac delta exists to simply ensure scattering for a mode occurs only at the allowed energies where there exists a mode. This interaction can either leave the neutron with less, or more if modes are populated, energy, called the neutron loss and gain interactions.

The tools required for understanding elastic and inelastic scattering events have now been presented, and so the discussion turns to the neutron source used and practical considerations for performing these experiments.

2.3 Neutron Time-of-Flight measurements

It turns out neutrons with wavelengths well-suited to scattering experiments also have energies of similar scales to the phonon modes, making them a particularly powerful tool for investigating the dynamics of materials. In this thesis, neutron scattering measurements were performed at the ISIS neutron source in the UK, a

pulsed spallation source. Protons are accelerated to energies of 800 MeV and made to collide with a target made from tungsten clad in tantalum. These collisions produce a large number of neutrons as the excited tantalum nuclei release energy in order to return to their ground state. Collisions happen with a frequency of 50 Hz, hence this is referred to as a pulsed source. At ISIS, once the protons are of sufficient energy they are directed towards one of two targets, target station 1 (TS1) and 2 (TS2). 1 in 5 pulses are sent to TS2 and the rest sent to TS1.

Since neutrons are produced in pulses at a well-defined time, it is possible to employ the mass of the neutron to perform time-of-flight measurements. Neutrons produced have a distribution of energies, and this is reflected in their speed since, unlike x-rays, the neutron has a non-zero mass. Since the time of creation, flight path, and the time detected are all known, each individual neutron's energy can be calculated, which allows simultaneous measurement of a wide range of wavelengths for diffraction.

Since the speed of the neutrons is related to their energy, the use of choppers allows experimental setups where either E_i or E_f is fixed. The time-of-flight technique is then used to determine the change in energy and hence the inelastic scattering can be measured. A schematic of a chopper selecting an incident energy can be seen in Fig. 2.5 [25].

To obtain neutrons that are useful for scattering as described in the previous section, it is crucial that the neutron wavelengths and energies are well-suited to studying length scales of the order Å. The neutrons produced are of too high energy, and are first slowed using a moderator. Moderators used were ambient water at 300 K producing thermal neutrons for measurements on MERLIN and SXD, liquid methane at 100 K on MARI and liquid hydrogen 20 K providing cold neutrons on LET. These

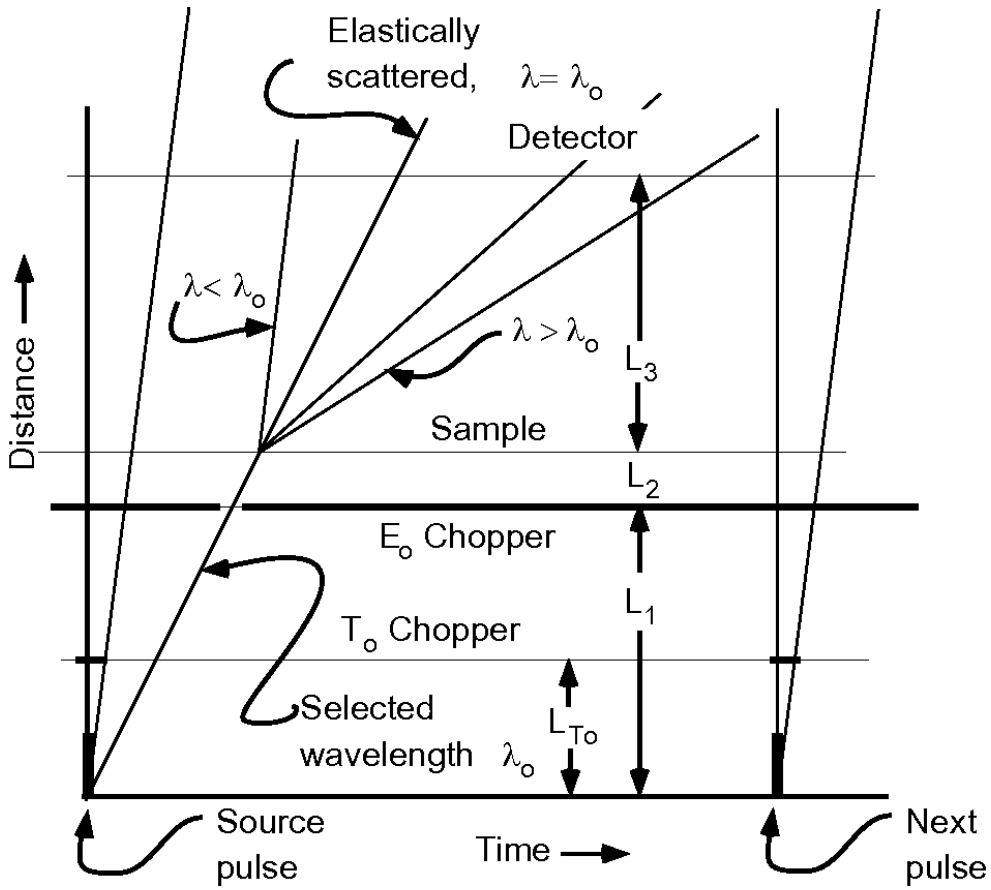


Figure 2.5: Selection of incident energy using choppers. Only neutrons of a specific speed are selected by the E_0 chopper. After scattering from the sample, the energy change can be determined based on the time-of-flight. [25]

machines will be discussed in more details later.

2.3.1 Time-of-Flight Diffractometers

As when introducing scattering theory, it is helpful to begin with the simpler case of elastic scattering on diffractometers. Recall the Bragg condition, $n\lambda = 2d\sin\theta$, which shows where Bragg peaks can be found. In a typical x-ray diffraction experiment, a monochromator is used to set λ, and then a detector measures a range of 2θ to determine Bragg peak positions.

With a pulsed neutron diffractometer the neutron time-of-flight can be exploited to simultaneously measure many different λ and hence, access large, three-dimensional volumes of reciprocal space. The sample is bathed in the neutron white beam, which is a distribution of neutron energies, and hence wavelengths. When a neutron reaches a detector, the raw data is the position and time-of-flight. From this it is possible to map each detected neutron to its wavelength and hence, the intensity measured at a specific $\vec{\boldsymbol{Q}}$.

This assumes the only scattering processes involved are elastic, which is not always a safe assumption. Exactly how measured data is mapped to reciprocal space, and the limitations of these assumptions, are discussed in greater detail in Section 5.2.

SXD technical information

A comprehensive list of the relevant technical information can be found in [26]. The key points are that neutrons with incident wavelength 0.2 Å to 10 Å are scattered and measured by eleven 64×64 pixel detectors, spanning an active area of 192×192 mm^2 with resolution 3×3 mm^2. A picture of the detectors can be seen in Fig. 2.6.

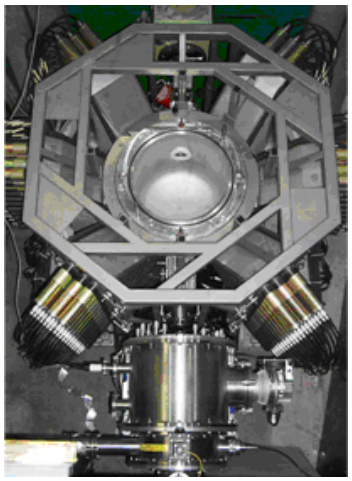


Figure 2.6: The detectors on SXD [27].

2.3.2 Time-of-Flight Spectrometers

As previously mentioned, if the flightpath and either E_i or E_f are known, the others can be deduced from the time-of-flight. In general, this is achieved either by fixing E_i (direct geometry), or E_f (indirect geometry), however this thesis focuses on the direct geometry case. For indirect scattering, the sample is bathed in white-beam and a monochromator backscatters neutrons with E_f into the detectors.

For the direct geometry case, incident energies are selected using a number of choppers, and in this thesis two types of choppers were used: disk and Fermi. The disk choppers are effectively a circular sheet of neutron absorbing material with a hole, which is rotated such that only neutrons of a specific velocity can pass. Multiple disk choppers are required to produce a monochromated beam, as only one would require a prohibitively large angular velocity.

More complicated arrangements of disk choppers can be used to slice each pulse of neutrons into a number of well-separated bunches. Thus it is possible to perform measurements for multiple E_i simultaneously, with the caveat that the selection of

these E_i is somewhat limited; it would not be possible to measure two very similar incident energies simultaneously, for example, as slicing the pulse into distinct bunches would not be possible [28]. This can be understood by looking at the distance-time diagram, shown in Fig. 2.7, detailing the role of 5 choppers on LET.

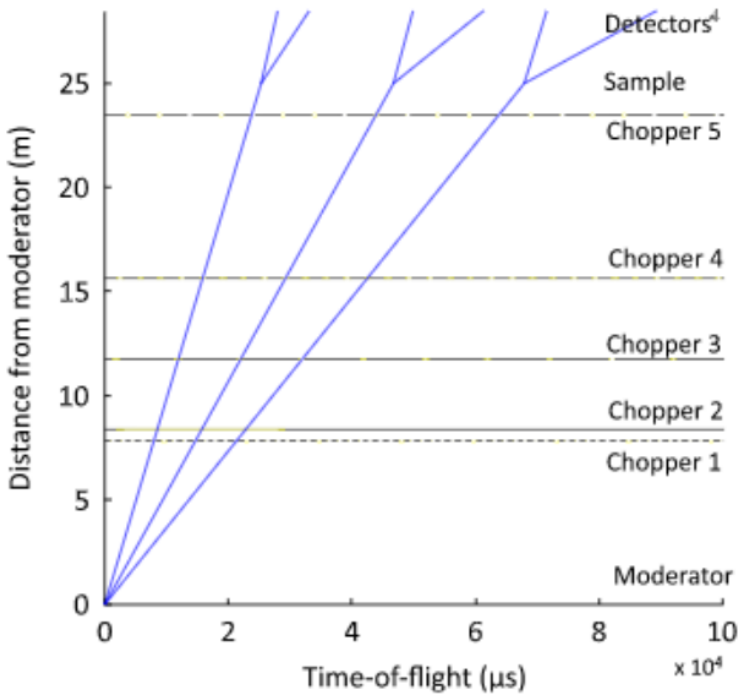


Figure 2.7: Distance-time diagram showing three separate E_i measured simultaneously, due to the many choppers that allow careful slicing of the neutron pulse into 3 bunches with $E_i = 5, 1.5$ and $0.7\,\text{meV}$ [28].

Finally, Fermi choppers are a cylindrical drum with curved slots running across its diameter [29]. With this design, the neutron absorbing section of the chopper is a much larger volume than when using disk choppers, which gives very low background.

A comprehensive list of the relevant technical information for the spectrometers MARI, MERLIN and LET can be found in [30, 31, 32]. The key points relevant to this thesis are discussed below.

MARI

Optimised for polycrystalline and powder measurements, neutrons with incident energies ranging 7 meV to 1000 meV are scattered into the low- and high-angle detector banks. The low-angle bank comprises an eight fold array of ^{3}He detectors covering $3-13°$. Unlike the other spectrometers, the detectors are not position sensitive hence the suitability of this instrument for powdered samples, as it is only possible to measure $|Q|$. The high-angle bank has detectors covering $12-135°$. For all measurements, the "s" Fermi chopper was used, which provides $\Delta E/E_i$ between 3 % to 8 % [30]. A schematic of MARI can be seen in Fig. 2.8

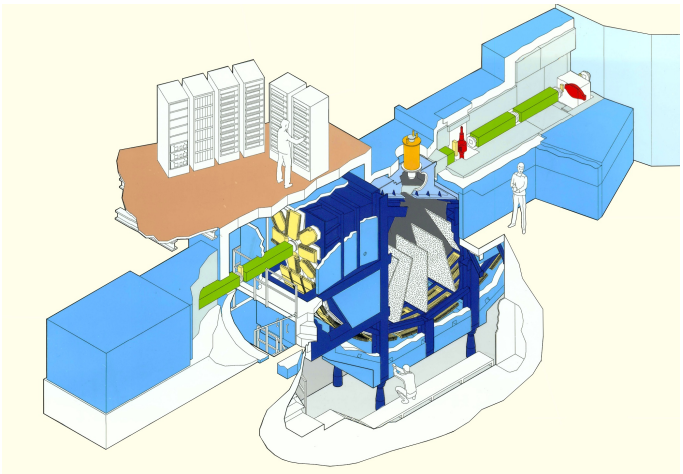


Figure 2.8: Schematic diagram of the MARI spectrometer [33].

MERLIN

Neutrons with incident energies 7 meV to 2000 meV are scattered into a detector bank of 3 m-tubes, position sensitive ^{3}He detectors (PSD), spanning a huge $-45-135°$ horizontal angle and $\pm 30°$ vertical angle, with energy resolutions of $\Delta E/E_i$ between 4 % to 7 % at the elastic line [31]. A schematic of MERLIN can be seen in Fig. 2.9

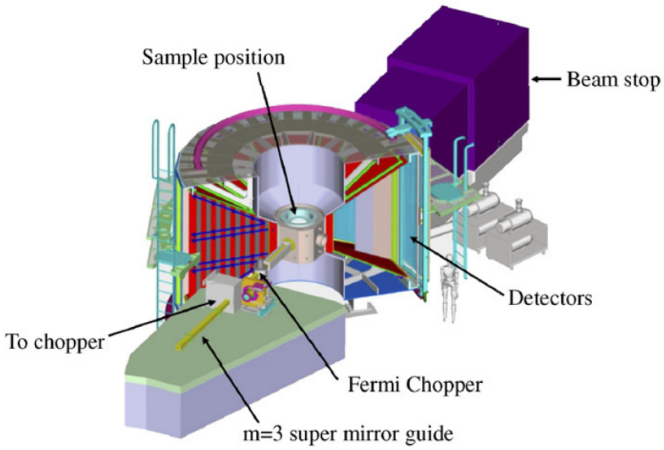


Figure 2.9: Schematic diagram of the MERLIN spectrometer [34].

LET

The cold neutrons provided on LET have incident energies ranging 0.5 meV to 30 meV. The detectors are similar to those on MERLIN, this time using PSD of 4 m and spanning $-40 - 140°$ horizontal angle and $\pm 30°$ vertical angle [32]. In the case of LET, great care has been taken to ensure gaps between detectors are kept to an absolute minimum [28]. A schematic of LET can be seen in Fig. 2.10

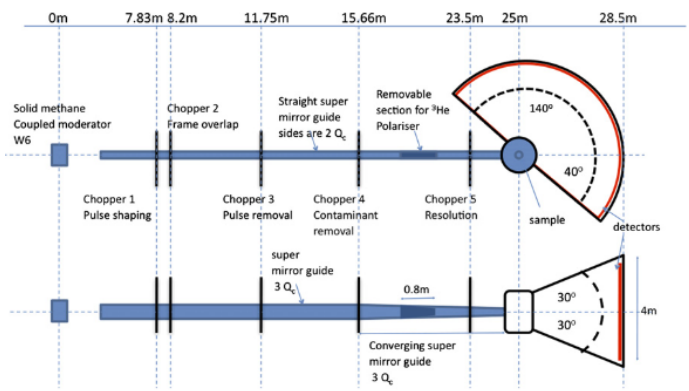


Figure 2.10: Schematic diagram of the LET spectrometer and series of choppers [28].

For all measurements performed, except the powder samples on MARI, samples are mounted on a rod which can be rotated through 360°. Vanadium measurements are performed to allow normalising the multiple detectors, as this scatters neutrons uniformly in all directions due to the Vanadium's near-zero coherent scattering length. Background measurements with identical sample environment (e.g. furnace, CCR) including sample mount were performed, and background subtractions performed in the time-of-flight domain before reducing the data to $S(\vec{\boldsymbol{Q}}, \omega)$.

The alignment of single crystal measurements was checked by visualising the measured elastic scattering. This shows missing Bragg peaks in symmetrically equivalent regions of reciprocal space if there is a small in-plane misalignment. By integrating over specific Bragg peaks it is possible to calculate the transformation matrix required to correct the nominally aligned $\vec{\boldsymbol{u}}$ and $\vec{\boldsymbol{v}}$. This was performed using the Horace software to obtain the true values of $\vec{\boldsymbol{u}}$ and $\vec{\boldsymbol{v}}$.

2.4 Thermal Conductivity – Laser Flash Method

Thermal diffusivity can be directly measured using the Laser flash method. The substrate is mounted horizontally in a furnace and one side irradiated by an energy pulse provided by the flash lamp. This pulse induces a homogenous rise in the temperature at the alternate side of the sample, which can be measured using a high-speed IR detector. A schematic diagram of the apparatus can be seen in Fig. 2.11.

The thermal diffusivity, heat capacity and, hence, thermal conductivity, can then be computed by the measured temperature rise as a function of time. The furnace allows measurements over a range of temperatures. A reference standard is used to calibrate measurements.

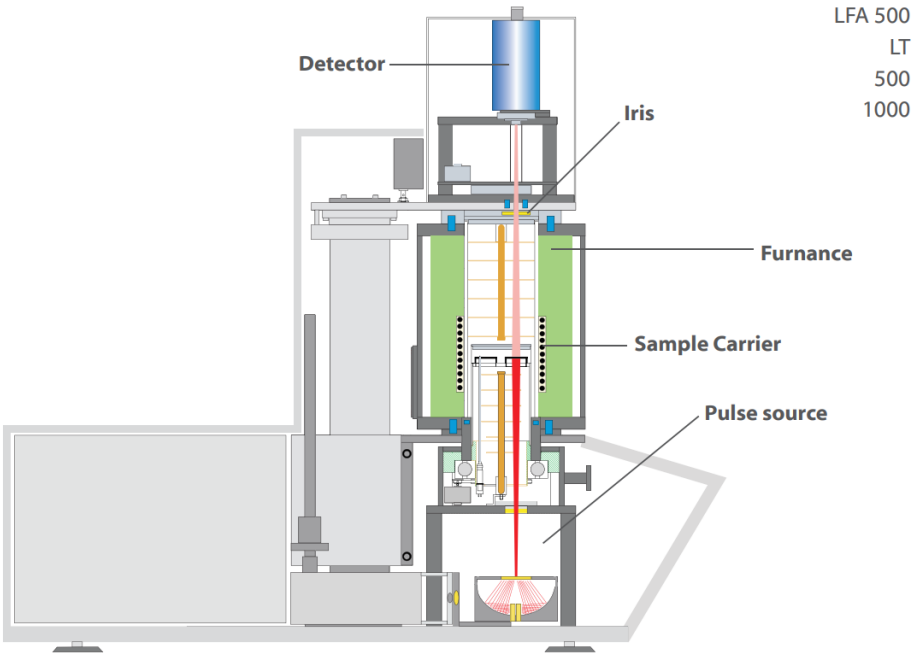


Figure 2.11: Schematic diagram of the Xenon LFA 500 apparatus used to measure thermal conductivities. The sample is placed in the sample carrier inside the furnace. The xenon lamp provides an energy pulse which induces heating in the sample. The IR radiation emitted from the opposing side of the substrate is measured by the high-speed detector using the focusing iris. [35]

For these measurements the Linseis LFA 500 was used, which provides a Xenon lamp pulse source and temperature range of −50 °C to 500 °C. A calibration reference with the same dimensions as the substrates, made from the alloy Inconel 600, was used along with a graphite sample holder. Crystals were made rough with a coarse sandpaper, and then coated in a thin layer of graphite, to minimise reflections of the incident xenon flash.

The measured data was analysed by Chris Nuttall at JM, and heat capacities and thermal conductivities extracted. For each temperature, multiple measurements were taken and reported values are the average with standard deviations.

Chapter 3

Phonon Dispersion of ZnO

3.1 Modelling

The thermal conductivity of ZnO is dominated by its large lattice contribution [36]. First-principles calculations allow modelling the phonon modes. A detailed derivation of density functional theory (DFT) is outside the scope of this thesis, nonetheless this section aims to give a brief introduction. Following this the model used to calculate phonons in ZnO is presented. Finally, initial benchmarks against other models in the literature are shown.

3.1.1 Brief Introduction to Density Functional Theory

The Hamiltonian for a system of M nuclei with atomic number Z_i at positions $\vec{\boldsymbol{R}}_i$, and N electrons at positions $\vec{\boldsymbol{r}}_i$, can be found by solving the many-body Schrödinger equation:

$$\begin{aligned}\hat{\mathcal{H}} = &-\sum_{i=1}^{N}\frac{\nabla_i^2}{2}+\frac{1}{2}\sum_{i\neq j}^{N}\frac{1}{|\vec{\boldsymbol{r}}_i-\vec{\boldsymbol{r}}_j|}\\ &-\sum_{I=1}^{M}\frac{\nabla_I^2}{2}+\frac{1}{2}\sum_{I\neq J}^{M}\frac{Z_I Z_J}{\left|\vec{\boldsymbol{R}}_I-\vec{\boldsymbol{R}}_J\right|}-\frac{1}{2}\sum_{i}^{N}\sum_{I}^{M}\frac{Z_I}{\left|\vec{\boldsymbol{r}}_i-\vec{\boldsymbol{R}}_I\right|},\end{aligned} \tag{3.1}$$

using Hartree atomic units. Solutions to this exist in a $3(N+M)$ dimensional Hilbert space, which leads to equations that quickly become intractable. Using the Born-Oppenheimer approximation [37], wherein the electronic and nuclear degrees of freedom of the wavefunction are separable, an equivalent problem to Eq. (3.1) can be written as:

$$\hat{\mathcal{H}} = -\sum_{i=1}^{N}\frac{\nabla_i^2}{2}+\frac{1}{2}\sum_{i\neq j}^{N}\frac{1}{|\vec{\boldsymbol{r}}_i-\vec{\boldsymbol{r}}_j|}-\sum_{i}^{N}V_n(\vec{\boldsymbol{r}}_i), \tag{3.2}$$

for a given nuclear configuration. Here V_n is the external potential the electrons feel due to the nuclei. Additional terms for the nuclei-nuclei interaction and kinetic energy of the nuclei would just change E by a constant, for a given equilibrium nuclei of position. The nuclei are no longer variables but instead parameters that define the potential.

It is convenient to write Eq. (3.2) as $\hat{\mathcal{H}} = \hat{F}+\hat{V}_n$, as $\hat{F}$ is the same for all N-electron systems. Ground state solutions are therefore determined by N and $V_n(\vec{\boldsymbol{r}})$.

Hohenberg and Kohn showed that the external potential is uniquely determined by the ground-state electronic density [38], which can be proved by contradiction starting by assuming there exists some potential $V'_{ext}(\vec{\boldsymbol{r}})$ with ground state $|\Psi'_0\rangle$ that gives rise to the same density $n(\vec{\boldsymbol{r}})$ [39].

As a consequence every electron density that is a ground state density, defines a functional $F[n] = \langle\Psi|\hat{F}|\Psi\rangle$ since $n(\vec{\boldsymbol{r}})$ defines both $N = \int d\vec{\boldsymbol{r}} n_0(\vec{\boldsymbol{r}})$ and the external potential. Thus there exists some functional of the electron density to obtain the energies:

$$E[n] = F[n] + \int d\vec{\boldsymbol{r}} V(\vec{\boldsymbol{r}}) n(\vec{\boldsymbol{r}}), \tag{3.3}$$

where $V(\vec{\boldsymbol{r}})$ is some arbitrary external potential. $E[n] \geq E_0$ from the variational principle [39].

Solving the Schrödinger equation can, therefore, be reframed as minimising the functional $E[n]$ with respect to the potential-generating densities.

The Kohn-Sham equations

The Kohn-Sham equations describe a fictitious system of non-interacting 'electrons':

$$\left[-\frac{1}{2}\nabla^2 + V_n(\vec{\boldsymbol{r}}) + V_H(\vec{\boldsymbol{r}}) + V_{XC}(\vec{\boldsymbol{r}})\right] \phi_i(\vec{\boldsymbol{r}}) = \epsilon_i \phi_i(\vec{\boldsymbol{r}}) \tag{3.4}$$

where:

$$V_n(\vec{\boldsymbol{r}}) = -\sum_I^M \frac{Z_I}{\left|\vec{\boldsymbol{r}} - \vec{\boldsymbol{R}}_I\right|}, \tag{3.5}$$

$$\nabla^2 V_H(\vec{\boldsymbol{r}}) = -4\pi n(\vec{\boldsymbol{r}}) \tag{3.6}$$

$$V_{XC}(\vec{\boldsymbol{r}}) = -\frac{\partial E_{XC}[n]}{\partial n}(\vec{\boldsymbol{r}}), \tag{3.7}$$

and

$$n(\vec{\boldsymbol{r}}) = \sum_i^N |\phi_i(\vec{\boldsymbol{r}})|^2. \tag{3.8}$$

The exchange-correlation term, E_{XC}, represents all of the change in energy due to electron interactions not accounted for by the Hartree potential and nuclei potential.

Since the potentials depend on $n(\vec{\boldsymbol{r}})$, which depends on $\phi(\vec{\boldsymbol{r}})$, Eq. (3.4) can be solved using a self-consistent method. An initial guess at the wavefunction $\phi(\vec{\boldsymbol{r}})$ is selected. $n(\vec{\boldsymbol{r}})$ is then computed and Eq. (3.4) is then solved for $\phi'(\vec{\boldsymbol{r}})$. Finally $\phi'(\vec{\boldsymbol{r}})$ is compared to $\phi(\vec{\boldsymbol{r}})$ and the process is iterated until convergence is achieved.

It can be shown that there exists an exchange-correlation functional, however unfortunately it is unknown [40]. A great deal of work has gone into developing approximate forms of this functional, some highly tailored to a specific system, others more suitable for a wide range of systems [41].

3.1.2 Phonon Mode Calculations

With the set of solvable equations obtained in Section 3.1.1, a large number of further calculations are possible. For example the forces on atoms can be calculated by exploring the energy landscape in response to some displacement, $\vec{\boldsymbol{u}}_{j,\alpha}$.

$$\vec{\boldsymbol{F}}_{j,\alpha} = -\frac{\partial E}{\partial \vec{\boldsymbol{u}}_{j,\alpha}}, \tag{3.9}$$

where j labels the atom and $\alpha \in \{\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}, \vec{\boldsymbol{z}}\}$ denotes one of three Cartesian directions. Nuclear positions can then be tweaked to minimise these forces, a process known as geometry optimisation.

To calculate lattice dynamics, the harmonic approximation is used. A Taylor expansion of the total energy yields:

$$E = E_0 + \sum_{j,\alpha} \frac{\partial E}{\partial \vec{\boldsymbol{u}}_{j,\alpha}} \cdot \vec{\boldsymbol{u}}_{j,\alpha} + \frac{1}{2} \sum_{j,\alpha,j',\alpha'} \vec{\boldsymbol{u}}_{j,\alpha} \cdot \Phi^{j,j'}_{\alpha,\alpha'} \cdot \vec{\boldsymbol{u}}_{j',\alpha'} + ..., \tag{3.10}$$

where $\vec{\boldsymbol{u}}_{j,\alpha}$ is a vector of displacements from equilibrium. Since the system is in

equilibrium, the second term is zero.

Assuming a plane-wave displacement from a phonon of wavevector $\vec{\boldsymbol{q}}$ and polarization vector $\vec{\boldsymbol{e}}_{j,\alpha}(\vec{\boldsymbol{k}},\nu)$:

$$\vec{\boldsymbol{u}}_{j,\alpha} = \vec{\boldsymbol{e}}_{j,\alpha}(\vec{\boldsymbol{k}},\nu)e^{i\vec{\boldsymbol{q}}\cdot\vec{\boldsymbol{r}}_{j,\alpha}-\omega_m t}, \tag{3.11}$$

yields the eigenvalue equation:

$$D^{j,j'}_{\alpha,\alpha'}(\vec{\boldsymbol{q}})\ \vec{\boldsymbol{e}}_{j,\alpha}(\vec{\boldsymbol{k}},\nu) = \omega^2_{m,\vec{\boldsymbol{q}}}\ \vec{\boldsymbol{e}}_{j,\alpha}(\vec{\boldsymbol{k}},\nu). \tag{3.12}$$

Solving the Kohn-Sham equations yields ground state energies, from which the Dynamical Matrix, $\vec{\boldsymbol{D}}$, can be obtained [42] using second order derivatives of the total energy, E, with respect to two atomic displacements, $\vec{\boldsymbol{u}}$, i.e.

$$\Phi^{j,j'}_{\alpha,\alpha'} = \frac{\partial^2 E}{\partial\vec{\boldsymbol{u}}_{j,\alpha}\partial\vec{\boldsymbol{u}}_{j',\alpha'}}, \tag{3.13a}$$

$$D^{j,j'}_{\alpha,\alpha'}(\vec{\boldsymbol{q}}) = \frac{1}{\sqrt{M_j M_{j'}}}\sum_j \Phi^{j,j'}_{\alpha,\alpha'}e^{-i\vec{\boldsymbol{q}}\cdot\vec{\boldsymbol{r}}_j}, \tag{3.13b}$$

where $\vec{\boldsymbol{R}}$ is the position vector and M_j the atomic mass.

The Hellmann–Feynman theorem makes calculating the first order derivatives of the total energy a quick calculation [43]. More concretely, for a given displacement λ the eigenvalue equation is:

$$\hat{H}_\lambda\left|\psi_\lambda\right\rangle = E_\lambda\left|\psi_\lambda\right\rangle. \tag{3.14a}$$

Left multiplying by $\langle\psi_\lambda|$ and differentiating yields:

$$\begin{aligned}\frac{dE_\lambda}{d\lambda} = \frac{d}{d\lambda}\left\langle\psi_\lambda\middle|\hat{H}_\lambda\middle|\psi_\lambda\right\rangle &= \left\langle\frac{d\psi_\lambda}{d\lambda}\middle|\hat{H}_\lambda\middle|\psi_\lambda\right\rangle + \left\langle\psi_\lambda\middle|\hat{H}_\lambda\middle|\frac{d\psi_\lambda}{d\lambda}\right\rangle + \left\langle\psi_\lambda\middle|\frac{d\hat{H}_\lambda}{d\lambda}\middle|\psi_\lambda\right\rangle \\ &= E_\lambda\frac{d}{d\lambda}\langle\psi_\lambda|\psi_\lambda\rangle + \left\langle\psi_\lambda\middle|\frac{d\hat{H}_\lambda}{d\lambda}\middle|\psi_\lambda\right\rangle \\ \frac{dE_\lambda}{d\lambda} &= \left\langle\psi_\lambda\middle|\frac{d\hat{H}_\lambda}{d\lambda}\middle|\psi_\lambda\right\rangle\end{aligned} \tag{3.14b}$$

The Hellmann-Feynman theorem can only be used to obtain first order derivatives. To compute the Dynamical Matrix requires second-order derivatives. Applying $\frac{d}{d\lambda}$ to Eq. (3.14b) yields:

$$\frac{d^2E_\lambda}{d\lambda^2} = \left\langle\frac{d\psi_\lambda}{d\lambda}\middle|\frac{d\hat{H}_\lambda}{d\lambda}\middle|\psi_\lambda\right\rangle + \left\langle\psi_\lambda\middle|\frac{d\hat{H}_\lambda}{d\lambda}\middle|\frac{d\psi_\lambda}{d\lambda}\right\rangle + \left\langle\psi_\lambda\middle|\frac{d^2E_\lambda}{d\lambda^2}\middle|\psi_\lambda\right\rangle. \tag{3.15}$$

The terms involving $\frac{d\psi_\lambda}{d\lambda}$ do not cancel and so the linear response of the wave-function, ψ_λ, with respect to some displacement, λ, must be computed. This can be achieved using either finite-displacement routines or using perturbation theory (DFPT) [44], which is much cheaper computationally [42].

3.1.3 Calculating Phonons in ZnO

ZnO possesses the simple hexagonal wurtzite structure, space group $P6_3mc$ (186). The unit cell consists of 4 atoms: two Zn with fractional positions $(1/3, 2/3, 0)$ and $(2/3, 1/3, 1/2)$; and two O at $(1/3, 2/3, 0.375)$ and $(2/3, 1/3, 0.875)$. CASTEP was

Atom	w
O1	0.377138
O2	0.877031
Zn1	-0.002026
Zn2	0.497857

a	3.293542
b	3.293625
c	5.316134
α	90.000018
β	89.999979
γ	119.994036

Table 3.1: Results of the geometry optimisation performed by CASTEP. Note the precision is as reported in the computed output files and does not imply uncertainty.

used to compute the phonon modes with the Local Density Approximation (LDA) and Generalized Gradient Approximation (GGA) functionals based on a literature review [45, 46]. First, a geometry optimisation was performed using the Broyden Fletcher Goldfarb Shanno (BFGS) algorithm with a total energy convergence tolerance of $2 \times 10^{-5}\,\mathrm{eV\,atom^{-1}}$. Lattice parameters and atomic fractional $\vec{\boldsymbol{c}}$ positions were relaxed, and can be seen in Table 3.1.

A phonon calculation was performed using norm-conserving pseudopotentials and specifying an $10 \times 10 \times 10$ K-point Monkhorst-Pack grid [47]. DFPT [44] was used to calculate the phonon modes in the first Brillouin zone with a $\vec{\boldsymbol{Q}}-$spacing of 0.01 reciprocal lattice units. These calculations were performed using the STFC SCARF compute cluster.

3.1.4 Validating Calculations

As the exchange-correlation functional is not known it is always important to benchmark and validate the results of calculations against other calculations and, ideally, empirical data. Calculated phonon eigenenergies can be plotted against empirical dispersion curves to give some confidence that the selected functional was suitable and calculations are physically meaningful. Neutron spectroscopy then allows direct

measurement of the phonon modes, and the scattering intensity can be calculated from the calculated phonon modes as described in Eq. (2.32).

As a first check, a literature review was performed to obtain empirical data of phonon dispersion energies, as well as first-principles calculations of the phonon eigenvectors [45]. These data and calculations were used as a benchmark to test three potential functionals: LDA; GGA, more specifically Perdew–Burke-Ernzerhof (PBE); and a screened exchange hybrid functional (sX) which has been shown to be particularly good for calculating the electronic band gap in ZnO [9].

Calculated eigenenergies are very similar for LDA and GGA, however the sX functional produces very different values that do not agree with experimental data. Results from the LDA calculation, compared with the reported experimental data, can be seen in Fig. 3.1. The low energy phonon-modes, which contribute most to the thermal parameters, show excellent agreement with empirical results. The higher energy modes are reasonable, however there appears to be a systematic over-estimation of the calculated energies.

At this point it is possible to dismiss the sX functional as a candidate for modelling phonon modes. Since LDA and GGA produce similar eigenenergies, selecting one requires a more stringent test of the calculated phonon modes. Inelastic scattering facilitates this, as measured intensities depend on the eigenvectors as $\vec{\boldsymbol{Q}} \cdot \vec{\boldsymbol{e}}_j(\vec{\boldsymbol{k}}, \nu)$ from Eq. (2.32). Figure 3.2 show the calculated phonons using LDA. These calculations reproduce those in the literature [45], however this is unsurprising as those calculations use the same functional. This motivates inelastic neutron scattering experiments to directly measure these intensities. The python implementation of Eq. (2.32) to produce these plots can be found in Appendix B.

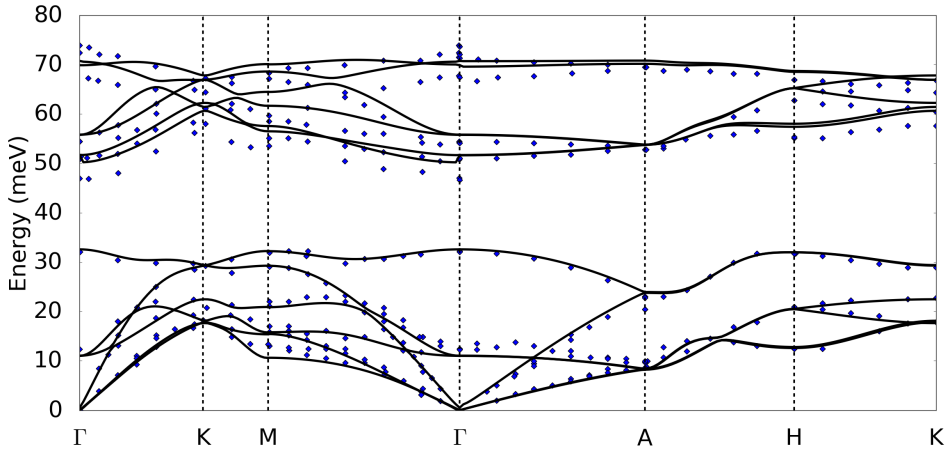


Figure 3.1: Comparison of calculated phonon energies (lines) with experimental data (points) reported in [45]. These phonon modes were calculated using the LDA functional, and are in good agreement with the reported data.

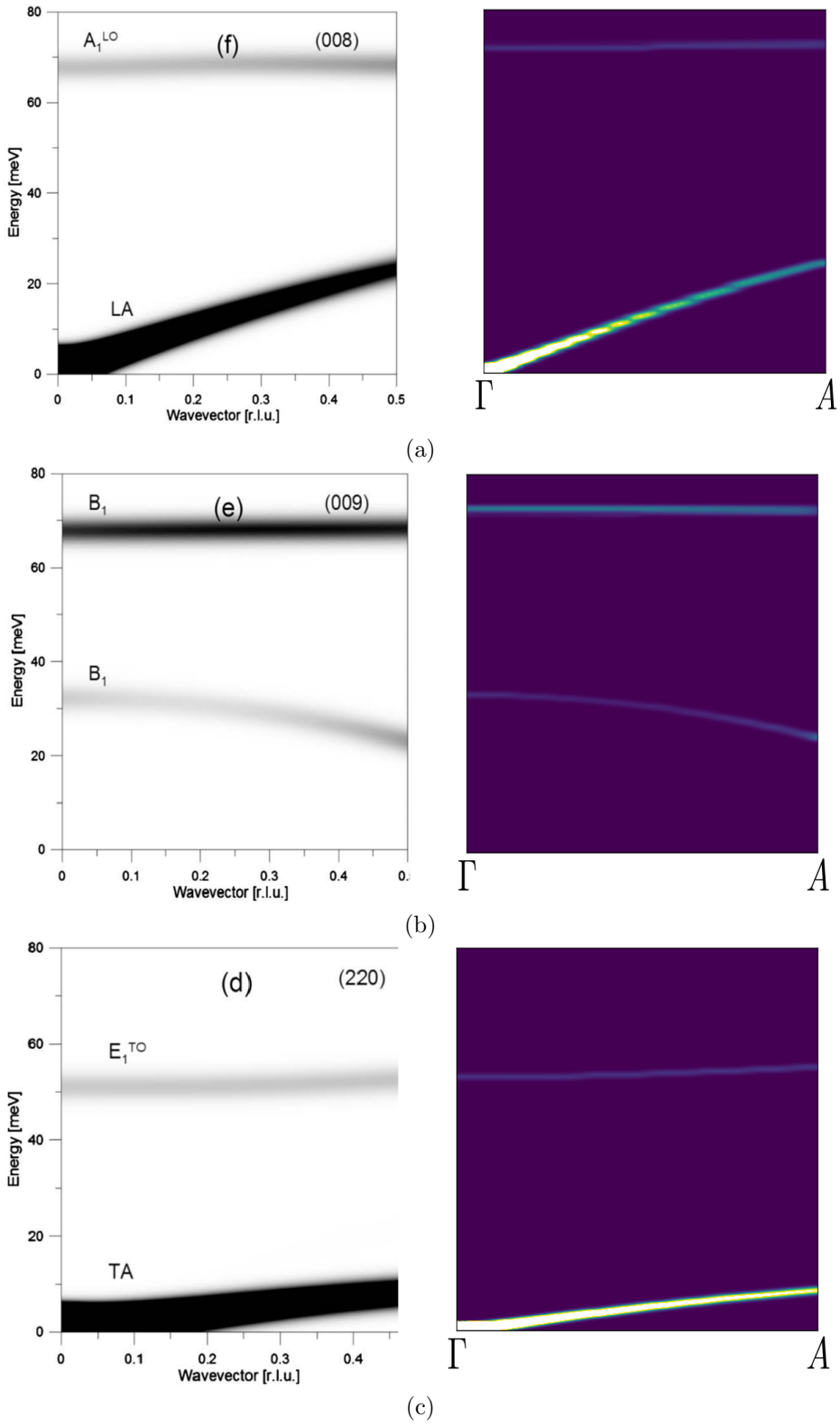


Figure 3.2: Preliminary phonon mode intensities in the (a) 008, (b) 009 and (c) 220 Brillouin zone along the $\Gamma - A$ direction. Left shows the results found in [45], right the calculated results for comparison.

3.2 Measurements on LET

A large, high-quality crystal ($10 \times 8 \times 8$ mm) of nominally stoichiometric ZnO was purchased from Goodfellow. Direct measurements of the phonon dispersion then allow verification of the first principles calculations described in 3.1.

3.2.1 Experimental Procedure

The proposed experiment was to measure the phonons at three temperatures, $T = 5$, 300 and 600 K to verify first-principles calculations. It is then possible to investigate the temperature dependence of the acoustic modes to attempt to extract phonon lifetimes for thermal conductivity calculations. During the experiment, there was a problem sourcing a suitable heat-rod to mount the sample limiting the upper temperature range to 300 K.

The crystal was mounted on aluminium pins and shielded with Cadmium, then placed in a Closed Cycle Refrigerator (CCR). Measurements were performed at two temperatures: 'base', $T = 5$ K, where broadening is minimised and line-shapes determined primarily by instrumental resolution; and room temperature, $T = 300$ K, to investigate phonon-phonon scattering processes.

The sample was aligned with the $(h, 0, l)$ crystallographic plane in the horizontal scattering plane, then rotated through 100° in 1° steps. Each orientation was measured for approximately 10 min. To optimize the flux and resolution the choppers were set to 200 Hz, yielding an incident energy of 30 meV.

The raw measured data was reduced into $S(\vec{\boldsymbol{Q}}, \omega)$ files for analysis using software provided by the ISIS excitations group designed for LET. The data were normalised by beam current, followed by data reduction, background subtraction and vanadium

measurements to correct and calibrate the detectors, as described in Chapter 2. Finally, the data were treated to account for the Bose factor.

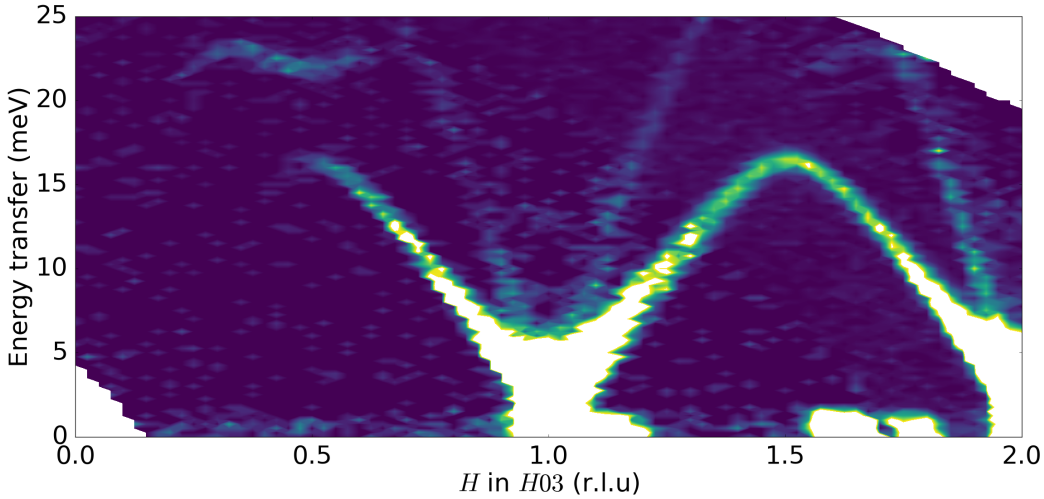
The data reduction was performed using the ISIS compute cluster due to the quantity of data to process.

