Predicting Phonons and Their Effects in 2D Quantum Devices

Mphys. year 5 Research Project
Heriot-Watt University
Submission date: 05/05/2022
Student: Laura Rintoul(H00293952)
Supervisor: Dr E. Gauger

1 Heriot Watt University Physics Department – Own Work Declaration

I confirm that all this work is my own except where indicated, and that I have:

· Clearly referenced/listed all sources as appropriate
· Referenced and put in inverted commas all quoted text (from books, web, etc)
· Given the sources of all pictures, data etc. that are not my own
· Not made any use of the essay(s) of any other student(s) either past or present
· Not sought or used the help of any external professional agencies for the work
· Acknowledged in appropriate places any help that I have received from others

Name: Laura Rintoul
Student Number: H00293952
Course/Programme: B21AX, 5th Year Mathematical Physics
Title of work: Predicting phonons and their effects in 2D quantum devices
Date: 21/06/21

2 Summary of previous work

This project builds on work done in my Honour's Project, "Towards Predicting the Phonons of 2D Quantum Devices" (10.6). In this project, the backbone of this project, an MVFF simulation, was built in Python 3. This was checked to function for the 1-Dimensional case, and verified to give expected results. The simulation was found to need adjustment for making work in 2D, though this was attempted to be integrated during the previous project. This project worked towards understanding phonons using the Phonon Spectral Density rather than the phonon dispersion as was done in the previous project.

Contents

3 Abstract

Practical quantum technologies rely on robust hardware. For many applications, semiconductor quantum nanostructures are ideal candidates, e.g. as physical implementations of quantum bits, or as emitters of non-classical light. However, phonons, i.e. vibrations in the underlying crystal structure, severely change the behaviour and operation of semiconductor quantum devices in ways that - in many cases - we do not yet fully understand. A novel class of promising nanostructures is based on two-dimensional materials such as graphene (the exploration of which awarded the Nobel Prize in 2010) and MoS2 (molybdenum disulfide). There are hopes of engineering specialist and novel materials by layering these materials. Surprisingly, the effects of phonons on quantum devices are largely determined by a single function, the so-called phonon spectral density. While analytic approximations to the spectral density are known for many conventional physical structures, the same is not the case for 2D material due to a number of additional and unusual features in their electron and phonon band structures. The spectral densities of various 1D and 2D structures were calculated using a modified valence force field model, a computational method of resolving phonons, and a pathway was set up for understanding their effects. The spectral density method used was a summing of Gaussians to solve degeneracy. Methods for improving the effectiveness of the methods employed are discussed.

4 Introduction

Phonons are known to have a large impact on the behaviour of materials in many manners, through their coupling to particles in materials, such as excitons[1] and electrons. The latter is of particular interest for quantum devices. Quantum devices are already known to be able to utilise phonons in their effect, not simply being aware of their impact, such as a source of decoherence in qubits[2][3], but utilising them for specific device applications[4][5][6]. However, this project aims for a more general understanding of how the material structure of the entire quantum device impacts its function for electron oriented quantum device. In a quantum device, the structure can be generalised to being composed of a system that we're interested in the behaviour of, such as a qu-bit, and an environment this system is coupled to, the material it exists in. The coupling we are interested in particularly is that of the phonons in this environment. The way we approach understanding such a problem is through utilising open quantum systems dynamics[7]. We can assume that the system is much smaller than the environment and so it will be independent of the system it's coupled to. The open quantum systems method means that the entire environment doesn't need to be computed in the same manner as the system, as this would be an extremely computationally intense problem, increasing in size as the environment increases in size. Instead, we obtain a master equation for the system for a reduced density matrix[8]. Applying this to a 2 level system in a phonon bath as we might expect in a quantum device, we get a Hamiltonian as in equation 1:

$$H = H_S + H_E + H_I \tag{1}$$

$$H_S = \frac{\epsilon}{2}(|e\rangle\langle e| - |g\rangle\langle g|)$$

$$H_E = \sum_k \omega_k b_k^\dagger b_k$$

$$H_I = \sum_k (g_k \sigma_+ b_k + g_k^* \sigma_- b_k^\dagger)$$

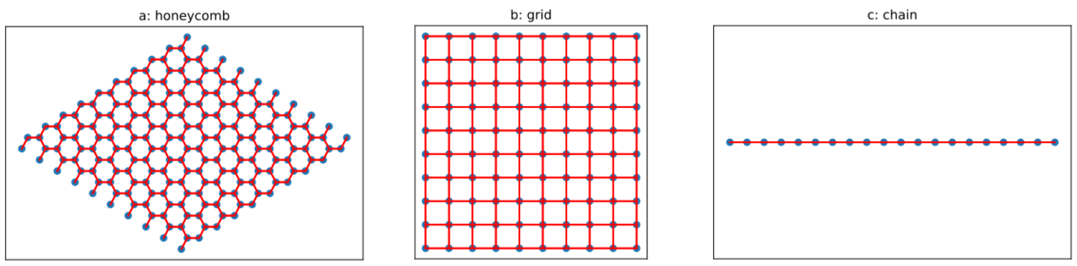
Through the the open quantum systems method, the master equation describes the impact of the phonons on a system as controlled by the phonon spectral density (equation 2).

$$J(w) = \sum_k |g_k|^2 \delta(\omega - \omega_k) = D(\omega)|g(\omega)|^2 \tag{2}$$

The time evolution and solution of a system under the effects of a spectral density is a known and very studied problem, with exactly known solutions [9]. In this project, iQUAPI and ACE were explored as computation methods, which are non-exact but much faster than the exact solutions[10]; though, this is non-exact approach is not the only way to analyse the effects of phonons. This is primarily utilised because of the high computational demands of the number of degrees of freedom employed by such a task. For smaller systems, it is reasonable to solve exactly [11]. This project took a microscopic approach to the systems studied, which is a less studied way of understanding phonons. Therefore, the task of understanding how phonons effect quantum devices can be reduced down to the computational calculation of the spectral density. While computationally demanding, this gives the opportunity for understanding much more complicated structures. In principle, if the system can be verified to yield useful results for simple structures, with sufficient computing power, the impact of phonons on arbitrary structures could be predicted; this is particularly helpful for quantum devices which may be less symmetrical than traditionally studied materials in structure. This is not to say that the effects of phonons are entirely understood, and there is still much incompleteness to the understanding they play, such as with pure dephasing[12].

This is of particular interest for 2D structures, as there is still a lot unknown about the actual phonon spectral densities and phonon behaviours of 2D materials, with much being discovered

Figure 1: Examples of the structures used in the project. a: A 2D honeycomb structure. This instance is 10*10 unit cells. b: a 2D grid structure. Again, this is 10*10. c: a 1D chain structure. This instance is a 20 atom chain.



and developed as time goes on[13]. In particular, there is already much known about phonon spectral densities within the context of 3D structures[14][15]; therefore the focus on the model's effectiveness within the context of 2D structures is more prudent, as well as less computationally intense. Models such as the one set out in this project can have their robustness tested against these and other methods[16][11].

4.1 Simulation of phonons in materials

Materials were simulated using a MVFF model[17] that had previously been implemented for 1 dimensional structures; in this project, this computer model was expanded to 3D. The model stores data about where lattice points are located, which are bonded and constants related to their bond strengths. The bonds are modelled as spring like with bond bending and bond stretching potentials. Three different types of materials were mainly worked with; a one dimensional chain of atoms, a 2D grid of atoms and a 2D honeycomb.

$$U = \frac{1}{2}\sum_{i\in N_A}[\sum_{j\in nn(i)}\frac{3}{8}\alpha_{ij}\frac{(r_{ij}^2 - d_{ij,0}^2)^2}{||d_i^2 j,0||} + \sum_{j,k\in nn(i)}^{k\neq j}\frac{3}{8}\beta_{ijk}\frac{(r_i j\cdot r_i k - d_i j,0\cdot d_i k,0)^2}{||d_{ij,0}||||d_{ik,0}||}] \tag{3}$$