3.2.2 Experimental Data

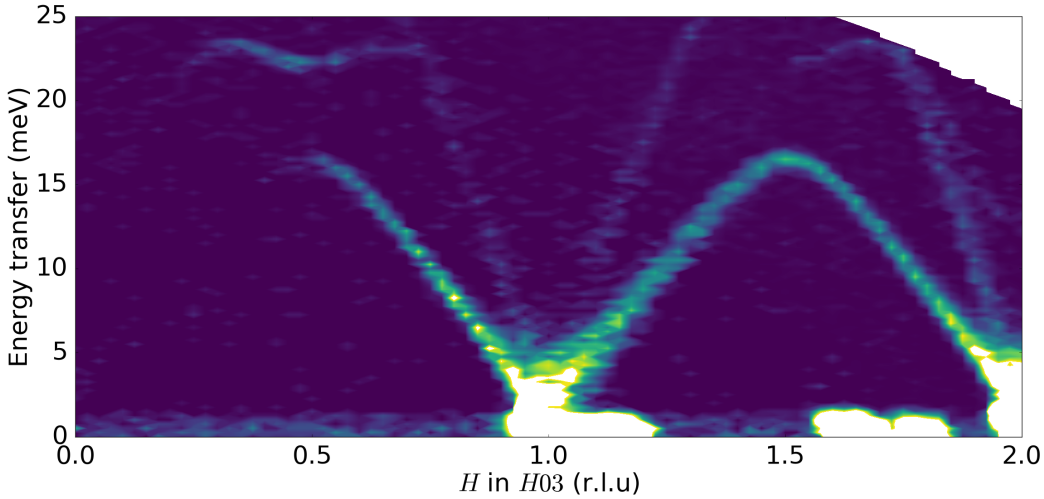
These data cover a wide range of reciprocal-energy space, so 1- and 2-dimensional cuts were obtained using the Horace software [48]. The measured scattering intensity, $S(\vec{\boldsymbol{Q}}, \omega)$, obtained at both temperatures was plotted for several slices along high symmetry directions, [H00] and [00L], then compared with the calculated phonon modes from Section 3.1.

Figure 3.3 shows the (H03) plane of the measured data for two temperatures, 5 and 300 K, compared with calculated intensities at 5 K. To aid comparison with empirical results, regions of reciprocal-energy space that were inaccessible in the experiment were masked in the calculated intensities.

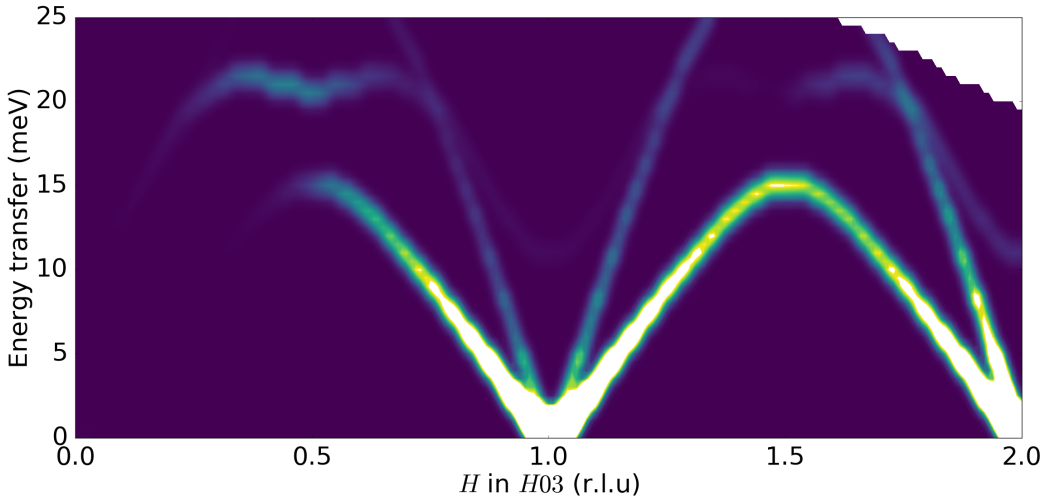
Figures 3.4 and 3.5 show similar plots for two different cuts, the (H02) and (20L) plane. Calculations are in very good agreement with the measured spectra. In the [00L] direction, there is a small oscillation in relative intensities calculated in the brightest mode which is inconsistent with measured data, nonetheless these calculations do a good job of reproducing the data.



(a)

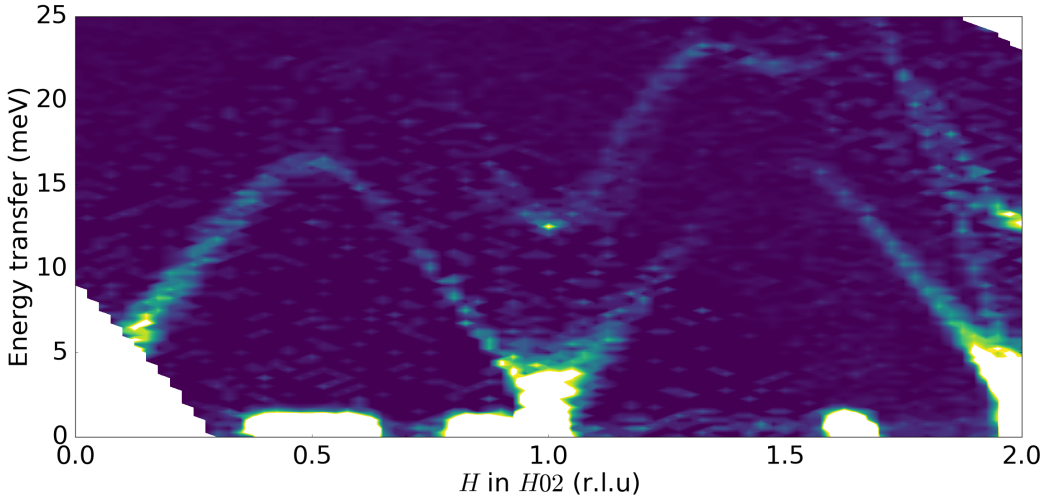


(b)

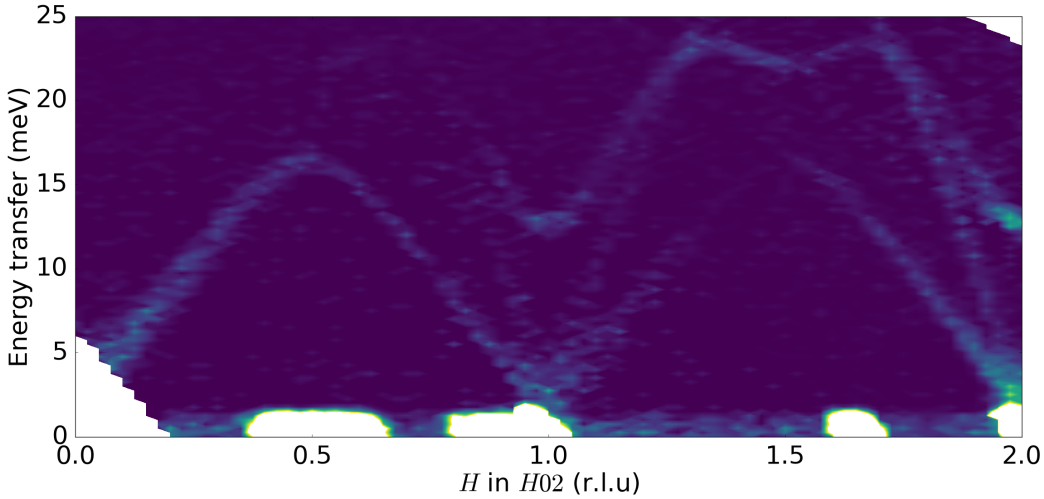


(c)

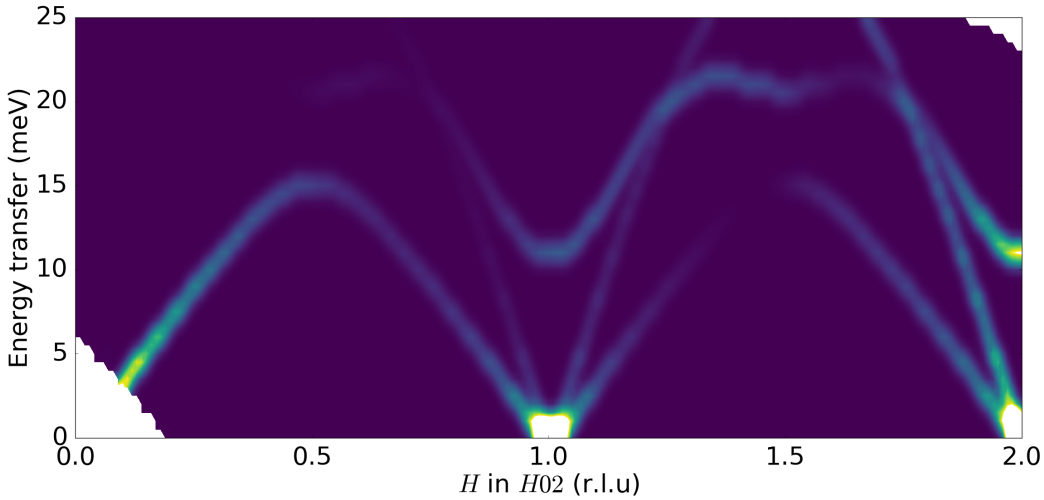
Figure 3.3: Empirical, (a, b), and calculated, (c), scattering intensity along (H03) with resolution $\Delta x = 0.025$ reciprocal lattice units and $\Delta E = 0.25\,\mathrm{meV}$ for: $T = 300\,\mathrm{K}$, (a); and $T = 5\,\mathrm{K}$ (b). Inaccessible regions have been masked in both measured and calculated results.



(a)

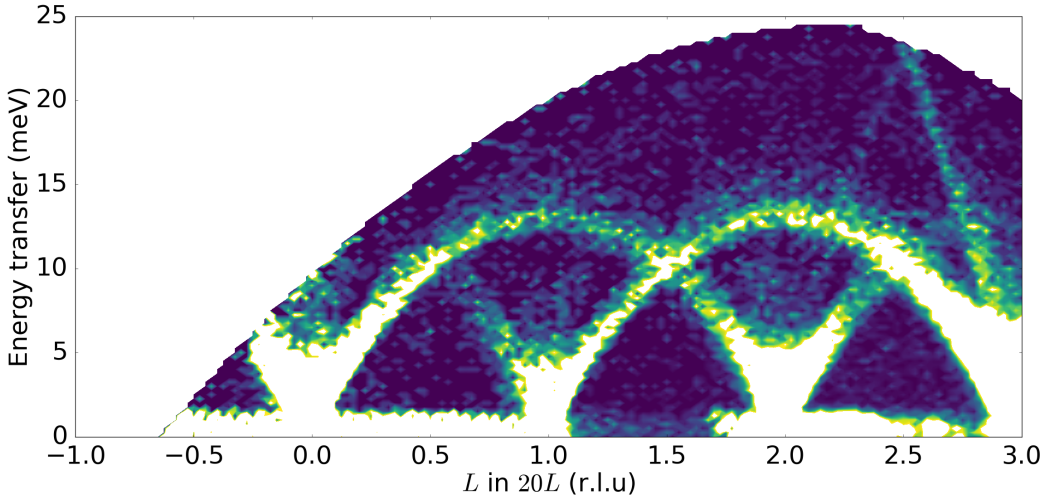


(b)

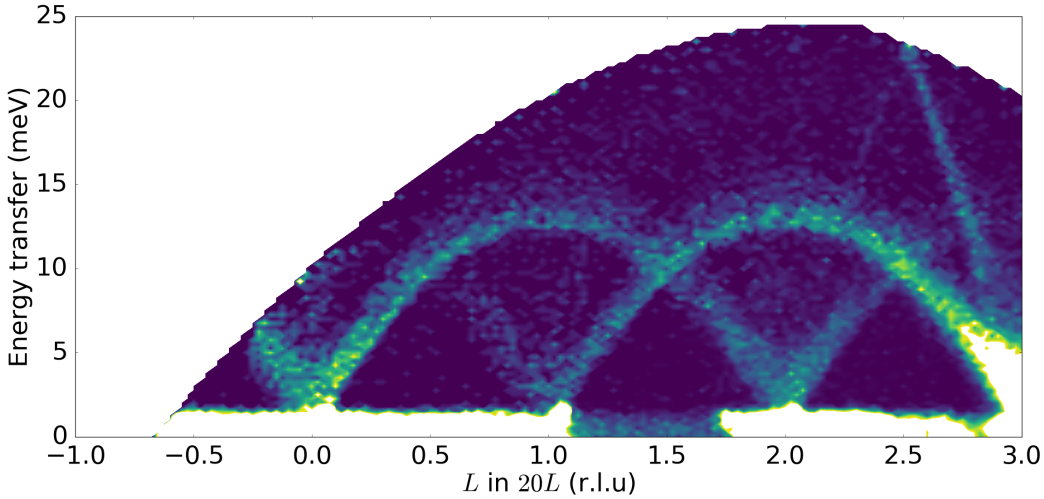


(c)

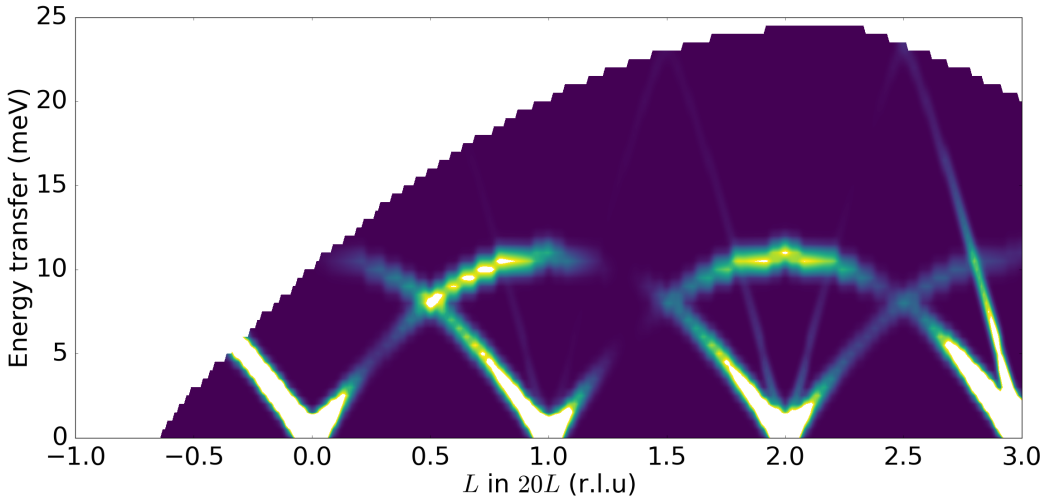
Figure 3.4: Empirical, (a, b), and calculated, (c), scattering intensity along (H02) with resolution $\Delta x = 0.025$ reciprocal lattice units and $\Delta E = 0.25\,\mathrm{meV}$ for: $T = 300\,\mathrm{K}$, (a); and $T = 5\,\mathrm{K}$ (b).



(a)



(b)



(c)

Figure 3.5: Empirical, (a, b), and calculated, (c), scattering intensity along (20L) with resolution $\Delta x = 0.025$ reciprocal lattice units and $\Delta E = 0.25\,\mathrm{meV}$ for: $T = 300\,\mathrm{K}$, (a); and $T = 5\,\mathrm{K}$ (b).

The phonon lifetimes determine their phonon line-widths, Fig. 3.6 shows a typical 1-d cut through an acoustic mode for the two temperatures, specifically at (-1.5, 0, 3). The temperature dependence appears to be dominated only by an increase in intensity. Measurements suggest the temperature has either little effect on the phonon line-widths, and thus lifetimes, or the line-widths are limited by instrumental resolution and not measurable.

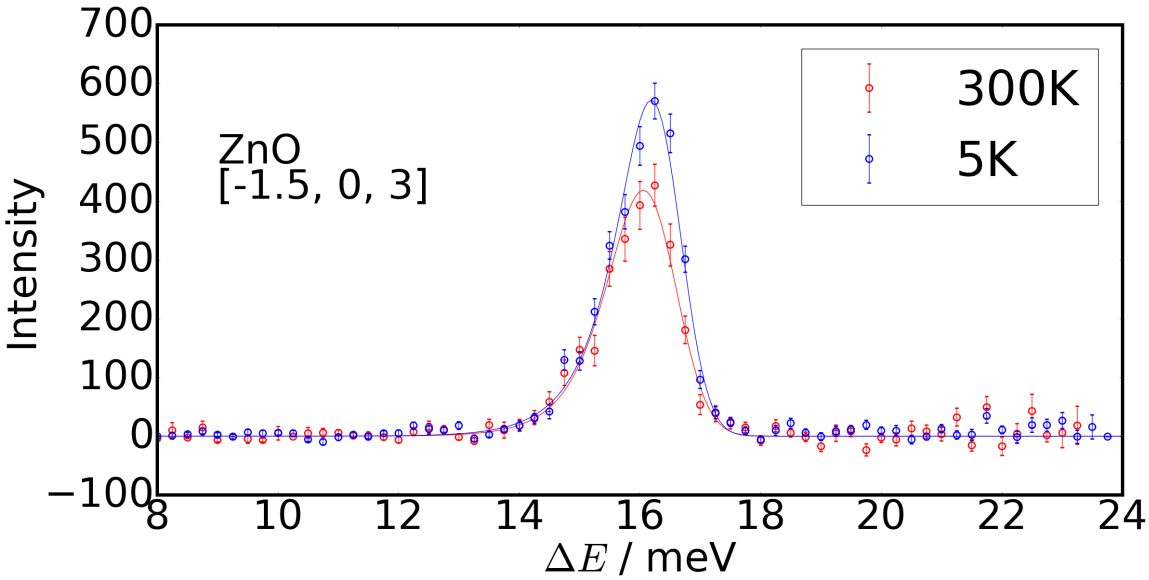


Figure 3.6: Energy cuts at (-1.5, 0, 3). Fits for the linewidths do not show convincing evidence of broadening.

3.3 Measurements on Merlin

In Section 3.2 the low-energy modes were measured and used to validate first-principles calculations. To further verify the model and investigate the temperature dependence of phonon lifetimes, measurements were performed on Merlin similar to those in Section 3.2. The higher energy neutrons on Merlin allow probing the full dispersion, as

the calculated phonon modes have energies below 80 meV.

3.3.1 Experimental Procedure

The same large crystal from the LET experiment was measured at three temperatures, $T = 5$, 300 and 550 K. Gadolinium was used for shielding to allow the high temperature measurements as cadmium melts at 590 K. The 's' chopper was set to 400 Hz yielding 3 separate incident energies due to repetition-rate multiplication. Thus simultaneous measurements were performed with: $E_i = 170$ meV, to measure the full dispersion; 59 meV, with better resolution covering the first band of phonons; and 29 meV, for direct comparison with the LET data.

The sample was, again, aligned with the $(h, 0, l)$ plane horizontal with orthogonal in-plane vectors, and this time rotated through 120° in 1° steps. The sample was mounted on a hot stick in a CCR to measure the three temperatures. Each orientation was measured for approximately 20 min. Data reduction, background subtraction and vanadium calibrations were performed as in Section 3.2. The $T = 300$ K measurement was only rotated through 77° due to beam loss, and so has less coverage.

3.3.2 Experimental Data

Misalignment Correction

To ensure measurements were successful the elastic scattering can be plotted as a sanity check that the sample was aligned as expected and rotated correctly. The true rotation of the crystal could be different to the nominal $\vec{\boldsymbol{u}}-$ and $\vec{\boldsymbol{v}}-$ vectors. This would result in a systematic error in the labelling of $\vec{\boldsymbol{Q}}$ during the data reduction.

Figure 3.7 shows a typical elastic line for the 170 meV before background sub-

traction. This plot shows there is a small misalignment in plane. Each orientation measured introduces a segment of the obtained spectra. Initial slices seem reasonably-aligned for $L \geq -4$, although the Bragg peaks are not quite aligned with integer L. For $L \leq -5$ the missing Bragg peaks indicate an in-plane misalignment.

By integrating over specific Bragg peaks it is possible to calculate the transformation matrix required to correct the nominally aligned $\vec{\boldsymbol{u}}$ and $\vec{\boldsymbol{v}}$. This was performed using the Horace software, on the 5 K data to minimise broadening which would introduce error to the correction calculation.

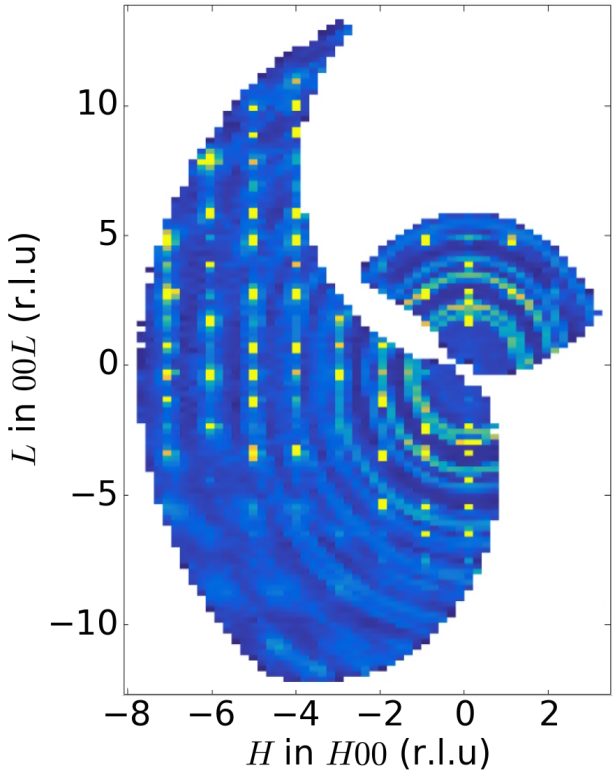


Figure 3.7: Plot of the elastic line obtained on Merlin, before background subtraction and misalignment corrections. For $L \leq -5$ the misalignment is clearly shown with missing Bragg peaks.

The result of applying the correction can be seen in Fig. 3.8. Misalignment cor-

rections were performed for all temperatures and gave a consistent result, since the crystal was not removed and remounted during the experiment. After corrections, there are no more missing Bragg peaks and they are aligned with the axis.

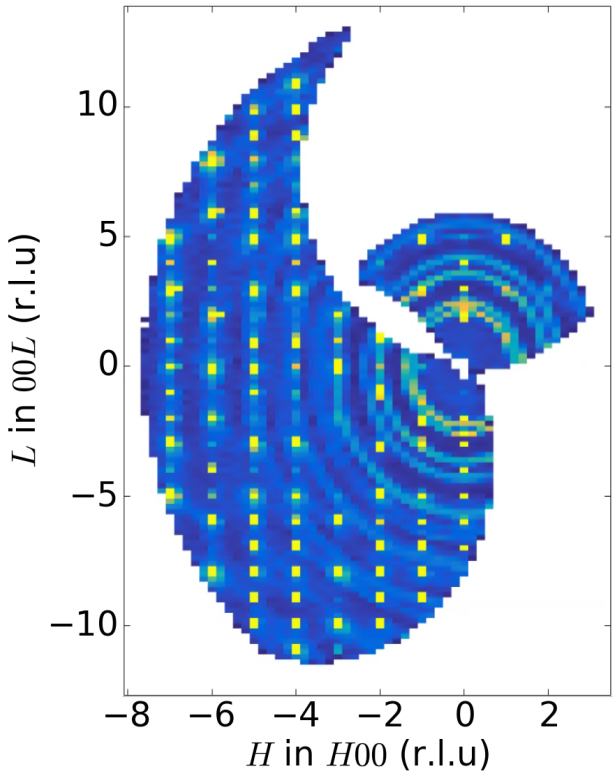
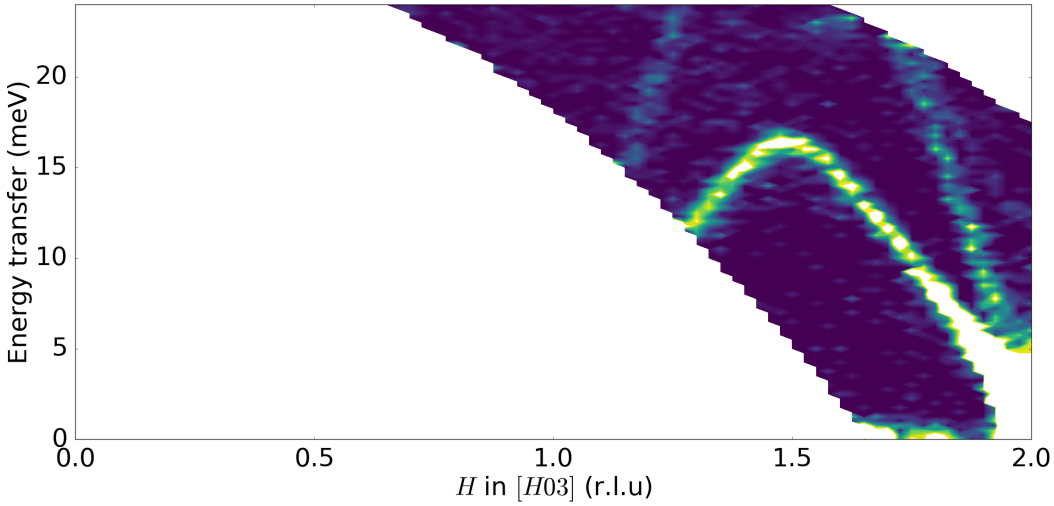


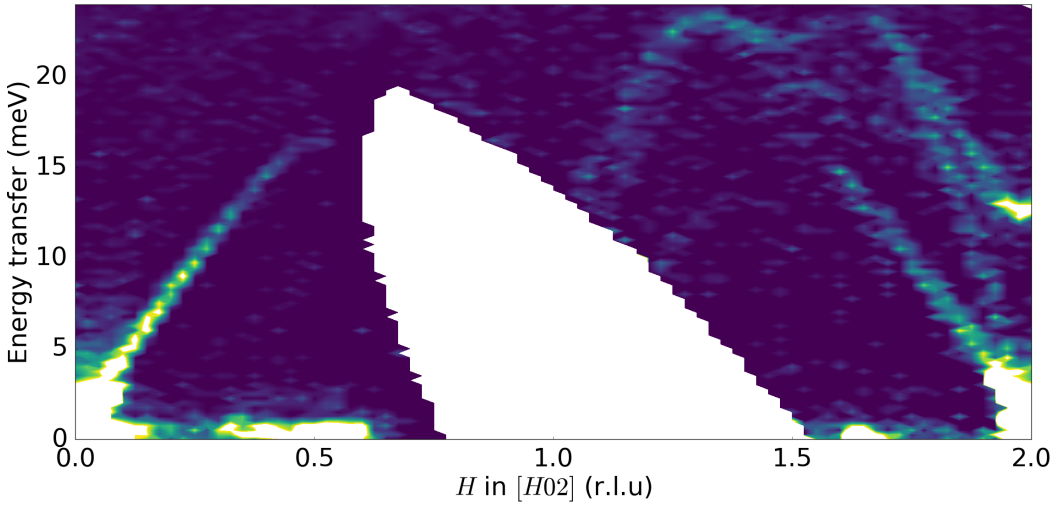
Figure 3.8: Plot of a typical elastic scattering (integrated over the energy range −0.2 meV to 0.2 meV) after misalignment corrections. There are no more missing Bragg peaks, and they are well aligned with integer H, L.

Comparisons With LET

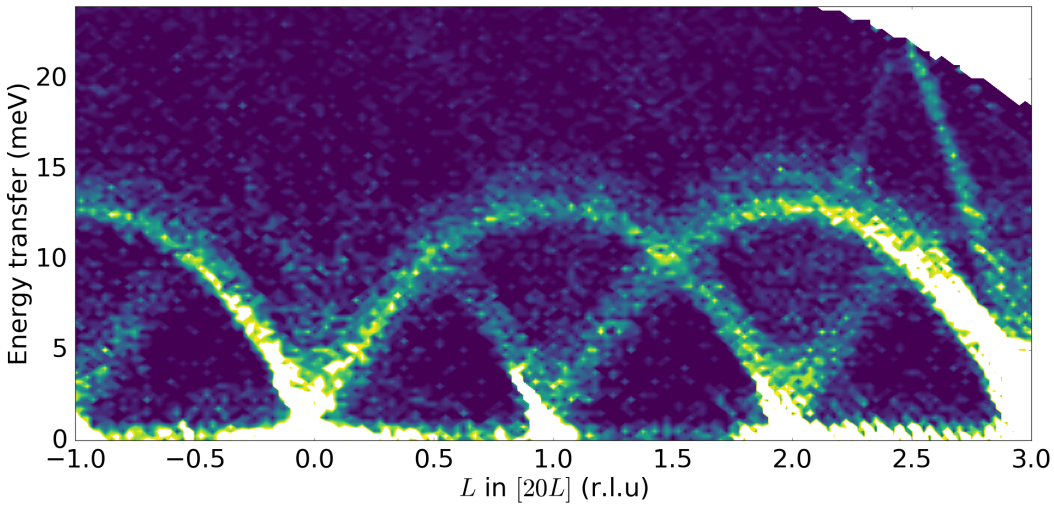
To further verify misalignment corrections – as well as the data collection, reduction, background subtraction and normalisation – equivalent slices were made for the 5 K, $E_i = 29$ meV data for comparisons against those discussed in Section 3.2, and can be seen in Fig. 3.9. These data are consistent with the LET data in Figs. 3.3 to 3.5.



(a) H03



(b) H02



(c) H01

Figure 3.9: $S(\vec{\boldsymbol{Q}}, \omega)$ measured on Merlin along $[H, 0, 3]$ (a), $[H, 0, 2]$ (b) and $[2, 0, L]$ (c). Measured at $T = 5\,\mathrm{K}$ with $E_i = 29\,\mathrm{meV}$. These slices are equivalent to Figs. 3.3 to 3.5, with the same resolution.

High Energy Phonons

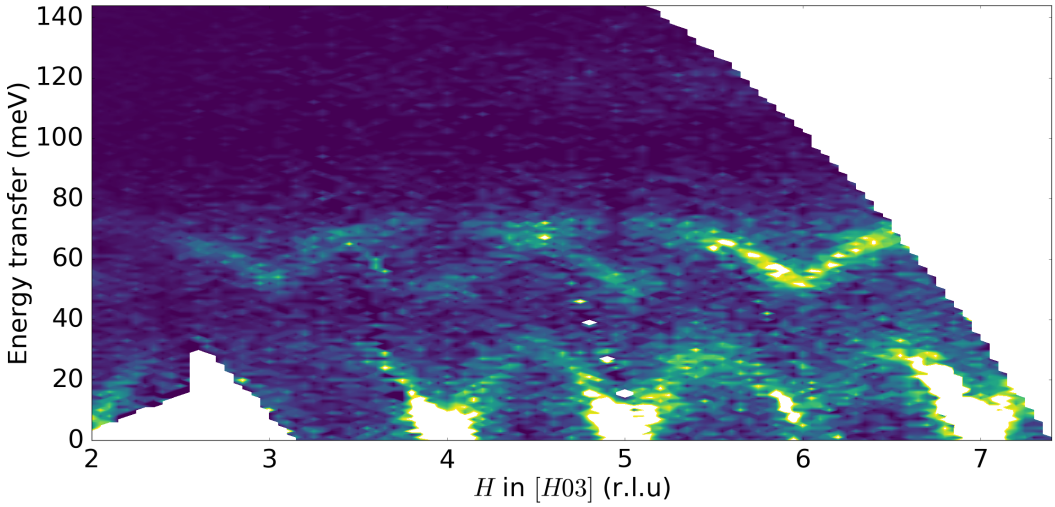
With the $E_i = 170\,\text{meV}$ measurements the full dispersion can be sampled and compared against the model discussed in Section 3.1. The dispersion along H03 can be seen in Fig. 3.10 which is, again, in very good agreement. The lowest energy modes cannot be distinguished from the background.

In the calculation there is a bright, flat mode present at 70 meV between H=−3.5 and H=−5.5 that does not appear as flat in the data, which immediately draws the eye, however it follows the measured dispersion quite closely and the changes in relative intensities are in excellent agreement.

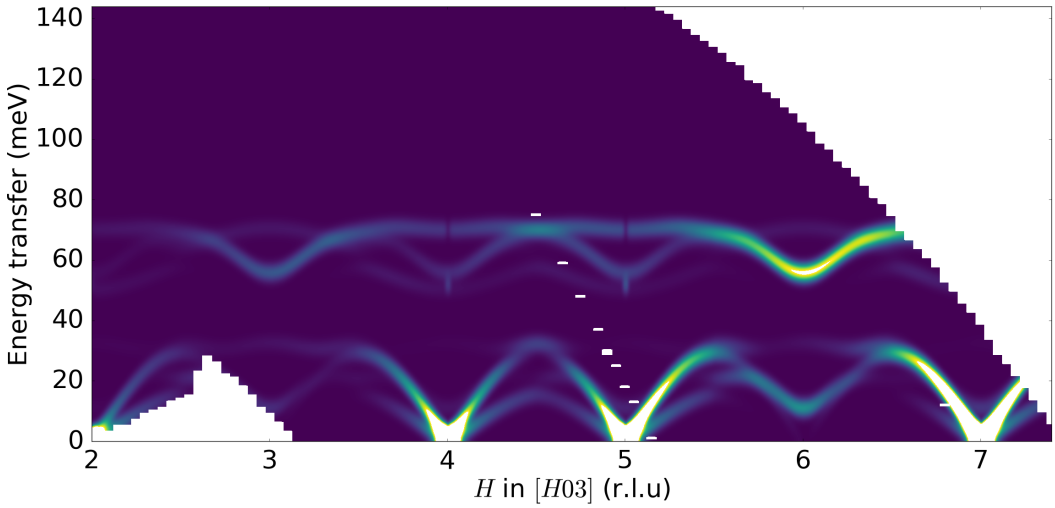
Similar plots for H02 and 10L can be seen in Figs. 3.11 and 3.12. The flat, additional mode present in calculations is consistently seen along H0X, however absent along X0L.

Measurements near the elastic Bragg peaks are very noisy due to the energy resolution for this incident energy, 6.8 meV to 11.9 meV, most clearly seen in Fig. 3.12. The calculated high-energy modes are in good agreement with the data. The low-energy data with better resolution, compared with LET measurements, shows excellent agreement with calculations. Since the acoustic modes typically play the largest role in the thermal conductivities [46], the model is well-suited for further calculations to investigate the thermoelectric properties of ZnO.

Finally, the [HH0] direction can be seen in Fig. 3.13 and is in equally good agreement with the others.

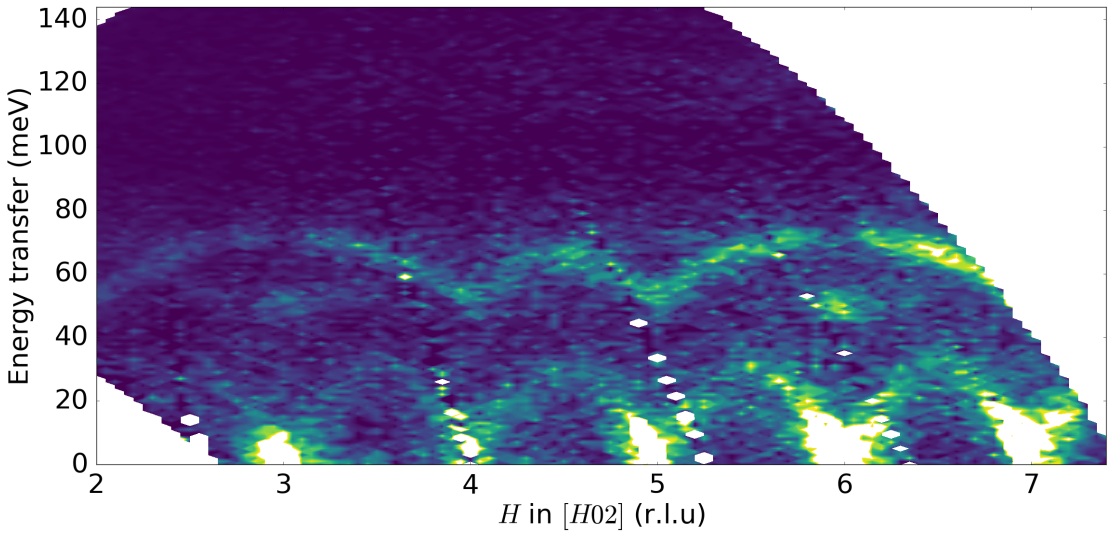


(a) Measured

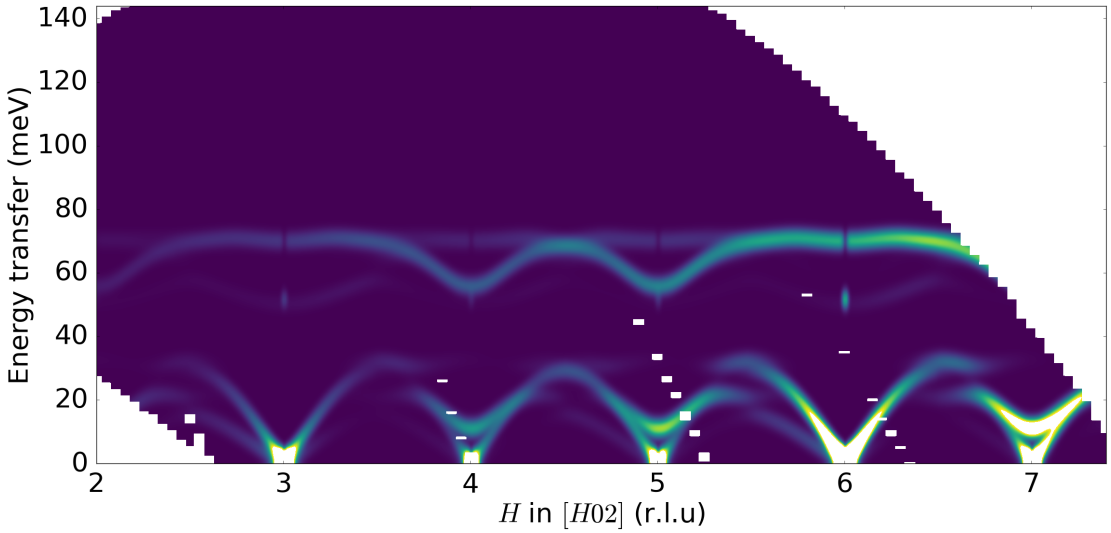


(b) Calculated

Figure 3.10: Phonon intensities measured (a) and calculated (b) at $T = 5\,\mathrm{K}$ with $E_i = 170\,\mathrm{meV}$. For these plots, the resolution was changed to $\Delta E = 1\,\mathrm{meV}$

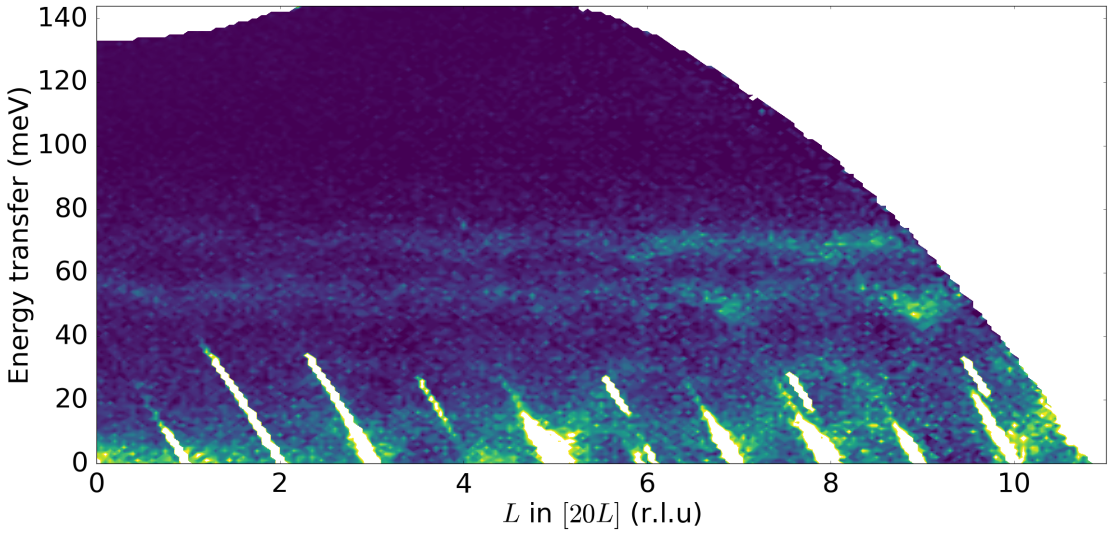


(a) Measured

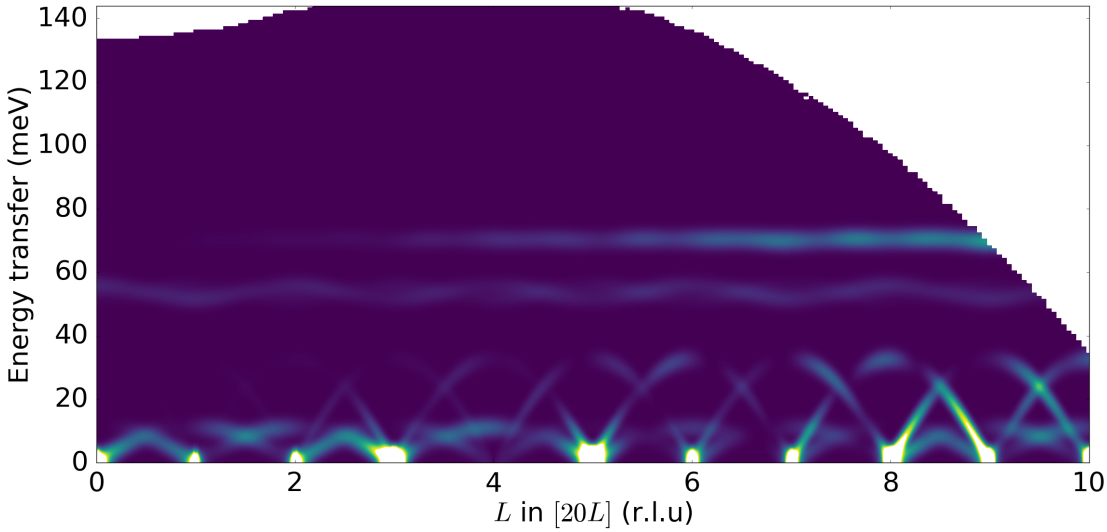


(b) Calculated

Figure 3.11: Phonon intensities measured (a) and calculated (b) at $T = 5\,\mathrm{K}$ with $E_i = 170\,\mathrm{meV}$. The high energy mode still has the anomalous feature.

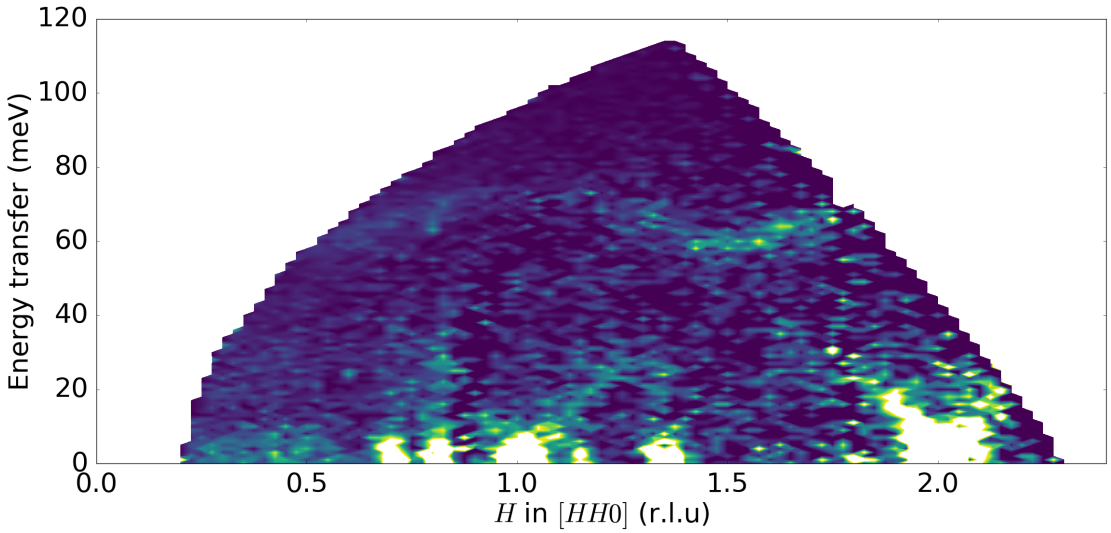


(a) Measured

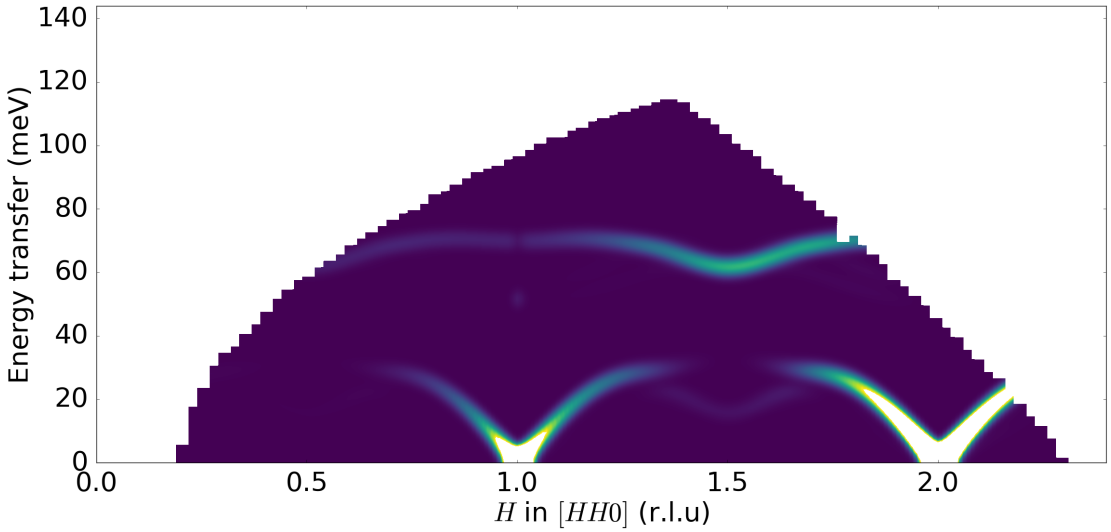


(b) Calculated

Figure 3.12: Phonon intensities measured (a) and calculated (b) at $T = 5\,\mathrm{K}$ with $E_i = 170\,\mathrm{meV}$.



(a) Measured

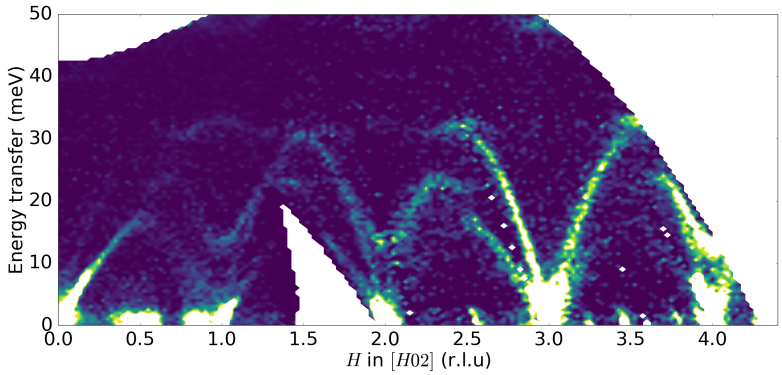


(b) Calculated

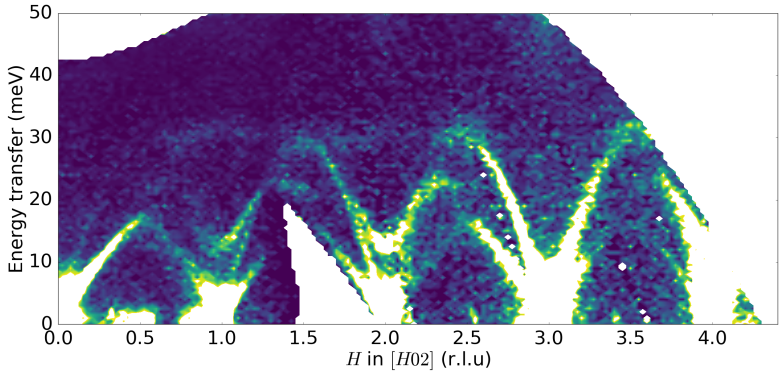
Figure 3.13: Phonon intensities measured (a) and calculated (b) at $T = 5\,\mathrm{K}$ with $E_i = 170\,\mathrm{meV}$.

Temperature Dependence

The H02 slice for the 59 meV data at the three temperatures can be seen in Fig. 3.14. From this the cut at $(2.5, 0, 2)$, shown in Fig. 3.15, was taken. Initial analysis of phonon line-widths was performed using a Gaussian to determine if there was detectable broadening.



(a)



(b)

Figure 3.14: $S(\vec{\boldsymbol{Q}}, \omega)$ measured on Merlin along H02 at $T = 5\,\mathrm{K}$ (a) and $T = 550\,\mathrm{K}$ (b), with $E_i = 59\,\mathrm{meV}$.

The lowest energy peak softens, i.e. shifts to lower energy, and broadens by 300 K, above which it is stable. The smallest peak does not appear to soften as much, although it does broaden noticeably by 550 K. The highest energy peak is very stable,

and then softens at 550 K.

Fits were attempted to determine the peak profile using a convolution of a Gaussian, Lorentzian and the nominal instrumental resolution, however it was not possible to separate broadening due to instrumental resolution from that of the phonons; the variance of these two fitted parameters was very large and inconsistent across datasets, and so no reasonable values could be extracted for lifetimes from these data.

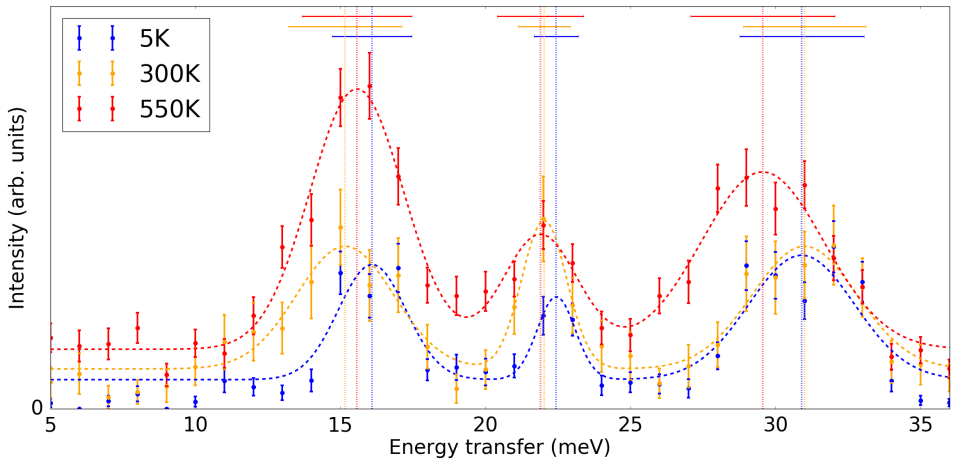


Figure 3.15: A slice at constant Q through $(-2.5, 0, 2)$ for 5 meV to 35 meV at $T = 5$ K (blue), 300 K (orange) and 550 K (red). The peak width and positions have been determined by fitting 3 gaussians. The dotted line shows the fitted peak centre, and the thick solid bar the FWHM.

Figure 3.16 shows a cut at $(1.5, 0, 3)$ at the three temperatures for the 29 meV data shown in Fig. 3.9a. This region was selected as the data is particularly clean and strong. The 5 K data was used to benchmark the instrumental resolution, and then phonon lifetimes extracted from 300 and 550 K. A 0.234 and 0.439 meV phonon broadening was fitted, from which lifetimes of 1.19 and 0.64 ps are obtained.

When comparing the 170 meV measurements with calculations, a weak, diffuse signal is noticeable at very high energies of approximately 120 meV. Figure 3.17 shows

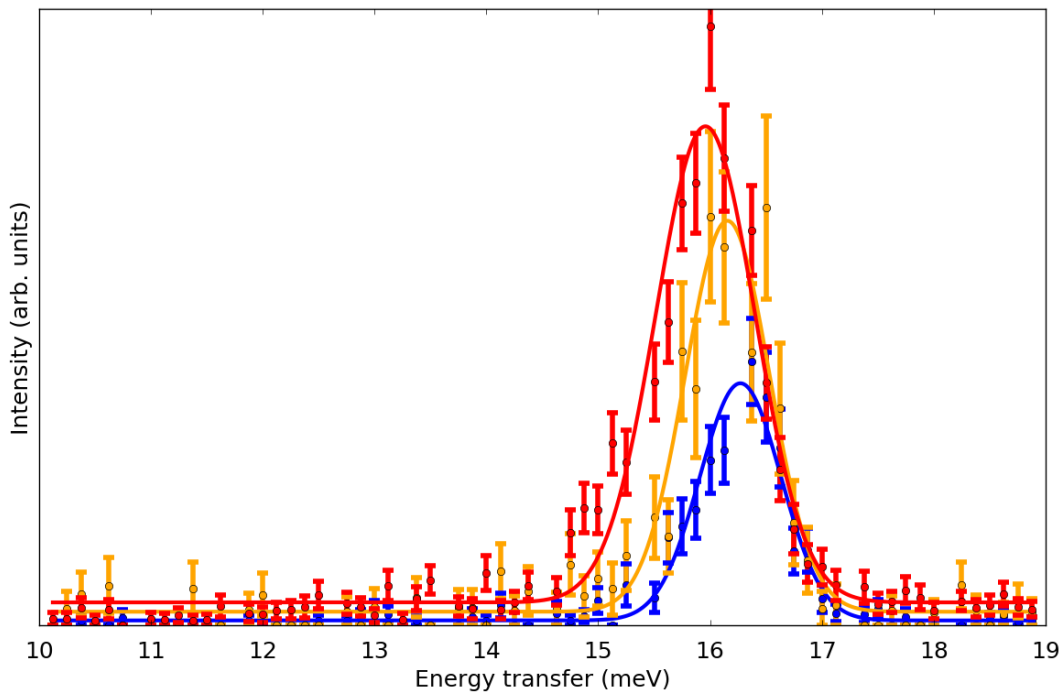
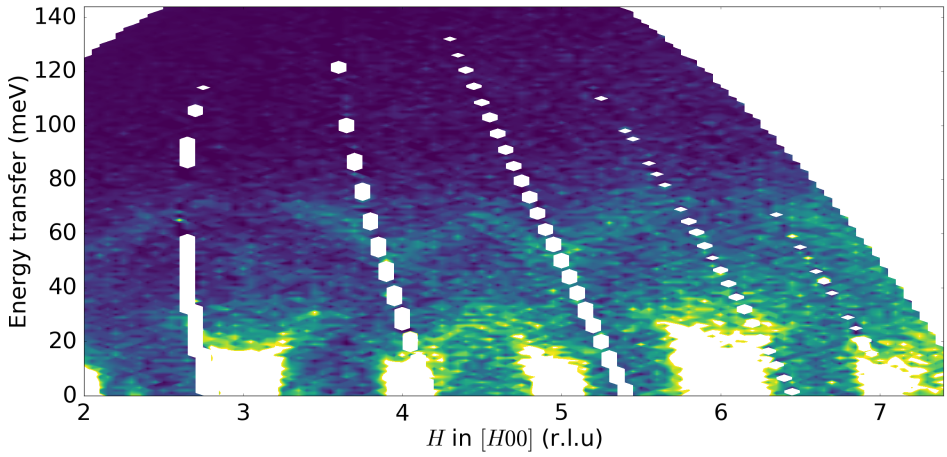
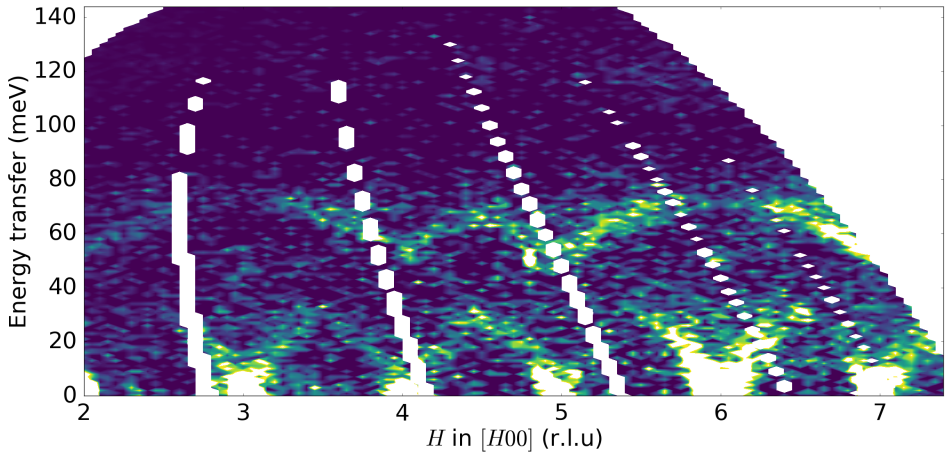


Figure 3.16: A slice at constant $Q_{hkl} = (1.5, 0, 3)$ for 10 meV to 19 meV at $T = 5$ K (blue), 300 K (orange) and 550 K (red). From these measurements the phonon lifetimes were extracted.

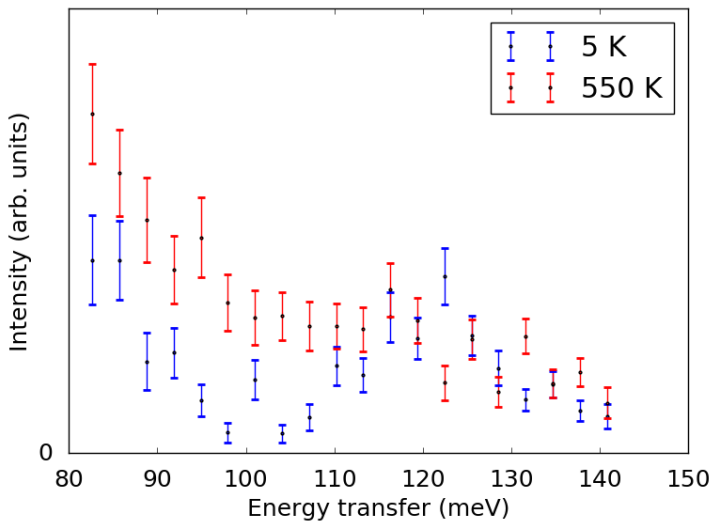
a plot along H00 with intensities plotted to emphasise the weak signal present. An integration over H between $4.5 - 5.5$ r.l.u. for $E \geq 80$ meV can be seen in Fig. 3.17c, and shows a peak in intensity ~ 120 meV. This suggests the source of this signal is phonon-phonon scattering, as the calculated dispersion has no phonons above 80 meV, and will be further investigated in Chapter 4.



(a)



(b)



(c)

Figure 3.17: (a, b): $S(\vec{\boldsymbol{Q}}, \omega)$ along H00 at $T = 550$ (a) and 5 K (b). There is a weak, diffuse signal present at $E = 120$ meV. (c): Integration over Q of the high-energy diffuse signal against E for $T = 5$ K (blue) and 550 K (red). The multi-phonon scattering is also visible at lower energies at 550 K.

3.4 Summary

The phonon dispersion of ZnO was modelled using Density Functional Theory. A number of different exchange-correlation functionals were trialled and compared to measurements performed on Merlin and LET. The LDA functional was selected, which gives excellent agreement with direct measurements of the phonon dispersion using INS.

Calculations were performed using CASTEP 17.1 with norm-conserving pseudopotentials and an $8 \times 8 \times 8$ Monkhorst-Pack grid. DFPT was then used to calculate phonon eigenvectors in the first Brillouin zone.

In the Merlin data, energy broadening corresponding to finite phonon lifetimes are detected at elevated T. The next chapter explores the consequences of this on the thermal conductivity.

Chapter 4

Thermal conductivity of ZnO

It is possible to suppress the thermal conductivity in ZnO by a factor 7 through nanostructuring [19]. In this chapter, the thermal conductivity of ZnO is investigated in bulk and nanostructured samples using inelastic neutron spectroscopy and laser flash measurements.

First the sample preparation, characterisation and experimental setup is discussed. The bulk and nanostructured measurements are presented individually. Following this, a comparison of a semi-empirical and *ab initio* model to extract thermal conductivities from the measured phonon density of states (PDOS) are presented.

Finally, thermal conductivities of bulk, single-crystal ZnO are investigated using laser flash measurements.

4.1 INS Measured on MARI

To investigate the effects of nanosized crystal grains on the lattice dynamics, as well as the multi-phonon scattering seen in Chapter 3, powder inelastic neutron scattering

measurements were performed on MARI for a range of temperatures on bulk and nanostructured samples.

4.1.1 Sample Preparation and Characterisation

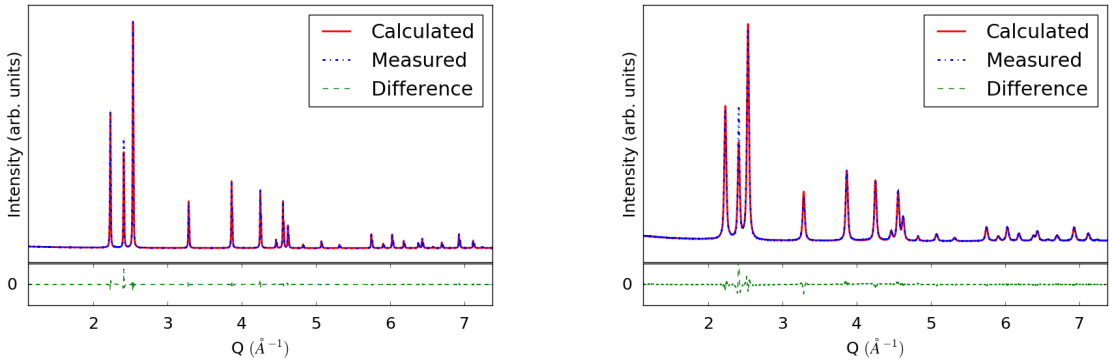
Two powders were produced by Dr. Chris Nuttall at Johnson Matthey (JM): one consisting 99.99% pure, ball-milled ZnO purchased from Sigma Aldrich (bulk); and another with nanosized grains synthesised using flame spray pyrolysis (FSP). For both, the total sample mass ∼30 g.

Characterisation of these powders was performed by JM. Average crystallite size was determined using XRD with a Bruker D8 Cu-source diffractometer and the TOPAS software package. Measured, and fitted, XRD data can be seen in Fig. 4.1, yielding 117(1) and 15.3(1) nm for bulk and FSP powders respectively. The goodness of these fits is reported by the so-called R factor, defined as

$$R = \frac{\sum ||F_{\text{obs}}| - F_{\text{calc}}|}{\sum |F_{\text{obs}}|} \tag{4.1}$$

For the fits of the bulk the FSP samples, R factors of 7.218 and 7.898 were obtained, respectively. The measured peak positions are consistent with the hexagonal wurtzite structure reported from the bulk sample in [19].

SEM characterisation of the FSP powder was also provided, a typical SEM image can be seen in Fig. 4.2. From these images, an average particle size of 28(8) nm was extracted. It should be noted that values obtained from SEM and XRD cannot be immediately compared, as SEM measures the average particle size, whilst XRD measures the average crystallite size; these values are only the same if the particles are all single crystals. Comparison with SEM data from the nano-structured sample



(a) (b)

Figure 4.1: Measured XRD of the bulk (a) and FSP (b) samples, used to determine average crystallite size. Difference between the fitted and measured peaks can be seen plotted in a dashed green line. Provided by JM.

reported by Jood *et al* shows a similar size distribution[19], thus this sample was deemed to be suitable for comparison with that in the reference.

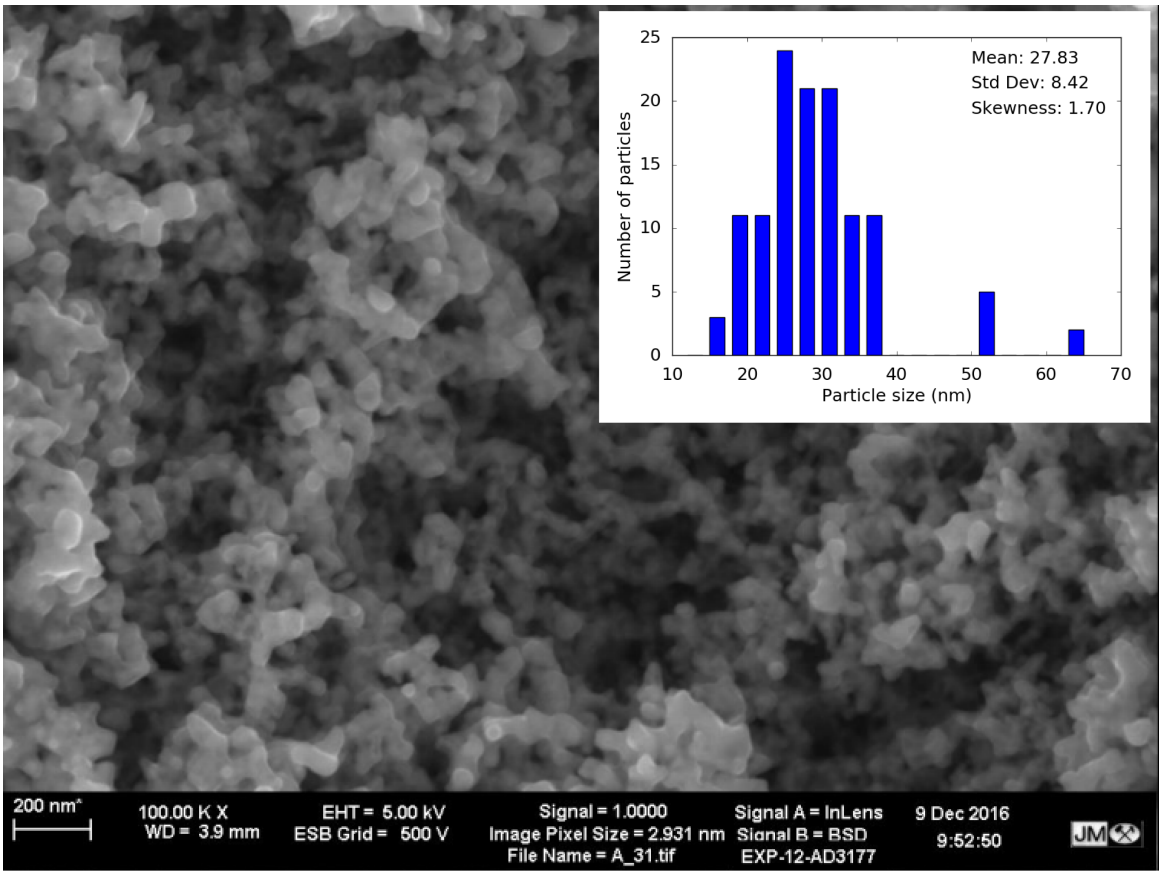


Figure 4.2: A typical SEM image of FSP powder. The size distribution extracted from this image can be seen as a histogram in the inset. Provided by JM.

4.1.2 Experimental Procedure

Powders were initially placed in a cylindrical can made of a thin sheet of aluminium. The sample can was secured to the mount using screws and a boron nitride piece to minimise unwanted scattering. The 's' chopper was used at 250 Hz with two incident energies, 40 and 80 meV at room temperature. Measurement times varied, ranging 12-24 hours to obtain suitable statistics. Background measurements of the empty can, and vanadium calibrations, were performed as usual.

Data reduction was performed using a number of Mantid scripts developed by the

excitations group. From the initial measurement it became clear some changes were required for continuation experiments. In the 80 meV data, intensity does not drop off to zero at high energy as expected from the calculated one-phonon dispersion. This incident energy was changed to 200 meV to investigate the multi-phonon scattering. To maintain reasonable resolution, the chopper frequency was changed to 400 Hz for the 200 meV measurements. The 40 meV measurements were kept to obtain high-resolution data of the acoustic phonons most involved with heat transport.

Samples were sealed inside annular cylindrical Nb cans with height and outer diameter of 45 mm. The annular thickness was selected such that the total volume in the can was only slightly larger than that of the samples. Loading the powdered samples in this way ensures the entire volume of sample will be bathed in the neutron beam, maximising scattering. The cans were mounted in a furnace and measured at $300, 500, 700$ and 900 K.

4.2 Bulk PDOS

The measured $S(Q, \omega)$ can be seen in Fig. 4.3 for $E_i = 40$ and 200 meV. At energy transfer near 0 meV the elastic line makes it difficult to investigate phonon modes due to the instrumental resolution. Data above 85% of E_i are extremely noisy and removed. As a result, the PDOS is best analysed in the range 5 meV to 35 meV using the high-resolution 40 meV data, whilst the 200 meV data provides coverage elsewhere.

The PDOS, $g(E)$, can be calculated from first principles as:

$$g(E) = B \sum_j \left\{ \frac{4\pi b_j^2}{m_j} \right\} g_j(E), \tag{4.2}$$

where B is a normalisation constant and b_j, m_j, $g_j(E)$ are the neutron scattering length, mass and partial density of states of the jth atom [49]. The partial density of states is obtained by summing the contribution from each phonon mode, ν, i.e.:

$$g_j(E) = \sum_\nu \int \frac{d\vec{\boldsymbol{k}}}{4\pi^3} |e_\nu(j)|^2 \delta(E - E_\nu(\vec{\boldsymbol{k}})), \tag{4.3}$$

where $\vec{\boldsymbol{k}}$ is the phonon's reduced wavevector, $e_\nu(j)$ the normalised phonon eigenvector of the νth mode for the jth atom and E_ν its associated energy.

The calculated PDOS obtained from Eq. (4.2) is idealised, and the obtained PDOS consists of only very sharp delta functions. A more realistic PDOS is obtained by applying broadening to account for the instrumental resolution, finite-size effects, scattering from phonons, etc.

To account for the phonon broadening, a damped harmonic oscillator model was used:

$$f(\omega) = \frac{1}{\pi Q_f \omega'} \frac{1}{(\omega'/\omega - \omega/\omega')^2 + 1/Q_f^2}, \tag{4.4}$$

where ω' is the frequency of the phonon mode and Q_f the quality factor [50].

The instrumental resolution of MARI is well characterised and can be calculated using a program called PyChop, provided by the excitations group [51].

In the 200 meV data shown in Fig. 4.3a there is scattering present with energy transfer up to 120 meV, visible at high Q (12 Å^{-1} to 16 Å^{-1}); this is most clearly seen in the PDOS in the right inset.

Since the phonon dispersion calculated in Section 3.1 shows the mode with the highest energy is approximately 75 meV, and the signal observed gets more intense as Q increases, this suggests multi-phonon scattering that needs to be accounted for to properly model these data.

This can be calculated by convolving the one-phonon PDOS with itself in order to get a 2-phonon component, $g_j^{(2)}(E)$, and then finding the total PDOS as:

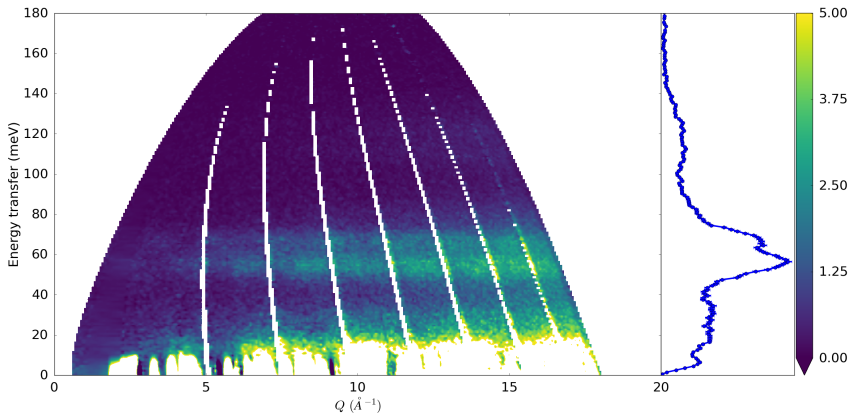
$$g_j^{(\text{total})}(E) = A(g_j^{(1)}(E) + Bg_j^{(2)}(E) + ..), \tag{4.5}$$

where A is an arbitrary scaling and B the weighting factor of the two-phonon component obtained from empirical fits. Thus, the PDOS was calculated as follows:

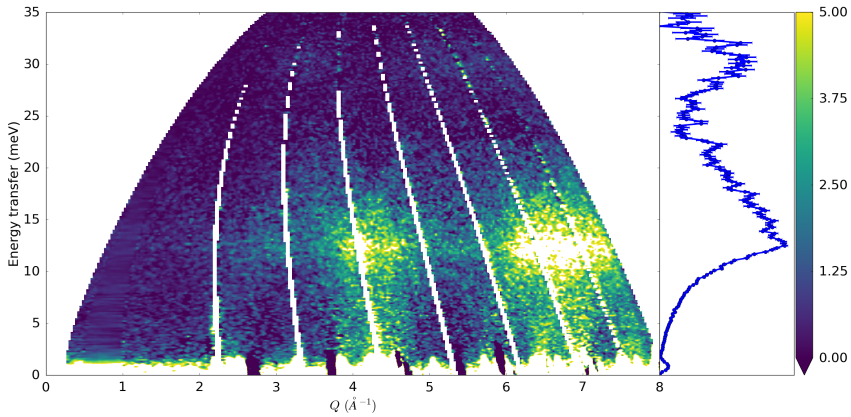
1. The idealised PDOS is calculated using Eqs. (4.2) and (4.3).
2. The idealised PDOS is convolved with Eq. (4.4) to apply phonon broadening.
3. The multi-phonon scattering is calculated using Eq. (4.5).
4. Finally, the instrumental resolution function obtained from PyChop is applied.

The measured and calculated PDOS at $E_i = 40$ and 200 meV for the bulk sample can be seen in Fig. 4.4, with a value of $B = 0.8(4)$. The PDOS has been plotted over the full range measured, using the high-resolution data in the range 5 meV to 35 meV.

At low energies, the damped harmonic oscillator is in very good agreement with the data, however this model does not work so well for the optic modes [50] and is unable to fit the higher energy data. It is noteworthy that this model works very well for the low-energy acoustic phonons of nanocrystalline $Si_{1-x}Ge_x$, but the authors acknowledge that the model fails at higher energies for the optic phonons [52].

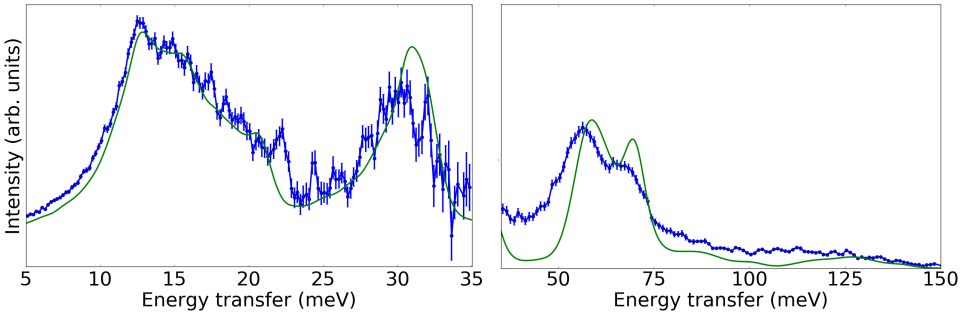


(a)



(b)

Figure 4.3: Measured $S(Q, \omega)$ for the bulk powder at $T = 300\,\mathrm{K}, E_i = 200$ (a), and $40\,\mathrm{meV}$ (b). The PDOS can also be seen, for reference on the right, plotted in blue, which has been computed from these data using the MARI reduceToDOS tools.

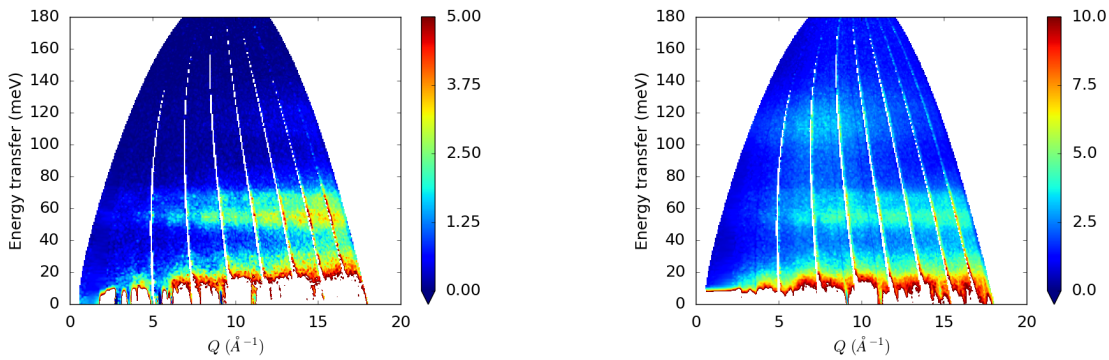


(a) (b)

Figure 4.4: Measured (blue) and calculated (green) PDOS for the bulk powder at $T = 300\,\mathrm{K}, E_i = 40$(a) and $200\,\mathrm{meV}$ (b). The damped harmonic oscillator model works well at lower energies, but is unable to fit the high energy data. IT should be noted that the weighting values of the one- and two-phonon component are fitted parameters in these plots.

4.3 Nanostructured ZnO

A comparison of the measured $S(Q,\omega)$ for the bulk and FSP powders can be seen in Fig. 4.5. There is a strong signal clearly visible at 120 meV at lower Q, between 5 Å^{-1} to 10 Å^{-1} for the FSP sample.



(a) (b)

Figure 4.5: Measured $S(Q,\omega)$ for bulk (a) and FSP (b). There is a clear signal at 120 meV present only in the FSP data.

4.3.1 Hydroxyl Contaminant

Examination of the FSP data shows significantly higher incoherent scattering, seen most clearly with slices through the elastic line. These slices can be seen for $T = 300, 500, 700$ and 900 K in Fig. 4.6, and show the difference is eliminated by $T = 900$ K.

A final measurement was performed on the FSP sample, at 330 K after the signal was eliminated, and shows the difference originally seen was no longer present. Comparison of the elastic slice with the bulk 300 K and FSP 330 K data shows little difference and can be seen in Fig. 4.7. This measurement was performed at 330 K as a compromise, as it was desired to keep the powder under vacuum making cooling to

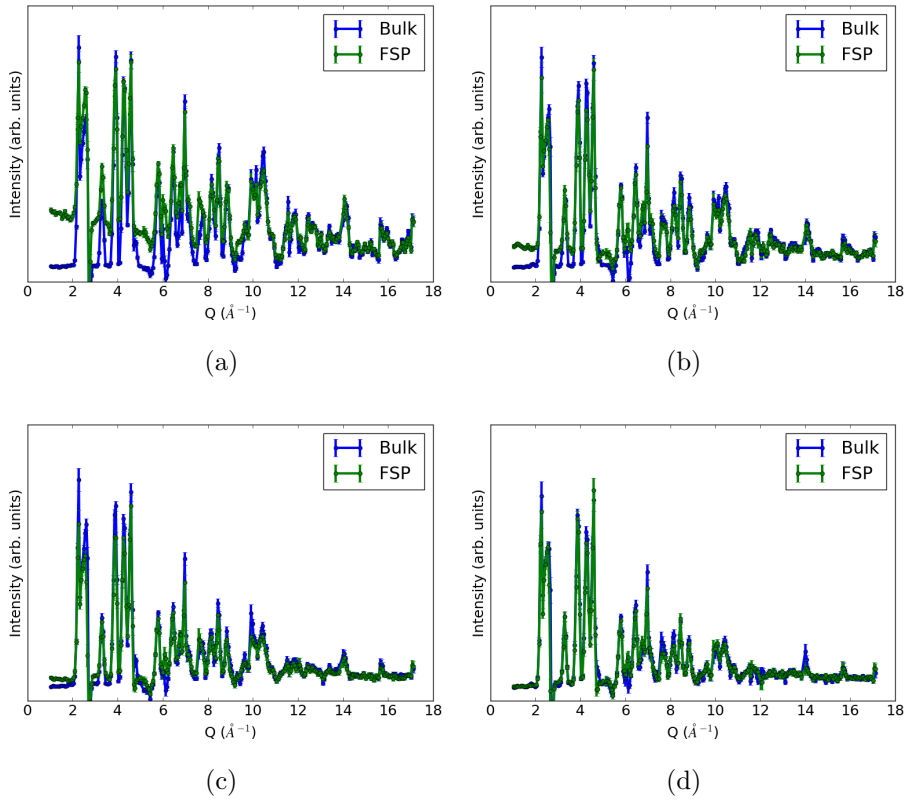


Figure 4.6: Elastic line measured at $T = 300$ (a), 500 (b), 700 (c) and 900 K (d). At low Q the increased incoherent signal is clearly present in (a), however this difference is eliminated by T =900 K.

room temperature infeasible during the remaining beam time.

The original FSP measurement appears to contain some contaminant removed by high-temperature annealing, which consequently leads to the FSP sample becoming very similar to the bulk. Understanding the measured FSP data requires correcting this contaminant. Unexpected hydrogen absorption and surface adsorption have been reported in metal-oxides, particularly in the case of nano-powders [53], and would explain the large increase in incoherent scattering.

The two room temperature FSP measurements were subtracted from each other to

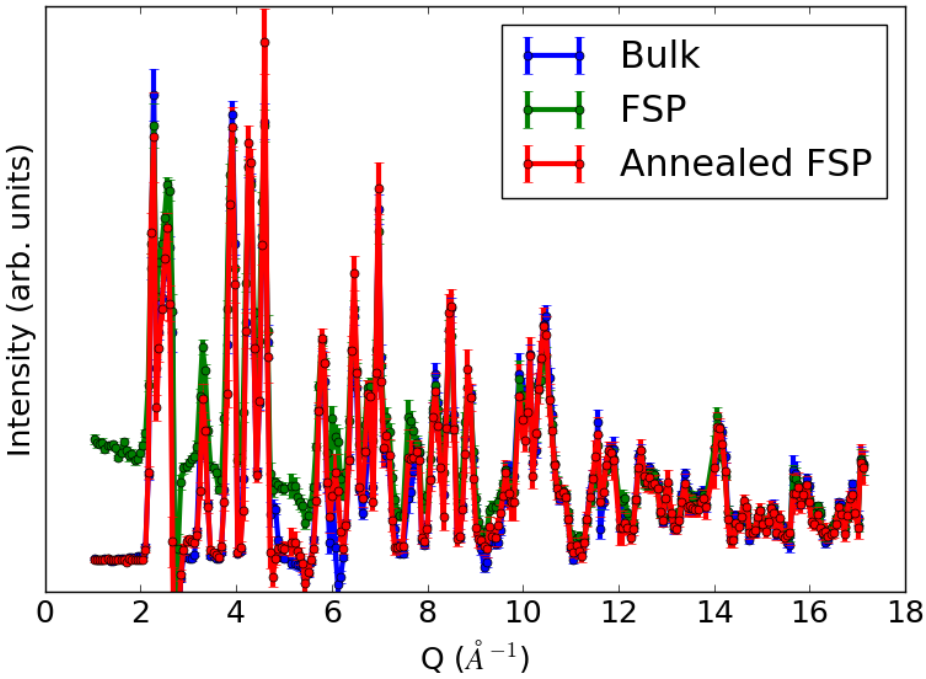


Figure 4.7: Elastic line measured at $T = 330$ (red) for the FSP powder after annealing at 900 K for 24 hours. The original measured data at 300 K can also be seen for FSP (green) and bulk (blue) for comparison.

obtain the difference due to annealing, and can be seen in Fig. 4.8. The dominating feature was identified as an in-plane bend of a surface zinc hydroxyl [54, 55] after discussions with Dr. Stewart Parker.

The incoherent scattering for a mode with energy, ω_0, can be calculated as:

$$S_{inc}(Q,\omega) \propto e^{-\langle u^2\rangle Q^2}\delta(\omega) + \langle u^2\rangle Q^2 e^{-\langle u^2\rangle Q^2}\delta(\omega-\omega_0) + ..., \tag{4.6}$$

where $\langle u^2\rangle$ is the mean square displacement [56].

An estimate of $\langle u^2\rangle$ can be obtained by fitting the Q dependence of the elastic scattering measured using the first term in Eq. (4.6). Similarly, the hydroxyl mode

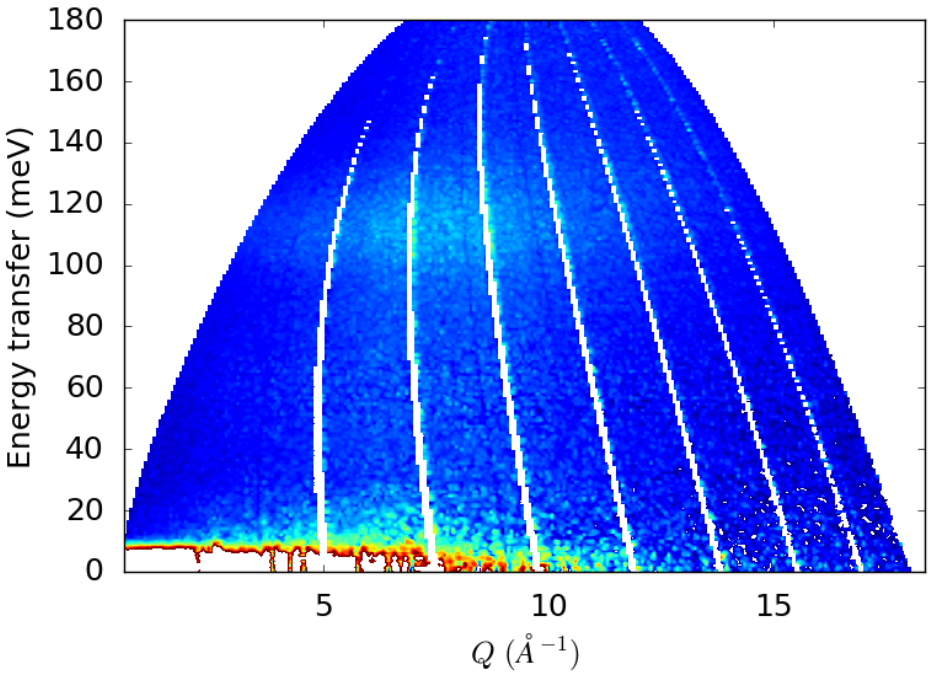


Figure 4.8: Result of subtracting $S(Q,\omega)$ measured for FSP before and after annealing.

can be accounted for by first determining ω_0, and then fitting the second term.

4.3.2 Hydroxyl Corrections

An energy slice, $\int_5^{11} S(Q,\omega)dQ$, was computed to obtain an estimate of ω_0. The slice can be seen in Fig. 4.9 and shows one well-defined peak, with $\omega_0 = 112.9(2)$ meV.

The mean square displacement, $\langle u^2 \rangle$, was fitted from the elastic data and a value of 1.59×10^{-2} Å^2 obtained. The fitted $\langle u^2 \rangle$ and ω_0 were used to calculate the inelastic slice, and is in excellent agreement with the data. Fits of the two constant-energy slices can be seen in Fig. 4.10.

With the hydroxyl contributions suitably fitted, it is possible to use the model to

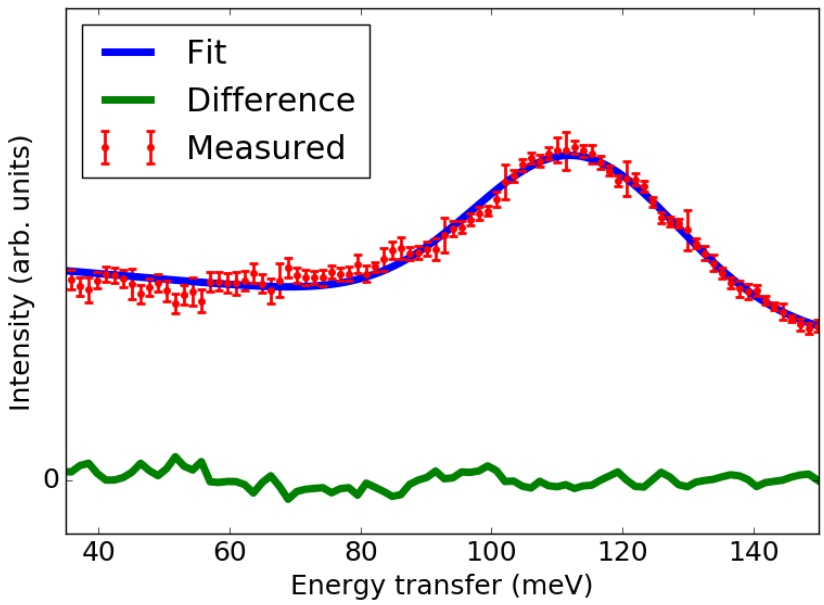
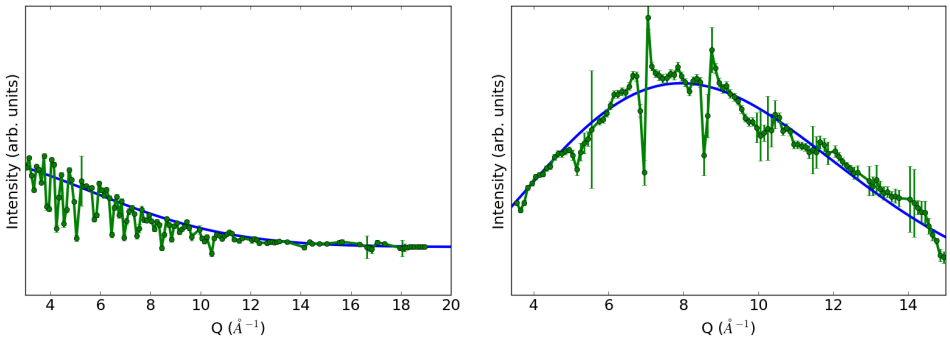


Figure 4.9: Energy slice at $Q = 8\,\text{Å}^{-1}$ with width $6\,\text{Å}^{-1}$ of the data shown in Fig. 4.8. The hydroxyl mode peak position is determined as 112.9(2) meV using a Gaussian fit with sloped background.

correct the originally measured $S(Q,\omega)$. The corrected 300 K $S(Q,\omega)$ can be seen in Fig. 4.11.