The problem of solving this was tackled in my previous project work: the mathematics are attached in appendix 10.6. The numpy sparse package was used to extract the phonons as the eigenstates of the system. 3D systems are easier to compare against for spectral densities, with a wide range of well studied real life data, but due to the phonon extraction ultimately being an eigenvalue problem this does not scale well with the size of the system; the domain for getting out of finite range effects is not achievable with the hardware available for this project for a 3D system. Periodic boundary conditions were also implemented, but only used on the 1D chain, due to strange effects arising on the 2D grid and the significantly increased mathematical difficulty of implementing correct periodicity for the honeycomb lattice. The structures used are shown in figure 1.

5 Materials and methods

Generally, spectral densities are calculated directly from k-space rather than microscopically. However, the computational approach here is to analyse the lattice microscopically to observe the degree of coupling of an electron to a phonon; this is observed through the deformation of the lattice by the phonon, and its effect on the electron band energy. This relationship is defined in equation 4 [18].

$$a^{Gap} = dE^{Gap}/dln(V) = VdE^{Gap}/dV \tag{4}$$

Where E^{Gap} is the electron band gap and V is the volume. For a 3D cubic unit cell, $V = l^3 => dV = 3l^2 dl$:

$$a^{Gap} = l/3dE^{Gap}/dl => dE^{Gap} = 3/la^{Gap}dl$$

Therefore, the coupling can be taken as proportional to deformation by the lattice. Because we are interested in qualitative analysis, the rate of proportionality is not as important.

The deformation can be taken as the change in bond length between two atoms.

$$d - d_0 = |(r_1 + dr_1) - (r_2 + dr_2)|^2 - |r_1 - r_2|^2 = |(r_1 - r_2) + (dr_1 - dr_2)|^2 - |r_1 - r_2|^2 \quad (5)$$

Where d is the bond length with perturbation, d_0 is the perturbed bond lengh, r is the atom location, and dr is the perturbation. Linearising gives:

$$dl = 2(r_1 - r_2)\dot{(}dr_1 - dr_2) + |dr_1 - dr_2|^2 \quad 2(r1 - r2) \cdot (dr1 - dr2) \quad (6)$$

We are interested in the total deformation over the space the electron occupies; weighted with the electron wavefunction, and summed over for all bonds. The electron is assumed to be gaussian in form. Therefore, we calculate $g(k)$ as:

$$g(k) = \sum_i sum_{j=nn(i)} 2(r_i - rj) \cdot (dr_i - dr_j)\phi(r_i) \quad (7)$$

Where k is some phonon, r_i is the equilibrium position of atom i, dr_i is the phonon displacement of atom i and $\phi(r_i)$ is the electron density over atom r_i. Due to degeneracy of phonons, in order to successfully sum in a meaningful way for a spectral density, the effects of these phonons had to be superimposed computationally. This meant that a modified definition of the spectral density was used, as in the end of equation 8. Sigma was generally adjusted manually in order to ensure continuity while preventing overlap as much as possible, to reduce muddying of results.

$$J(\omega) \equiv |g_k|^2 \delta(\omega - \omega_k) = D(\omega)|g(\omega)|^2 = \sum_k |g(\omega_k)|^2 e^{-\frac{1}{2}\frac{(\omega - \omega_k)^2}{\sigma}} \quad (8)$$

To summarise, the effective computational method was looping over each phonon and calculating the overlap with the electron wavefunction at each lattice point, by adding a gaussian of height related to the energy and width a parameter controlled for resolution.

5.1 1D Material spectral density analytical modelling

As well as being implemented computationally, the above microscopic approach was used to return an analytical model, so that results could be fitted to give a metric of accuracy of the theoretical results.