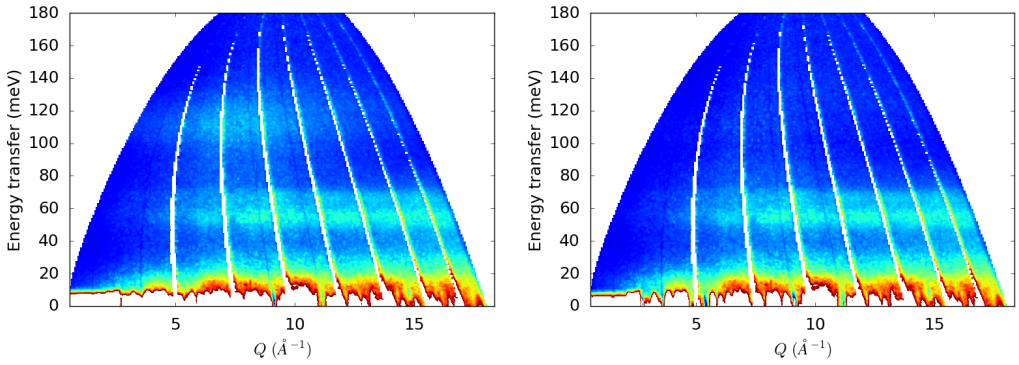
A comparison of the corrected PDOS for the annealed, original and bulk samples can be seen in Fig. 4.12. The corrected PDOS does show significant differences from both the bulk and annealed FSP powders most notably in additional broadening. The similarity in the FSP annealed data, also seen in Fig. 4.7, suggests this sample is now of similar crystallite size as the bulk sample, however it was not possible to characterise the FSP sample after annealing. A more careful investigation of the effects of annealing on crystallite size could lead to further insights into potential thermoelectric applications, as this suggests high-temperature use such as in car exhausts would be infeasible. Multi-phonon scattering seen around 115 meV, previously obscured by



(a) (b)

Figure 4.10: Slices of the contaminated FSP data for $E = 0$ (a) and 113 meV (b). Calculations for the fitted $\langle u^2 \rangle$ can be seen in blue.

the hydroxyl mode, is still visible.



(a) (b)

Figure 4.11: The 300 K data measured for the FSP sample before (a) and after (b) corrections for the zinc surface hydroxyls.

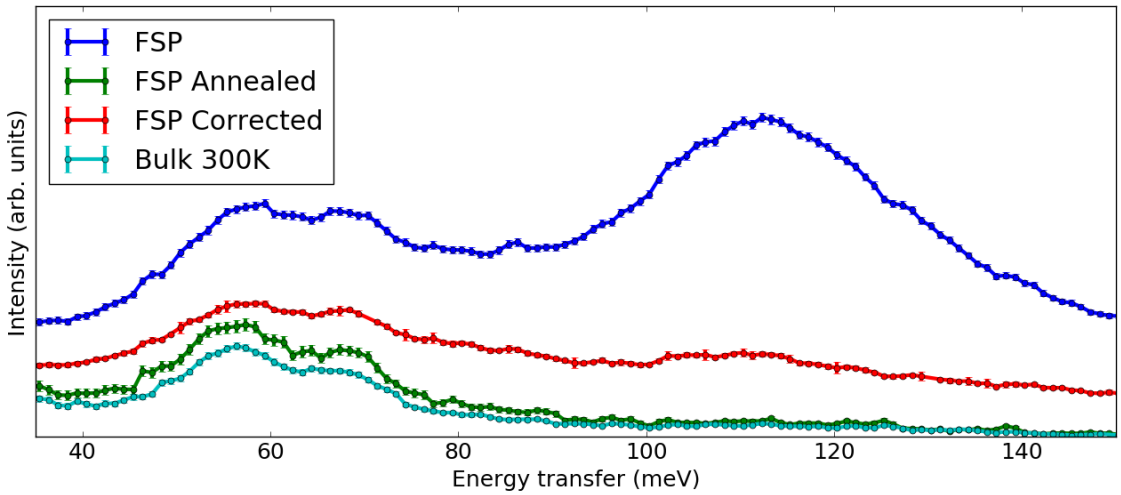
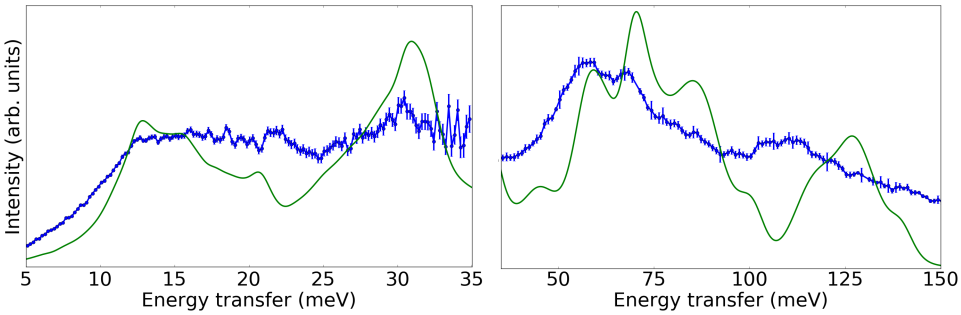


Figure 4.12: The PDOS for the FSP data after hydroxyl corrections. The FSP (blue), corrected (red), annealed (green) and bulk (cyan) PDOS can be seen for comparison. The FSP after annealing and the bulk samples both look very similar. The corrected PDOS still has some additional scattering at 115 meV and shows clear differences from the annealed measurement. It was unfortunately not possible to empirically characterize the annealed FSP sample, and it is only suspected the two are now very similar due to measurements of the elastic line.

4.3.3 Calculated PDOS

With these corrections it is now possible to model the PDOS, as done in the end of Section 4.2. A plot equivalent to Fig. 4.4 can be seen in Fig. 4.13. The model is unable to get good agreement with the FSP data, even for the high-resolution low energy data.



(a) (b)

Figure 4.13: Measured (blue) and calculated (green) PDOS for the FSP powder at $T = 300\,\mathrm{K}$, $E_i = 40$(a) and $200\,\mathrm{meV}$ (b).

4.4 Alternative PDOS Modelling

It was not possible to get good agreement with the high-energy data using the damped harmonic oscillator model for any values of Q_f for either sample. The $40\,\mathrm{meV}$ data is well-reproduced, however only for bulk. Alternative models are presented which describe the phonon broadening due to phonon lifetimes.

The simplest, somewhat crude model uses a single lifetime (henceforth the "fixed-lifetime model"). In this model it is assumed the lifetimes can be described by some average value, $\langle\tau\rangle$. Calculating the PDOS is equivalent to the steps described in Section 4.2, however instead of convolving with Eq. (4.4), a Gaussian was used. An

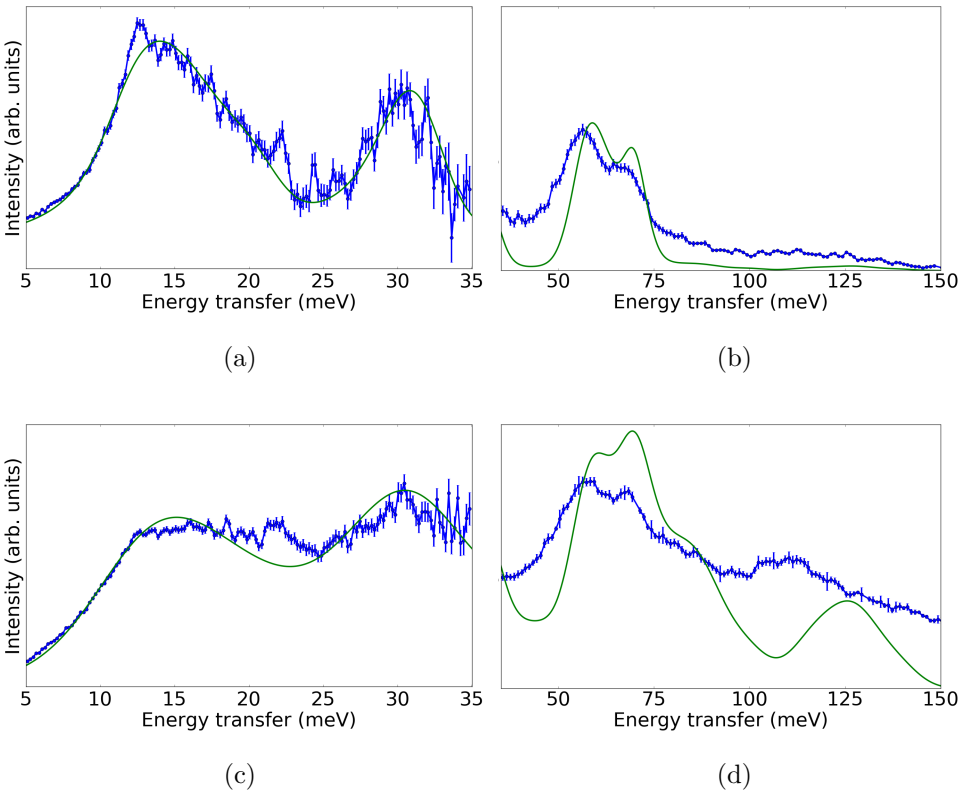


Figure 4.14: Measured (blue) and calculated (green) PDOS for the bulk (a, b) and FSP (c, d) powder at $E_i = 40$ (a) and 200 meV (b) using the fixed lifetimes model.

average phonon lifetime of 0.92 ps was obtained from this fit for the bulk powder, which is not far from the values obtained from the Merlin data.

Plots of this model for the Bulk, and FSP, samples can be seen in Fig. 4.14 for both incident energies. The fixed lifetime gives a reasonable fit for the 40 meV Bulk and FSP data, however it still struggles to fit the higher energy modes. At this point it is interesting to note that whilst the peak at 70 meV appears well aligned with the data, the multi-phonon peak measured is clearly softer. An anharmonic model for the multi-phonon scattering may provide better agreement.

A phonon's lifetime has associated with it a mean free path (MFP), $\bar{x}$, which can

be calculated from the group velocity as:

$$\tau(\vec{\boldsymbol{k}}, \nu) = \frac{\bar{x}}{v_g(\vec{\boldsymbol{k}}, \nu)}. \tag{4.7}$$

Group velocities can be determined *ab initio* from the gradient of the dispersion. From these considerations it is not surprising fits to the higher energy modes are less successful. The optic modes have low group velocities which may have very different lifetimes to those of the acoustic phonons.

To capture this a more sophisticated model, the fixed MFP model, was used. The suppression of thermal conductivity in the nanostructured powders is suspected to be due to finite-size effects, and so this can be described as the MFP being limited by the crystallite size: 15.3 nm for FSP and 117 nm for bulk.

In this model, the phonon broadening widths are based on lifetimes determined from the phonon's group velocity. As a result acoustic modes with large v_g have much shorter lifetimes than the optic modes. Results using the novel, fixed MFP, model can be seen in Fig. 4.15.

This model has the most consistent agreement, and is the only model to give reasonable fits for the 200 meV data. The acoustic modes with the greatest v_g are not well described in this model. Fitted parameters from these models can be directly related to the thermal conductivity, a major advantage over the model used in Section 4.2.

Attempts were made to go beyond the harmonic two-phonon approximation, however these calculations were difficult and unfruitful, and were not further investigated.

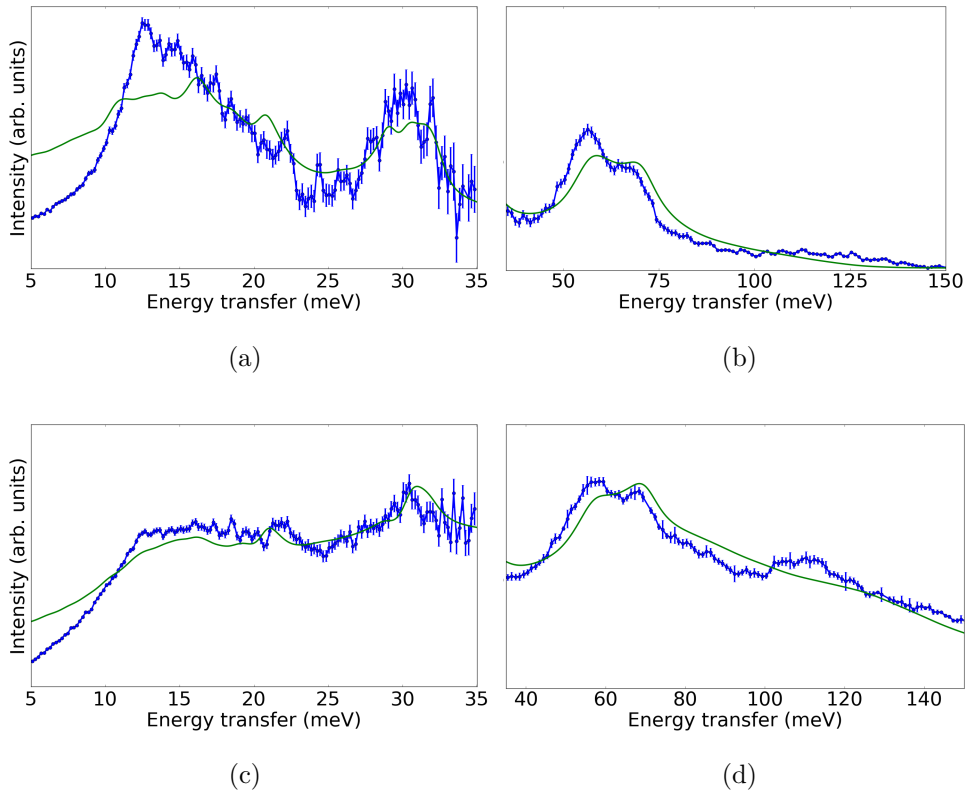


Figure 4.15: Measured (blue) and calculated (green) PDOS for the bulk (a, b) and FSP (c, d) powder at $E_i = 40$ (a) and 200 meV (b) using the fixed MFP model.

4.4.1 Calculating the Thermal Conductivity

The lattice contribution to the thermal conductivity can be calculated as:

$$\kappa_L = \sum_{\nu,Q} c_\nu(Q,T) v_g(Q,\nu)^2 \tau(Q,T,\nu) \tag{4.8}$$

where $v_g(Q,\nu)$ is the group velocity, $c_\nu(Q,T)$ the specific heat capacity and $\tau(Q,T,\nu)$ the lifetime of the ν-th phonon [57]. From the two models discussed in Section 4.4, we can approximate Eq. (4.8) in one of two ways using Eq. (4.7), i.e.:

$$\kappa_L \approx \sum_{\nu,Q} c_\nu(Q,T) v_g(Q,\nu)^2 \langle\tau\rangle \tag{4.9}$$

$$\approx \sum_{\nu,Q} c_\nu(Q,T) v_g(Q,\nu) \langle x\rangle \,, \tag{4.10}$$

where $\langle\tau\rangle$ and $\langle x\rangle$ are the fixed lifetimes or MFP from the model. The average lifetime, $\langle\tau\rangle$, must be determined empirically, however in the fixed MFP approach, $\langle x\rangle$ is simply a parameter, in this case taken to be the crystallite size. Thus the fixed MFP model provides an *ab initio* method for calculating the thermal conductivities.

The specific heat capacity can be calculated from the dispersion assuming Bose statistics, simply as:

$$c_{\vec{\boldsymbol{k}},\nu} = \frac{k_B}{V}\left(\frac{E(\vec{\boldsymbol{k}},\nu)}{k_B T}\right)^2 \frac{e^{\frac{E(\vec{\boldsymbol{k}},\nu)}{k_B T}}}{(e^{\frac{E(\vec{\boldsymbol{k}},\nu)}{k_B T}} - 1)^2}, \tag{4.11}$$

where $E(\vec{\boldsymbol{k}},\nu)$ is the eigenenergy of the ν-th phonon mode.

4.5 Single-Crystal Thermal Conductivities

Measurements of the thermal conductivity of single-crystal ZnO were performed *in situ* using the xenon-flash method described in Section 2.4 at Johnson Matthey with Dr. Chris Nuttall. The apparatus used provides measurements of the heat capacity and thermal diffusivity from which the thermal conductivity can be obtained.

Three small, thin ($1 \times 10 \times 10\,\mathrm{mm}$) single-crystals of ZnO, purchased from Good-Fellow, were measured. Two of the substrates were left as-grown with [001] and [100]

aligned normal to the square face. The third substrate, also with [001] normal, was given a O-annealing treatment described in more detail in Section 5.1.

4.5.1 Density Measurements

For these measurements the densities of the single-crystal substrates were measured using an Archimedes balance; for which a schematic diagram can be seen in Fig. 4.16.

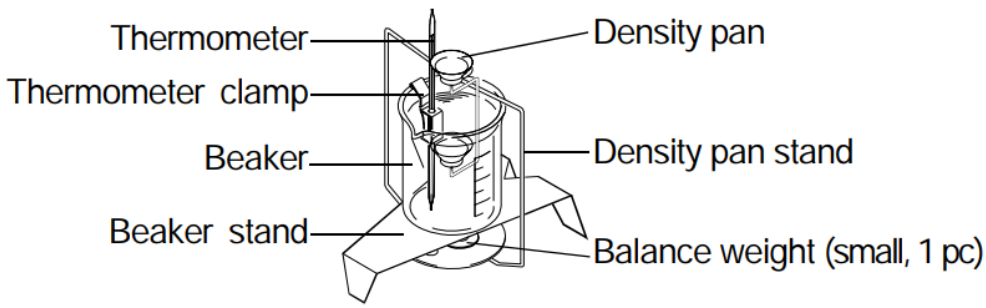


Figure 4.16: Schematic diagram of an Archimedes Balance used for determining the density of materials. There are two trays to place the sample, labelled the density pan. One tray places the sample above the liquid whilst the other is submerged. Temperature readings from the thermometer are used to obtain the density of the liquid, and is the largest source of error in these measurements. [58]

This technique exploits Archimedes' principle, which states "a body immersed (partially or fully) in a liquid (or gas) is subject to an upward force equal to the weight of the liquid (or gas) it displaces.", in order to determine the densities of solids [58]. A well-characterised liquid, thermometer and set of measuring scales are used to measure the weight of a body both in air and in the liquid. To measure the densities of the ZnO subtrates, a beaker of ethanol at 19 °C was used. All substrates thicknesses were measured with a vernier caliper at 6 different positions in different directions, and found to be uniform and very close to the nominal thickness, with a measured

range of thicknesses from 0.987 mm to 1.020 mm. Measured thicknesses and densities can be seen in Table 4.1.

Sample	Density (g cm^{-3})	Thickness (mm)
As-grown [100]	5.56(2)	1.020(6)
As-grown [001]	5.55(5)	0.987(5)
O2 [001]	5.54(6)	0.988(4)

Table 4.1: Densities and thicknesses of the ZnO substrates measured using vernier callipers and the Archimedes balance. Reported values are the aggregate of 6 measurements.

4.5.2 Heat Capacity

The LFA 500 is not optimised for measurements of the heat capacity, particularly for translucent samples like the single crystals used [35]. Nonetheless the heat capacity is a straightforward *ab initio* calculation and this provides another test of the first-principle calculations. The heat capacity measured, C_P, and calculated, C_V, for bulk ZnO can be seen in Fig. 4.17 and shows reasonable agreement to within about 10%.

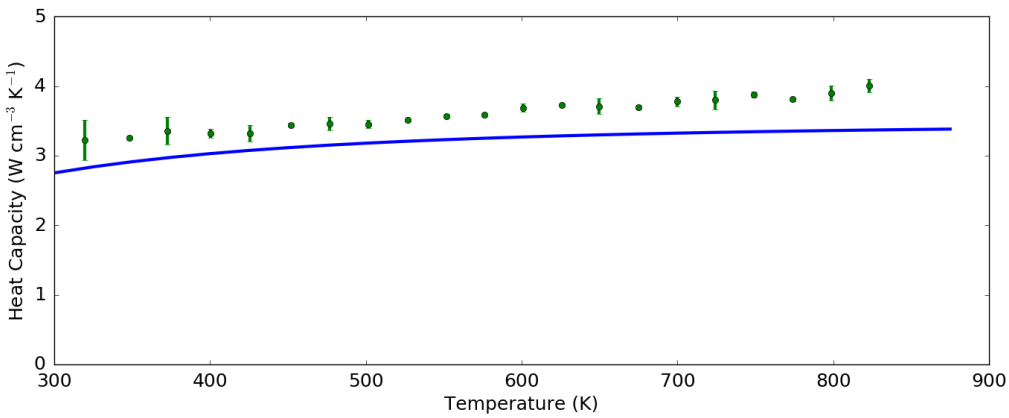


Figure 4.17: ZnO Heat Capacity measured using the LFA 500 (green), and calculated (blue) from first principles.

4.5.3 Thermal Conductivity

The thermal conductivities for the [001], [100] and O_2 (also [001]) substrates can be seen in Fig. 4.18. Whilst there is a clear systematic increase in measured thermal conductivities in the [001] direction, annealing in oxygen appears to have little effect on the thermal conductivity. Hence, intrinsic defects have very little effect, but the thermal conductivity is anisotropic.

In addition, Fig. 4.18 includes a number of thermal conductivities reported in the literature. The bulk, single-crystal measurements performed are consistent with those reported for polycrystalline ZnO [36], which lie between the [001] and [100] measurements due to powder averaging. Calculations of the bulk thermal conductivity at 300 K using the two models in Eq. (4.9) can also be seen as stars. The fixed lifetime model gives a reasonable estimate of the thermal conductivity, whilst the *ab initio* model performs extremely well. The measurement of the nanostructured sample reported with ultra-low thermal conductivity [19] is included as a cross, and the calculation using the *ab initio* model is in excellent agreement.

It was not possible to perform measurements of the FSP thermal conductivities, since the FSP sample turns into the bulk after annealing, and measurements require sufficiently densified pellets [35] which would be fabricated using a hot press, inducing annealing. Calculating the thermal conductivity using the fixed MFP model is not helpful without benchmarks, and comparisons with the measured PDOS, as done for the 300 K data, are not feasible without additional measurements of the powder after annealing to correctly subtract the hydroxyl contaminant and obtain sensible estimates of $\langle x \rangle$.

It is not possible to calculate the temperature dependence of the thermal conduc-

tivities from these data using the fixed lifetime model, as the instrumental resolution and flux make it difficult to fit phonon lifetimes sufficiently accurately. The phonon lifetimes extracted from the Merlin data in Section 3.3 were also used to obtained estimates of the thermal conductivity at 300 and 550 K. The room temperature value is in good agreement, however this decreases at higher temperatures. It is believed that at higher temperatures there is more multi-phonon scattering, which this model does not include. Both values obtained from the Merlin data give good qualitative agreement, which is as good as one can expect from a single phonon. It would be more reliable to estimate the average phonon lifetime from many more phonons at different locations on the dispersion.

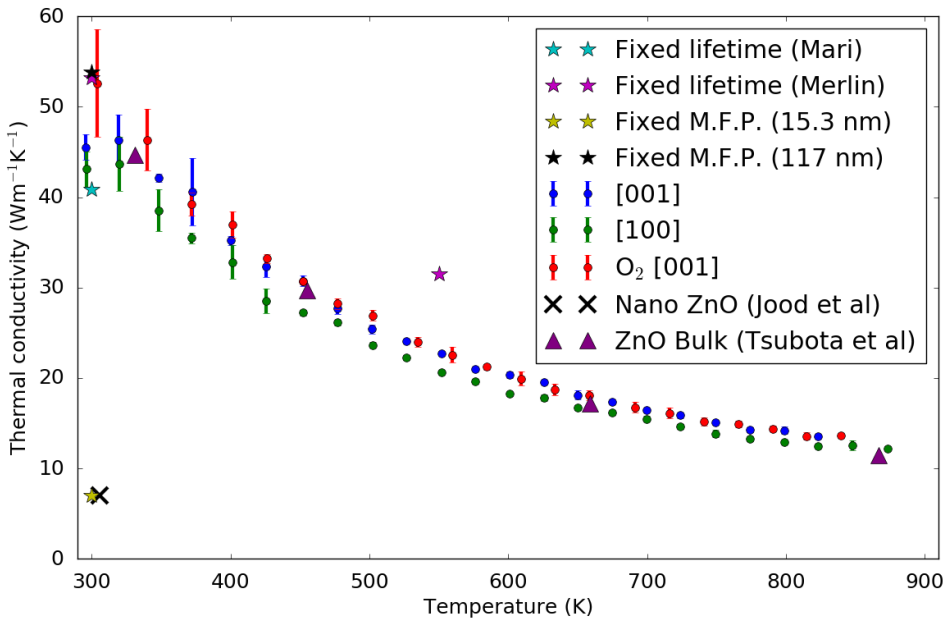


Figure 4.18: Thermal conductivities of the as-grown substrate along [100] (green); [001] (blue); and the oxygen annealed substrate (red), also along [001], measured with the LFA-500. Measurements reported for the bulk (triangles) [36] and nanostructured (X) [19] samples can be seen for comparison and show these measurements are consistent with those reported. Calculations of the thermal conductivities can be seen (stars) using the fixed lifetime model, with lifetimes extracted from the MARI (cyan) and Merlin (purple) data. The Fixed M.F.P. models for bulk (black) and nanostructured (yellow) ZnO can also be seen.

4.6 Summary

Two powders were prepared to investigate the effect of nanostructuring on the lattice vibrations, and hence thermal conductivities. One powder, named bulk, with nominal crystallite size of 117 nm was compared with a nano-crystallite powder, named FSP, with nominal crystallite size 15.3 nm.

Initial measurements show the presence of additional incoherent scattering in the nano-structured powder which was attributed to zinc surface hydroxyls. These impurities can be completely removed by annealing under vacuum at 900 K for 24 hours, however this also induces growth of crystallites leading to a very similar PDOS to the bulk sample. Since any heat treatment changes the nanostructured grain size, leading eventually towards bulk properties, this is not promising for high temperature thermoelectric applications, however this does not rule out uses near room-temperature, for example personal monitors.

The hydroxyl contaminated data was corrected for by modelling the hydroxyl contribution as incoherent scattering from a harmonic oscillator. The corrected PDOS show significant differences from the same sample after annealing at 900 K.

Attempts to fit the PDOS using the dampled harmonic oscillator model work very well for the low-energy, bulk data, however this model was unable to correctly describe any of the other datasets, including the low-energy FSP data.

A novel model, based on considerations of the phonon lifetimes from a mean free path due to crystallite size, was compared and this fixed MFP model shows the most consistent agreement with all the measured PDOS. The fixed MFP model allows calculation of the thermal conductivities which are in excellent agreement with these measurements, and other similar measurements reported in the literature. It is

difficult to perform these calculations at higher temperatures as it is only possible to analyse the corrected, room temperature FSP data due to the surface hydroxyls. Since the fixed MFP model gives such good agreement with measurements, this suggests the finite-size effects dominate for samples with ultra-low thermal conductivity.

It is worth pointing out that it is possible to lower the thermal conductivity further by doping with additional benefits for electrical aspects of the thermoelectric properties [19]. However, given that doping is accompanied by a change in the aspect ratio and a reduction in the average grain size, it is likely that nanocrystal size effects are still the dominant factor in determining the thermal conductivity.

Measurements of the thermal conductivities for the FSP powder could provide further insight, however it is difficult to obtain sufficiently densified pellets from the powder without hot-pressing, which could induce similar growing of crystallites as seen when annealing at high temperature to remove surface hydroxyls.

The thermal conductivities of thin single-crystal substrates were measured, and showed oxygen annealing has little effect on the thermal conductivity. Thermal conductivities were calculated for the bulk sample using a crude model of a single average lifetime and show a remarkably good agreement with experimental data given the simplicity of the model.

The observed red-shift of the multi-phonon scattering is a clear indication of anharmonic effects, and it is likely that these would need to be taken into account to obtain a more accurate model of the thermal conductivity. However, the harmonic model employed here is remarkably successful in describing the experimental thermal conductivity, and is sufficient to demonstrate the dominance of crystallite size over intrinsic defects.

Chapter 5

Defects in ZnO

ZnO is an important semiconductor beyond thermoelectric applications, and is widely used in piezoelectric transducers, acoustooptic media, conductive gas sensors, transparent conductive electrodes and varistors [59]. It can be easily doped n-type, but p-type doping is difficult [6]. This has been attributed to the nature of its intrinsic defects [7].

The crystal structure and stoichiometry can be determined using diffraction. Diffuse scattering can be used to better understand defects in the crystal, for example whether vacancies are randomly distributed or prefer to form superstructures in a non-stoichiometric system [11]. One-phonon excitations also give rise to a diffuse signal in Laue neutron time-of-flight experiments [60], which complicates analysis as separating these two components is not trivial.

The experimental procedure for SXD measurements is outlined in Section 5.1. The two sources of diffuse scattering, structural defects and mislabelled inelastic processes, are examined in Sections 5.2 and 5.3. The intrinsic defect structure determined using a combination of *ab initio* and semi-empirical classical models is shown in Section 5.4.

5.1 Experimental Procedure

Two high quality, thin ($2 \times 8 \times 8$ mm) single crystals were purchased from Goodfellow. One crystal was annealed in an oxygen atmosphere at 700 K for 24 h, hereafter named O-annealed , to investigate potential oxygen defects reported from first-principle calculations [9]. The other, left as purchased, was named as-grown . The samples were measured at two temperatures: $T = 30$ and 300 K.

The large crystal measured in Section 3.2 was also measured on SXD at three temperatures: $T = 300, 600$ and 900 K. Inelastic scattering increases with temperature and structural diffuse scattering remains roughly constant, provided the concentration of defects is fixed, the diffuse intensity should decrease slightly with the Debye-Waller factor. Thus measuring the temperature dependence can aid distinguishing inelastic features from those coming from defects.

Samples were mounted on aluminium pins and secured using a small quantity of thin aluminium tape for the substrates, and wire for the large crystal. The sample mount was then shielded using cadmium for $T = 30$ and 300 K, and gadolinium for furnace measurements. For the thin crystals, samples were cooled to 30 K using a CCR. Measurements of the large crystal were performed in a furnace. For all measurement configurations, a null scattering V sample had been previously measured to correct for incident flux. Background measurements were performed on the empty sample mount, including fastening aluminium wire. Typical measurements consist of five or more orientations for at least three hours per orientation.

These data were processed using SXD2001, software provided by the crystallography group at ISIS [61]. This software processes the raw data as follows:

1. Normalise the raw data, V/Nb and background measurements by beam current.

2. Subtract normalised background from the normalised data.

3. Normalise the background-subtracted data with the V data.

4. Pixel detectors' angular positions and t.o.f. are mapped to reciprocal space using the modelled geometry of SXD.

5. The volumetric data is exported to allow further analysis and visualisation using other programs.

For each sample measured at a given temperature, the orientation which led to the brightest peaks in the high-resolution detectors was used to calculate the **UB** matrix. Peak positions were determined using a 3D Gaussian ellipsoid fit and then used to refine both the instrument model and **UB** matrix. This **UB** matrix can be used for the other orientations using a suitable transformation matrix followed by an additional refinement iteration to account for any difference between the nominal and actual orientation.

An obtained **UB** matrix was determined to be suitable once the following conditions were met:

1. It simultaneously indexes the majority of peaks measured in all detectors.

2. The obtained lattice parameters are suitably close to $a = b = 3.2\,\text{Å}, c = 5.2\,\text{Å}, \alpha = \beta = 90°, \gamma = 120°$.

3. The above conditions are satisfied starting from the same **UB** matrix for all orientations measured of that sample at that temperature.

With a suitable **UB** matrix, the measured elastic scattering can be plotted in reciprocal space and compared with calculations. The measurements from different

orientations are combined together and symmetrised, a process where symmetrically equivalent points are folded onto each other to improve counting statistics. Integrated Bragg peak intensities can be used to perform structure refinements to determine the stoichiometry of ZnO, as well as to infer the presence of additional scatterers at other sites in the unit cell. Structural diffuse scattering can then give deeper insight into the nature of these defects.

5.2 Inelastic Scattering on SXD

5.2.1 Origin of Inelastic Scattering

The inelastic scattering detected on SXD, and other similar diffractometers, has been well explained in [60] and is detailed here for reference. The geometry of a given detector pixel is defined by the distance to the sample mount and the angular position of the pixel: the longitude, δ; and latitude, ν. Let the direction of the neutron beam be along $\vec{\boldsymbol{y}}$, such that:

$$\vec{\boldsymbol{k}}_i = \frac{2\pi}{\lambda_i}\begin{pmatrix}0\\1\\0\end{pmatrix}, \qquad \vec{\boldsymbol{k}}_f = \frac{2\pi}{\lambda_f}\begin{pmatrix}\sin\delta\cos\nu\\ \cos\delta\cos\nu\\ \sin\nu,\end{pmatrix} \tag{5.1}$$

where λ_i, λ_f are the incident and final wavelengths respectively and, by definition, equal for an elastic scattering event.

For a single crystal the scattering vector can be obtained as:

$$\vec{\boldsymbol{Q}} = \vec{\boldsymbol{k}}_f - \vec{\boldsymbol{k}}_i = 2\pi[\mathbf{T}][\mathbf{UB}]\begin{pmatrix}h\\k\\l\end{pmatrix}, \tag{5.2}$$

where $\mathbf{T}$ is a transformation matrix describing the goniometer settings, $\mathbf{UB}$ the orientation matrix of the crystal, and h, k, l the Miller indices.

Since SXD is a time-of-flight diffractometer, multiple different wavelengths are measured simultaneously. Any individual neutron in the beam, labelled i, travels from the source to the sample in time $t_{i,1}$, after which it is scattered into a specific detector. A signal occurs in a detector pixel after some additional time, $t_{i,2}$, yielding a total travel time $t_i = t_{i,1} + t_{i,2}$. The de Broglie wavelength can then be identified, since:

$$E = \frac{mv_i^2}{2} = \frac{h^2}{2m}\frac{1}{\lambda_i^2}, \tag{5.3}$$

where m is the neutron mass and E_i, v_i, λ_i the energy, velocity and wavelength of any particular neutron.

Neutrons measured on SXD are placed into histograms based on their time-of-flight with bin-widths of 1 ms, unlike LET which uses event-mode to not require time-binning at the expense of larger datasets. Each area detector then has a histogram of measured intensity, which can be mapped to a volume of reciprocal space and indexed in terms of (h, k, l) using Eqs. (5.1) and (5.2).

These arguments all hold given the measured scattering is strictly elastic, which unfortunately is not the case. Inelastic scattering can still occur in the sample and will lead to changes in the time-of-flight, due to the experimental setup this change in time-of-flight is indistinguishable from elastic scattering with different wavelength.

These mislabelled neutrons lead to additional, spurious features in the diffraction pattern, often named thermal diffuse scattering [60]. It is important to account for this additional signal in order to properly isolate and analyse diffuse scattering from

structural defects. It is not feasible to extract the phonon eigenvectors from these measurements, however it is possible to calculate what the diffuse signal would look like given a sufficiently well-converged CASTEP calculation and accurately refined **UB** matrix.

Using the results of the calculation of the phonon dispersion from Section 3.1, a program produced by Matthias Gutmann was used to calculate this scattering using the geometry of SXD. The details of the calculational method can be found in ref [60] and goes outside the scope of this thesis. However, it is important to note that these calculations are both extremely computationally expensive and sensitive to the crystal orientation and scattering geometry.

5.2.2 Measured Inelastic Scattering

The measurements yield large amounts of volumetric data, and 2-dimensional slices were plotted for visualisation. Arc-like features characteristic of inelastic scattering can be seen in Fig. 5.1, which shows the plane $(h, k, 4)$ for the thin crystal annealed in oxygen, measured at 30 K. The reason why arc-like features emerging from Bragg reflections is expected for inelastic scattering is because the $1/\omega$ factor in the expression for the phonon structure factor in Eq. (2.32) is largest here.

In contrast, the same plane can be seen in Fig. 5.2 for the as-grown sample, and does not have these characteristic arc-like features. In this representation of the data, the position in $\vec{\boldsymbol{Q}}$ of the inelastic scattering depends upon the particular instrumental geometry and the orientation of the sample. Hence it is not possible to recombine data according to the underlying symmetry of the reciprocal lattice for inelastic scattering. Great care was taken to ensure the orientation of samples were the same for different

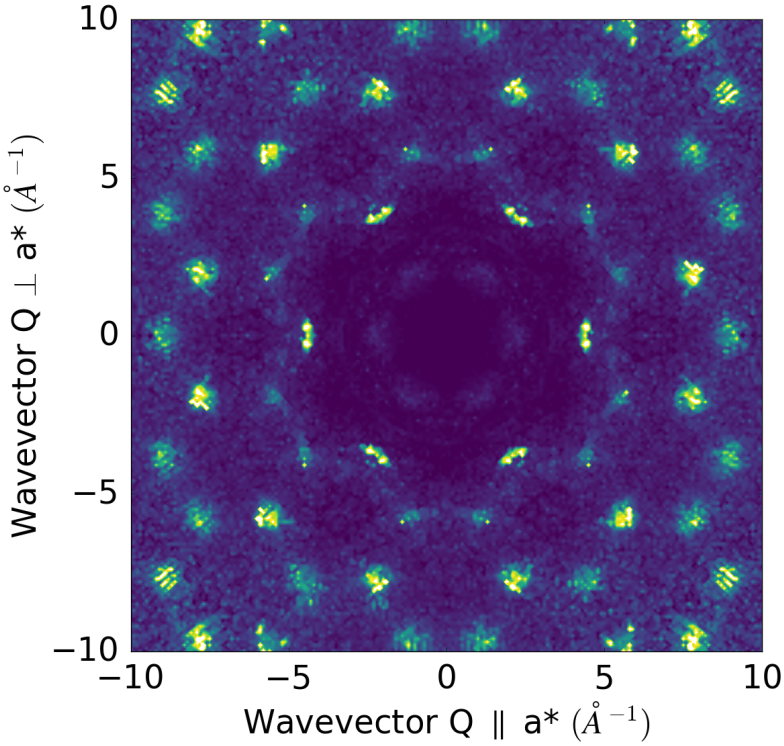


Figure 5.1: Scattering measured in the $(h, k, 4)$ plane for the O-annealed sample at $T = 30\,\mathrm{K}$. There are arc-like features characteristic of thermal diffuse scattering extending out of Bragg peaks. Axis units are $\mathrm{\AA}^{-1}$.

temperatures, so that these features could be compared.

Whilst these arc-like shapes are typical of thermal diffuse scattering, their presence alone is insufficient to claim they are inelastic in origin. First-principles calculations that reproduce these signals can give a high level of confidence that the signal is, indeed, inelastic. Calculating the inelastic contribution from first-principles is a formidable challenge [60], so the annealed data was examined in order to find a region with strong, characteristic diffuse scattering that varies with temperature in a single detector. Calculations for the specific orientation and detector can be seen in Fig. 5.3

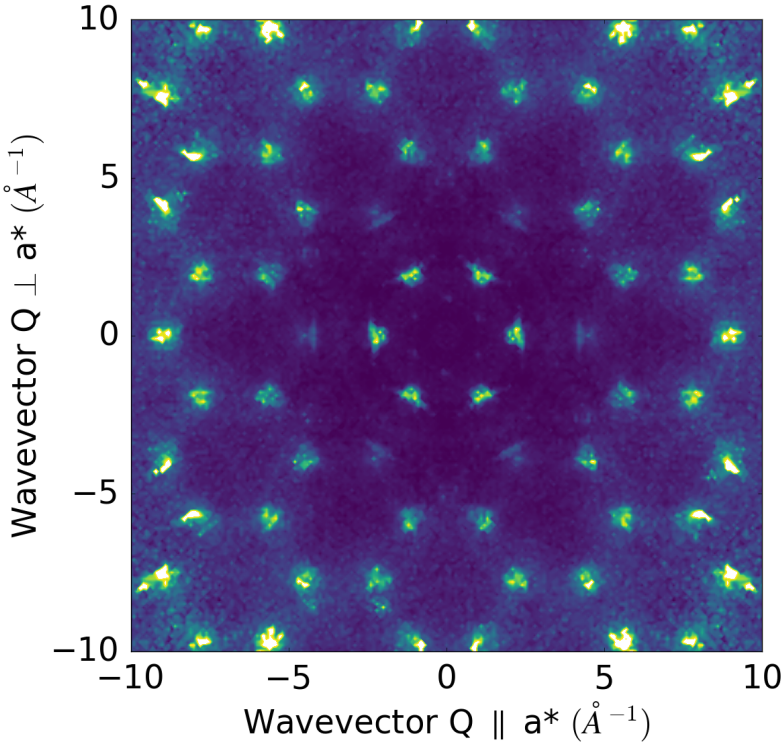
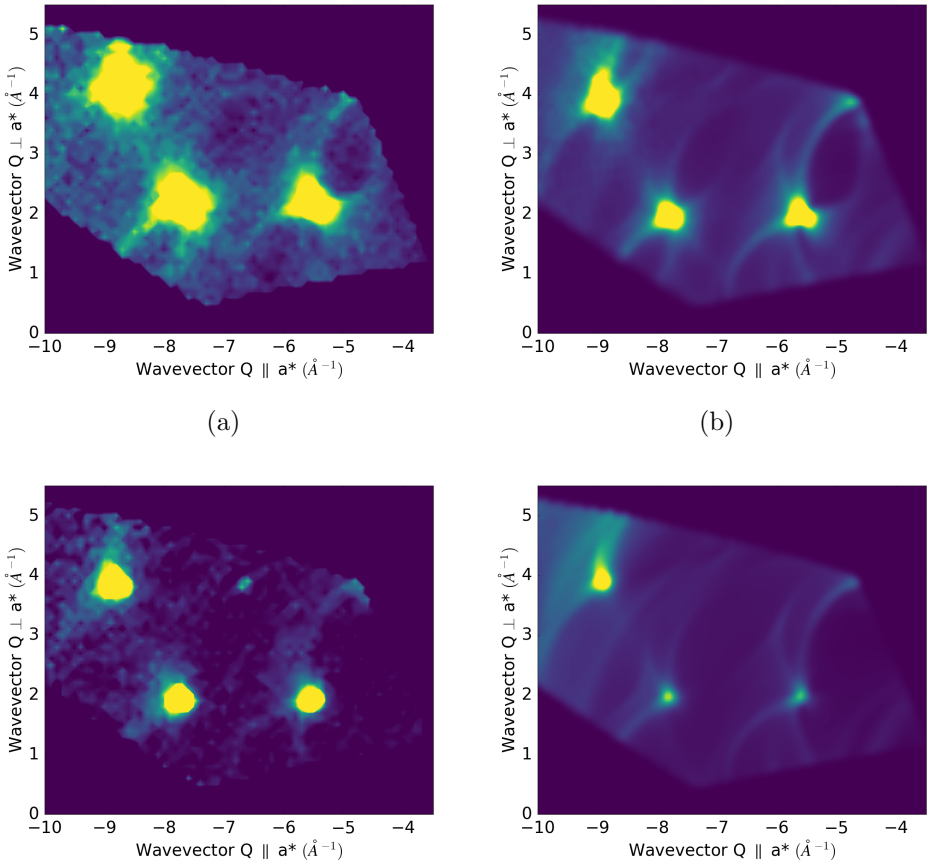


Figure 5.2: Scattering measured in the $(h, k, 4)$ plane for the as-grown sample at $T = 30\,\mathrm{K}$.

for both temperatures. At 300 K additional arcs appear extending out of Bragg peaks, which show neutron energy-gain scattering events forbidden at 5 K due to unoccupied modes.

The diffraction pattern measured for the large crystal in the same region as Figs. 5.3a and 5.3b at $T = 300$ and 900 K can be seen in Fig. 5.4 and shows similar characteristic features.

Features present only in the thin, oxygen-annealed sample are also visible in the larger, as-grown sample. Calculations of all detectors were performed at $T = 300\,\mathrm{K}$, and can be seen in Fig. 5.4c.



(a) (b) (c) (d)

Figure 5.3: Inelastic scattering in the $(hk3)$ plane of ZnO. The data measured at $T = 300\,\mathrm{K}$ (a) has additional arcs not seen at $T = 30\,\mathrm{K}$ (c) due to the occupation of states. These arcs are well-reproduced from first-principles calculations (b, 300 K; d, 30 K).

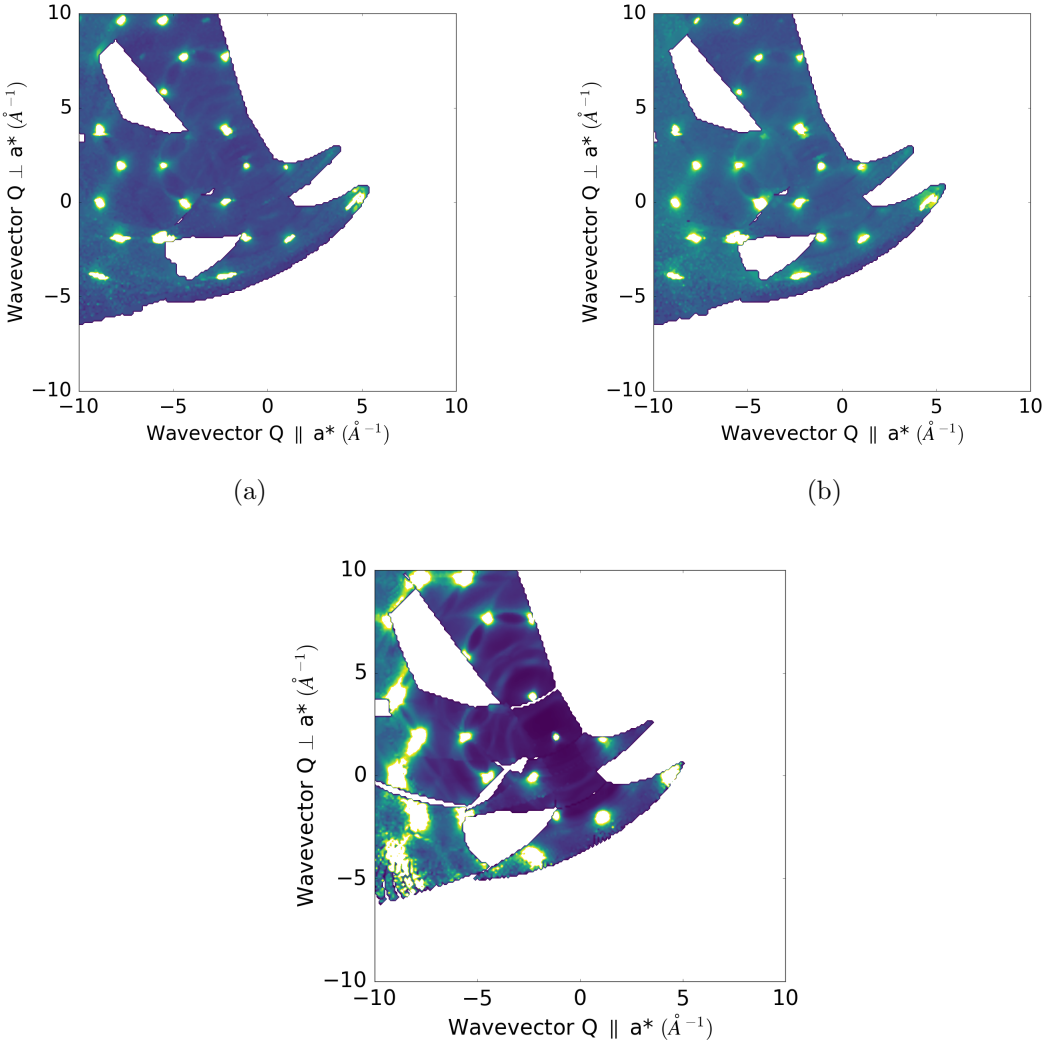


Figure 5.4: The diffraction pattern measured (a, 300 K; and b, 900 K) and inelastic contribution calculated (c, 300 K) in the $(h, k, 4)$ plane for the large, as-grown sample. Arc-like features are well reproduced, but diffuse features remain unaccounted for, for example at $(-1, 2, 4)$. The flat background is noticeably higher in the measured data, suggesting the presence of incoherent scatterers.

5.3 Structural Diffuse Scattering

The data for the as-grown sample contains diffuse scattering not accounted for by inelastic scattering, for example at $(-1, 2, 4)$ in Fig. 5.4. A semi-classical 'balls-and-springs' model was used to calculate the structural diffuse scattering. A C++ program was written which allows investigating potentials defects and implements the 'ball-and-spring' model, which is based on the approach used in ref [11].

5.3.1 Defect Modelling

Given a starting crystal, a large supercell is constructed from periodic repetitions of the unit cell. Atoms are modelled as points connected by springs to their nearest, and next-nearest, neighbours. Atoms are defined by their species, position, occupancy and, optionally, charge. Defects – such as vacancies, inclusions and/or displacements – can be inserted into the supercell which is then allowed to relax following a 'balls-and-springs' model. A trial displacement is selected at random and the change in Hooke energy computed as a sum over the displaced atoms and its neighbours, labelled ν, as:

$$\Delta E = \sum_{\nu} \frac{1}{2} k_{\nu} (\Delta x_{\nu})^2, \tag{5.4}$$

where k_{ν} is the spring constant between the displaced atom and its neighbours, and Δx_{ν} the displacement of the atom from its equilibrium position. The trial displacement is then accepted or rejected using the typical Monte-Carlo Metropolis condition [62].

In the original model presented in ref [11], the spring constants were free variables in the model that were fitted empirically. In this thesis, the model was modified to

utilize the calculated phonon modes used extensively in Chapters 3 and 4. With this modification, the force constants become parameters, obtained from DFT calculations reducing the degrees of freedom of the model. After the spring relaxation, the elastic scattering is calculated using Eq. (2.19). The force constants obtained for these calculations are listed in Appendix A.

The O-annealed sample shows mostly inelastic features, whilst the as-grown has additional structural diffuse scattering which obscures the characteristic arcs from inelastic scattering. The fact that oxygen annealing appears to eliminate the structural diffuse scattering suggests that oxygen vacancies are the dominant defects. The stability of oxygen vacancies is supported by first-principles calculations [63].

The introduction of random oxygen vacancies in the lattice with no other changes leads to completely flat diffuse scattering. Additional displacements of nearby ions are required to give diffuse scattering peaks. O vacancies typically induce displacement of nearby ions due to the change of Coulomb field [11].

The C++ program written was designed to be flexible for a number of potential defects; it can apply a number of arbitrary changes to the supercell, called *mutations*, which can have side-effects described by other mutators that are chained together. The crystal is initialised by specifying atomic species and position to create an instance of the `Crystal` class. From the unit cell, helper methods can be used to create a `SuperCell` instance which duplicates the unit cell for a number of desired repetitions. The `CrystalMutator` abstract class allows flexible introduction of defects. The *mutator* classes implement two methods: `bool filter(Atom)` and `bool process(Atom, SuperCell *)`, which define for which atoms this mutator is valid, and how to perform the mutation. When performing mutations and relaxations, the `EfficientCrystalRelaxor` can be used, which pre-indexes nearest neighbours to min-

imise computational cost. Example usage, and the C++ source code, can be found in Appendix C.

5.3.2 Comparison with Measurements

Calculations were performed using a supercell of $64 \times 64 \times 64$ unit cells. O atoms were selected at random to be removed, and nearest neighbours displaced away from the vacant site. Calculations were performed for oxygen occupations of 5% vacancies with initial displacements of 0.1 Å before relaxing, values chosen from structural refinements presented later in this chapter. The same plane as in Fig. 5.2 can be seen in Fig. 5.5 and shows distinct wall-like features, including the diffuse features around the Bragg peak at $(-1, 2, 4)$. The structural diffuse scattering lies in the same regions as the inelastic scattering, for example the features at high $\vec{\boldsymbol{Q}}$, which makes separating the two more challenging.

Inclusion of H, simply by adding it to assembly of atoms without additional consideration of displacements, does not lead to noticeable changes in the diffraction pattern. This is due to these calculations not taking incoherent scattering into account, and the hydrogen inclusion not leading to significant distortions of the lattice after the Monte-Carlo relaxation.

These wall-like features are similar to the scattering observed in the as-grown thin samples shown in Fig. 5.2. The non-symmetrised plot of the $(hk0)$ plane can be seen in Fig. 5.6 for the as-grown and O-annealed samples at 300 K. The as-grown sample shows strong diffuse scattering, particularly at high Q, however it lacks the characteristic arcs which can be seen much more clearly in the O-annealed sample.

1-dimensional cuts allow a more quantitative analysis. The integrated intensity for

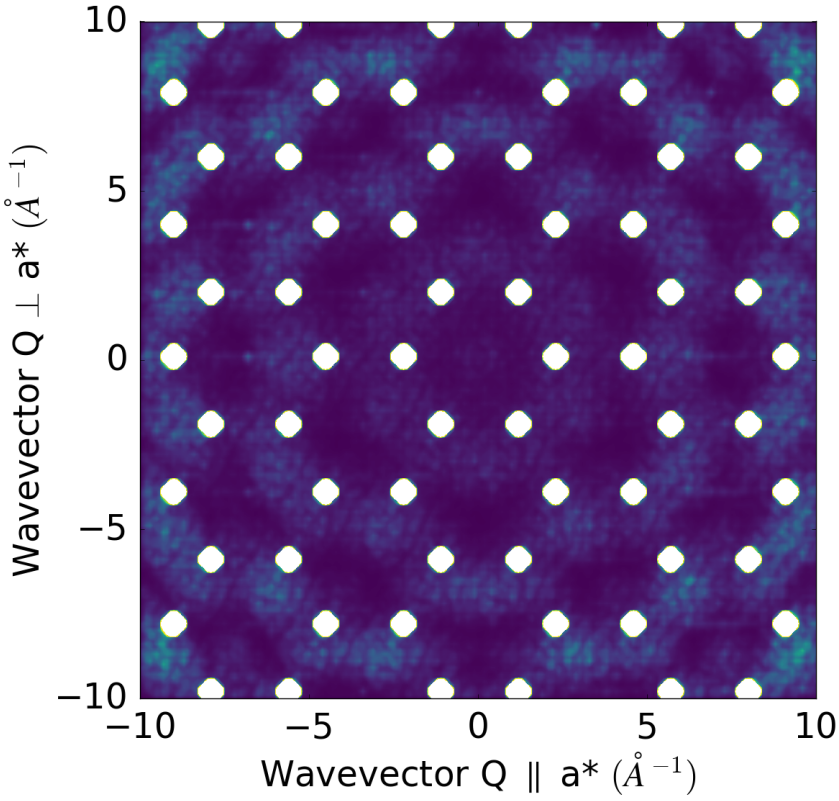
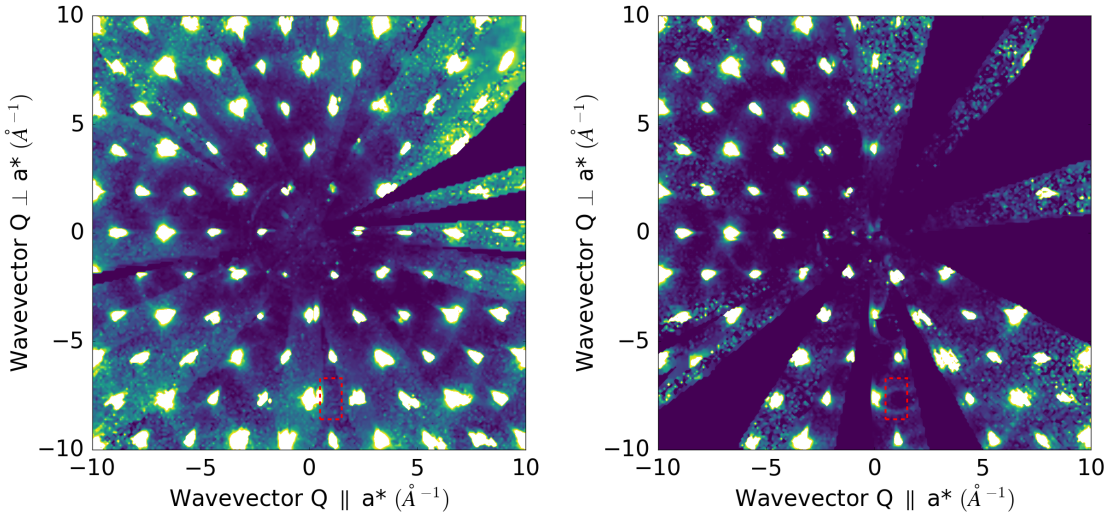


Figure 5.5: Structural diffuse scattering calculated for the $(h, k, 3.7)$ plane for ZnO with 5% O vacancies and nearest-neighbour displacements of ∼0.1 Å. Wall-like features can be seen between Bragg peaks.

the dashed red box can be seen in Fig. 5.7 for the as-grown and O-annealed samples at $T = 30$ and 300 K. The O-annealed data clearly shows the presence of an additional signal at 300 K due to occupied phonon modes. In contrast these peaks are not visible in the as-grown data, instead the overall intensity is increased due to structural diffuse scattering.

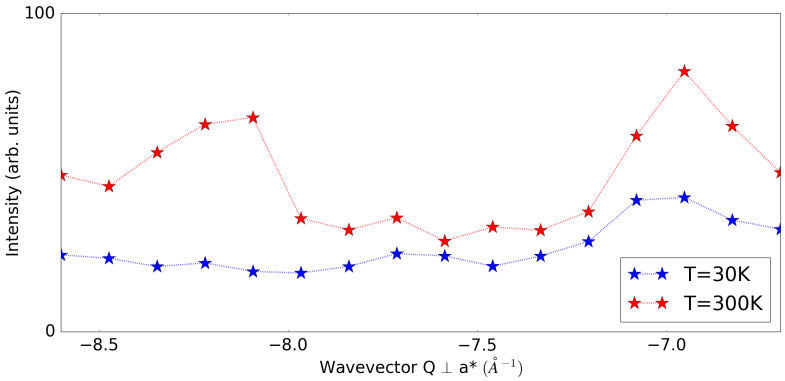
The structural diffuse features are most clearly visible in the as-grown sample. Annealing in oxygen makes the structural diffuse scattering become negligible, leav-



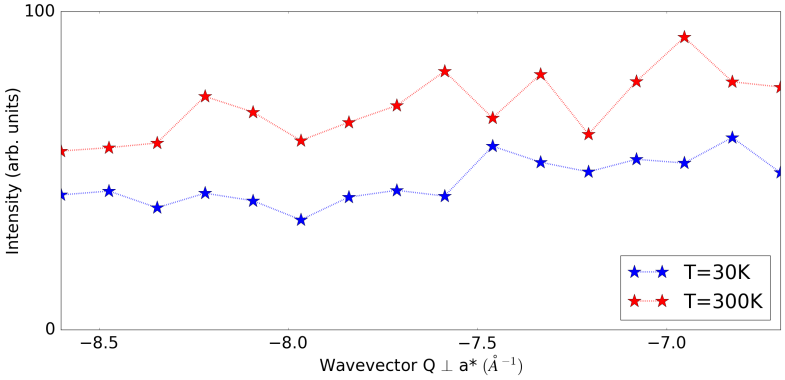
(a) (b)

Figure 5.6: The measured diffraction in the $(h, k, 0)$ plane for the as-grown (a) and O-annealed (b) samples. The as-grown data has stronger diffuse scattering, particularly wall-like features at high Q, but lacks the characteristic arcs seen in the O-annealed sample.

ing diffuse features only from mislabelled inelastic scattering events. The remaining inelastic signal can be confidently labelled as such, as first-principles calculations reproduce them very well. Since the thermal diffuse scattering simulations were for perfect, stoichiometric ZnO this suggests the O-annealed is actually closest to stoichiometric ZnO. Structural diffuse calculations indicate $\sim 5\%$ O vacancies in the as-grown sample, which is consistent with the idea that O annealing leads to stoichiometric ZnO.



(a)



(b)

Figure 5.7: Line profile of the measured diffraction for O-annealed (a) and as-grown (b) ZnO along $[0.5, Q_{\perp a^*}, 0]$. The phonon contribution in a) is rather pronounced, particularly in the 300 K plot. For the as-grown sample the additional structural diffuse scattering makes it difficult to see the inelastic contribution as clearly.

5.4 Intrinsic Defects in ZnO

Refinements were performed to determine the crystal structure. Indexed Bragg peaks and their measured intensities were exported for analysis using the Jana2006 software package [64]. This program uses atomic coordinates, species and thermal displacement parameters to model and fit the Bragg peak intensities.

5.4.1 Fourier Maps

An initial model of ZnO_{1-x} was unable to yield convincing refinements. The structure factor, F_{hkl}, is the Fourier transform of the atomic scattering length density, however since only intensities are measured the phase problem does not allow such straightforward analysis.

In a refinement, the quality is measured by the so-called R-factor (lower is better):

$$R = \frac{\sum_{hkl} ||F_{\text{obs}}(hkl)| - |F_{\text{calc}}(hkl)||}{\sum_{hkl} |F_{\text{obs}}(hkl)|}, \tag{5.5}$$

where $F_{\text{calc}}(hkl)$ is the calculated structure factor, $|F_{\text{obs}}(hkl)|^2$ the measured intensity, and the sum carried out over the measured, indexed Bragg peaks [24].

From the calculated structure factor it is possible to create Fourier difference maps which can give insight into the source of features not accounted for in the model, in real space as a scattering density, $\rho(x, y, z)$. This effectively ignores the phase problem [20] by using only the calculated phases since they cannot be observed. Furthermore, in principle the summation in Eq. (5.5) would be over all possible hkl, however only a finite amount are accessible, thus there are no guarantees on the obtained scattering density, or other parameters, derived thereof.

To investigate the difference in the as-grown sample, Fourier maps were calculated by modelling stoichiometric ZnO for calculations of $F_{\text{calc}}(hkl)$. Fourier maps for $T = 30\,\text{K}$ can be seen in Fig. 5.8, and show negative scattering lengths both at the oxygen sites and at an interstitial site. Since Zn and O both have a positive neutron scattering length [23], this suggests the presence of another scatterer. Hydrogen, with its large incoherent cross section and negative scattering length, fits this as well as results in Section 5.2. Discrepancies at the O site can be accounted for equivalently with O vacancies, as this will lower the effective scattering length of atoms at that site.

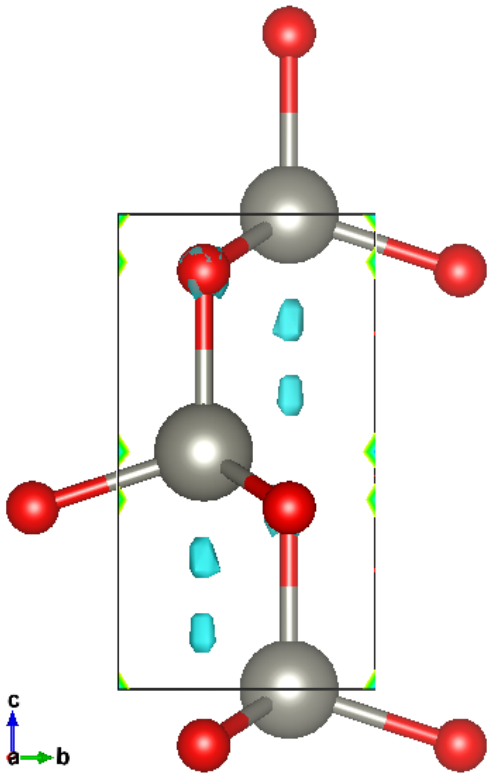


Figure 5.8: Fourier difference map for the as-grown sample plotted with the nominal positions of oxygen (red) and zinc (grey). These were computed against a model of perfect, stoichiometric ZnO and show negative scattering density in interstitial sites (cyan). There is also negative scattering density present on the oxygen sites, suggesting O vacancies.

5.4.2 Complimentary X-ray Measurements

After the measurements on SXD, small pieces (approximately $0.3 \times 0.2 \times 0.05$ mm) were cleaved from the samples and the x-ray diffraction patterns measured in-house using a Xcalibur single-crystal diffractometer.

The diffractometer uses a molybdenum source and monochromator to produce x-rays of a single wavelength, 0.709 Å. The scattering triangle is then defined by the 4-axis goniometer, and scattered x-rays are collected on a large CCD camera covered with a scintillation screen.

The diffractometer is mostly automated, data collection and initial analysis is performed using the software CrysAlis^{Pro}. The Experiment begins by screening samples to find a well-diffracting sample, for which the software can automatically determine a suitable unit cell and **UB** matrix. A number of possible strategies for orientations can be calculated to survey reciprocal space efficiently based on the crystal symmetries, however in this case redundant data were measured to allow advanced absorption corrections calculations provided by the software. Peaks are indexed based on their positions and goniometer settings, since the incident wavelength is fixed. These indexed peaks can then be exported for structural refinements using Jana2006, equivalent to the process in Section 5.3.

5.4.3 Structure Refinements

Since the x-rays scatter off electrons instead of nuclei, the atomic form factor per atom is proportional to the number of electrons, and thus these measurements are insensitive to hydrogen, and refinements were performed only on the oxygen stoichiometry. For the neutron data hydrogen interstitials were included with fractional

position $(1/3, 2/3, z)$ where z is fitted. This position was selected from the Fourier maps, initially with $z = 0.76$. The refinements give a good R-factor, the quality of fit, of approximately 5 for x-rays and 9 for neutrons. Full details of the refined hydrogen occupation, position and oxygen occupation can be seen in Table 5.1.

	AsGrown			Annealed		
	Neutrons		Xray	Neutrons		X-rays
	30 K	300 K	300 K	30 K	300 K	300 K
R	8.89	9.94	5.26	9.0	8.26	5.31
x in ZnO_xH_y	0.963(5)	0.941(7)	0.96(2)	0.992(7)	1.023(12)	1.0(2)
y in ZnO_xH_y	0.223(5)	0.211(10)	N/A	0.191(7)	0.235(12)	N/A
Hydrogen z pos.	0.7617(4)	0.7559(11)	N/A	0.7599(5)	0.7593(9)	N/A

Table 5.1: Structure refinements for ZnO_xH_y. Oxygen occupations are consistent between x-ray and neutron measurements for both samples. The as-grown sample has approximated 95% O occupancies, consistent with results from Section 5.3. From the Fourier maps, a hydrogen atom was inserted with fractional position $(1/3, 2/3, z)$.

As suspected from Section 5.3, the O-annealed sample appears most similar to stoichiometric ZnO, whilst the as-grown sample has 5% oxygen vacancies. Hydrogen interstitials appear in both samples, consistent with results from Chapter 4.

5.5 Summary

The intrinsic defect structure of ZnO was determined using neutron diffraction for three samples: two thin crystals of nominally as-grown and O-annealed ; and a larger crystal of as-grown at a range of temperatures from 5 K to 900 K.

To properly analyse the structural diffuse scattering, the thermal diffuse scattering was modelled using the same phonon modes obtained in Chapter 3 and showed good agreement with measurements. Structural diffuse calculations using a Monte-

Carlo model show similar structural diffuse features with 5% oxygen vacancies and displacements of the vacancy's nearest-neighbours by ~ 0.1 Å. Refinements of the stoichiometry show as-grown with oxygen occupations of 0.95, verified by independent x-ray measurements, and consistent with structural diffuse scattering calculations. Hydrogen interstitials were identified from the presence of incoherent scattering and negative scattering lengths in the Fourier maps. The intrinsic defect structure of ZnO can be seen in Fig. 5.9.

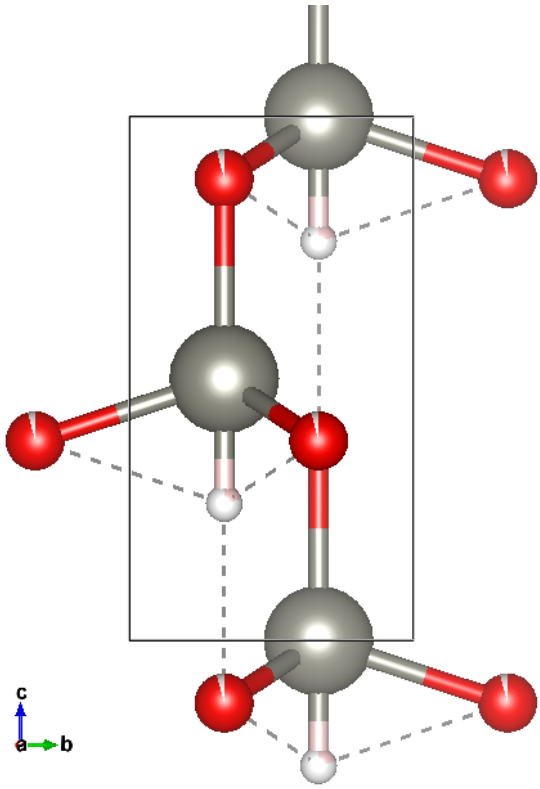


Figure 5.9: Final refined structure for as-grown ZnO. The atom colours are as in Fig. 5.8, with the addition of hydrogen (pink). The occupation of these sites is represented by how filled the atom is. Occupations are approximately: Zn, 100%; O, 95%; and H, 20%.

Chapter 6

Summary and Future Outlook

In this thesis the thermal conductivity and intrinsic defect structure of ZnO has been studied using neutron scattering techniques and *ab initio* DFT calculations.

The stoichiometry for oxygen was found to be 95% from x-ray and neutron structure refinements and is further validated by Monte Carlo simulations of the diffuse neutron scattering. The Monte Carlo simulations of the diffuse scattering, alone, are not sufficient to understand the diffuse scattering, as there is additional diffuse scattering from inelastic excitations. *Ab initio* simulations of the inelastic diffuse scattering are in excellent agreement with these data, and are also consistent with INS on LET, Merlin and MARI.

The presence of hydrogen at interstitial sites was observed in the single-crystal neutron diffraction data, and again with INS measurements of powders. The phonon density of states measured is well described by the calculated phonon modes using a model parametrized by a single mean free path. These mean free paths were used to calculate the thermal conductivity from first principles and are in excellent agreement with empirical results.

Calculations using the crystallite size reported for the nanostructured ZnO sample with ultra-low thermal conductivity, the original motivation of this thesis, are in excellent agreement. Aluminium doping leads to a further reduction in average grain size of approximately a factor 2, thus the further reduction in thermal conductivity by a factor 2 can be understood as the result, primarily, of finite-size effects. For the bulk sample, calculated values are extremely close to those measured *in situ*, and also in remarkable agreement with calculations assuming a model with a single, fixed lifetime as a variable instead of the mean free path, which yields an average lifetime of 0.92 ps.

It is possible to extract the lifetime of one phonon mode in the Merlin data, which was determined to be 1.19 ps, in reasonable agreement with the value obtained with the powder data. It was not possible to extract many different lifetimes as the broadening effects are quite subtle. A mode with particularly clean data, sufficiently isolated from other modes, and with sufficient intensity and coverage, was the only region it was possible to extract lifetimes from. It would be interesting to obtain much higher quality energy cuts to determine phonon lifetimes over a range of different modes across select positions in the first Brillouin zone, using an instrument like IN8. These measurements would allow further investigation of the capabilities and shortcomings of the fixed mean free path model, to give better calculations for the thermal conductivity, including its temperature dependence. Furthermore it would be interesting to see how this approach works for similar materials, other semi-conductors with a relatively simple structure, and with a thermal conductivity dominated by lattice dynamics, for example GaN.

Measurements of the nanostructured powder at a number of temperatures ranging from 300 K to 900 K show that crystallite size increases rapidly during annealing and

becomes bulk-like by 900 K. Furthermore, any zinc surface hydroxyls in the sample are removed after this heat treatment. Unfortunately this rules out ZnO for reasonably high-temperature applications such as the proposed car exhaust. A more careful study of this behaviour, i.e. how rapidly crystallite size grows with temperature, is required to better understand the useful temperature range for thermoelectric applications.

Finally, measurements of the inelastic scattering show strong, anharmonic phonon-phonon scattering which these models do not capture correctly. It would be interesting to see the effect calculations going beyond the simple harmonic approximation have on the phonon density of states, particularly for the high energy modes. This could give further insight into the thermoelectric properties, particularly at higher temperatures where these effects have been shown to become more pronounced.

Bibliography

[1] Ü Özgür, Ya I. Alivov, C. Liu, A. Teke, M. A. Reshchikov, S. Doğan, V. Avrutin, S. J. Cho, and H. Morkǫ. A comprehensive review of ZnO materials and devices. *Journal of Applied Physics*, 98(4):1–103, 2005.

[2] D.C. Look. Recent advances in ZnO materials and devices. *Materials Science and Engineering: B*, 80(1-3):383–387, March 2001.

[3] Robert Triboulet. The scope of the ZnO growth. *Proceedings of SPIE*, 4412:1–8, October 2000.

[4] Yuki Orikasa, Naoaki Hayashi, and Shigetoshi Muranaka. Effects of oxygen gas pressure on structural, electrical, and thermoelectric properties of (ZnO)3In2O3 thin films deposited by RF magnetron sputtering. *Journal of Applied Physics*, 103(11):113703, 2008.

[5] D.C. Look. Recent advances in ZnO materials and devices. *Materials Science and Engineering: B*, 80(1):383 – 387, 2001.

[6] D. C. Look, B. Claflin, Ya. I. Alivov, and S. J. Park. The future of ZnO light emitters. *physica status solidi (a)*, 201(10):2203–2212, 2004.

[7] S. B. Zhang, S.-H. Wei, and Alex Zunger. Intrinsic n-type versus p-type doping asymmetry and the defect physics of ZnO. *Phys. Rev. B*, 63:075205, Jan 2001.

[8] C W Bunn. The lattice-dimensions of zinc oxide. *Proceedings of the Physical Society*, 47(5):835–842, September 1935.

[9] S. J. Clark and J. Robertson. Calculation of semiconductor band structures and defects by the screened exchange density functional. *Physica Status Solidi (B) Basic Research*, 248(3):537–546, 2011.

[10] Chris G. Van de Walle. Hydrogen as a cause of doping in zinc oxide. *Phys. Rev. Lett.*, 85:1012–1015, Jul 2000.

[11] Gabriele Sala, Matthias Gutmann, D. Prabhakaran, D. Pomaranski, C. Mitchelitis, J.B. Kycia, Dan Porter, Claudio Castelnovo, and Jon Goff. Vacancy defects and monopole dynamics in oxygen-deficient pyrochlores. *Nature Materials*, 13:488–493, 5 2014.

[12] Zhen-Hua Ge, Li-Dong Zhao, Di Wu, Xiaoye Liu, Bo-Ping Zhang, Jing-Feng Li, and Jiaqing He. Low-cost, abundant binary sulfides as promising thermoelectric materials. *Materials Today*, 19(4):227–239, May 2016.

[13] Fj DiSalvo. Thermoelectric cooling and power generation. *Science*, 285(5428):703–6, 1999.

[14] R. Hyde and P. Stevenson. The potential for recovering and using surplus heat from industry. 2014.

[15] G. Jeffrey Snyder and Eric S. Toberer. Complex thermoelectric materials. *Nature Materials*, 7:105 EP –, Feb 2008. Review Article.

[16] Joseph McDonald and Lee Jones. Demonstration of tier 2 emission levels for heavy light-duty trucks. *SAE International Technical Paper Series*, pages 776–4841, 06 2000.

[17] Aaron D. LaLonde, Yanzhong Pei, Heng Wang, and G. Jeffrey Snyder. Lead telluride alloy thermoelectrics. *Materials Today*, 14(11):526 – 532, 2011.

[18] Yoshihiro Inoue, Masaki Okamoto, Toshio Kawahara, Yoichi Okamoto, and Jun Morimoto. Thermoelectric Properties of Amorphous Zinc Oxide Thin Films Fabricated by Pulsed Laser Deposition. *Materials Transactions*, 46(7):1470–1475, 2005.

[19] P Jood, R J Mehta, Y Zhang, G Peleckis, X Wang, R W Siegel, T Borca-Tasciuc, S X Dou, and G Ramanath. Al-doped zinc oxide nanocomposites with enhanced thermoelectric properties. *Nano Lett*, 11(10):4337–4342, 2011.

[20] D.S. Sivia. *Elementary Scattering Theory: For Xray And Neutron Users*. Oxford University Press, 2011.

[21] J. R. Hook. *Solid State Physics (Manchester Physics Series)*. Wiley, jun 2013.

[22] Rolf Hempelmann. *Quasielastic Neutron Scattering and Solid State Diffusion (Oxford Series on Neutron Scattering in Condensed Matter)*. Oxford University Press, 2000.

[23] Varley F. Sears. Neutron scattering lengths and cross sections. *Neutron News*, 3(3):26–37, 1992.

[24] Martin T. Dove. *Structure and Dynamics: An Atomic View of Materials (Oxford Master Series in Physics)*. Oxford University Press, 2003.

[25] J. M. Carpenter, C.-K. Loong, and Marie-Louise Saboungi. Neutron-scattering instruments: spectrometers. In *Elements of Slow-Neutron Scattering*, pages 204–236. Cambridge University Press.

[26] SXD technical information. `https://www.isis.stfc.ac.uk/Pages/SXD-technical-information.aspx`. Accessed: 2019-09-10.

[27] SXD detectors. `https://www.isis.stfc.ac.uk/Gallery/sxd.jpg`. Accessed: 2019-09-10.

[28] R.I. Bewley, J.W. Taylor, and S.M. Bennington. LET, a cold neutron multi-disk chopper spectrometer at ISIS. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 637(1):128–134, May 2011.

[29] J. Peters, J.D.M. Champion, G. Zsigmond, H.N. Bordallo, and F. Mezei. Using fermi choppers to shape the neutron pulse. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 557(2):580 – 584, 2006.

[30] MARI technical specification. `https://www.isis.stfc.ac.uk/Pages/MARI-technical-specification.aspx`. Accessed: 2019-09-10.

[31] MERLIN technical information. `https://www.isis.stfc.ac.uk/Pages/Merlin-technical-information.aspx`. Accessed: 2019-09-10.

[32] LET technical information. `https://www.isis.stfc.ac.uk/Pages/Let-technical-information.aspx`. Accessed: 2019-09-10.

[33] MARI schematic diagram. . Accessed: 2019-09-10.

[34] R.I. Bewley, R.S. Eccleston, K.A. McEwen, S.M. Hayden, M.T. Dove, S.M. Bennington, J.R. Treadgold, and R.L.S. Coleman. MERLIN, a new high count rate spectrometer at ISIS. *Physica B: Condensed Matter*, 385-386:1029 – 1031, 2006.

[35] Linseis. *Light Flash Analysis*. Linseis, LINSEIS GmbH Germany, Vielitzerstr. 43, 95100 Selb, 2019. Accessed online: `https://www.linseis.com/wp-content/uploads/2019/04/LINSEIS-LFA-500_v5_compressed.pdf`.

[36] Toshiki Tsubota, Michitaka Ohtaki, Koichi Eguchi, and Hiromichi Arai. Transport properties and thermoelectric performance of (zn1–ymgy)1–xalxo. *J. Mater. Chem.*, 8:409–412, 1998.

[37] M. Born and R. Oppenheimer. Zur quantentheorie der molekeln. *Annalen der Physik*, 389(20):457–484, 1927.

[38] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, Nov 1964.

[39] Feliciano Giustino. *Materials Modelling using Density Functional Theory: Properties and Predictions*. Oxford University Press, jul 2014.

[40] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, Nov 1965.

[41] Veronika Brázdová. *Atomistic Computer Simulations: A Practical Guide*. Wiley-VCH, apr 2013.

[42] Stefano Baroni, Stefano de Gironcoli, Andrea Dal Corso, and Paolo Giannozzi. Phonons and related crystal properties from density-functional perturbation theory. *Rev. Mod. Phys.*, 73:515–562, Jul 2001.

[43] R. P. Feynman. Forces in molecules. *Phys. Rev.*, 56:340–343, Aug 1939.

[44] K. Refson, P. R. Tulip, and Stewart J. Clark. Variational density-functional perturbation theory for dielectrics and lattice dynamics. *Phys. Rev. B*, 73:155114, 2006.

[45] J. Serrano, F. J. Manjón, a. H. Romero, A. Ivanov, M. Cardona, R. Lauck, A. Bosak, and M. Krisch. Phonon dispersion relations of zinc oxide: Inelastic neutron scattering and ab initio calculations. *Physical Review B*, 81(17):174304, 2010.

[46] Xufei Wu, Jonghoon Lee, Vikas Varshney, Jennifer L. Wohlwend, Ajit K. Roy, and Tengfei Luo. Thermal conductivity of wurtzite zinc-oxide from first-principles lattice dynamics - a comparative study with gallium nitride. *Scientific Reports*, 6:22504 EP –, Mar 2016. Article.

[47] Hendrik J. Monkhorst and James D. Pack. Special points for brillouin-zone integrations. *Phys. Rev. B*, 13:5188–5192, Jun 1976.

[48] R.A. Ewings, A. Buts, M.D. Le, J. van Duijn, I. Bustinduy, and T.G. Perring. Horace: Software for the analysis of data from single crystal spectroscopy experiments at time-of-flight neutron instruments. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 834:132 – 142, 2016.

[49] R. Mittal, M. K. Gupta, S. L. Chaplot, M. Zbiri, S. Rols, H. Schober, Y. Su, Th Brueckel, and T. Wolf. Spin-phonon coupling in K0.8Fe1.6Se2 and KFe2Se2: Inelastic neutron scattering and ab initio phonon calculations. *Physical Review B - Condensed Matter and Materials Physics*, 87(18):1–9, 2013.

[50] B. Fultz, C. C. Ahn, E. E. Alp, W. Sturhahn, and T. S. Toellner. Phonons in nanocrystalline ^{57}fe. *Phys. Rev. Lett.*, 79:937–940, Aug 1997.

[51] PyChop documentaion. `https://docs.mantidproject.org/nightly/interfaces/PyChop.html`. Accessed: 2019-09-07.

[52] Zhifeng Ren. High performance bulk thermoelectric materials. Technical report, March 2013.

[53] M. Wilde, K. Fukutani, M. Naschitzki, and H.-J. Freund. Hydrogen absorption in oxide-supported palladium nanocrystals. *Phys. Rev. B*, 77:113412, Mar 2008.

[54] Heshmat Noei, Hengshan Qiu, Yuemin Wang, Martin Muhler, and Christof Wöll. Hydrogen loading of oxide powder particles: A transmission IR study for the case of zinc oxide. *ChemPhysChem*, 11(17):3604–3607, November 2010.

[55] Jennifer Strunk, Kevin Kähler, Xinyu Xia, and Martin Muhler. The surface chemistry of ZnO nanoparticles applied as heterogeneous catalysts in methanol synthesis. *Surface Science*, 603(10-12):1776–1783, June 2009.

[56] J. Tomkinson. *Inelastic incoherent neutron scattering spectroscopy of hydrogen vibrations in metals and molecules.* Adam Hilger, United Kingdom, 1988.

[57] D. J. Voneshen, K. Refson, E. Borissenko, M. Krisch, A. Bosak, A. Piovano, E. Cemal, M. Enderle, M. J. Gutmann, M. Hoesch, M. Roger, L. Gannon, A. T.

Boothroyd, S. Uthayakumar, D. G. Porter, and J. P. Goff. Suppression of thermal conductivity by rattling modes in thermoelectric sodium cobaltate. *Nature Materials*, 12(11):1028–1032, August 2013.

[58] A&D Company. *AD-1653 Density Determination Kit Instruction Manual*. A&D Company, Limited., A&D Instruments Limited, European Head Office, 2012. Accessed online: `https://www.aandd.jp/products/manual/balances/ad1653.pdf`.

[59] Robert Triboulet. Scope of ZnO growth. *Proc SPIE*, 4412:1–8, 08 2001.

[60] Matthias J. Gutmann, Gabriella Graziano, Sanghamitra Mukhopadhyay, Keith Refson, and Martin von Zimmerman. Computation of diffuse scattering arising from one-phonon excitations in a neutron time-of-flight single-crystal Laue diffraction experiment. *Journal of Applied Crystallography*, 48(4):1122–1129, Aug 2015.

[61] M. Gutmann. SXD2001 - a program for treating data from TOF neutron single-crystal diffraction. *Acta Crystallographica Section A*, 61(a1):c164, Aug 2005.

[62] Christian P. Robert. *The Metropolis–Hastings Algorithm*, pages 1–15. American Cancer Society, 2015.

[63] Anderson Janotti and Chris G. Van de Walle. Oxygen vacancies in ZnO. *Applied Physics Letters*, 87(12):122102, 2005.

[64] Petrícek Václav, Dusek Michal, and Palatinus Lukás. *zkri*, volume 229, chapter Crystallographic Computing System JANA2006: General features, page 345. 2019 2014. 5.

Appendices

Appendix A

ZnO Force Constants

Atom		O1			O2		
		x	y	z	x	y	z
O1	x	14.037656	0.003467	-0.103452	-0.22855	0.048834	-0.358682
O1	y	0.003467	14.026205	0.052116	0.048859	-0.1721	0.2072
O1	z	-0.103452	0.052116	14.090954	-0.034028	0.019672	-0.469981
O2	x	-0.22855	0.048859	-0.034028	14.028913	0.003393	0.103392
O2	y	0.048834	-0.1721	0.019672	0.003393	14.017511	-0.052146
O2	z	-0.358682	0.2072	-0.469981	0.103392	-0.052146	13.702774
Zn1	x	-0.0497	-0.006475	-0.000134	-6.23803	4.274199	2.974503
Zn1	y	-0.00639	-0.057192	0.000198	4.275086	-1.304812	-1.717638
Zn1	z	0.000043	0.000171	-0.487345	2.92105	-1.686601	0.06458
Zn2	x	-6.237767	4.274082	-2.974153	1.225213	-0.027753	0.000217
Zn2	y	4.274906	-1.304743	1.717416	-0.027592	1.19278	-0.000326
Zn2	z	-2.92103	1.686553	0.06502	-0.000353	0.00008	-9.895889

Table A.1: The force experienced by an atom in response to a displacement of an oxygen atom.

Atom		Zn1			Zn2		
		x	y	z	x	y	z
O1	x	-0.0497	-0.00639	0.000043	-6.237767	4.274906	-2.92103
O1	y	-0.006475	-0.057192	0.000171	4.274082	-1.304743	1.686553
O1	z	-0.000134	0.000198	-0.487345	-2.974153	1.717416	0.06502
O2	x	-6.23803	4.275086	2.92105	1.225213	-0.027592	-0.000353
O2	y	4.274199	-1.304812	-1.686601	-0.027753	1.19278	0.00008
O2	z	2.974503	-1.717638	0.06458	0.000217	-0.000326	-9.895889
Zn1	x	11.559857	0.007093	0.104793	0.006727	-0.108457	0.665483
Zn1	y	0.007093	11.556864	-0.053406	-0.108377	-0.118587	-0.384319
Zn1	z	0.104793	-0.053406	11.499448	0.051978	-0.030014	-0.220702
Zn2	x	0.006727	-0.108377	0.051978	11.515479	0.007064	-0.104878
Zn2	y	-0.108457	-0.118587	-0.030014	0.007064	11.512572	0.053276
Zn2	z	0.665483	-0.384319	-0.220702	-0.104878	0.053276	11.442804

Table A.2: Force constants for ZnO calcutated using CASTEP. This matrix shows the force experienced by an Atom in response to a displacement of a Zinc atom.