5.1.1 electron phonon interaction, g(k)

We define $g(k)$ as the deformation of an electron band gap by a phonon of wavevector k in a lattice.

$$g(k) = \int \psi(x) \cdot dv_k(x) dx$$

Where dv is proportional to the change in bond length, which is proportional to deformation of the lattice, and $\psi(x)$ is the wavevector of the electron. In the discrete numerical model, this is taken as a sum over all lattice points' individual contribution, but here this is taken as an integral over all space instead. The electron is assumed to be of a gaussian centred at x = 0; the periodic boundary conditions employed mean that this location isn't particularly important, and in the numerical simulation the electron is also placed atop a lattice point.

$$\psi(x) = Ae^{-1/2(\frac{x}{r})^2}$$

5.1.2 transfer to continuous form for dv_k

Previously, in the discrete model, we defined dv_k for a given point as:

$$dv_k(i) = \sum_{nn(i)} (\underline{r}(i) - \underline{r}(nn(i))) \cdot (\underline{v}_k(i) - \underline{v}_k(nn(i)))$$

Here, $v(i)$ is the eigenvector oscillation value at the point i and r is its location. In the 1D example, this is summing over the two nearest neighbours, with the same bond length b on each side (with a sign change). Here, we also drop the notation of underlines as we are collapsing the general definition of dv_k down to the 1D case.

$$dv_k(i) = bv_k(i), v_k(i+1)) - b(v_k(i), v_k(i-1)) = b(v_k(i+1) - v_k(i-1))$$

Therefore, for the continuous case, $dv_k(x)$ is taken to be:

$$dv_k(x) = v_k(x+b) - v_k(x-b)$$

5.1.3 1D Chain phonons

For a 1D periodic chain's phonons and dispersion, the frequency w and the wavefunction v_k are known to be[19]:

$$w(k) = \frac{1}{\sqrt{2m}} sin(\frac{ka}{2})$$

$$v_k(x) = Ae^{i(kx-wt)}$$

Where m is the mass of the atoms in the chain, a is the lattice spacing and k is the wave-vector of the phonon. A is some constant to be normalised over the distribution of the phonon.

5.1.4 substitution into $g(k)$

Because the spring constants as defined in the analytical vs computational model aren't the same, the solution derived here will be proportional rather than exact; therefore, for the sake of brevity, constants will be collected as A.

$$g(k) = \int \psi(x)(v_k(x+b) - v_k(x-b))dx$$

$$= \int Ae^{-1/2(\frac{x}{r})^2}(e^{i(k(x+b)-wt)} - e^{i(k(x-b)-wt)})dx$$

$$= Ce^{-iwt} sin(kb) e^{-\frac{1}{2}k^2r^2}$$

$$\therefore |g(k)|^2 \propto \sin(kb)^2 e^{-k^2r^2}$$

The sinusoidal part is surprising, but potentially explains the oscillations towards the tail of the 2D spectral density (section 5.3).

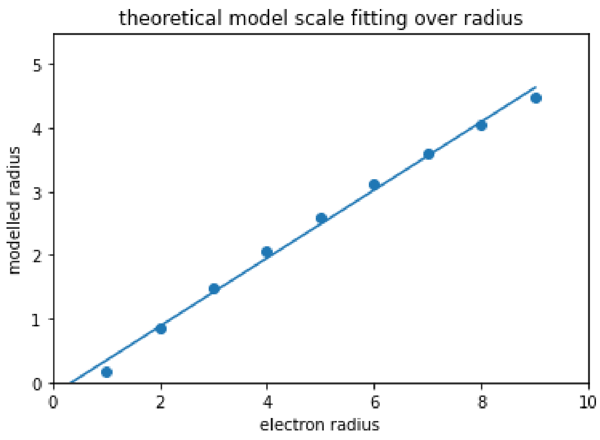


Figure 2: Fitting of analytical fit radius against the radius used to calculate computationally.