Appendix B

Python Scripts

B.1 AtomicFormFactor

```
import numpy as np
import matplotlib.pyplot as plt
import os

"""
  This program approximates the atomic form factor for x-ray
   ↪ scattering  for a number of elements and isotopes
  For more information see: http://lampx.tugraz.at/~hadley/
   ↪ ss1/crystaldiffraction/atomicformfactors/formfactors.
   ↪ php

  The equation used:

```

```
  sum(i=1,4)[a_i * exp(-b_i * (q/4pi)^2)] + c

  Where the coefficients have been determined empirically

  The coefficients for a wide range of elements can be found in the accompanying file: coefficients.dat
  In order to add more elements to this, simply determine their coefficients and place those in the file
  format:
  Symbol (for lookup)  a1  b1  a2  b2  a3  b3  a4  b4  c
  note the file is TAB DELIMITED, no spaces!
"""

__author__ = 'TimLehner'

RESOURCE_FOLDER = os.path.join(os.path.dirname(__file__), 'resources')
COEFFICIENTS_FILE = os.path.join(RESOURCE_FOLDER, 'coefficients.dat')


def _load_equation_coefficients(filename=COEFFICIENTS_FILE):
  assert isinstance(filename, str)

```

```
31    data = np.genfromtxt(filename, delimiter='\t', dtype=None)
32    return data
33
34
35 def _match_element(element_symbol, filename=COEFFICIENTS_FILE
    ↪ ):
36    """
37    Looks up coefficients a_i, b_i, c from a file for a given
    ↪ element
38
39    :param element_symbol: String      - Symbol to search for in
    ↪  first column of filename, e.g. "H", "H1-" etc
40    :param filename: String        - location of symbols and
    ↪ related coefficients, default: coefficients.dat
41    :return: [float ...]          - Returns the coefficients a_i
    ↪ , b_i, c for i in [1, 4]
42    """
43
44    assert isinstance(element_symbol, str)
45    assert isinstance(filename, str)
46
47    all_available_data = _load_equation_coefficients(filename)
48
49    for current_row in all_available_data:
```

```
50      if current_row[0] == element_symbol:
51        return current_row
52
53    # Didn't find it, prepare error message...
54    message = ("Could not find coefficients for element \"" +
      ↪ element_symbol +
55          "\", please ensure they exist in " + filename)
56    # Check for typos / unmatched element
57    alternatives = search_element(element_symbol)
58    if len(alternatives) > 0:
59      message += "\nDid you mean one of: " + ", ".join(
      ↪ alternatives)
60    raise LookupError(message)
61
62
63  def verify_element_is_valid(element_symbol, filename=
      ↪ COEFFICIENTS_FILE):
64    assert isinstance(element_symbol, str)
65    assert isinstance(filename, str)
66
67    all_available_data = _load_equation_coefficients(filename)
68
69    for current_row in all_available_data:
70      if current_row[0] == element_symbol:
```

```
71        return True
72    return False
73
74
75 def search_element(element_symbol_pattern):
76    """
77    Returns list of names of all elements resembling argument.
78    This method allows easy searching for supported elements.
79
80    :param element_symbol_pattern: String      - As in
      ↪ match_element() or ElementXRayFormFactor.__init__()
81    :return:  [String ...]            - List of supported
      ↪ elements resembling element_symbol_pattern
82    """
83
84    assert isinstance(element_symbol_pattern, str)
85
86    element_symbol_pattern = str(element_symbol_pattern)
87    all_available_data = _load_equation_coefficients()
88
89    matches = []
90
91    for current_row in all_available_data:
92      if element_symbol_pattern in current_row[0]:
```

```
        matches.append(current_row[0])
    return matches


def print_all_available_element_names():
    all_names = search_element('')
    print ("The atomic X-ray form factor can be approximated for the following:" +
        "\n" + ", ".join(all_names))


class ElementScatterFactor(object):
    # Wrapper to allow selection of X-rays/Neutrons at instantiation.

    def __init__(self, element_symbol, using_neutrons=False, coh_b=0):
        self.scatter_factor_calculator = _ElementXRayFormFactor(element_symbol)
        if using_neutrons:
            self.scatter_factor_calculator.use_neutrons(coh_b)

    def f(self, q_magnitude=0):
        return self.scatter_factor_calculator.f(q_magnitude)
```

```
113
114
115 class _ElementXRayFormFactor(object):
116   def __init__(self, element_symbol):
117     """
118     Given element_symbol looks up the relevant coefficients
    ↪ in coefficients.dat]
119
120     Example Usage:
121       H =  ElementXRayFormFactor("H")
122
123     Errors:
124       Throws LookupError if given symbol does not match
    ↪ anything in coefficients.dat
125
126     :type self: _ElementXRayFormFactor
127     :type element_symbol: String    - Specifies which
    ↪ elements, e.g. "H", "H1-", "Zn" etc (see coefficients.
    ↪ dat)
128     """
129
130     assert isinstance(element_symbol, str)
131
132     data = _match_element(element_symbol)
```

```
133     self.a = [data[1], data[3], data[5], data[7]]
134     self.b = [data[2], data[4], data[6], data[8]]
135     self.c = data[9]
136     self._using_neutrons = False
137     self.coh_b = 0
138     self.element_symbol = element_symbol
139
140     # print "Found atom with {0} {1} {2}".format(self.a, self
    ↪ .b, self.c)
141
142   def use_neutrons(self, coh_b):
143     """
144     Use this function to return neutron scattering length
    ↪ only
145
146     examples:
147       Zn : 5.680
148       O  : 5.803
149
150     :param coh_b: float
151     :return: void
152     """
153     self._using_neutrons = True
154     self.coh_b = coh_b
```

```
155
156 def use_xrays(self):
157     self._using_neutrons = False
158
159 def f(self, q_magnitude=0, enable_q=True):
160     """
161     Returns the atomic form factor f(Q) for a given Q
162
163     Calculated as
164         sum(i=1,4)[a_i * exp(-b_i * (q/4pi)^2)] + c,
165     where the coefficients have been determined empirically.
166     For more info see: http://lampx.tugraz.at/~hadley/ss1/
    ↪ crystaldiffraction/atomicformfactors/formfactors.php
167
168     :param q_magnitude: float        - Magnitude of Momentum
    ↪ Transfer Vector
169     :param enable_q: bool      - If False, f(Q) = f(0) for all
    ↪  Q, default: True
170     :return: float         - Atomic Form Factor
171     """
172     if self._using_neutrons:
173         return np.zeros_like(q_magnitude) + self.coh_b
174     assert isinstance(enable_q, bool)
175
```

```
    f = 0
    if not enable_q:
      q_magnitude = 0
    for i in range(0, 4):
      f += (self.a[i] * np.exp(-self.b[i] * (q_magnitude /
    ↪ (4. * np.pi))**2))
    f += self.c
    return f


if __name__ == "__main__":

  xray = ElementScatterFactor("O")

  print xray.f(0)

  # Example usage

  # You can find all available isotopes, to add more update
    ↪ the coefficients.dat file accordingly.
  print_all_available_element_names()

  # simply refer to the element with its symbol
```

```
# you can use the method search_element() or print_all_available_element_names()
elements = ["Zn", "O1-", "O2-"]

q_range = np.linspace(0., 25., num=1000)  # this equation is valid for 0 <= q <= 25 Angstrom^-1

for element in elements:
    # load the element
    current_element = ElementScatterFactor(element)
    # get the f(q) value with the method... f(q)
    # here q can be an array for efficiency
    # noinspection PyTypeChecker
    f_n = current_element.f(q_range)
    plt.plot(q_range, f_n, label=element)

plt.title("Atomic form factor approximation for: " + ", ".join(elements))
plt.xlabel(r"$Q / \AA^{-1}$")
plt.ylabel(r"$f(Q)$")
plt.legend()

# noinspection PyTypeChecker
print ElementScatterFactor("O").f(15)
```

```python

plt.show()
```

B.2 geometry.py

```python
import numpy as np


def toRad(degree):
  return degree * np.pi / 180.


def rotate(theta, axis):
  n = axis / magnitude(axis)
  c = 1 - np.cos(theta)
  s = np.sin(theta)

  return np.array([
    [1 - c + c * n[0] ** 2, c * n[0] * n[1] - s * n[2], c * n[0] * n[2] + s * n[1]],
    [c * n[0] * n[1] + s * n[2], 1 - c + c * n[1] ** 2, c * n[1] * n[2] - s * n[0]],
    [c * n[0] * n[2] - s * n[1], c * n[1] * n[2] + s * n[0], 1 - c + c * n[2] ** 2]
```

```
  ])


def rotateX(theta):
  return np.array([
    [1, 0, 0],
    [0, np.cos(theta), -np.sin(theta)],
    [0, np.sin(theta), np.cos(theta)]
  ])


def rotateY(theta):
  return np.array([
    [np.cos(theta), 0, np.sin(theta)],
    [0, 1, 0],
    [-np.sin(theta), 0, np.cos(theta)]
  ])


def rotateZ(theta):
  return np.array([
    [np.cos(theta), -np.sin(theta), 0],
    [np.sin(theta), np.cos(theta), 0],
    [0, 0, 1]
```

```
  ])


def magnitude(vector):
  return np.sqrt(np.dot(vector, vector))


def angle_between(v1, v2):
  # a.b = |a||b|cos(t)
  return np.arccos(np.dot(v1, v2) / (magnitude(v1) *
   ↪ magnitude(v2)))
```

B.3 Bravais.py

```
  import numpy as np
import math
import warnings

from objects.Crystals.helpers.geometry import rotateX,
   ↪ rotateY, rotateZ, toRad, magnitude, angle_between

__author__ = 'Tim Lehner'


```

```
class Bravais(object):
    """
    The Bravais object is for doing calculations concerning the Bravais Lattice.
    Once initialized both the Bravais and Reciprocal lattice vectors can be accessed as:

        BravaisObject.bravais[i]
        BravaisObject.reciprocal[i]

    where i is a, b, c (*) respectively (for reciprocal).

    Other Bravais calculations supported are:

        BravaisObject.get_q(h, k, l)                    - Returns the q vector [q_x, q_y, q_z]
        BravaisObject.get_q_mag(h, k, l)                - Returns magnitude of q vector from get_q(h, k, l)
        BravaisObject.get_maximum_accessible_q_space(wavelength) - Returns [h_max, k_max, l_max]

    """

    def __init__(self, a, b, c):
```

```
    """
    Initialize a new Bravais object.

    :param a: array     - 3D numpy array of form [a_x, a_y, a_z]
    :param b: array     - [b_x, b_y, b_z]
    :param c: array     - [c_x, c_y, c_z]
    """
    self.bravais = np.vstack([a, b, c])
    self.reciprocal = self._calculate_reciprocal()

@classmethod
def fromABCAlphaBetaGamma(cls, mag_a, mag_b, mag_c, alpha, beta, gamma):
    """

    :param mag_a: double  - Lattice parameter a in angstrom
    :param mag_b: double  - Lattice parameter b in angstrom
    :param mag_c: double  - Lattice parameter c in angstrom
    :param alpha: double  - Lattice angle alpha in degree
    :param beta: double  - Lattice angle beta in degree
    :param gamma: double  - Lattice angle gamma a in degree
    :return: Bravais object
    """
```

```
51
52      rot_matrix_1 = rotateZ(toRad(gamma))
53      rot_matrix_2 = rotateY(toRad(-alpha))
54
55      a_vector = np.array([mag_a, 0, 0])
56      b_vector = np.matmul(rot_matrix_1, a_vector) * (mag_b /
    ↪ float(mag_a))
57      c_vector = np.matmul(rot_matrix_2, a_vector) * (mag_c /
    ↪ float(mag_a))
58
59      return cls(a_vector, b_vector, c_vector)
60
61
62  def get_q(self, h, k=None, l=None):
63      """ Returns the q vector for a given h, k, l in cartesian
    ↪  coordinates
64
65      :param h: int
66      :param k: int
67      :param l: int
68      :return: array - [q_x, q_y, q_z]
69      """
70      if k is None and l is None:
71          return np.dot(h, self.reciprocal)
```

```
    else:
        return (h * self.reciprocal[0] +
            k * self.reciprocal[1] +
            l * self.reciprocal[2])

    def get_r(self, u, v=None, w=None):
        """ Returns the r vector for a given u, v, w in cartesian
        ↪ coordinates

            :param u: int
            :param v: int
            :param w: int
            :return: array - [r_x, r_y, r_z]
            """
        if v is None and w is None:
            return np.dot(u, self.bravais)
        else:
            return (u * self.bravais[0] +
                v * self.bravais[1] +
                w * self.bravais[2])

    def get_hkl(self, q_vector):
        inverse_lattice = np.linalg.inv(self.reciprocal)
        return np.dot(q_vector, inverse_lattice)
```

```python

def get_uvw(self, r_vector):
    inverse_lattive = np.linalg.inv(self.bravais)
    return np.dot(r_vector, inverse_lattive)

def get_volume(self):
    return np.dot(self.bravais[0], np.cross(self.bravais[1], self.bravais[2]))

def get_q_mag(self, h, k=None, l=None):
    """
    Returns the magnitude of the q vector, as defined in get_q(h, k, l) above

    :param h: int
    :param k: int
    :param l: int
    :return: float
    """
    if k is None and l is None:
        self.get_magnitude(self.get_q(h))
    return self.get_magnitude(self.get_q(h, k, l))

def get_max_hkl_with_qmag(self, qmin=0, qmax=20):
```

```
    """
    Returns h_max, k_max, l_max with qmin <= |Q| <= qmax

    :param qmin: int
    :param qmax: int
    :return:
    """
    i = 0
    j = 0
    k = 0

    # We're going to encounter warning when we leave the accessible space
    # since this will cause us to compute arcsin(x) where |x| > 1
    # we expect this to happen, in fact it will ALWAYS happen running this
    # section, as a result the warning is suppressed here.
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        while qmin <= self.get_q_mag(i, j, k) <= qmax:
            i += 1
        i_max = i - 1
        i = 0
```

```
        while qmin <= self.get_q_mag(i, j, k) <= qmax:
            j += 1
        j_max = j - 1
        j = 0
        while qmin <= self.get_q_mag(i, j, k) <= qmax:
            k += 1
        k_max = k - 1
        return [i_max, j_max, k_max]



    def get_maximum_accessible_q_space(self, wavelength):
        """
        Returns the maximum values of h, k, l that will have allowed Bragg reflections for the given wavelength.

        Note [h_max, n, n], [n, k_max, n], [n, n, l_max] are allowed ONLY for n = 0.

        :param wavelength: float - Wavelength, units Angstrom
        :return: [h_max, k_max, l_max]
        """
        i = 0
        j = 0
```

```
    k = 0
    # We're going to encounter warning when we leave the accessible space
    # since this will cause us to compute arcsin(x) where |x| > 1
    # we expect this to happen, in fact it will ALWAYS happen running this
    # section, as a result the warning is suppressed here.
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        while not math.isnan(self.get_2_theta(i, j, k, wavelength)):
            i += 1
        i_max = i
        i = 0
        while not math.isnan(self.get_2_theta(i, j, k, wavelength)):
            j += 1
        j_max = j
        j = 0
        while not math.isnan(self.get_2_theta(i, j, k, wavelength)):
            k += 1
        k_max = k
```

```
        return [i_max, j_max, k_max]

    @staticmethod
    def get_magnitude(q):
        """
        Returns the magnitude of a given vector in cartesian coordinates

        :param q: vector [v_x, v_y, v_z]
        :return:
        """
        return np.sqrt(q[0] ** 2 + q[1] ** 2 + q[2] ** 2)

    @staticmethod
    def get_angle(a, b):
        """
        Returns the angle between given vectors a, b in cartesian coordinates

        This is computed using the relation:

            A . B = |A||B|cos(theta)

            theta = arccos( (A . B) / (|A||B|))
```

```
200
201     :param a: vector [a_x, a_y, a_z]
202     :param b: vector [b_x, b_y, b_z]
203     :return:
204     """
205     return np.arccos((np.dot(a, b)) / float(Bravais.
    ↪ get_magnitude(a) * Bravais.get_magnitude(b)))
206
207   def get_2_theta(self, h, k, l, wavelength=0.5):
208     """
209     Returns the 2Theta values for a given h, k, l at
    ↪ specified wavelength
210
211     :param h: int
212     :param k: int
213     :param l: int
214     :param wavelength: float - Units Angstrom
215     :return: float [0, 180] -  2 Theta in degrees.
216     """
217     d_inverse = self.get_d_inverse(h, k, l)
218     return 2 * (np.arcsin(wavelength * d_inverse / 2.) * 180.
    ↪  / np.pi)
219
220   def get_d_spacing(self, h, k, l):
```

```
221         return 1. / self.get_d_inverse(h, k, l)
222
223     def get_d_inverse(self, h, k, l):
224         """
225         Returns the d spacing for a given h, k, l plane.
226
227         See https://www.scribd.com/document/333810781/
        ↪ xtalgeometry-pdf for more information on this method
228
229         :param h: int
230         :param k: int
231         :param l: int
232         :return: float
233         """
234
235         volume = self._get_volume_triclinic()
236         s11 = self._get_sii_triclinic(1)
237         s22 = self._get_sii_triclinic(2)
238         s33 = self._get_sii_triclinic(3)
239         s12 = self._get_sij_triclinic(1)
240         s23 = self._get_sij_triclinic(2)
241         s13 = self._get_sij_triclinic(3)
242
243         return np.sqrt(1. / volume ** 2 * (
```

```
244         s11 * h ** 2 +
245         s22 * k ** 2 +
246         s33 * l ** 2 +
247         2 * s12 * h * k +
248         2 * s23 * k * l +
249         2 * s13 * h * l))
250
251     def _calculate_reciprocal(self):
252         """
253         ! ! ! FOR INTERNAL USE ONLY - Use object.reciprocal[i]
        ↪ instead ! ! !
254
255         Calculates the reciprocal lattice vectors a*, b*, c*
256         This is automatically called when the Bravais object is
        ↪ instantiated and saved as
257
258             BravaisObject.reciprocal
259
260         Calculates as:
261
262             i* = 2 * pi / (i . (j x k)) * (j x k)
263
264             where i, j, k are x, y, z cyclically
265
```

```
    :return: 3x3 array: [0] = a*, [1] = b*, [2] = c*, [0][0]
 ↪ = a*_x etc.
    """
    a = self.bravais[0]
    b = self.bravais[1]
    c = self.bravais[2]

    scaling = 2 * np.pi / np.dot(a, np.cross(b, c))

    a_star = scaling * np.cross(b, c)
    b_star = scaling * np.cross(c, a)
    c_star = scaling * np.cross(a, b)

    return np.vstack([a_star, b_star, c_star])

  def _get_sii_triclinic(self, i):
    cyclic_lookup_lengths, cyclic_lookup_angles = self.
 ↪ _get_abc_alpha_beta_gamma()

    index1 = (i % 3)
    index2 = ((i + 1) % 3)
    index3 = ((i + 2) % 3)

```

```
    return (cyclic_lookup_lengths[index1] ** 2 * cyclic_lookup_lengths[index2] ** 2 *
        math.sin(cyclic_lookup_angles[index3]) ** 2)

  def _get_sij_triclinic(self, i):
    """

    :param i: [1, 2, 3] maps to S_{12}, S_{23}, S_{13} respectively
    :return: S_ij
    """
    cyclic_lookup_lengths, cyclic_lookup_angles = self._get_abc_alpha_beta_gamma()

    index1 = ((i + 2) % 3)
    index2 = (i % 3)
    index3 = ((i + 1) % 3)

    return (cyclic_lookup_lengths[index1] * cyclic_lookup_lengths[index2] * cyclic_lookup_lengths[index3] ** 2 *
        (np.cos(cyclic_lookup_angles[index1]) * np.cos(cyclic_lookup_angles[index2]) -
         np.cos(cyclic_lookup_angles[index3])
```

```
305             )
306         )
307
308     def _get_volume_triclinic(self):
309         cyclic_lookup_lengths, cyclic_lookup_angles = self.
    ↪ _get_abc_alpha_beta_gamma()
310
311         a, b, c = cyclic_lookup_lengths[0], cyclic_lookup_lengths
    ↪ [1], cyclic_lookup_lengths[2]
312         alpha, beta, gamma = cyclic_lookup_angles[0],
    ↪ cyclic_lookup_angles[1], cyclic_lookup_angles[2]
313
314         return (a * b * c *
315             np.sqrt(1 - math.cos(alpha) ** 2 - math.cos(beta) **
    ↪ 2 - math.cos(gamma) ** 2 +
316                 2 * math.cos(alpha) * math.cos(beta) * math.cos(
    ↪ gamma))
317             )
318
319     def _get_abc_alpha_beta_gamma(self):
320         a = Bravais.get_magnitude(self.bravais[0])
321         b = Bravais.get_magnitude(self.bravais[1])
322         c = Bravais.get_magnitude(self.bravais[2])
```

```
    alpha = Bravais.get_angle(self.bravais[0], self.bravais[2])
    beta = Bravais.get_angle(self.bravais[1], self.bravais[2])
    gamma = Bravais.get_angle(self.bravais[0], self.bravais[1])
    return [a, b, c], [alpha, beta, gamma]

  def get_reduced_hkls(self, hkls):
    """
    Given arbitrary hkls, return the reduced wavevector k, defined as:

    Q = k + v    where Q, k, v are all 3 dimensional vectors, in hkl:

    Q = [h_Q, k_Q, l_Q]
    k = [h_k, k_k, l_k]    h_k, k_k, l_k in [0, 1]
    v = [h_v, k_v, l_v]    h_v, k_v, l_v all integer
    :param qvector:
    :return:
    """
    reduced_h_val = np.array(hkls) - np.array(hkls, dtype=int)
```

```
    delta = (-1 * (reduced_h_val > 0.5)) + (1 * (reduced_h_val < -0.5))
    return reduced_h_val + delta

  def get_reduced_wavevector(self, qvector):
    return self.get_q(self.get_reduced_hkls(qvector))
```

B.4 Atoms.py

```
from calculators.ScatteringLengths.AtomicFormFactor import ElementScatterFactor
from Bravais import Bravais

__author__ = 'Tim Lehner'


class Atom(object):
  # Details atoms (fractional location, name, scattering form factor)

  def __init__(self, name, scattering_form_factor, position, mass=-1):
    """

```

```
    :param name: String
    :param scattering_form_factor: ElementScatterFactor
    :param position: [float, float, float] - Fractional coordinate
    :param mass: Atomic mass in a.u., leave as -1 if unneeded
    :return: Atom
    """
    assert isinstance(scattering_form_factor, ElementScatterFactor)
    # assert isinstance(position, list)
    assert len(position) == 3

    self.mass = mass
    self.name = name
    self.fi = scattering_form_factor  # Should be of type calculators
    self.loc = position
    self.scatterFactorScale = 1

  def get_loc(self, bravais):
    """ Given a Bravais lattice (see Bravais class) returns the cartesian coordinates of the atom.

```

```
    e.g. an atom with fractional coordinates (0.5, 0, 0) in a cubic lattice, a = 10 would return (5, 0, 0)

    :param bravais: Bravais object
    :return: Location in cartesian (x, y, z) coordinates.
    """
    assert isinstance(bravais, Bravais)
    return self.loc[0] * bravais.bravais[0] + self.loc[1] * bravais.bravais[1] + self.loc[2] * bravais.bravais[2]

  def set_scatter_scale(self, scale):
    # print "changed scatter scale to " + str(scale)
    self.scatterFactorScale = scale

  def get_fi(self, q_magnitude=0):
    """
    Returns the atomic form factor f(Q) for a given Q

    :param q_magnitude: float        - Magnitude of Momentum Transfer Vector
    :return: float        - Atomic Form Factor
    """
    return self.scatterFactorScale * self.fi.f(q_magnitude)

```

```
53  def debye_waller_factor(self, q_vector):
54    # TODO: Actually implement this
55    return 1
56
57  def get_mass(self, units="si"):
58    """
59    Sometimes it is useful to have the mass of the atom, but
    ↪ this is not always necessary
60
61    The option is available, the mass must be set at
    ↪ instantiation, otherwise this will raise AttributeError
62
63    :param units: If units = "si" (default), returns kg,
    ↪ otherwise atomic mass units.
64    :return: mass (kg or a.m.u.)
65    """
66    if self.mass == -1:
67      raise AttributeError("The mass of this atom has not
    ↪ been set!")
68    a = self.mass
69    if units.lower() == "si":
70      a *= 1.660539e-27
71    return a
72
```

```
  def use_neutrons(self, coh_b):
    new_scatter_factor = ElementScatterFactor(self.fi.scatter_factor_calculator.element_symbol, True, coh_b)
    self.fi = new_scatter_factor
```

B.5 PhononEigencector.py

```
import numpy as np


class PhononEigenvector(object):
  def __init__(self, mode, atom_number, x_real, x_imag, y_real, y_imag, z_real, z_imag):
    self.mode = mode
    self.atom_number = atom_number
    self.vector = np.array([x_real + 1j * x_imag, y_real + 1j * y_imag, z_real + 1j * z_imag])

__author__ = 'TimLehner'
```

B.6 PhononQPoint.py

```
import numpy as np

```

```
3 from PhononEigenvector import PhononEigenvector
4
5
6 class PhononQPoint(object):
7   """
8   PhononQPoint handles a single Q point of a CASTEP .phonon
    ↪ file
9
10  There are 3 main parameters of interest:
11
12    q_vector     -   Q vector in cartesian co-ordinates
13    eigenvalues   -   list of numbers, 1st eigenvalues is
    ↪ accessed with 0 index, as usual
14    eigenvector_dict -  Dictionary of dictionaries. Accessed
    ↪ as eigenvector_dict[mode_number][atom_number]
15  """
16
17  def __init__(self, hkl):
18    """
19    Instantiates a PhononQPoint object, ready for populating
    ↪ with eigenvalue/vector pairs.
20    """
21    self.hkl = hkl
22    self.eigenvalues = []
```

```
23      self.eigenvector_dict = dict()  # [Mode number][Atom
    ↪ number]
24
25  def get_normalised_eigenvector(self, mode_number,
    ↪ atom_number):
26      evect = self.eigenvector_dict[mode_number][atom_number]
27      factor = np.sqrt(np.vdot(evect, evect))
28
29      evect *= 1. / factor
30      return evect
31
32  def add_eigenvalue(self, e_value):
33      self.eigenvalues.append(float(e_value))
34
35  def add_eigenvector(self, e_vector):
36      assert isinstance(e_vector, PhononEigenvector)
37
38      mode_number = e_vector.mode
39      atom_number = e_vector.atom_number
40      vector = e_vector.vector
41
42      if mode_number not in self.eigenvector_dict:
43          self.eigenvector_dict[mode_number] = dict()
44      self.eigenvector_dict[mode_number][atom_number] = vector
```

```


class SimplePhononQPoint(object):
  def __init__(self, hkl_or_phonon_q_point):

    if isinstance(hkl_or_phonon_q_point, PhononQPoint):
      a = hkl_or_phonon_q_point
    else:
      a = PhononQPoint(hkl_or_phonon_q_point)

    self.hkl = a.hkl
    self.eigenvalues = a.eigenvalues
    self.eigenvector_dict = a.eigenvector_dict


__author__ = 'TimLehner'
```

B.7 PhononReader.py

```
import numpy as np
import parse
from copy import deepcopy

import matplotlib
```

```python

font = {'size': 40}
matplotlib.rc('font', **font)

from PhononQPoint import PhononQPoint
from PhononEigenvector import PhononEigenvector
from calculators.ScatteringLengths.AtomicFormFactor import verify_element_is_valid, ElementScatterFactor
from objects.Crystals.Bravais import Bravais
from objects.Crystals.Atoms import Atom


def _memoize(f):
  memo = {}

  def helper(self, x):
    memoize_index = "{0:.3f}:{1:.3f}:{2:.3f}".format(*x)
    if memoize_index not in memo:
      memo[memoize_index] = f(self, x)
    return memo[memoize_index]

  return helper

class PhononReader(object):
```

```
def __init__(self, filename, mass_dict=None, cohb_dict=None):
    """
    PhononReader is instantiated using a .phonon file from a CASTEP calculation

    PhononReader parses the file and creates an object with all the associated eigenvectors/values as well as the
    bravais lattice and atom locations

    You may want to consider using PhononFileObject (see below)

    Example usage:
      see if __name__=="__main__": code block


    :param filename: String - Path to .phonon file from CASTEP calculation
    :return: initialized PhononReader Object
    """

    f = open(filename, "r")

```

```
48      # Read the file into memory
49      self.file_content = [x.strip('\n') for x in f.readlines()
    ↪ ]
50      f.close()
51
52      # Initialize object attributes
53      self.q_points = []
54      self.q_point_info = None
55      self.atoms = []
56
57
58      # Now we need to find the Bravais Lattice and Atom
    ↪ locations
59      never_initialised = True
60      for i in range(0, len(self.file_content)):
61        if "frequencies in " in self.file_content[i].lower():
62          self.using_frequency = "cm-1" in self.file_content[i
    ↪ ].lower()
63
64        # bravais lattice always has exactly 3 vectors
65        if "unit cell vectors" in self.file_content[i].lower():
66          v1 = self.file_content[i + 1].split()
67          v2 = self.file_content[i + 2].split()
68          v3 = self.file_content[i + 3].split()
```

```
69
70          a = np.array([float(v1[0]), float(v1[1]), float(v1
    ↪ [2])])
71          b = np.array([float(v2[0]), float(v2[1]), float(v2
    ↪ [2])])
72          c = np.array([float(v3[0]), float(v3[1]), float(v3
    ↪ [2])])
73          self.bravais = Bravais(a, b, c)
74          never_initialised = False
75
76      atom_info = None
77
78      if "fractional co-ordinates" in self.file_content[i].
    ↪ lower():
79          # get all lines containing information on atoms
80          for j in range(i, len(self.file_content)):
81              if "end" in self.file_content[j].lower():
82                  atom_info = self.file_content[i + 1:j]
83                  break
84      if atom_info is not None:
85          for line in atom_info:
86              values = line.split()
87              position = [float(values[1]), float(values[2]),
    ↪ float(values[3])]
```

```
88          ion = str(values[4])
89          if verify_element_is_valid(ion):
90            atomic_form_factor = ElementScatterFactor(ion)
91            if (mass_dict is not None):
92              new_atom = Atom(ion, atomic_form_factor,
    ↪ position, mass=mass_dict[ion])
93            else:
94              new_atom = Atom(ion, atomic_form_factor,
    ↪ position)
95            if (cohb_dict is not None):
96              new_atom.use_neutrons(cohb_dict[ion])
97            self.atoms.append(new_atom)
98          else:
99            raise LookupError("Unsupported atom: " + ion +
100                      ", please put the appropriate data in
    ↪ coefficients.dat")
101
102
103
104   if never_initialised:
105     raise LookupError("Could not find a bravais lattice in
    ↪ " + filename)
106
107   self.load_all_q_points()
```

```
    self.file_content = []

  def get_all_q_point_information(self):
    """
    Goes through the .phonon file and splits each q-pt into its own array of strings

    :return: list of array of strings, each array contains a block of q-pt information
    """
    if self.q_point_info is not None:
      return self.q_point_info

    start_indicies = []
    for i in range(0, len(self.file_content)):
      if "q-pt" in self.file_content[i].lower():
        start_indicies.append(i)

    q_point_info = []
    for i in range(0, len(start_indicies) - 1):
      curr_info = self.file_content[start_indicies[i]: start_indicies[i + 1]]
      q_point_info.append(curr_info)

```

```
    curr_info = self.file_content[start_indicies[-1]:]
    q_point_info.append(curr_info)

    self.q_point_info = q_point_info

    return q_point_info

  def load_new_q_point(self, q_point):
    """

    :param q_point: Array of strings,   first element: line starting q-pt,
                                last element: line before next line starting q-pt
    :return: PhononQPoint.    Also adds PhononQPoint to self.
    """
    # CASTEP computes eigenvalues as frequency, units cm^-1
    # For calculations, it is much more useful to use Energy, units meV
    # E = hc / lamda -> 1cm^-1 = 0.1239842 meV

    val = self.get_hkl_values(q_point[0])
    q_vector = np.array([float(val[1]), float(val[2]), float(val[3])])
```

```
    new_q = PhononQPoint(q_vector)

    need_to_add_eigenvalues = True
    need_to_add_eigenvectors = True

    i = 1
    while need_to_add_eigenvalues:
      if i == len(q_point):
        raise LookupError("Error in file")

      if "eigenvectors" in q_point[i].lower():
        i += 2  # skip 2 lines to first e-vector info
        break

      p = parse.search("{:^} {:^} ", q_point[i])
      if self.using_frequency:
        eigenvalue = float(p[1]) * 0.1239842
      else:
        eigenvalue = float(p[1])
      new_q.add_eigenvalue(eigenvalue)
      i += 1

    while need_to_add_eigenvectors:
      if i == len(q_point):
```

```
173         break
174
175     p = q_point[i].split()
176     p_casted = np.zeros(8)
177     p_casted[0] = int(p[0])
178     p_casted[1] = int(p[1])
179     for j in range(2, len(p_casted)):
180         p_casted[j] = float(p[j])
181
182     e_vector = PhononEigenvector(p_casted[0], p_casted[1],
    ↪ p_casted[2], p_casted[3],
183                                  p_casted[4], p_casted[5], p_casted[6],
    ↪ p_casted[7])
184     new_q.add_eigenvector(e_vector)
185     i += 1
186
187   self.q_points.append(new_q)
188   return new_q
189
190 @staticmethod
191 def get_hkl_values(current_line):
192   p = parse.search("q-pt={:^} {:^} {:^} {:^} ",
    ↪ current_line)
193   return p
```

```python

def load_all_q_points(self):
    all_q_points = self.get_all_q_point_information()
    for current_q_point in all_q_points:
        self.load_new_q_point(current_q_point)

def get_q_turning_points(self):
    previous_d_hkl = [-1, -1, -1]

    interesting_index = []
    interesting_hkl = []
    interesting_dhkl = []

    for i in range(1, len(self.q_points) - 1):
        d_hkl = self.q_points[i].hkl - self.q_points[i - 1].hkl
        if Bravais.get_magnitude(d_hkl - previous_d_hkl) > 1e-3:
            interesting_index.append(i - 1)
            interesting_hkl.append(np.round(self.q_points[i - 1].hkl, 2).tolist())
            interesting_dhkl.append(np.round(d_hkl, 2).tolist())

        previous_d_hkl = deepcopy(d_hkl)

```

```
    interesting_index.append(len(self.q_points))
    interesting_hkl.append(np.round(self.q_points[-1].hkl, 2)
↪ .tolist())
    interesting_dhkl.append(np.round((self.q_points[-2].hkl -
↪  self.q_points[-1].hkl), 2).tolist())

    previous_index = -2
    splitting_points_found = 0

    final_index = []
    final_hkl = []
    final_dhkl = []
    for i in range(0, len(interesting_index)):
      if interesting_index[i] - previous_index == 1:
        splitting_points_found += 1
      else:
        final_index.append(interesting_index[i] -
↪ splitting_points_found)
        final_hkl.append(interesting_hkl[i])
        final_dhkl.append(interesting_dhkl[i])
      previous_index = interesting_index[i]
    return final_index, final_hkl, final_dhkl

  def get_q_mpgrid_boundaries(self):
```

```
237
238     min_h, min_k, min_l = 10, 10, 10
239     max_h, max_k, max_l = -10, -10, -10
240
241     for qpoint in self.q_points:
242       assert isinstance(qpoint, PhononQPoint)
243
244       if qpoint.hkl[0] < min_h:
245         min_h = qpoint.hkl[0]
246       if qpoint.hkl[1] < min_k:
247         min_k = qpoint.hkl[1]
248       if qpoint.hkl[2] < min_l:
249         min_l = qpoint.hkl[2]
250
251       if qpoint.hkl[0] > max_h:
252         max_h = qpoint.hkl[0]
253       if qpoint.hkl[1] > max_k:
254         max_k = qpoint.hkl[1]
255       if qpoint.hkl[2] > max_l:
256         max_l = qpoint.hkl[2]
257
258     return [min_h, min_k, min_l], [max_h, max_k, max_l]
259
260   def get_q_mpgrid_step_size(self):
```

```
    min_dh, min_dk, min_dl = 10, 10, 10
    for i in range (1, len(self.q_points)):
      current_qpoint = self.q_points[i - 1]
      next_qpoint = self.q_points[i]

      dhkl = np.absolute(next_qpoint.hkl - current_qpoint.hkl
  ↪ )

      if dhkl[0] < min_dh and dhkl[0] != 0:
        min_dh = dhkl[0]
      if dhkl[1] < min_dk and dhkl[1] != 0:
        min_dk = dhkl[1]
      if dhkl[2] < min_dl and dhkl[2] != 0:
        min_dl = dhkl[2]

    return [min_dh, min_dk, min_dl]

  @_memoize
  def get_nearest_qpoint_to_hkls(self, hkls):
    hkls = np.abs(self.bravais.get_reduced_hkls(hkls))

    full_list_dist = [self.bravais.get_magnitude(np.absolute(
  ↪ np.abs(value.hkl) - hkls)) for value in self.q_points]

```

```
    index = np.argmin(full_list_dist)
    return self.q_points[index]

@_memoize
def get_nearest_qpoint_to_qvector(self, qvector):
    hkl_indicies = self.bravais.get_reduced_hkls(qvector)

    full_list_dist = [self.bravais.get_magnitude(np.absolute(
    ↪ value.hkl - hkl_indicies)) for value in self.q_points]

    index = np.argmin(full_list_dist)
    return self.q_points[index]

def convert_to_omega(self, eigenenergy):
    """

    :param eigenenergy: energy to convert to w. Units [meV]
    ↪ -> [rad. s^-1]
    :return: w in rad s^-1
    """
    inverse_wavelength = eigenenergy / 0.1239842  #  convert
    ↪ meV to cm^-1

```

```
    # w = 2 pi f = 2 pi c / lambda. [c] = cms^-1, [lambda] = cm

    return 2 * np.pi * 29979245800 * inverse_wavelength


class PhononFileObject(object):
  """
  A reduced version of PhononReader, can be instantiated using either a PhononReader or matching PhononReader __init__

  Useful parameters:

    PhononFileObject.bravais  -  Bravais object, for calculating reciprocal lattice, q_vectors, d spacings etc
    PhononFileObject.atoms    -  Atoms object, contains element name, fractional position, calculating f_i
    PhononFileObject.q_points  -  List of Q-Point information, including h,k,l value, eigenvectors/values

  """
  def __init__(self, filename):
```

```
320
321     if isinstance(filename, PhononReader):
322       a = filename
323     else:
324       a = PhononReader(filename)
325     self.bravais = a.bravais
326     self.q_points = a.q_points
327     self.atoms = a.atoms
328
329     self._a = a
330
331   def get_q_turning_points(self):
332     return self._a.get_q_turning_points()
333
334   def get_q_mpgrid_boundaries(self):
335     return self._a.get_q_mpgrid_boundaries()
336
337   def get_q_mpgrid_step_size(self):
338     return self._a.get_q_mpgrid_step_size()
339
340   def get_nearest_qpoint_to_qvector(self, qvect):
341     return self._a.get_nearest_qpoint_to_qvector(qvect)
342
343 __author__ = 'TimLehner'
```

```
#"""
```

B.8 PlotPhononIntensities.py

```
import numpy as np
import matplotlib

font = {'size': 60}
matplotlib.rc('font', **font)

import matplotlib.pyplot as plt
from scipy.ndimage.filters import gaussian_filter

from readers.PhononFileReader.PhononReader import
    ↪ PhononReader
from readers.PhononFileReader.PhononQPoint import
    ↪ PhononQPoint
from objects.Crystals.Atoms import Atom


class PhononSQWCalculator(object):
  def __init__(self, phonon_file, mode=0):
    assert isinstance(phonon_file, PhononReader)
    self.phonon_file = phonon_file
```

```
19        self.mode = mode
20
21    def getEigenvalues(self):
22        eigen = []
23        for qpoint in self.phonon_file.q_points:
24            assert isinstance(qpoint, PhononQPoint)
25            eigen.append(np.array(qpoint.eigenvalues))
26        return np.array(eigen).transpose()
27
28    def getSQWMeshgrid(self, brillouin_zone_offset, temp=300,
      ↪ method=0):
29        """
30
31        :param brillouin_zone_offset: some vector, R with integer
      ↪  hkl, such that Q = R + k, k within first BZ
32        :param temp: temperature, in Kelvin
33        :return:
34        """
35        cmTomeV = 8.06554
36        hbar = 6.582119514e-13  # meV s
37        N = 1e23
38        eigenvalues = self.getEigenvalues().transpose()
39        qrange = np.linspace(0, 1, len(eigenvalues))
```

```
40    erange = np.linspace(-np.max(eigenvalues.flatten()) *
      ↪ 1.1, np.max(eigenvalues.flatten()) * 1.1, 400)
41
42    XX, YY = np.meshgrid(qrange, erange)
43
44    SQW = np.zeros_like(XX)
45    index_q = 0
46    for i in range(0, len(self.phonon_file.q_points)):
47      qpoint = self.phonon_file.q_points[i]
48      assert isinstance(qpoint, PhononQPoint)
49      Q = self.phonon_file.bravais.get_q(qpoint.hkl +
      ↪ brillouin_zone_offset)
50      for v in range(0, len(eigenvalues[0])):
51        eigenenergy = eigenvalues[i][v]
52        omegav = eigenenergy * cmTomeV
53        contribution = 0
54        for j in range(0, len(self.phonon_file.atoms)):
55          atom = self.phonon_file.atoms[j]
56          assert isinstance(atom, Atom)
57          coh_b = atom.get_fi(Q)
58          mass = atom.get_mass()
59          eigenvectorORIG = []
60          if self.mode == 0:
61            # print "method 1"
```

```
            eigenvectorORIG = np.conj(qpoint.eigenvector_dict[v + 1][j + 1])
        elif self.mode == 1:
            # print "method 2"
            eigenvectorORIG = (qpoint.eigenvector_dict[v + 1][j + 1])
        eigenvector = np.zeros_like(eigenvectorORIG)
        if method == 0:
            eigenvector[0] = eigenvectorORIG[0]
            eigenvector[1] = eigenvectorORIG[1]
        elif method == 1:
            eigenvector[0] = eigenvectorORIG[1]
            eigenvector[1] = eigenvectorORIG[0]

        eigenvector[2] = eigenvectorORIG[2]

        Tj = atom.debye_waller_factor(Q)
        structure = np.exp(1j * np.dot(Q, atom.get_loc(self.phonon_file.bravais)))
        phonon = np.dot(Q, eigenvector)
        contribution += coh_b / np.sqrt(mass) * phonon * structure * Tj
    contribution = np.vdot(contribution, contribution)
    contribution *= 1. / omegav * N * hbar / 2.
```

```
        contribution = np.real(contribution)
        self.insertIntoMesh(qrange[index_q], eigenenergy, contribution * self.bose_distribution(eigenenergy, temp),
                    SQW, qrange, erange)
        self.insertIntoMesh(qrange[index_q], -eigenenergy,
                    contribution * (self.bose_distribution(eigenenergy, temp) + 1),
                    SQW, qrange, erange)
    index_q += 1
  return XX, YY, SQW[::-1]

@staticmethod
def bose_distribution(energy, T):
  boltzmann = 8.6173303e-2
  return 1. / (np.exp(energy / (boltzmann * T)) - 1)

def insertIntoMesh(self, Q, E, intensity, mesh, q_range, e_range):
  index_x = np.argmin(np.abs(q_range - Q))
  index_y = np.argmin(np.abs(e_range - E))

  print "Target e: {0}, got {1}".format(E, e_range[index_y])
```

```
101
102     mesh[index_y][index_x] += intensity
103
104
105 class PhononPlotter(object):
106   def __init__(self, phonon_file, mode=0):
107     self.phonon_file = phonon_file
108     self.calculator = PhononSQWCalculator(phonon_file, mode)
109
110   def plotEigenvalues(self, plotObj=plt):
111     pass
112     # evals = self.calculator.getEigenvalues()
113     # for i in range(0, len(evals)):
114     # plotObj.plot(x_range, evals[i,:], 'black', label="mode
    ↪ {0}".format(i))
115
116   def plotEigenvectors(self, brillouin_zone_offset, maxlevel
    ↪ =1., temperature=300, plotObj=plt, no_broadening=False,
117                 max_energy=80):
118     x, y, SQWmesh = self.calculator.getSQWMeshgrid(
    ↪ brillouin_zone_offset, temperature)
119
120     # rescale to max itnensity = 1 for plotting purposes
121     # SQWmesh /= np.max(SQWmesh.flatten()) / 1.
```

```
122
123     # apply gaussian filter to smooth image
124     if not no_broadening:
125       SQWmesh = gaussian_filter(SQWmesh[len(SQWmesh) / 2::],
    ↪ 1)
126
127     print np.shape(x)
128     print np.shape(y)
129     print np.shape(SQWmesh)
130     plotObj.contourf(SQWmesh, levels=np.linspace(0, 5e37,
    ↪ 255), cmap='viridis')
131
132
133 def plot_LET_like(PhononPlotterPositive,
    ↪ PhononPlotterNegative, bzStart=np.array([2, 0, -2]),
    ↪ bzEnd=np.array([2, 0, 0]),
134             bzStep=np.array([0, 0, 1]), max_steps=100,
135             maxlevel=1., temp=300, plotObj=plt, no_broadening=
    ↪ False, max_energy=80):
136   steps = 0
137   start = np.copy(bzStart)
138   printLabels = [np.dot(bzStart, bzStep)]
139   while (start != bzEnd).any() and max_steps > 0:
140     start += bzStep
```

```
    steps += 1
    max_steps -= 1
    printLabels.append(np.dot(start, bzStep))

  if max_steps < 0:
    print "WARNING MANY STEPS REQUIRED ARE YOU SURE?"

  start = np.copy(bzStart)

  sqwTotal = None

  for i in range(0, steps):
    sqwPosMesh = PhononPlotterPositive.calculator.getSQWMeshgrid(start, temp=temp)[2]
    start += bzStep
    sqwNegMesh = np.fliplr(PhononPlotterNegative.calculator.getSQWMeshgrid(start, temp=temp)[2])

    if sqwTotal is None:
      sqwTotal = np.hstack([sqwPosMesh[:, :-1], sqwNegMesh])
    else:
      sqwTotal = np.hstack([sqwTotal, sqwPosMesh[:, :-1], sqwNegMesh])

```

```
162  # apply gaussian filter to smooth image
163  if no_broadening:
164    SQWmesh = sqwTotal[len(sqwTotal) / 2::]
165  else:
166    SQWmesh = gaussian_filter(sqwTotal[len(sqwTotal) / 2::],
     ↪ 1)
167
168  plotObj.contourf(SQWmesh, levels=np.linspace(0, maxlevel,
     ↪ 255), interpolation="None")
169
170  plotObj.yticks(np.linspace(0, len(SQWmesh[0]) / 80. *
     ↪ max_energy, 3), np.linspace(0, max_energy, 3))
171  plotObj.ylim([0, max_energy * len(SQWmesh[0]) / 80.])
172
173  plotObj.xticks(np.linspace(0, len(SQWmesh), steps + 1),
     ↪ printLabels)
174  plotObj.xlim([0, len(SQWmesh)])
175  plotObj.title("Simulation")
176
177
178 if __name__ == "__main__":
179  mass_dict = {"O": 15.999, "Zn": 65.39}  # in atomic mass
     ↪ units
180  cohb_dict = {"O": 5.803, "Zn": 5.680}  # For using neutrons
```

```

# Set phonon_file to the CASTEP output .phonon file
phonon_file = "C:\\Users\\Tim\\Documents\\PhD\\Scripts\\ZnOMaster\\calculators\\PhononScatteringIntensity\\ga.phonon"
phonons = PhononReader(phonon_file, mass_dict, cohb_dict)

fig = plt.figure()
sqw_plotter = PhononPlotter(phonons)
BZ_OFFSET = [0, 0, 8]
sqw_plotter.plotEigenvectors(np.array(BZ_OFFSET), maxlevel=1e-3, temperature=10)

x_tick_labels = [r"$\Gamma$", r"$A$"]
plt.yticks([])
plt.xticks([0, 50], x_tick_labels)
plt.show()
```

Appendix C

Balls-and-Springs Monte Carlo Structural Diffuse Scattering Simulator

C.1 ChainedMutator.cpp

```
//
// Created by Tim on 28/03/2018.
//

#include <iostream>
#include "ChainedMutator.h"

void ChainedMutator::mutateCrystal(SuperCell *crystalToModify) {
    // *crystalToModify* is any FilterableAtoms object that will be
   modified by this call
    // *filter* is some function that takes Atom as argument and returns
```

```
    true is the atom should be selected as an atom to make vacant
    // *targetOccupation* is the goal occupancy of atoms selected by *filter*,
    //      e.g. if want ZnO_{0.8}, targetOccupation = 0.8 and *filter* should select O atoms
    bool exit = false;
    while (!exit) {
        for (Atom &a : crystalToModify->getUnfilteredList()) {
            if (filter(a)) {
                if (process(a, crystalToModify) && next != nullptr) {
                    next->process(a, crystalToModify);
                }
                if (canFinishEarly() && isFinished(crystalToModify)) {
                    exit = true;
                    break;
                }
            }
        }
        exit = isFinished(crystalToModify);
    }
}

void ChainedMutator::setNext(ChainedMutator *next) {
    ChainedMutator::next = next;
}

bool ChainedMutator::isFinished(SuperCell *crystalToModify) {
    return true;
}

```

```
bool ChainedMutator::canFinishEarly() {
    return false;
}
```

C.2 CrystalMutator.cpp

```
//
// Created by Tim on 14/03/2018.
//

#include "CrystalMutator.h"
#include <iostream>

void
CrystalMutator::mutateCrystal(SuperCell *crystalToModify) {
    // *crystalToModify* is any FilterableAtoms object that will be modified by this call
    // *filter* is some function that takes Atom as argument and returns true is the atom should be selected as an atom to make vacant
    // *targetOccupation* is the goal occupancy of atoms selected by *filter*,
    //      e.g. if want ZnO_{0.8}, targetOccupation = 0.8 and *filter* should select O atoms
    for (Atom &a : crystalToModify->getUnfilteredList()) {
        if (filter(a)) {
            if (process(a, crystalToModify)) {
                changesMade++;
            };
        }
    }
```

```
    std::cout << "Made " << changesMade << " changes to the crystals" << std::endl;

}
```

C.3 CycleSuperCell.cpp

```
//
// Created by Tim on 21/03/2018.
//

#include "CycleSuperCell.h"

void CycleSuperCell::execute(int nCycles, SuperCell &superCell) {
    Vector3i supercellDims = superCell.getSupercellSize();
    for (int currentCycle = 0; currentCycle < nCycles; currentCycle++) {
        for (int n1 = 0; n1 < supercellDims[0]; n1++) {
            for (int n2 = 0; n2 < supercellDims[1]; n2++) {
                for (int n3 = 0; n3 < supercellDims[2]; n3++) {
                    process_subcell(Eigen::Vector3i(n1, n2, n3), superCell);
                }
            }
        }
    }
}
```

C.4 CrystalEnergyCalculator.cpp

```
#include <chrono>
```

```
#include "../objects/SuperCell.h"
#include "CrystalEnergyCalculator.h"
#include "../Calculators/SpringEnergy/SpringEnergyStrategy.h"
#include "../Calculators/SpringEnergy/ZnOSprings.h"

//
// Created by Tim on 23/04/2018.
//
EfficientCrystalRelaxor::EfficientCrystalRelaxor(SuperCell &superCell,
    SpringEnergyStrategy *strategy, int nneighbours,
                                                     int numberOfCycles) :
    strategy(strategy), nneighbours(nneighbours),

    numberOfCycles(numberOfCycles) {
     std::cout << "Allocating memory for MC calculations. This may take a
     while..." << std::endl;
     elementToId = superCell.getTypeIdFromElementMap();
     idToElement = superCell.getElementFromTypeIdMap();
     b = const_cast<Bravais *     >(&superCell.getBravais());
     setupArrays(superCell);
     std::cout << "Ready to calculate" << std::endl;
     acceptedMoves = 0;
     rejectedMoves = 0;
     superCellSize = superCell.getSupercellSize().cast<float>();
}

EfficientCrystalRelaxor::~EfficientCrystalRelaxor() {
     free(hAtomX);
     free(hAtomY);
     free(hAtomZ);
```

```
    free(hAtomType);
    free(hCharge);
    free(hOccupancy);
    free(hNeighbourIndex);
    free(hElemIndex);
    free(hOrigAtomType);
    free(hUnitCellIndex);
}

void EfficientCrystalRelaxor::setupArrays(SuperCell &superCell) {
    // All the data pertaining to the crystal is loaded into double/int arrays
    // This makes calculations and iterating the object lightning quick

    int boxSizeX = superCell.getSupercellSize()[0];
    int boxSizeY = superCell.getSupercellSize()[1];
    int boxSizeZ = superCell.getSupercellSize()[2];
    atoms_per_unit_cell = superCell.getSupercellCrystals()[0][0][0]->getBasis().size();

    number_of_atoms = boxSizeX * boxSizeY * boxSizeZ * atoms_per_unit_cell;

    size_t size = number_of_atoms * sizeof(double);
    size_t size_int = number_of_atoms * sizeof(int);

    hAtomX = (double *) malloc(size);
    hAtomY = (double *) malloc(size);
    hAtomZ = (double *) malloc(size);
    hCharge = (double *) malloc(size);
```

```
55     std::cout << "Setting up neighbour array for " << nneighbours << "
    neighbours" << std::endl;
56     hNeighbourIndex = (int *) malloc(size_int * nneighbours);
57     hAtomType = (int *) malloc(size_int);
58     hOrigAtomType = (int *) malloc(size_int);
59     hElemIndex = (int *) malloc(size_int);
60     hUnitCellIndex = (int *) malloc(size_int);
61     hOccupancy = (double *) malloc(size);
62
63     std::vector <std::vector<NeighbourDirections>> neighbours =
    getNeighbourStructure(superCell);
64
65     for (int i = 0; i < boxSizeX; i++) {
66         for (int j = 0; j < boxSizeY; j++) {
67             for (int k = 0; k < boxSizeZ; k++) {
68                 for (int l = 0; l < atoms_per_unit_cell; l++) {
69                     int address = getAddress(superCell, i, j, k, l);
70                     Atom currAtom = superCell.getSupercellCrystals()[i][
    j][k]->getBasis()[l];
71                     Bravais *bravais = const_cast<Bravais *>(&superCell.
    getBravais());
72
73                     hElemIndex[address] = currAtom.getElementIndex();
74                     hAtomX[address] = currAtom.getAbsolutePosition(
    bravais)[0];
75                     hAtomY[address] = currAtom.getAbsolutePosition(
    bravais)[1];
76                     hAtomZ[address] = currAtom.getAbsolutePosition(
    bravais)[2];
77                     hUnitCellIndex[address] = currAtom.getUnitCellIndex
```

```
    ();
                    hAtomType[address] = superCell.getTypeIdFromElement(currAtom.getAtomType());
                    hCharge[address] = currAtom.getCharge();
                    hOccupancy[address] = currAtom.getOccupancy();

                    for (int neighbourIndex = 0; neighbourIndex < nneighbours; neighbourIndex++) {
                        int newAddress = getNeighbourAddress(superCell, neighbours[l][neighbourIndex], currAtom);
                        hNeighbourIndex[address * nneighbours + neighbourIndex] = newAddress;
                    }
                }
            }
        }
    }

}

void EfficientCrystalRelaxor::analyseNeighbourStructure(SuperCell & superCell) {
    for (int i = 0; i < atoms_per_unit_cell; i++) {
        Atom currAtom = superCell.getSupercellCrystals()[1][1][1]->getBasis()[i];
        std::vector <NeighbourDirections> atomNeighbours;

        for (Atom a : superCell.getNearestNeighbours(currAtom, nneighbours)) {
            atomNeighbours.push_back(NeighbourDirections(currAtom, a));
```

```
        }
        _neighbourStructure.push_back(atomNeighbours);
    }
}

std::vector <std::vector<NeighbourDirections>> &EfficientCrystalRelaxor
    ::getNeighbourStructure(SuperCell &superCell) {
    if (_neighbourStructure.size() == 0) {
        analyseNeighbourStructure(superCell);
    }

    return _neighbourStructure;
}

int EfficientCrystalRelaxor::getNeighbourAddress(SuperCell &superCell,
    NeighbourDirections &neighbourDirs,
                                                    Atom &focusAtom) {

    int newSupercellX =
            (focusAtom.getSupercellIndex()[0] + neighbourDirs.
    getSupercellXOffset()) % superCell.getSupercellSize()[0];
    int newSupercellY =
            (focusAtom.getSupercellIndex()[1] + neighbourDirs.
    getSupercellYOffset()) % superCell.getSupercellSize()[1];
    int newSupercellZ =
            (focusAtom.getSupercellIndex()[2] + neighbourDirs.
    getSupercellZOffset()) % superCell.getSupercellSize()[2];

    if (newSupercellX < 0) newSupercellX += superCell.getSupercellSize()
    [0];
```

```
124     if (newSupercellY < 0) newSupercellY += superCell.getSupercellSize()
    [1];
125     if (newSupercellZ < 0) newSupercellZ += superCell.getSupercellSize()
    [2];
126
127     int atomIndex = focusAtom.getUnitCellIndex() + neighbourDirs.
    getUnitCellIndexOffset();
128     return getAddress(superCell, newSupercellX, newSupercellY,
    newSupercellZ, atomIndex);
129 }
130
131 int
132 EfficientCrystalRelaxor::getAddress(SuperCell &superCell, int
    supercellXIndex, int supercellYIndex, int supercellZIndex,
133                                     int unitCellIndex) {
134     int boxSizeY = superCell.getSupercellSize()[1];
135     int boxSizeZ = superCell.getSupercellSize()[2];
136     int atomsPerUnitCell = superCell.getSupercellCrystals()[0][0][0]->
    getBasis().size();
137     int address =
138             (supercellXIndex * boxSizeY * boxSizeZ + supercellYIndex *
    boxSizeZ + supercellZIndex) * atomsPerUnitCell +
139             unitCellIndex;
140     return address;
141 }
142
143 void EfficientCrystalRelaxor::relax(double temperature) {
144
145     for (int i = 0; i < numberOfCycles; i++) {
146         int atomToMove = RandomGenerator::instance().rand(
```

```
    number_of_atoms);
147         relax(temperature, atomToMove);
148     }
149
150 //    std::cout << "Made " << acceptedMoves + rejectedMoves << "
    displacements, of which " << acceptedMoves << " were accepted" << std
    ::endl;
151 }
152
153 double *EfficientCrystalRelaxor::getHAtomX() const {
154     return hAtomX;
155 }
156
157 double *EfficientCrystalRelaxor::getHAtomY() const {
158     return hAtomY;
159 }
160
161 double *EfficientCrystalRelaxor::getHAtomZ() const {
162     return hAtomZ;
163 }
164
165 double *EfficientCrystalRelaxor::getHOccupancy() const {
166     return hOccupancy;
167 }
168
169 double *EfficientCrystalRelaxor::getHCharge() const {
170     return hCharge;
171 }
172
173 int *EfficientCrystalRelaxor::getHAtomType() const {
```

```
    return hAtomType;
}

int *EfficientCrystalRelaxor::getHNeighbourIndex() const {
    return hNeighbourIndex;
}

std::string getElementFromAtomName(std::string name) {
    std::string parsed = "";
    for (int i = 0; i < name.length(); i++) {
        if (isalpha(name[i])) {
            parsed += name[i];
        }
    }
    return parsed;
}

Atom EfficientCrystalRelaxor::getAtom(int atomIndex) {
    Vector3f uvwTot = b->getUVW(hAtomX[atomIndex], hAtomY[atomIndex],
    hAtomZ[atomIndex]);
    Vector3f superCellIndex = uvwTot.array().floor();
    Vector3f fracPos = uvwTot - superCellIndex;
    std::string type = idToElement[hAtomType[atomIndex]];
    std::string element = getElementFromAtomName(type);
    Atom returnAtom(type, hElemIndex[atomIndex],
                    element,
                    fracPos, superCellIndex.cast<int>(), (int) hCharge[
    atomIndex], hOccupancy[atomIndex]);
    returnAtom.setUnitCellIndex(hUnitCellIndex[atomIndex]);
    return returnAtom;
```

```
}

double EfficientCrystalRelaxor::springEnergy(int atomIndex) {
    Atom focusAtom = getAtom(atomIndex);
    double totalNeighbourEnergy = 0;
    for (int neighbourIndex = 0; neighbourIndex < nneighbours; neighbourIndex++) {
        int neighbourAddress = atomIndex * nneighbours + neighbourIndex + 1;
        int trueNeighbourIndex = hNeighbourIndex[neighbourAddress];
        if (trueNeighbourIndex > 0 && trueNeighbourIndex < (number_of_atoms * nneighbours)) {
            Atom neighbourAtom = getAtom(trueNeighbourIndex);
            totalNeighbourEnergy += strategy->calculateNeighbourEnergy(focusAtom, neighbourAtom);
        }
    }
    return totalNeighbourEnergy;
}

std::list<int> EfficientCrystalRelaxor::getNeighbourAddresses(SuperCell &superCell, int focusAtomIndex) {
    std::list<int> neighbourIndecies;
    Atom focusAtom = getAtom(focusAtomIndex);
    for (NeighbourDirections dir : _neighbourStructure[focusAtom.getUnitCellIndex()]) {
        neighbourIndecies.push_back(getNeighbourAddress(superCell, dir, focusAtom));
    }
    return neighbourIndecies;
```

```
}

int *EfficientCrystalRelaxor::getHAtomUnitCellIndex() const {
    return hUnitCellIndex;
}

void EfficientCrystalRelaxor::displace(int atomIndex, double displaceX,
    double displaceY, double displaceZ) {
    hAtomX[atomIndex] += displaceX;
    hAtomY[atomIndex] += displaceY;
    hAtomZ[atomIndex] += displaceZ;
}

void EfficientCrystalRelaxor::relax(double temperature, int atomIndex) {

    // get original energy
    double energy1 = springEnergy(atomIndex);

    // generate random displacement

    double shift = 0.02; // max displacement

    double dx = (2 * RandomGenerator::instance().rand() - 1) * shift;
    double dy = (2 * RandomGenerator::instance().rand() - 1) * shift;
    double dz = (2 * RandomGenerator::instance().rand() - 1) * shift;

    hAtomX[atomIndex] += dx;
    hAtomY[atomIndex] += dy;
    hAtomZ[atomIndex] += dz;

```

```
    double energy2 = springEnergy(atomIndex);
//    std::cout << "Changed energy " << energy1 << " vs " << energy2 << std::endl;
    if (energy2 < energy1) {
        // This move led to a lower energy, accept the move
        acceptedMoves += 1;
    } else {
        // The move led to a higher energy, accept based on Metropolis condition
        double rnum = RandomGenerator::instance().rand();
        double compareEnergy = exp(-(energy2 - energy1) / (boltzmannConst * temperature));
//        std::cout << "Comparing " << rnum << " with " << compareEnergy << std::endl;
        if (rnum <= compareEnergy)       // Metropolis Condition !!!
            acceptedMoves += 1;
        else {
            rejectedMoves += 1;
            hAtomX[atomIndex] -= dx;
            hAtomY[atomIndex] -= dy;
            hAtomZ[atomIndex] -= dz;
        };
    }
}

void EfficientCrystalRelaxor::printAtomLocs() {
    for (int i = 0; i < number_of_atoms; i++) {
        if (hOccupancy[i] != 0) {
            std::cout << hAtomType[i] << "\t" << hAtomX[i] << "\t" << hAtomY[i] << "\t" << hAtomZ[i] << std::endl;
```

```
279         }
280     }
281 }
282
283 int EfficientCrystalRelaxor::getNumberOfAtoms() const {
284     return number_of_atoms;
285 }
286
287 EfficientCrystalRelaxor::EfficientCrystalRelaxor(SuperCell &superCell) :
        EfficientCrystalRelaxor(superCell,
288
                                new ZnOSprings(
289
                                        &superCell), 4,
290
                                10) {
291
292 }
293
294 Vector3f EfficientCrystalRelaxor::getSuperCellSize() const {
295     return superCellSize;
296 }
297
298
299 NeighbourDirections::NeighbourDirections(int supercellXOffset, int
        supercellYOffset, int supercellZOffset,
300                                          int unitCellIndexOffset) :
        supercellXOffset(supercellXOffset),
301
        supercellYOffset(supercellYOffset),
```

```
302
    supercellZOffset(supercellZOffset),
303
    unitCellIndexOffset(unitCellIndexOffset) {}
304
305 int NeighbourDirections::getSupercellXOffset() const {
306     return supercellXOffset;
307 }
308
309 int NeighbourDirections::getSupercellYOffset() const {
310     return supercellYOffset;
311 }
312
313 int NeighbourDirections::getSupercellZOffset() const {
314     return supercellZOffset;
315 }
316
317 int NeighbourDirections::getUnitCellIndexOffset() const {
318     return unitCellIndexOffset;
319 }
320
321 NeighbourDirections::NeighbourDirections(Atom focusAtom, Atom
    neighbourAtom) {
322     supercellXOffset = neighbourAtom.getSupercellIndex()[0] - focusAtom.
    getSupercellIndex()[0];
323     supercellYOffset = neighbourAtom.getSupercellIndex()[1] - focusAtom.
    getSupercellIndex()[1];
324     supercellZOffset = neighbourAtom.getSupercellIndex()[2] - focusAtom.
    getSupercellIndex()[2];
325
```

```
    unitCellIndexOffset = neighbourAtom.getUnitCellIndex() - focusAtom.getUnitCellIndex();
}
```

C.5 CrystalFactory.cpp

```
//
// Created by Tim on 28/09/2017.
//

#include "CrystalFactory.h"

Crystal CrystalFactory::ZnO() {
    Bravais b(3.35, 5.22);
    return ZnO(b);
}

Crystal CrystalFactory::Y2Ti2O7() {
    std::vector<Atom> atoms;

    Bravais b(10.120);

    atoms.push_back(Atom("Y", 1, "Y", Vector3f(0.125, 0.625, 0.125), 3));
    atoms.push_back(Atom("Y", 2, "Y", Vector3f(0.375, 0.875, 0.125), 3));
    atoms.push_back(Atom("Y", 3, "Y", Vector3f(0.625, 0.125, 0.125), 3));
    atoms.push_back(Atom("Y", 4, "Y", Vector3f(0.875, 0.375, 0.125), 3));
```

```
21     atoms.push_back(Atom("Y", 5, "Y", Vector3f(0.125, 0.875, 0.375), 3))
       ;
22     atoms.push_back(Atom("Y", 6, "Y", Vector3f(0.375, 0.625, 0.375), 3))
       ;
23     atoms.push_back(Atom("Y", 7, "Y", Vector3f(0.625, 0.375, 0.375), 3))
       ;
24     atoms.push_back(Atom("Y", 8, "Y", Vector3f(0.875, 0.125, 0.375), 3))
       ;
25     atoms.push_back(Atom("Y", 9, "Y", Vector3f(0.125, 0.125, 0.625), 3))
       ;
26     atoms.push_back(Atom("Y", 10, "Y", Vector3f(0.375, 0.375, 0.625), 3)
       );
27     atoms.push_back(Atom("Y", 11, "Y", Vector3f(0.625, 0.625, 0.625), 3)
       );
28     atoms.push_back(Atom("Y", 12, "Y", Vector3f(0.875, 0.875, 0.625), 3)
       );
29     atoms.push_back(Atom("Y", 13, "Y", Vector3f(0.125, 0.375, 0.875), 3)
       );
30     atoms.push_back(Atom("Y", 14, "Y", Vector3f(0.375, 0.125, 0.875), 3)
       );
31     atoms.push_back(Atom("Y", 15, "Y", Vector3f(0.625, 0.875, 0.875), 3)
       );
32     atoms.push_back(Atom("Y", 16, "Y", Vector3f(0.875, 0.625, 0.875), 3)
       );
33
34     atoms.push_back(Atom("Ti", 1, "Ti", Vector3f(0.125, 0.125, 0.125),
       4));
35     atoms.push_back(Atom("Ti", 2, "Ti", Vector3f(0.375, 0.375, 0.125),
       4));
36     atoms.push_back(Atom("Ti", 3, "Ti", Vector3f(0.625, 0.625, 0.125),
```

```
4));
 atoms.push_back(Atom("Ti", 4, "Ti", Vector3f(0.875, 0.875, 0.125),
4));
 atoms.push_back(Atom("Ti", 5, "Ti", Vector3f(0.125, 0.375, 0.375),
4));
 atoms.push_back(Atom("Ti", 6, "Ti", Vector3f(0.375, 0.125, 0.375),
4));
 atoms.push_back(Atom("Ti", 7, "Ti", Vector3f(0.625, 0.875, 0.375),
4));
 atoms.push_back(Atom("Ti", 8, "Ti", Vector3f(0.875, 0.625, 0.375),
4));
 atoms.push_back(Atom("Ti", 9, "Ti", Vector3f(0.125, 0.625, 0.625),
4));
 atoms.push_back(Atom("Ti", 10, "Ti", Vector3f(0.375, 0.875, 0.625),
4));
 atoms.push_back(Atom("Ti", 11, "Ti", Vector3f(0.625, 0.125, 0.625),
4));
 atoms.push_back(Atom("Ti", 12, "Ti", Vector3f(0.875, 0.375, 0.625),
4));
 atoms.push_back(Atom("Ti", 13, "Ti", Vector3f(0.125, 0.875, 0.875),
4));
 atoms.push_back(Atom("Ti", 14, "Ti", Vector3f(0.375, 0.625, 0.875),
4));
 atoms.push_back(Atom("Ti", 15, "Ti", Vector3f(0.625, 0.375, 0.875),
4));
 atoms.push_back(Atom("Ti", 16, "Ti", Vector3f(0.875, 0.125, 0.875),
4));

 atoms.push_back(Atom("O2", 1, "O", Vector3f(0, 0.79515, 0), -2));
 atoms.push_back(Atom("O2", 2, "O", Vector3f(0, 0.20485, 0), -2));
```

```
53     atoms.push_back(Atom("O2", 3, "O", Vector3f(0.20485, 0, 0), -2));
54     atoms.push_back(Atom("O2", 4, "O", Vector3f(0.25, 0.25, 0.04515),
    -2));
55     atoms.push_back(Atom("O2", 5, "O", Vector3f(0.29515, 0.5, 0), -2));
56     atoms.push_back(Atom("O2", 6, "O", Vector3f(0.5, 0.29515, 0), -2));
57     atoms.push_back(Atom("O2", 7, "O", Vector3f(0.5, 0.70485, 0), -2));
58     atoms.push_back(Atom("O2", 8, "O", Vector3f(0.70485, 0.5, 0), -2));
59     atoms.push_back(Atom("O2", 9, "O", Vector3f(0.79515, 0, 0), -2));
60     atoms.push_back(Atom("O2", 10, "O", Vector3f(0.75, 0.75, 0.04515),
    -2));
61     atoms.push_back(Atom("O2", 11, "O", Vector3f(0, 0, 0.20485), -2));
62     atoms.push_back(Atom("O2", 12, "O", Vector3f(0, 0.5, 0.29515), -2));
63     atoms.push_back(Atom("O2", 13, "O", Vector3f(0.04515, 0.25, 0.25),
    -2));
64     atoms.push_back(Atom("O2", 14, "O", Vector3f(0.25, 0.04515, 0.25),
    -2));
65     atoms.push_back(Atom("O2", 15, "O", Vector3f(0.25, 0.45485, 0.25),
    -2));
66     atoms.push_back(Atom("O2", 16, "O", Vector3f(0.5, 0.5, 0.20485), -2)
    );
67     atoms.push_back(Atom("O2", 17, "O", Vector3f(0.45485, 0.25, 0.25),
    -2));
68     atoms.push_back(Atom("O2", 18, "O", Vector3f(0.5, 0, 0.29515), -2));
69     atoms.push_back(Atom("O2", 19, "O", Vector3f(0.54515, 0.75, 0.25),
    -2));
70     atoms.push_back(Atom("O2", 20, "O", Vector3f(0.75, 0.54515, 0.25),
    -2));
71     atoms.push_back(Atom("O2", 21, "O", Vector3f(0.75, 0.95485, 0.25),
    -2));
72     atoms.push_back(Atom("O2", 22, "O", Vector3f(0.95485, 0.75, 0.25),
```

```
-2));
 atoms.push_back(Atom("O2", 23, "O", Vector3f(0, 0.29515, 0.5), -2));
 atoms.push_back(Atom("O2", 24, "O", Vector3f(0, 0.70485, 0.5), -2));
 atoms.push_back(Atom("O2", 25, "O", Vector3f(0.20485, 0.5, 0.5), -2)
);
 atoms.push_back(Atom("O2", 26, "O", Vector3f(0.25, 0.25, 0.45485),
-2));
 atoms.push_back(Atom("O2", 27, "O", Vector3f(0.29515, 0, 0.5), -2));
 atoms.push_back(Atom("O2", 28, "O", Vector3f(0.25, 0.75, 0.54515),
-2));
 atoms.push_back(Atom("O2", 29, "O", Vector3f(0.5, 0.20485, 0.5), -2)
);
 atoms.push_back(Atom("O2", 30, "O", Vector3f(0.5, 0.79515, 0.5), -2)
);
 atoms.push_back(Atom("O2", 31, "O", Vector3f(0.70485, 0, 0.5), -2));
 atoms.push_back(Atom("O2", 32, "O", Vector3f(0.75, 0.75, 0.45485),
-2));
 atoms.push_back(Atom("O2", 33, "O", Vector3f(0.75, 0.25, 0.54515),
-2));
 atoms.push_back(Atom("O2", 34, "O", Vector3f(0.79515, 0.5, 0.5), -2)
);
 atoms.push_back(Atom("O2", 35, "O", Vector3f(0, 0.5, 0.70485), -2));
 atoms.push_back(Atom("O2", 36, "O", Vector3f(0, 0, 0.79515), -2));
 atoms.push_back(Atom("O2", 37, "O", Vector3f(0.04515, 0.75, 0.75),
-2));
 atoms.push_back(Atom("O2", 38, "O", Vector3f(0.25, 0.54515, 0.75),
-2));
 atoms.push_back(Atom("O2", 39, "O", Vector3f(0.25, 0.95485, 0.75),
-2));
 atoms.push_back(Atom("O2", 40, "O", Vector3f(0.5, 0, 0.70485), -2));
```

```
    atoms.push_back(Atom("O2", 41, "O", Vector3f(0.45485, 0.75, 0.75), -2));
    atoms.push_back(Atom("O2", 42, "O", Vector3f(0.5, 0.5, 0.79515), -2));
    atoms.push_back(Atom("O2", 43, "O", Vector3f(0.54515, 0.25, 0.75), -2));
    atoms.push_back(Atom("O2", 44, "O", Vector3f(0.75, 0.04515, 0.75), -2));
    atoms.push_back(Atom("O2", 45, "O", Vector3f(0.75, 0.45485, 0.75), -2));
    atoms.push_back(Atom("O2", 46, "O", Vector3f(0.95485, 0.25, 0.75), -2));
    atoms.push_back(Atom("O2", 47, "O", Vector3f(0.25, 0.75, 0.95485), -2));
    atoms.push_back(Atom("O2", 48, "O", Vector3f(0.75, 0.25, 0.95485), -2));

    atoms.push_back(Atom("O1", 1, "O", Vector3f(0, 0.5, 0), -2));
    atoms.push_back(Atom("O1", 2, "O", Vector3f(0.5, 0, 0), -2));
    atoms.push_back(Atom("O1", 3, "O", Vector3f(0.25, 0.75, 0.25), -2));
    atoms.push_back(Atom("O1", 4, "O", Vector3f(0.75, 0.25, 0.25), -2));
    atoms.push_back(Atom("O1", 5, "O", Vector3f(0, 0, 0.5), -2));
    atoms.push_back(Atom("O1", 6, "O", Vector3f(0.5, 0.5, 0.5), -2));
    atoms.push_back(Atom("O1", 7, "O", Vector3f(0.25, 0.25, 0.75), -2));
    atoms.push_back(Atom("O1", 8, "O", Vector3f(0.75, 0.75, 0.75), -2));

    return Crystal(b, atoms);
}


```

```
Crystal CrystalFactory::_Y2Ti2O7_broken(){
    std::vector<Atom> atoms;

    Bravais b(10.120);

    atoms.push_back(Atom("Y", 1, "Y", Vector3f(0.124123, 0.625, 0.125), 3));
    atoms.push_back(Atom("Y", 2, "Y", Vector3f(0.375, 0.875, 0.125), 3));
    atoms.push_back(Atom("Y", 3, "Y", Vector3f(0.625, 0.125, 0.125), 3));
    atoms.push_back(Atom("Y", 4, "Y", Vector3f(0.875, 0.375, 0.125), 3));
    atoms.push_back(Atom("Y", 5, "Y", Vector3f(0.125, 0.875, 0.375), 3));
    atoms.push_back(Atom("Y", 6, "Y", Vector3f(0.375, 0.625, 0.375), 3));
    atoms.push_back(Atom("Y", 7, "Y", Vector3f(0.625, 0.375, 0.375), 3));
    atoms.push_back(Atom("Y", 8, "Y", Vector3f(0.875, 0.125, 0.375), 3));
    atoms.push_back(Atom("Y", 9, "Y", Vector3f(0.125, 0.125, 0.625), 3));
    atoms.push_back(Atom("Y", 10, "Y", Vector3f(0.375, 0.375, 0.625), 3));
    atoms.push_back(Atom("Y", 11, "Y", Vector3f(0.625, 0.625, 0.625), 3));
    atoms.push_back(Atom("Y", 12, "Y", Vector3f(0.875, 0.875, 0.625), 3));
    atoms.push_back(Atom("Y", 13, "Y", Vector3f(0.125, 0.375, 0.875), 3)
```

```
);
 atoms.push_back(Atom("Y", 14, "Y", Vector3f(0.375, 0.125, 0.875), 3)
);
 atoms.push_back(Atom("Y", 15, "Y", Vector3f(0.625, 0.875, 0.875), 3)
);
 atoms.push_back(Atom("Y", 16, "Y", Vector3f(0.875, 0.625, 0.875), 3)
);

 atoms.push_back(Atom("Ti", 1, "Ti", Vector3f(0.125, 0.125, 0.125),
4));
 atoms.push_back(Atom("Ti", 2, "Ti", Vector3f(0.375, 0.375, 0.125),
4));
 atoms.push_back(Atom("Ti", 3, "Ti", Vector3f(0.625, 0.625, 0.125),
4));
 atoms.push_back(Atom("Ti", 4, "Ti", Vector3f(0.875, 0.875, 0.125),
4));
 atoms.push_back(Atom("Ti", 5, "Ti", Vector3f(0.125, 0.375, 0.375),
4));
 atoms.push_back(Atom("Ti", 6, "Ti", Vector3f(0.375, 0.125, 0.375),
4));
 atoms.push_back(Atom("Ti", 7, "Ti", Vector3f(0.625, 0.875, 0.375),
4));
 atoms.push_back(Atom("Ti", 8, "Ti", Vector3f(0.875, 0.625, 0.375),
4));
 atoms.push_back(Atom("Ti", 9, "Ti", Vector3f(0.125, 0.625, 0.625),
4));
 atoms.push_back(Atom("Ti", 10, "Ti", Vector3f(0.375, 0.875, 0.625),
4));
 atoms.push_back(Atom("Ti", 11, "Ti", Vector3f(0.625, 0.125, 0.625),
4));
```

```
146     atoms.push_back(Atom("Ti", 12, "Ti", Vector3f(0.875, 0.375, 0.625),
        4));
147     atoms.push_back(Atom("Ti", 13, "Ti", Vector3f(0.125, 0.875, 0.875),
        4));
148     atoms.push_back(Atom("Ti", 14, "Ti", Vector3f(0.375, 0.625, 0.875),
        4));
149     atoms.push_back(Atom("Ti", 15, "Ti", Vector3f(0.625, 0.375, 0.875),
        4));
150     atoms.push_back(Atom("Ti", 16, "Ti", Vector3f(0.875, 0.125, 0.875),
        4));
151
152     atoms.push_back(Atom("O2", 1, "O", Vector3f(0, 0.79515, 0), -2));
153     atoms.push_back(Atom("O2", 2, "O", Vector3f(0, 0.20485, 0), -2));
154     atoms.push_back(Atom("O2", 3, "O", Vector3f(0.20485, 0, 0), -2));
155     atoms.push_back(Atom("O2", 4, "O", Vector3f(0.25, 0.25, 0.04515),
        -2));
156     atoms.push_back(Atom("O2", 5, "O", Vector3f(0.29515, 0.5, 0), -2));
157     atoms.push_back(Atom("O2", 6, "O", Vector3f(0.5, 0.29515, 0), -2));
158     atoms.push_back(Atom("O2", 7, "O", Vector3f(0.5, 0.70485, 0), -2));
159     atoms.push_back(Atom("O2", 8, "O", Vector3f(0.70485, 0.5, 0), -2));
160     atoms.push_back(Atom("O2", 9, "O", Vector3f(0.79515, 0, 0), -2));
161     atoms.push_back(Atom("O2", 10, "O", Vector3f(0.75, 0.75, 0.04515),
        -2));
162     atoms.push_back(Atom("O2", 11, "O", Vector3f(0, 0, 0.20485), -2));
163     atoms.push_back(Atom("O2", 12, "O", Vector3f(0, 0.5, 0.29515), -2));
164     atoms.push_back(Atom("O2", 13, "O", Vector3f(0.04515, 0.25, 0.25),
        -2));
165     atoms.push_back(Atom("O2", 14, "O", Vector3f(0.25, 0.04515, 0.25),
        -2));
166     atoms.push_back(Atom("O2", 15, "O", Vector3f(0.25, 0.45485, 0.25),
```

```
-2));
167  atoms.push_back(Atom("O2", 16, "O", Vector3f(0.5, 0.5, 0.20485), -2)
);
168  atoms.push_back(Atom("O2", 17, "O", Vector3f(0.45485, 0.25, 0.25),
-2));
169  atoms.push_back(Atom("O2", 18, "O", Vector3f(0.5, 0, 0.29515), -2));
170  atoms.push_back(Atom("O2", 19, "O", Vector3f(0.54515, 0.75, 0.25),
-2));
171  atoms.push_back(Atom("O2", 20, "O", Vector3f(0.75, 0.54515, 0.25),
-2));
172  atoms.push_back(Atom("O2", 21, "O", Vector3f(0.75, 0.95485, 0.25),
-2));
173  atoms.push_back(Atom("O2", 22, "O", Vector3f(0.95485, 0.75, 0.25),
-2));
174  atoms.push_back(Atom("O2", 23, "O", Vector3f(0, 0.29515, 0.5), -2));
175  atoms.push_back(Atom("O2", 24, "O", Vector3f(0, 0.70485, 0.5), -2));
176  atoms.push_back(Atom("O2", 25, "O", Vector3f(0.20485, 0.5, 0.5), -2)
);
177  atoms.push_back(Atom("O2", 26, "O", Vector3f(0.25, 0.25, 0.45485),
-2));
178  atoms.push_back(Atom("O2", 27, "O", Vector3f(0.29515, 0, 0.5), -2));
179  atoms.push_back(Atom("O2", 28, "O", Vector3f(0.25, 0.75, 0.54515),
-2));
180  atoms.push_back(Atom("O2", 29, "O", Vector3f(0.5, 0.20485, 0.5), -2)
);
181  atoms.push_back(Atom("O2", 30, "O", Vector3f(0.5, 0.79515, 0.5), -2)
);
182  atoms.push_back(Atom("O2", 31, "O", Vector3f(0.70485, 0, 0.5), -2));
183  atoms.push_back(Atom("O2", 32, "O", Vector3f(0.75, 0.75, 0.45485),
-2));
```

```
184     atoms.push_back(Atom("O2", 33, "O", Vector3f(0.75, 0.25, 0.54515),
        -2));
185     atoms.push_back(Atom("O2", 34, "O", Vector3f(0.79515, 0.5, 0.5), -2)
        );
186     atoms.push_back(Atom("O2", 35, "O", Vector3f(0, 0.5, 0.70485), -2));
187     atoms.push_back(Atom("O2", 36, "O", Vector3f(0, 0, 0.79515), -2));
188     atoms.push_back(Atom("O2", 37, "O", Vector3f(0.04515, 0.75, 0.75),
        -2));
189     atoms.push_back(Atom("O2", 38, "O", Vector3f(0.25, 0.54515, 0.75),
        -2));
190     atoms.push_back(Atom("O2", 39, "O", Vector3f(0.25, 0.95485, 0.75),
        -2));
191     atoms.push_back(Atom("O2", 40, "O", Vector3f(0.5, 0, 0.70485), -2));
192     atoms.push_back(Atom("O2", 41, "O", Vector3f(0.45485, 0.75, 0.75),
        -2));
193     atoms.push_back(Atom("O2", 42, "O", Vector3f(0.5, 0.5, 0.79515), -2)
        );
194     atoms.push_back(Atom("O2", 43, "O", Vector3f(0.54515, 0.25, 0.75),
        -2));
195     atoms.push_back(Atom("O2", 44, "O", Vector3f(0.75, 0.04515, 0.75),
        -2));
196     atoms.push_back(Atom("O2", 45, "O", Vector3f(0.75, 0.45485, 0.75),
        -2));
197     atoms.push_back(Atom("O2", 46, "O", Vector3f(0.95485, 0.25, 0.75),
        -2));
198     atoms.push_back(Atom("O2", 47, "O", Vector3f(0.25, 0.75, 0.95485),
        -2));
199     atoms.push_back(Atom("O2", 48, "O", Vector3f(0.75, 0.25, 0.95485),
        -2));
200
```

```
    atoms.push_back(Atom("O1", 1, "O", Vector3f(0, 0.5, 0), -2));
    atoms.push_back(Atom("O1", 2, "O", Vector3f(0.5, 0, 0), -2));
    atoms.push_back(Atom("O1", 3, "O", Vector3f(0.25, 0.75, 0.25), -2));
    atoms.push_back(Atom("O1", 4, "O", Vector3f(0.75, 0.25, 0.25), -2));
    atoms.push_back(Atom("O1", 5, "O", Vector3f(0, 0, 0.5), -2));
    atoms.push_back(Atom("O1", 6, "O", Vector3f(0.5, 0.5, 0.5), -2));
    atoms.push_back(Atom("O1", 7, "O", Vector3f(0.25, 0.25, 0.75), -2));
    atoms.push_back(Atom("O1", 8, "O", Vector3f(0.75, 0.75, 0.75), -2));

    return Crystal(b, atoms);
}

Crystal CrystalFactory::ZnO(Bravais &b) {
    std::vector<Atom> atoms;

    Atom O1("O1", 1, "O", Vector3f(1./3., 2./3., 3./8.), -2);
    Atom O2("O2", 2, "O", Vector3f(2./3., 1./3., 7./8.), -2);
    Atom Zn1("Zn1", 1, "Zn", Vector3f(1./3., 2./3., 0), 2);
    Atom Zn2("Zn2", 2, "Zn", Vector3f(2./3., 1./3., 0.5), 2);

    atoms.push_back(Zn1);
    atoms.push_back(Zn2);
    atoms.push_back(O1);
    atoms.push_back(O2);

    return Crystal(b, atoms);
}
```

C.6 RandomGenerator.cpp

```
//
// Created by Tim on 14/03/2018.
//

#include <iostream>
#include <sstream>
#include "RandomGenerator.h"

RandomGenerator::RandomGenerator(unsigned int seed) {
//    std::cout << "PRNG seeded with " << seed << std::endl;
    this->seed = seed;
    _reseed(seed);
    _legacy_init();
}

RandomGenerator::RandomGenerator() : RandomGenerator(12345) {}

int RandomGenerator::_legacy_rand() {
    // Do not use this method, only exists for legacy testing purposes
    // use rand() instead;
    return std::rand();
}

double RandomGenerator::rand() {
    std::uniform_real_distribution<double> dis(0.0, 1.0);
    return dis(engine);
}

void RandomGenerator::_legacy_init() {
    for (int i = 0; i < 1; i++) {
```

```
        _legacy_rs[i] = _legacy_rand();
        _legacy_rnums[i] = 0;
    }
}


double RandomGenerator::_legacy_marsaglia() {
    return _legacy_marsaglia(0);
}

double RandomGenerator::_legacy_marsaglia(int whichR) {
    // whichR should be between 0-5 to select r1-r6 appropriately
    _legacy_rs[whichR] = _legacy_MWCcoeff * (_legacy_rs[whichR] &
   4294967295) + (_legacy_rs[whichR] >> 32);
    return (double) _legacy_rs[whichR] / 4294967295. / _legacy_MWCcoeff;
}

int RandomGenerator::rand(int max) {
    // generatures uid random numbers in [0,max)
    return (int) std::floor(rand() * max) % max;
}

std::string RandomGenerator::writeReport() {
    std::stringstream sstream;
    sstream << "# PRNG SEED : " << seed;
    return sstream.str();
}
```

C.7 RotationHelper.cpp

```
//
// Created by Tim on 26/09/2017.
//

#include "RotationHelper.h"

Matrix3f RotationHelper::rotateX(double theta) {
    // Returns a 3d Rotation Matrix to perform rotation about cartesian
    x
    Matrix3f m;
    m << 1, 0, 0,
            0, std::cos(theta), -std::sin(theta),
            0, std::sin(theta), std::cos(theta);
    return m;
}

Matrix3f RotationHelper::rotateY(double theta) {
    // Returns a 3d Rotation Matrix to perform rotation about cartesian
    y
    Matrix3f m;
    m << std::cos(theta), 0, std::sin(theta),
            0, 1, 0,
            -std::sin(theta), 0, std::cos(theta);
    return m;
}

Matrix3f RotationHelper::rotateZ(double theta) {
    // Returns a 3d Rotation Matrix to perform rotation about cartesian
    z
    Matrix3f m;
```

```


    m << std::cos(theta), -std::sin(theta), 0,
            std::sin(theta), std::cos(theta), 0,
            0, 0, 1;
    return m;
}

float RotationHelper::magnitude(Vector3f vect) {
    return std::sqrt(vect.dot(vect));
}

float RotationHelper::angleBetween(Vector3f vectA, Vector3f vectB) {
    return std::acos(vectA.dot(vectB) / (magnitude(vectA) * magnitude(
    vectB)));
}

float RotationHelper::toRad(double thetaDegree) {
    return thetaDegree * M_PI / 180.;
}

float RotationHelper::toDegree(double thetaRads) {
    return thetaRads * 180. / M_PI;
}
```