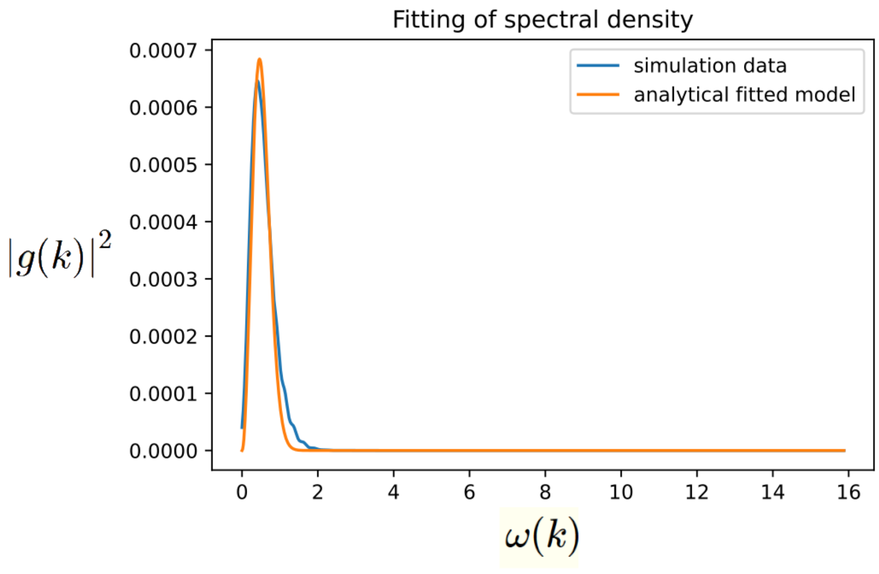


Figure 3: An example of a fit of the analytical model to the computational results for the 1D chain.

5.2 1D material spectral density calculation

In the 1D chain case, bond stretching and stetching constants of $\alpha = 1$ and $\beta = 1$ were used respectively. Periodic boundary conditions were found to give results closer to the analytical theory. As can be seen in figure 3, the fit is not precise, especially close to zero; this can be seen as due to the requirement for summing using Gaussian's leading to leaking over, with increased impact on areas with high rates of change. However, despite this, there was still successful fitting found between the analytically derived model and the computational phonon spectral density calculation, as can be seen in figure 2.

5.3 2D Material spectral density calculation

For the Grid case, there was a generally similar shape found about zero; however, there was an unusual decaying coupling to higher energies as well. This is a surprising result given the expectation for coupling mainly to low energies. Whether this is a problem with the simulation or a property of the structure requires further investigation. Either way, at the size that was

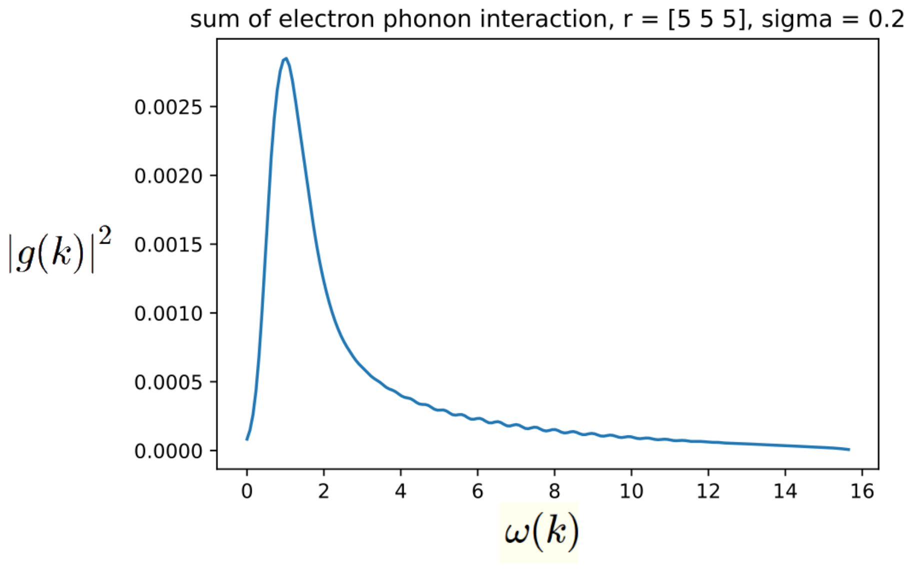


Figure 4: Example of results for the grid case

simulated up to (50 by 50), this behaviour was convergent. The oscillatory tail end of the graph could be derived from the oscillations as observed in the analytical solution to the 1D case being washed out at higher energies, or could simply be the peaks of the Gaussians being summed over. This problem of increased smoothness versus washing out of effects is one of the main issues faced by this method of using Gaussians to sum over the phonons.

For the honeycomb case, there were a wide range of results obtained for spectral density behaviour; the largest case that could be calculated was the 40 by 40 unit cells case, which is shown in figure 5. This complexity may be a problem with theory or computational methods, or it may be due to the increased complexity of the phonon bands giving rise to genuinely strange phonon spectral density effects, as can be seen in figure 6[20]. In terms of understanding the reasons for the issues with computational methods, it is known that the size of the sample being simulated has a very large effect on the phonon spectral density[21], so it is very possible that this is simply a problem of insufficient resolution that would converge as the grid structure did at sufficient size.

6 Discussion

While the computational power was insufficient to yield consistent results for honeycomb structures, the results for grid structures and 1 dimensional chains were similar and consistent with analytical modelling. Due to the high dependency of the coupling to the way that the spectral density is shaped about zero, the results were insufficiently large in order to retain the quality required to understand the effect that these structures would actually have on 2 level systems. However, with larger structures being simulated and higher resolution, phonon spectral densities could be used to predict the behaviour of quantum devices. Increased resolution could also determine whether error between the analytical model and the computational model were due to the method of solving for degeneracy, for which σ could be decreased in high resolution, or a discrepancy between the analytical and computational model. However, the issue of effectiveness for use in simulating the impact on quantum devices can be refined further. The primary coupling to the phonon spectral density of an electron takes place in the low energy region[22], so increasing resolution in the region about zero is the most important. Therefore, the method of solving the issue of degeneracy may need further refinement. About zero is the steepest part

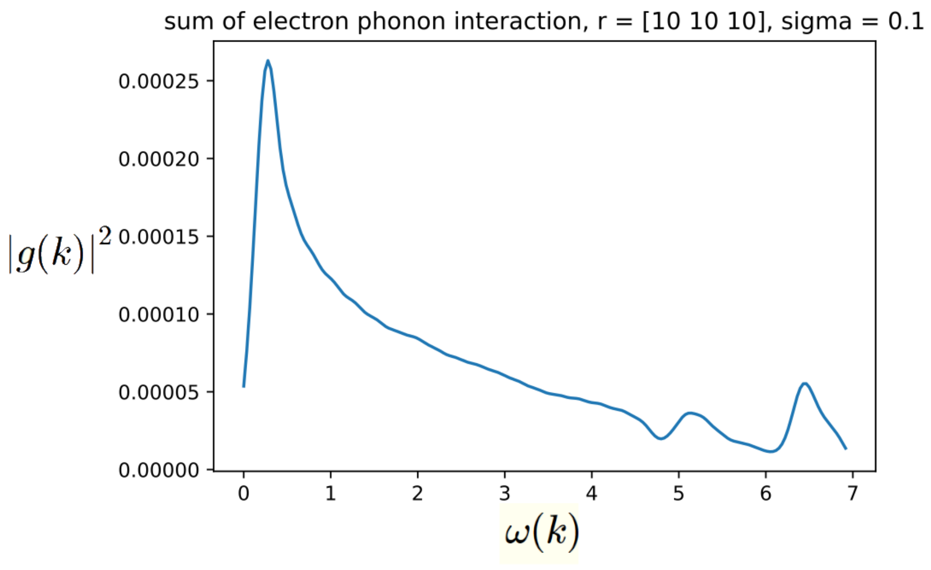


Figure 5: An example of a fit of the analytical model to the computational results for the 1D chain.