C.8 Atom.cpp

```
//
// Created by Tim on 27/09/2017.
//
```

```
4
5 #include <iostream>
6 #include "Atom.h"
7
8 #include "Bravais.h"
9
10 Atom::Atom(const std::string &atomType, int atomIndex, const std::string
     &elementType,
11            const Vector3f &fractionalPosition, const Vector3i &
    supercellIndex, int charge, double occupancy) :
12        elementType(elementType), fractionalPosition(fractionalPosition)
    , supercellIndex(supercellIndex),
13        charge(charge), occupancy(occupancy), atomType(atomType) {
14        atomName = atomType;
15        _atom_index = atomIndex;
16        _unit_cell_index = 0;
17        setUID();
18 }
19
20 Atom::Atom(const std::string &atomType, int atomIndex, const std::string
     &elementType, const Vector3f &fractionalPosition, int charge)
21        : Atom(atomType, atomIndex, elementType, fractionalPosition,
    Vector3i(0, 0, 0), charge, 1.0) {}
22
23
24 const std::string &Atom::getAtomName() const {
25     return atomName;
26 }
27
28 const std::string &Atom::getElementType() const {
```

```
29     return elementType;
30 }
31
32 const Vector3f &Atom::getFractionalPosition() const {
33     return fractionalPosition;
34 }
35
36 void Atom::setSupercellIndex(int supercellA, int supercellB, int
    supercellC) {
37     Atom::supercellIndex[0] = supercellA;
38     Atom::supercellIndex[1] = supercellB;
39     Atom::supercellIndex[2] = supercellC;
40 }
41
42 Vector3f Atom::getAbsolutePosition(Bravais *ref) {
43     return ref->getPosition(fractionalPosition) +
44            ref->getPosition(supercellIndex[0], supercellIndex[1],
    supercellIndex[2]);
45 }
46
47 const Vector3i &Atom::getSupercellIndex() const {
48     return supercellIndex;
49 }
50
51 double Atom::getCharge() const {
52     return charge;
53 }
54
55 bool Atom::operator==(const Atom &rhs) const {
56     return atomName == rhs.atomName &&
```

```
            atomType == rhs.atomType &&
            elementType == rhs.elementType &&
            fractionalPosition == rhs.fractionalPosition &&
            charge == rhs.charge &&
            occupancy == rhs.occupancy;
}

bool Atom::operator!=(const Atom &rhs) const {
    return !(rhs == *this);
}

const std::string Atom::getUID() const {
    return UID;
}

void Atom::setUID() {
    std::stringstream sstream;
    sstream << atomName << "_" << _atom_index << "_"
            << supercellIndex[0] << "_"
            << supercellIndex[1] << "_"
            << supercellIndex[2];
    UID = sstream.str();
}

void Atom::setFractionalPosition(const Vector3f &fractionalPosition) {
    Atom::fractionalPosition = fractionalPosition;
}

void Atom::setSupercellIndex(const Vector3i &supercellIndex) {
    Atom::supercellIndex = supercellIndex;
```

```
87 }
88
89 void Atom::setCharge(double charge) {
90     Atom::charge = charge;
91 }
92
93 void Atom::setOccupancy(double occupancy) {
94     Atom::occupancy = occupancy;
95 }
96
97 const std::string &Atom::getAtomType() const {
98     return atomType;
99 }
100
101 double Atom::getOccupancy() const {
102     return occupancy;
103 }
104
105 void Atom::setElementType(const std::string &elementType) {
106     Atom::elementType = elementType;
107 }
108
109 void Atom::setAtomName(const std::string &atomName) {
110     Atom::atomName = atomName;
111 }
112
113 int Atom::getUnitCellIndex() {
114     return _unit_cell_index;
115 }
116
```

```
void Atom::setUnitCellIndex(int newIndex) {
    _unit_cell_index = newIndex;

}

int Atom::getElementIndex() const {
    return _atom_index;
}
```

C.9 Bravais.cpp

```
//
// Created by Tim on 26/09/2017.
//

#include <iostream>
#include "Bravais.h"


// Helper functions

float TOLERANCE = 1e-3;

bool almostEqual(float lhs, float rhs) {
    return std::abs(lhs - rhs) < TOLERANCE;
}

// Bravais Class

Bravais::Bravais(const Matrix3f &bravais) : bravais(bravais) {
```

```
    initialiseReciprocalsInverses();
}
Bravais::Bravais(const float aMag, const float bMag, const float cMag,
    const float alpha, const float beta,
                 const float gamma) {

    Vector3f aVect(aMag, 0, 0);

    Matrix3f rotVect = RotationHelper::rotateZ(RotationHelper::toRad(
    gamma));

    Vector3f bVect = rotVect * aVect;

    rotVect = RotationHelper::rotateY(RotationHelper::toRad(-alpha));

    Vector3f cVect = rotVect * aVect;

    while (!almostEqual(RotationHelper::toRad(alpha), RotationHelper::
    angleBetween(bVect, cVect))) {
        std::cout << "rotating" << std::endl;
        rotVect = RotationHelper::rotateX(RotationHelper::toRad(1));
        cVect = rotVect * cVect;
    }

    aVect *= aMag / RotationHelper::magnitude(aVect);
    bVect *= bMag / RotationHelper::magnitude(bVect);
    cVect *= cMag / RotationHelper::magnitude(cVect);

    bravais.row(0) = aVect;
    bravais.row(1) = bVect;
```

```
    bravais.row(2) = cVect;

    // Verify obtained vectors match input requirements
    assert(almostEqual(aMag, RotationHelper::magnitude(aVect)));
    assert(almostEqual(bMag, RotationHelper::magnitude(bVect)));
    assert(almostEqual(cMag, RotationHelper::magnitude(cVect)));
    assert(almostEqual(RotationHelper::toRad(alpha), RotationHelper::angleBetween(bVect, cVect)));
    assert(almostEqual(RotationHelper::toRad(beta), RotationHelper::angleBetween(aVect, cVect)));
    assert(almostEqual(RotationHelper::toRad(gamma), RotationHelper::angleBetween(aVect, bVect)));

    initialiseReciprocalsInverses();
}

Bravais::Bravais(const Vector3f aVect, const Vector3f bVect, const Vector3f cVect) {
    bravais.row(0) = aVect;
    bravais.row(1) = bVect;
    bravais.row(2) = cVect;
    initialiseReciprocalsInverses();
}

Matrix3f Bravais::getReciprocal() const {
    return reciprocal;
}

Bravais::Bravais(const float aMag) : Bravais(aMag, aMag, aMag, 90, 90, 90) {
```

```
    // Constructs a cubic bravais lattice with lattice parameter aMag
}

Bravais::Bravais(const float aMag, const float bMag) : Bravais(aMag,
    aMag, bMag, 90, 90, 120) {
    // Constructs a hexagonal bravais lattice with lattice parameter
    aMag
}

Vector3f Bravais::getQ(float h, float k, float l) const {
    Matrix3f recip = getReciprocal();
    return Vector3f(h, k, l).transpose() * recip;
}
Vector3f Bravais::getQ(Vector3f hkls) const {
    return getQ(hkls[0], hkls[1], hkls[2]);
}

Vector3f Bravais::getPosition(float u, float v, float w) const {
    return Vector3f(u, v, w).transpose() * getBravais();
}

Vector3f Bravais::getPosition(Vector3f fractionalPosition) const {
    return getPosition(fractionalPosition[0], fractionalPosition[1],
    fractionalPosition[2]);
}

bool Bravais::operator==(const Bravais &rhs) const {
    return bravais == rhs.bravais;
}

```

```
bool Bravais::operator!=(const Bravais &rhs) const {
    return !(rhs == *this);
}

const Matrix3f &Bravais::getBravais() const {
    return bravais;
}

Vector3f Bravais::getUVW(float rx, float ry, float rz) const {
    return Vector3f(rx, ry, rz).transpose() * getBravaisInverse().matrix
    ();
}

Matrix3f Bravais::getBravaisInverse() const {
    return bravaisInverse;
}

Matrix3f Bravais::getReciprocalInverse() const {
    return reciprocalInverse;
}

Vector3f Bravais::getHKL(float qx, float qy, float qz) const {
    Vector3f qvect(qx, qy, qz);
    return qvect.transpose() * getReciprocalInverse().matrix();
}

Vector3f Bravais::getHKL(Vector3f qVector) const {
    return getHKL(qVector[0], qVector[1], qVector[2]);
}

```

```
std::string Bravais::writeReport() const{
    std::stringstream report;

    Matrix3f recip = getReciprocal();

    report << "# A : " << bravais(0,0) << " "<< bravais(0, 1) << " "<<
   bravais(0, 2) << " " << std::endl;
    report << "# B : " << bravais(1,0) << " "<< bravais(1, 1) << " "<<
   bravais(1, 2) << " " << std::endl;
    report << "# C : " << bravais(2,0) << " "<< bravais(2, 1) << " "<<
   bravais(2, 2) << " " << std::endl;
    report << "# A* : " << recip(0,0) << " "<< recip(0, 1) << " "<<
   recip(0, 2) << " " << std::endl;
    report << "# B* : " << recip(1,0) << " "<< recip(1, 1) << " "<<
   recip(1, 2) << " " << std::endl;
    report << "# C* : " << recip(2,0) << " "<< recip(2, 1) << " "<<
   recip(2, 2) << " ";

    return report.str();
}

Vector3f Bravais::getUVW(Vector3f position) const {
    return getUVW(position[0], position[1], position[2]);
}
```

C.10 Crystal.cpp

```
//
// Created by Tim on 28/09/2017.
//
```

```

#include <iostream>
#include <map>
#include "Crystal.h"


Crystal::Crystal(const Bravais &bravaisLattice, const std::vector<Atom>
    &basis) : bravaisLattice(bravaisLattice) {
    for (int i = 0; i < basis.size(); i++) {
        Atom newAtoms = basis[i];
        newAtoms.setUnitCellIndex(i);
        this->basis.push_back(newAtoms);
    }
}

std::string Crystal::writeCellFile() {
    std::stringstream sstream;

    sstream << "%BLOCK lattice_cart\n\tANG\n";

    sstream << "\t\t" << std::fixed << std::setprecision(14) <<
    getBravaisLattice().getBravais().row(0) << "\n";
    sstream << "\t\t" << std::fixed << std::setprecision(14) <<
    getBravaisLattice().getBravais().row(1) << "\n";
    sstream << "\t\t" << std::fixed << std::setprecision(14) <<
    getBravaisLattice().getBravais().row(2) << "\n";
    sstream << "%ENDBLOCK lattice_cart\n\n%BLOCK positions_frac\n";

    for (Atom currentAtom : basis) {
        sstream << "\t" << currentAtom.getElementType() << "\t" << std::
```

```
    fixed << std::setprecision(15) << currentAtom.getFractionalPosition()
    .transpose() << "\n";
30     }
31     sstream << "%ENDBLOCK positions_frac\n";
32     return sstream.str();
33 }
34
35 std::vector<Atom *> Crystal::findNeighbours(const Atom &toAtom, int
    numberOfNeighbours, bool (*filter)(Atom)) {
36     // TODO : implement this....
37
38     std::multimap<double, const Atom*> allNeighboursMultiMap;
39
40
41     for (const Atom &atom : getBasis()) {
42         if (atom != toAtom && filter(atom)) {
43             allNeighboursMultiMap.insert(std::pair<double, const Atom*>(
    distance(toAtom, atom), &atom));
44         }
45     }
46
47     std::vector<Atom*> neighbours;
48
49     for (std::pair<double, const Atom*> candidate :
    allNeighboursMultiMap) {
50         Atom *newNeighbour = const_cast<Atom*>(candidate.second);
51         neighbours.push_back(newNeighbour);
52         if (neighbours.size() == numberOfNeighbours) break;
53     }
54
```

```
    return neighbours;
}

std::vector<Atom *> Crystal::findNeighbours(const Atom &toAtom, int
    numberOfNeighbours) {
    auto nofilter = [](Atom a) { return true;};
    return findNeighbours(toAtom, numberOfNeighbours, nofilter);
}

double Crystal::distance(const Atom &atom1, const Atom &atom2) {
    Vector3f cartesian1 = bravaisLattice.getPosition(atom1.
    getFractionalPosition());
    Vector3f cartesian2 = bravaisLattice.getPosition(atom2.
    getFractionalPosition());

    return RotationHelper::magnitude(cartesian2 - cartesian1);
}

void Crystal::setSuperCellIndex(int supercellA, int supercellB, int
    supercellC) {
    for (Atom &a : basis) {
        a.setSupercellIndex(supercellA, supercellB, supercellC);
        a.setUID();
    }

}

bool Crystal::operator==(const Crystal &rhs) const {
    if (basis.size() != rhs.basis.size()) {
        return false;
```

```
    }

    for (int i = 0; i < basis.size(); i++) {
        if (basis[i] != rhs.getBasis()[i]) {
            return false;
        }
    }

    return bravaisLattice == rhs.bravaisLattice;
}

bool Crystal::operator!=(const Crystal &rhs) const {
    return !(rhs == *this);
}

const Bravais &Crystal::getBravaisLattice() const {
    return bravaisLattice;
}

const std::vector<Atom> &Crystal::getBasis() const {
    return basis;
}

void Crystal::updateAtom(const Atom &newAtom) {
    Atom mutableAtom = newAtom;
    for (Atom &a : basis) {
        if (a.getUID() == newAtom.getUID()) {
            a = newAtom;
            return;
        }
```

```
    }
    mutableAtom.setUnitCellIndex(basis.size() + 1);
    basis.push_back(mutableAtom);
}

std::vector<Atom> Crystal::getUnfilteredList() {
    return getBasis();
}
```

C.11 SuperCell.cpp

```
//
// Created by Tim on 08/10/2017.
//

#include <iostream>
#include <algorithm>
#include "SuperCell.h"
#include "../helpers/CrystalMaths.h"

SuperCell::SuperCell(Crystal baseCrystal, int size_a, int size_b, int
    size_c) {
    if (size_a < 3 || size_b < 3 || size_c < 3) {
        throw SUPERCELL_TOO_SMALL;
    }

    this->size_a = size_a;
    this->size_b = size_b;
    this->size_c = size_c;

```

```
19
20      supercellCrystals.resize(size_c);
21      for (int i = 0; i < size_c; i++) {
22          supercellCrystals[i].resize(size_b);
23          for (int j = 0; j < size_b; j++) {
24              supercellCrystals[i][j].resize(size_a);
25          }
26      }
27
28      for (int i = 0; i < size_a; i++) {
29          for (int j = 0; j < size_b; j++) {
30              for (int k = 0; k < size_c; k++) {
31                  supercellCrystals[i][j][k] = new Crystal(baseCrystal);
32                  supercellCrystals[i][j][k]->setSuperCellIndex(i, j, k);
33              }
34          }
35      }
36
37      int uniqueTypesFound = 1;
38      for (Atom a : supercellCrystals[0][0][0]->getBasis()) {
39          if (!_legacy_type.count(a.getAtomName())) {
40              _legacy_type.insert(std::pair<std::string, int>(a.
    getAtomName(), uniqueTypesFound));
41              _legacy_type_map.insert(std::pair<int, std::string>(
    uniqueTypesFound, a.getAtomName()));
42              uniqueTypesFound++;
43          }
44      }
45  }
46
```

```
const std::vector<std::vector<std::vector<Crystal *>>> &SuperCell::
    getSupercellCrystals() const {
    return supercellCrystals;
}

std::string SuperCell::writeToFile() {
    std::stringstream sstream;

    sstream << "%BLOCK lattice_cart\n\tANG\n";

    Vector3f aVect = supercellCrystals[0][0][0]->getBravaisLattice().
    getBravais().row(0);
    Vector3f bVect = supercellCrystals[0][0][0]->getBravaisLattice().
    getBravais().row(1);
    Vector3f cVect = supercellCrystals[0][0][0]->getBravaisLattice().
    getBravais().row(2);

    aVect *= size_a;
    bVect *= size_b;
    cVect *= size_c;

    sstream << "\t\t" << std::fixed << std::setprecision(14) << aVect.
    transpose() << "\n";
    sstream << "\t\t" << std::fixed << std::setprecision(14) << bVect.
    transpose() << "\n";
    sstream << "\t\t" << std::fixed << std::setprecision(14) << cVect.
    transpose() << "\n";
    sstream << "%ENDBLOCK lattice_cart\n\n%BLOCK positions_frac\n";

    for (int i = 0; i < size_a; i++) {
```

```
        for (int j = 0; j < size_b; j++) {
            for (int k = 0; k < size_b; k++) {
                for (const Atom &a : supercellCrystals[i][j][k]->getBasis()) {
                    Vector3i supercellIndex = a.getSupercellIndex();
                    sstream << "\t" << a.getElementType() << "\t" << std::fixed << std::setprecision(15)
                            << (a.getFractionalPosition()[0] + supercellIndex[0]) / size_a << "\t"
                            << (a.getFractionalPosition()[1] + supercellIndex[1]) / size_b
                            << "\t" << (a.getFractionalPosition()[2] + supercellIndex[2]) / size_c << "\n";
                }
            }
        }
    }
    sstream << "%ENDBLOCK positions_frac\n";
    return sstream.str();
}

std::vector<Atom> SuperCell::getAllAtomsInNearbyCells(Atom toAtom) {
    return getAllAtomsInNearbyCells(toAtom, [](Atom a) {
        return true;
    });
}

std::vector<Atom> SuperCell::getAllAtomsInNearbyCells(Atom toAtom, std::function<bool(Atom)> filterFunc) {
    Vector3i centralIndex = toAtom.getSupercellIndex();
```

```
    int cellsToVisitA[3] = {mod((centralIndex[0] - 1), size_a),
                            (centralIndex[0]),
                            mod((centralIndex[0] + 1), size_a)};

    int cellsToVisitB[3] = {mod(centralIndex[1] - 1, size_b),
                            (centralIndex[1]),
                            mod(centralIndex[1] + 1, size_b)};
    int cellsToVisitC[3] = {mod(centralIndex[2] - 1, size_c),
                            (centralIndex[2]),
                            mod(centralIndex[2] + 1, size_c)};

    std::vector<Atom> atoms;
    Bravais bravais = getBravais();
    Vector3f supercellDims = bravais.getPosition(size_a, size_b, size_c);

    for (int i : cellsToVisitA) {
        for (int j : cellsToVisitB) {
            for (int k : cellsToVisitC) {
//                std::cout << "checkign neighbours in " << i << j << k << std::endl;
                std::vector<Atom> filtered = supercellCrystals[i][j][k]->filter(filterFunc);

                for (std::vector<Atom>::iterator it = filtered.begin(); it != filtered.end(); it++) {
                    Vector3f delta = distance(*it, toAtom);
                    if (    true || (
                            std::abs(delta[0]) < std::abs(supercellDims
```

```
[0]) / 2. &&
                std::abs(delta[1]) < std::abs(supercellDims[1]) / 2. &&
                std::abs(delta[2]) < std::abs(supercellDims[2]) / 2.
                        )) {
            atoms.push_back(*it);
        } else {
//              std::cout << "rejected " << delta[0] << ", " << delta[1] << ", " << delta[2] << " not " << supercellDims[0] << ", " << supercellDims[1] << ", " << supercellDims[2] << std::endl;
        }
    }
  }
 }
}

 return atoms;
}

std::string SuperCell::write_legacy() {
 // superIndexX+1   superIndexY+1   superIndexY+1   imole   atom_index_unitcell itype   x   y   z   spinx   spiny   spinz   charge
 std::stringstream sstream;

 for (int i = 0; i < size_a; i++) {
    for (int j = 0; j < size_b; j++) {
        for (int k = 0; k < size_b; k++) {
            for (int atomIndex = 0; atomIndex < supercellCrystals[i
```

```
][j][k]->getBasis().size(); atomIndex++) {
                    const Atom &a = supercellCrystals[i][j][k]->getBasis
()[atomIndex];
                    std::string final_spacer = "  ";
                    if (a.getCharge() > 0) {
                        final_spacer += " ";
                    }

                    int itype = _legacy_type.find(a.getAtomName())->
second;

                    if (a.getOccupancy() == 0) {
                        itype = 5;
                    }

                    sstream << i + 1 << " " << j + 1 << " " << k + 1
                            << " " << 1 << " " << atomIndex + 1 << " "
<< itype
                            << std::setprecision(6) << std::fixed
                            << "   " << a.getFractionalPosition()[0] <<
"   " << a.getFractionalPosition()[1] << "   "
                            << a.getFractionalPosition()[2]
                            << "   " << 0. << "   " << 0. << "   " << 0.
 << final_spacer << a.getCharge() << std::endl;
                }
            }
        }
    }
    return sstream.str();
}
```

```
167
168 std::vector<Atom> SuperCell::getAllAtoms() {
169     std::vector<Atom> atoms;
170     for (int i = 0; i < size_a; i++) {
171         for (int j = 0; j < size_b; j++) {
172             for (int k = 0; k < size_b; k++) {
173                 for (int atomIndex = 0; atomIndex < supercellCrystals[i
    ][j][k]->getBasis().size(); atomIndex++) {
174                     atoms.push_back(supercellCrystals[i][j][k]->getBasis
    ()[atomIndex]);
175                 }
176             }
177         }
178     }
179     return atoms;
180 }
181
182 void SuperCell::updateAtom(const Atom &newAtom) {
183     Vector3i superCellIndex = newAtom.getSupercellIndex();
184     supercellCrystals[superCellIndex[0]][superCellIndex[1]][
    superCellIndex[2]]->updateAtom(newAtom);
185 }
186
187 std::vector<Atom> SuperCell::getUnfilteredList() {
188     return getAllAtoms();
189 }
190
191 Vector3i SuperCell::getSupercellSize() {
192     return Eigen::Vector3i(size_a, size_b, size_c);
193 }
```

```

Vector3f SuperCell::distance(Atom &a1, Atom &a2) {
    // Calculates the distance, implementing Periodic Bounday Conditions
     (PBC)

    Bravais b = getBravais();

    Vector3f pos1 = a1.getAbsolutePosition(&b);
    Vector3f pos2 = a2.getAbsolutePosition(&b);

    Vector3f delta = pos2 - pos1;

    Vector3f boxlength = b.getPosition(size_a, size_b, size_c).cwiseAbs
    ();

    // Implement PBCs
    if (delta[0] > boxlength[0] * 0.5) delta[0] -= boxlength[0];
    if (delta[0] <= -boxlength[0] * 0.5) delta[0] += boxlength[0];

    if (delta[1] > boxlength[1] * 0.5) delta[1] -= boxlength[1];
    if (delta[1] <= -boxlength[1] * 0.5) delta[1] += boxlength[1];

    if (delta[2] > boxlength[2] * 0.5) delta[2] -= boxlength[2];
    if (delta[2] <= -boxlength[2] * 0.5) delta[2] += boxlength[2];

    return delta;
}

const Bravais &SuperCell::getBravais() {
    return supercellCrystals[0][0][0]->getBravaisLattice();
```

```
}

std::vector<Atom>
SuperCell::getNearestNeighbours(Atom toAtom, int numberOfNeighbours, std
    ::function<bool(Atom)> filterFunc) {
    std::vector<Atom> neighbours = getAllAtomsInNearbyCells(toAtom,
    filterFunc);

    auto comparator = [&](Atom a1, Atom a2) {
        double distance1 = magnitude(distance(a1, toAtom));
        double distance2 = magnitude(distance(a2, toAtom));
        return distance1 < distance2;
    };

    int numberFound = neighbours.size();
    int trueNumberOfNeighbours = std::min(numberFound,
    numberOfNeighbours);

    std::sort(neighbours.begin(), neighbours.end(), comparator);

    // Avoid the nearest neighbour as this is actually the atom itself
    with distance 0
    std::vector<Atom> returnAtoms(neighbours.begin() + 1, neighbours.
    begin() + trueNumberOfNeighbours + 1);
    return returnAtoms;
}


const bool defaultCompare(Atom a) {
    return true;
```

```
}

std::vector<Atom> SuperCell::getNearestNeighbours(Atom toAtom, int
    numberOfNeighbours) {
    std::vector<Atom> neighbours;

//    for (int i = 0; i < numberOfNeighbours; i++) {
//        neighbours.push_back(getAllAtoms()[i]);
//    }
//    return neighbours;
//
    return getNearestNeighbours(toAtom, numberOfNeighbours,
    defaultCompare);
}


int SuperCell::getTypeIdFromElement(std::string element) {
    return _legacy_type[element];
}

std::string SuperCell::getElementFromTypeId(int elementId) {
    return _legacy_type_map[elementId];
}

const std::map<std::string, int> &SuperCell::getTypeIdFromElementMap()
    const {
    return _legacy_type;
}

const std::map<int, std::string> &SuperCell::getElementFromTypeIdMap()
```

```
    const {
     return _legacy_type_map;
}
```

C.12 ChainedMutator.h

```
//
// Created by Tim on 28/03/2018.
//

#ifndef TBALLSNSPRINGS_CHAINEDMUTATOR_H
#define TBALLSNSPRINGS_CHAINEDMUTATOR_H


#include "CrystalMutator.h"

class ChainedMutator : public CrystalMutator {
public:
    void mutateCrystal(SuperCell *crystalToModify) override;
    void setNext(ChainedMutator *next);

    virtual bool isFinished(SuperCell *crystalToModify);
    virtual bool canFinishEarly();

    ChainedMutator *next = nullptr;

};


#endif //TBALLSNSPRINGS_CHAINEDMUTATOR_H
```

C.13 CrystalMutator.h

```
//
// Created by Tim on 14/03/2018.
//

#ifndef TBALLSNSPRINGS_VACANCYCREATOR_H
#define TBALLSNSPRINGS_VACANCYCREATOR_H


#include "../objects/FilterableAtoms.h"
#include "../objects/SuperCell.h"

class CrystalMutator {
    // Allows for arbitrary crystal processing based on some seletion criteria (filter function) and some
    // process to do to the crystal on all filtered atoms (process function)

    // See SimpleO1Vacancy for examples

public:
    virtual void mutateCrystal(SuperCell *crystalToModify);
    virtual bool filter(Atom) { return true; }

    virtual bool process(Atom, SuperCell *) = 0;

    int changesMade = 0;

};
```

```


#endif //TBALLSNSPRINGS_VACANCYCREATOR_H
```

C.14 CycleSuperCell.h

```
//
// Created by Tim on 21/03/2018.
//

#ifndef TBALLSNSPRINGS_CYCLESUPERCELL_H
#define TBALLSNSPRINGS_CYCLESUPERCELL_H


#include "../objects/SuperCell.h"

class CycleSuperCell {
public:
    // This class allows iterating over every subcell in a super and performing some action

    // Method runs over all subcells in a Supercell and performs the process_subcell operation on them
    void execute(int nCycles, SuperCell &superCell);

    // Implement this method for the given crystal
    virtual void process_subcell(Vector3i supercellIndex, SuperCell &superCell) = 0;
};

```

```

#endif //TBALLSNSPRINGS_CYCLESUPERCELL_H
```

C.15 CrystalEnergyCalculator.h

```
//
// Created by Tim on 23/04/2018.
//
#ifndef TBALLSNSPRINGS_CRYSTALENERGYCALCULATOR_H
#define TBALLSNSPRINGS_CRYSTALENERGYCALCULATOR_H

#include <iostream>
#include "../objects/FilterableAtoms.h"
#include "RandomGenerator.h"
#include "../objects/SuperCell.h"
#include "CrystalMaths.h"
#include "../Calculators/SpringEnergy/SpringEnergyStrategy.h"

#include <Eigen/Dense>

using Eigen::Vector4i;

class NeighbourDirections;

class EfficientCrystalRelaxor {
public:
    const double boltzmannConst = 8.6173e-5;

    explicit EfficientCrystalRelaxor(SuperCell &superCell);

```

```
    EfficientCrystalRelaxor(SuperCell &superCell, SpringEnergyStrategy *
   strategy, int nneighbours, int numberOfCycles);

    virtual ~EfficientCrystalRelaxor();

    std::vector<std::vector<NeighbourDirections>> &getNeighbourStructure
   (SuperCell &superCell);

    double *getHAtomX() const;

    double *getHAtomY() const;

    double *getHAtomZ() const;

    double *getHOccupancy() const;

    double *getHCharge() const;

    int *getHAtomType() const;

    int *getHNeighbourIndex() const;

    int *getHAtomUnitCellIndex() const;

    int getAddress(SuperCell &superCell, int supercellXIndex, int
   supercellYIndex, int supercellZIndex, int unitCellIndex);

    int getNeighbourAddress(SuperCell &superCell, NeighbourDirections &
   neighbourDirs, Atom &focusAtom);

```

```
52     std::list<int> getNeighbourAddresses(SuperCell &superCell, int
    focusAtomIndex);
53
54     int getNumberOfAtoms() const;
55
56     Vector3f getSuperCellSize() const;
57
58     Atom getAtom(int atomIndex);
59
60     void relax(double temperature);
61     void relax(double temperature, int atomIndex);
62
63     void displace(int atomIndex, double displaceX, double displaceY,
    double displaceZ);
64
65     double springEnergy(int atomIndex);
66
67     void printAtomLocs();
68
69 private:
70     void analyseNeighbourStructure(SuperCell &superCell);
71
72     void setupArrays(SuperCell &superCell);
73
74     SpringEnergyStrategy *strategy;
75
76     double *hAtomX, *hAtomY, *hAtomZ, *hOccupancy, *hCharge;
77     int *hAtomType, *hNeighbourIndex, *hElemIndex, number_of_atoms,
    atoms_per_unit_cell, nneighbours, *hOrigAtomType, *hUnitCellIndex;
78
```

```
    int rejectedMoves, acceptedMoves, numberOfCycles;

    Vector3f superCellSize;

    std::vector<std::vector<NeighbourDirections>> _neighbourStructure;
    std::map<int, std::string> idToElement;
    std::map<std::string, int> elementToId;
    Bravais *b;

};


class NeighbourDirections {
public:
    NeighbourDirections(int supercellXOffset, int supercellYOffset, int
   supercellZOffset, int unitCellIndexOffset);
    NeighbourDirections(Atom a1, Atom a2);

    int getSupercellXOffset() const;

    int getSupercellYOffset() const;

    int getSupercellZOffset() const;

    int getUnitCellIndexOffset() const;

private:
    int supercellXOffset, supercellYOffset, supercellZOffset,
   unitCellIndexOffset;
};
```

```
107
108
109 //double calculateSpringEnergy(SuperCell &superCell, Vector4i atomCoord,
        SpringEnergyStrategy *strategy) {
110 //    int i = atomCoord[0];
111 //    int j = atomCoord[1];
112 //    int k = atomCoord[2];
113 //    int l = atomCoord[3];
114 //
115 //    Atom focusAtom = superCell.getSupercellCrystals()[i][j][k]->
       getBasis()[l];
116 //
117 //    double totalEnergy = 0;
118 //    for (Atom a : superCell.getNearestNeighbours(focusAtom, 20)) {
119 //        for (SpringEnergyClauses clause : strategy->getClauses()) {
120 //            if (clause.validate(a, focusAtom)) {
121 //                totalEnergy += clause.calculateNeighbourEnergy(a,
       focusAtom);
122 //                break;
123 //            }
124 //        }
125 //    }
126 //
127 //    return totalEnergy;
128 //}
129
130
131 #endif //TBALLSNSPRINGS_CRYSTALENERGYCALCULATOR_H
```

C.16 CrystalFactory.h

```
//
// Created by Tim on 28/09/2017.
//

#ifndef EIGENTUT_CRYSTALFACTORY_H
#define EIGENTUT_CRYSTALFACTORY_H


#include "../objects/Crystal.h"

class CrystalFactory {
public:

    static Crystal ZnO(Bravais &b);
    static Crystal ZnO();
    static Crystal Y2Ti2O7();

    // For testing
    static Crystal _Y2Ti2O7_broken();

};


#endif //EIGENTUT_CRYSTALFACTORY_H
```

C.17 CrystalMaths.h

```
//
```

```
// Created by Tim on 21/03/2018.
//

#ifndef TBALLSNSPRINGS_CRYSTALMATHS_H
#define TBALLSNSPRINGS_CRYSTALMATHS_H

inline int mod(int a, int b) {
    // Returns a % b with negatives rolling back around
    // e.g. mod(-1, 4) = 3

    if (a < 0) a += b;
    return a % b;
}

inline double magnitude(VectorXf v) {
    double sqSum = 0;
    for (int i = 0; i < v.size(); i++) {
        sqSum += std::pow(v[i], 2);
    }
    return std::sqrt(sqSum);
}

#endif //TBALLSNSPRINGS_CRYSTALMATHS_H
```

C.18 RandomGenerator.h

```
//
// Created by Tim on 14/03/2018.
//

```

```
5  #ifndef TBALLSNSPRINGS_RANDOMGENERATOR_H
6  #define TBALLSNSPRINGS_RANDOMGENERATOR_H
7
8
9  #include <random>
10
11 class RandomGenerator {
12     // Random generator ensures singleton access to PRNG
13
14 public:
15     static RandomGenerator& instance(unsigned int seed)
16     {
17         static RandomGenerator INSTANCE(seed);
18         return INSTANCE;
19     }
20
21     static RandomGenerator& instance() {
22         return instance(12345);
23     }
24
25     void _reseed(unsigned int seed) {
26         engine.seed(seed);
27         std::srand(seed);
28     }
29
30     static RandomGenerator* _getTestingInstance(unsigned int seed) {
31         // THIS METHOD OVERRIDES DESIRED SINGLETON BEHAVIOUR
32         // This method should only be used for testing purposes.
33
34         RandomGenerator *inst = new RandomGenerator(seed);
```

```
        return inst;
    }

    // Delete unacceptable methods to avoid accidental copies of
   singleton.
    RandomGenerator(RandomGenerator const &) = delete;
    void operator=(RandomGenerator const &)  = delete;

    int _legacy_rand();
    void _legacy_init();
    double _legacy_marsaglia();
    double _legacy_marsaglia(int whichR);


    virtual double rand();
    int rand(int maxN);

    std::string writeReport();

protected:
    RandomGenerator();
    explicit RandomGenerator(unsigned int seed);

private:

    std::mt19937 engine;

    int seed;

    unsigned long long int _legacy_rs[6];
```

```
    double _legacy_rnums[6];
    const unsigned long long int _legacy_MWCcoeff = 2141354214;
};

#endif //TBALLSNSPRINGS_RANDOMGENERATOR_H
```

C.19 RotationHelper.h

```
//
// Created by Tim on 26/09/2017.
//

#ifndef EIGENTUT_ROTATIONHELPER_H
#define EIGENTUT_ROTATIONHELPER_H

#include <Eigen/Dense>
#define _USE_MATH_DEFINES
#include <cmath>

using namespace Eigen;

class RotationHelper {

public:

    static Matrix3f rotateX(double theta);
    static Matrix3f rotateY(double theta);
    static Matrix3f rotateZ(double theta);

    static float angleBetween(Vector3f vectA, Vector3f vectB);
```

```
    static float magnitude(Vector3f vect);

    static float toDegree(double thetaRads);
    static float toRad(double thetaDegree);

};


#endif //EIGENTUT_ROTATIONHELPER_H
```

C.20 Atom.h

```
//
// Created by Tim on 27/09/2017.
//

#ifndef EIGENTUT_ATOM_H
#define EIGENTUT_ATOM_H

#include <Eigen/Dense>

using Eigen::Vector3f;
using Eigen::Vector3i;

class Bravais;

class Atom {
public:
    // TODO: Implement spin
    Atom(const std::string &atomType, int atomIndex, const std::string &
```

```
elementType, const Vector3f &fractionalPosition,
      const Vector3i &supercellIndex, int charge, double occupancy);

 Atom(const std::string &atomType, int atomIndex, const std::string &
elementType, const Vector3f &fractionalPosition, int charge);

 const std::string &getAtomName() const;

 const std::string &getElementType() const;

 const Vector3f &getFractionalPosition() const;

 const Vector3i &getSupercellIndex() const;

 Vector3f getAbsolutePosition(Bravais *ref);

 int getElementIndex() const;

 void setSupercellIndex(int supercellA, int supercellB, int
supercellC);

 bool operator==(const Atom &rhs) const;

 bool operator!=(const Atom &rhs) const;

 double getCharge() const;

 void setUnitCellIndex(int newIndex);

 const std::string &getAtomType() const;
```

```
46
47     double getOccupancy() const;
48
49     const std::string getUID() const;
50
51     void setUID();
52
53     void setFractionalPosition(const Vector3f &fractionalPosition);
54
55     void setSupercellIndex(const Vector3i &supercellIndex);
56
57     void setCharge(double charge);
58
59     void setOccupancy(double occupancy);
60
61     void setElementType(const std::string &elementType);
62
63     void setAtomName(const std::string &atomName);
64
65     int getUnitCellIndex();
66
67 private:
68     std::string atomName, atomType, elementType;
69     std::string UID;
70     Vector3f fractionalPosition;
71     Vector3i supercellIndex;
72     double charge;
73     double occupancy;
74     int _atom_index, _unit_cell_index;
75
```

```
};


#endif //EIGENTUT_ATOM_H
```

C.21 Bravais.h

```
//
// Created by Tim on 26/09/2017.
//

#ifndef EIGENTUT_BRAVAIS_H
#define EIGENTUT_BRAVAIS_H


#include <Eigen/Dense>
#include <Eigen/LU>
#include "../helpers/RotationHelper.h"

using namespace Eigen;

class Bravais {
public:

    Bravais(const Matrix3f &bravais);  // Explicit constructor
    Bravais(const Vector3f aVect, const Vector3f bVect, const Vector3f
   cVect); // constuctor given 3 vectors

    // given arbitrary lattice parameters
    Bravais(const float aMag, const float bMag, const float cMag, const
```

```
    float alpha, const float beta, const float gamma);

    // cubic
    Bravais(const float aMag);

    // hexagonal
    Bravais(const float aMag, const float cMag);

    void initialiseReciprocalsInverses() {
        bravaisInverse = bravais.inverse();

        Vector3f aVect = bravais.row(0);
        Vector3f bVect = bravais.row(1);
        Vector3f cVect = bravais.row(2);

        double volume = aVect.dot(bVect.cross(cVect));
        double prefactor = 2 * M_PI / volume;

        Vector3f aStar = prefactor * bVect.cross(cVect);
        Vector3f bStar = prefactor * cVect.cross(aVect);
        Vector3f cStar = prefactor * aVect.cross(bVect);

        reciprocal.row(0) = aStar;
        reciprocal.row(1) = bStar;
        reciprocal.row(2) = cStar;

        reciprocalInverse = reciprocal.inverse();
    };

    const Matrix3f &getBravais() const;
```

```
    Matrix3f getBravaisInverse() const;
    Matrix3f getReciprocal() const;
    Matrix3f getReciprocalInverse() const;

    Vector3f getQ(float h, float k, float l) const;
    Vector3f getQ(Vector3f hkls) const;
    Vector3f getHKL(float qx, float qy, float qz) const;
    Vector3f getHKL(Vector3f qvector) const;
    Vector3f getPosition(float u, float v, float w) const;
    Vector3f getPosition(Vector3f fractionalPosition) const;

    Vector3f getUVW(float rx, float ry, float rz) const;
    Vector3f getUVW(Vector3f position) const;

    bool operator==(const Bravais &rhs) const;

    bool operator!=(const Bravais &rhs) const;

    std::string writeReport() const;

private:
    Matrix3f bravais;
    Matrix3f reciprocal;
    Matrix3f bravaisInverse;
    Matrix3f reciprocalInverse;

};


#endif //EIGENTUT_BRAVAIS_H
```

C.22 Crystal.h

```
//
// Created by Tim on 28/09/2017.
//

#ifndef EIGENTUT_CRYSTAL_H
#define EIGENTUT_CRYSTAL_H

#include <list>
#include <Eigen/Dense>
#include <iomanip>
#include <vector>

#include "Bravais.h"
#include "Atom.h"
#include "FilterableAtoms.h"

class Crystal : public FilterableAtoms {
public:
    Crystal(const Bravais &bravaisLattice, const std::vector<Atom> &
    basis);

    std::string writeCellFile();

    const Bravais &getBravaisLattice() const;

    const std::vector<Atom> &getBasis() const;

    std::vector<Atom> getUnfilteredList();
```

```
28
29     Atom* getAtomByUID(std::string UID) const;
30
31     std::vector<Atom*> findNeighbours(const Atom& toAtom, int
    numberOfNeighbours, bool (*filter) (Atom));
32     std::vector<Atom*> findNeighbours(const Atom& toAtom, int
    numberOfNeighbours);
33
34     void updateAtom(const Atom& newAtom);
35
36     double distance(const Atom& atom1, const Atom& atom2);
37
38     void setSuperCellIndex(int supercellA, int supercellB, int
    supercellC);
39
40     bool operator==(const Crystal &rhs) const;
41
42     bool operator!=(const Crystal &rhs) const;
43
44 private:
45     Bravais bravaisLattice;
46     std::vector<Atom> basis;
47
48 };
49
50
51 #endif //EIGENTUT_CRYSTAL_H
```

C.23 FilterableAtoms.h

```
//
// Created by Tim on 14/03/2018.
//

#ifndef TBALLSNSPRINGS_FILTERABLEATOMS_H
#define TBALLSNSPRINGS_FILTERABLEATOMS_H


#include <vector>
#include "Atom.h"

class FilterableAtoms {
public:
    virtual std::vector<Atom> getUnfilteredList() = 0;

    virtual void updateAtom(const Atom &a) = 0;

    std::vector<Atom> filter(std::function<bool(Atom)> filterFunc) {
        return filter(filterFunc, getUnfilteredList());
    }

    static std::vector<Atom> filter(std::function<bool(Atom)> filterFunc
    , std::vector<Atom> listToFilter) {
        std::vector<Atom> filteredList;
        for (Atom a : listToFilter) {
            if (filterFunc(a)) {
                filteredList.push_back(a);
            }
        }
        return filteredList;
```

```
    }

};


#endif //TBALLSNSPRINGS_FILTERABLEATOMS_H
```

C.24 SuperCell.h

```
//
// Created by Tim on 08/10/2017.
//

#ifndef EIGENTUT_SUPERCELL_H
#define EIGENTUT_SUPERCELL_H


#include <map>
#include "Crystal.h"


class SuperCell : public FilterableAtoms {
// Given a Crystal, creates an nxmxl supercell of the crystal
public:
    SuperCell(Crystal baseCrystal, int size_a, int size_b, int size_c);

    const std::vector<std::vector<std::vector<Crystal *>>> &
   getSupercellCrystals() const;

    enum errors {SUPERCELL_TOO_SMALL, ATOMS_CLOSE_TO_EDGE, NUM_ERRORS};
```

```
21
22     std::vector<Atom> getAllAtomsInNearbyCells(Atom toAtom);
23     std::vector<Atom> getAllAtomsInNearbyCells(Atom toAtom, std::
    function<bool(Atom)> filterFunc);
24
25     std::vector<Atom> getNearestNeighbours(Atom toAtom, int
    numberOfNeighbours, std::function<bool(Atom)> filterFunc);
26     std::vector<Atom> getNearestNeighbours(Atom toAtom, int
    numberOfNeighbours);
27
28     std::string writeToFile();
29
30     std::string write_legacy();
31
32     std::vector<Atom> getAllAtoms();
33
34     void updateAtom(const Atom &newAtom);
35
36     std::vector<Atom> getUnfilteredList();
37
38     Vector3i getSupercellSize();
39
40     Vector3f distance(Atom &a1, Atom &a2);
41
42     const Bravais &getBravais();
43
44     int getTypeIdFromElement(std::string element);
45     std::string getElementFromTypeId(int elementId);
46
47     const std::map<std::string, int> &getTypeIdFromElementMap() const;
```

```
48
49     const std::map<int, std::string> &getElementFromTypeIdMap() const;
50
51 private:
52     int size_a, size_b, size_c;
53
54     std::map<std::string, int> _legacy_type;
55     std::map<int, std::string> _legacy_type_map;
56
57     std::vector<std::vector<std::vector<Crystal*>>> supercellCrystals;
58 };
59
60
61 #endif //EIGENTUT_SUPERCELL_H
```