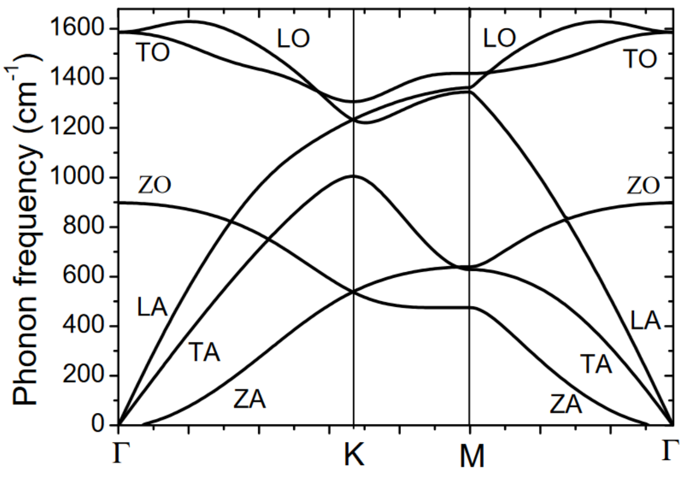


Figure 6: An example of a fit of the analytical model to the computational results for the 1D chain.

of the plots, so therefore the most impacted by washout by neighbouring gaussians in the addition method used. Therefore, the effectiveness of alternative methods of solving the degeneracy issue should be investigated, such as binning or a check for if an energy is sufficiently close to another giving adding of energy. For further testing of the robustness of the modelling, different constants could be experimented with to determine consistency. Also, modelling of analytical solutions in 2D, while more complicated, could further validate consistency between analytical and computational modelling. With even further increased computational power and optimisation to code, there is also potential for obtaining spectral densities for 3D structures; however, the speed of increase of degrees of freedom and how slowly 2D structures tended towards consistent results may mean that this model is limited in what kind of 3D structures could be usefully analysed. If results were obtained, then there would be possibility for comparison with known 3D phonon spectral densities due to the much greater experimental knowledge in existence on the topic. With the existence of other results for 2D dispersions as well, while it may be difficult to verify phonon spectral densities, there is room for confirmation of the fundamental behaviour of the mechanics employed[23].

7 Conclusion

In conclusion, an MVFF method was used to predict the phonons of 1D and 2D structures. The microscopic model was successful in producing consistent results for spectral densities in 1 dimensional and some 2 dimensional structures. This was internally verified as working computationally for the 1D case, and needs further work with regards to the 2D case. This project has built a general model for computation of these phonon spectral densities, which with sufficient computational strength and optimisation, could lead to a more coherent understanding of the effects and behaviour of phonons in 2D quantum devices.

8 References

References

1. Zhang, J., Zhu, C. & Liang, W. Benchmarking calculations of spectral densities for the diagonal and nondiagonal exciton-phonon coupling of tetracene crystal. *Chemical Physics Letters* **681,** 7–15. ISSN: 0009-2614. `https://www.sciencedirect.com/science/article/pii/S0009261417304591` (2017).
2. Chen, S.-H. The decoherence time of anisotropic quantum dot qubit: Effects of electric field, electron–phonon interactions, and impurities. *Superlattices and Microstructures* **154,** 106884. ISSN: 0749-6036. `https://www.sciencedirect.com/science/article/pii/S0749603621000823` (2021).
3. Makri, N. Exploiting classical decoherence in dissipative quantum dynamics: Memory, phonon emission, and the blip sum. *Chemical Physics Letters* **593,** 93–103. ISSN: 0009-2614. `https://www.sciencedirect.com/science/article/pii/S0009261413014802` (2014).
4. Afsaneh, E. & Harouni, M. B. Generation of entanglement between quantum dot molecule with the presence of phonon effects in a voltage-controlled junction. *Physics Letters A* **409,** 127525. ISSN: 0375-9601. `https://www.sciencedirect.com/science/article/pii/S0375960121003893` (2021).
5. Cecoi, E., Ciornea, V., Isar, A. & Macovei, M. A. Entanglement of a laser-driven pair of two-level qubits via its phonon environment. *Journal of the Optical Society of America. B, Optical physics* **35,** 1127. ISSN: 0740-3224 (2018).

6. Sadeghi, H. Theory of electron, phonon and spin transport in nanoscale quantum devices. *Nanotechnology* **29,** 373001. ISSN: 0957-4484 1361-6528. `http://dx.doi.org/10.1088/1361-6528/aace21` (2018).
7. *Open Quantum Systems II* 244. ISBN: 978-3-540-30992-5 (Springer, Berlin, Heidelberg, 2006).
8. Nazir, A. & McCutcheon, D. P. S. Modelling exciton-phonon interactions in optically driven quantum dots. *Journal of physics. Condensed matter* **28.** JPCM-105992.R1, 103002–103002. ISSN: 0953-8984. `https://spiral.imperial.ac.uk:8443/bitstream/10044/1/27863/2/1511.01405v1.pdf` (2016).
9. Chenu, A., Shiau, S.-Y. & Combescot, M. Two-level System coupled to Phonons: Full Analytical Solution. *Physical review. B, Condensed matter and materials physics* **99,** 366–382. ISSN: 1098-0121 (2018).
10. Cygorek, M. *et al.* Simulation of open quantum systems by automated compression of arbitrary environments. *Nature Physics.* ISSN: 1745-2481. `https://doi.org/10.1038/s41567-022-01544-9` (2022).
11. Stroscio, M., Dutta, M., Kahn, D., Kim, K. W. & Komirenko, S. Phonons in nanostructures: device applications. *Physica B: Condensed Matter* **316-317,** 8–11. ISSN: 0921-4526. `https://www.sciencedirect.com/science/article/pii/S0921452602004180` (2002).
12. Echeverri-Arteaga, S., Vinck-Posada, H., Villas-Bôas, J. M. & Gómez, E. A. Pure dephasing vs. Phonon mediated off-resonant coupling in a quantum-dot-cavity system. *Optics communications* **460,** 125115. ISSN: 0030-4018 (2020).
13. Novoselov, K. S., Mishchenko, A., Carvalho, A. & Castro Neto, A. H. 2D materials and van der Waals heterostructures. *Science* **353.** 1095-9203 Novoselov, K S Mishchenko, A Carvalho, A Castro Neto, A H Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. Review United States 2016/07/30 Science. 2016 Jul 29;353(6298):aac9439. doi: 10.1126/science.aac9439., aac9439. ISSN: 0036-8075 (2016).
14. Papanicolaou, N. I., Lagaris, I. E. & Evangelakis, G. A. Modification of phonon spectral densities of the (001) copper surface due to copper adatoms by molecular dynamics simulation. *Surface Science* **337,** L819–L824. ISSN: 0039-6028. `https://www.sciencedirect.com/science/article/pii/0039602895006389` (1995).
15. Hofmann, A. *et al.* Phonon spectral density in a GaAs/AlGaAs double quantum dot. *Physical Review Research* **2.** `https://doi.org/10.1103%2Fphysrevresearch.2.033230` (Aug. 2020).
16. J. Marquina C. Power, J. G. & Broto, J. Ab Initio Study of the Electronic and Vibrational Properties of 1-nm-Diameter Single-Walled Nanotubes. *Advances in Materials Physics and Chemistry* **Vol. 3,** 178–184 (2013).
17. Paul, A., Luisier, M. & Klimeck, G. Modified valence force field approach for phonon dispersion: from zinc-blende bulk to nanowires. *Journal of Computational Electronics* **9,** 160–172. ISSN: 1572-8137. `https://doi.org/10.1007/s10825-010-0332-9` (2010).
18. Bardeen, J. & Shockley, W. Deformation Potentials and Mobilities in Non-Polar Crystals. *Physical Review* **80.** PR, 72–80. `https://link.aps.org/doi/10.1103/PhysRev.80.72` (1950).
19. Avila, A. & Reyes Romero, D. Phonon multiplexing through 1D chains. *Brazilian Journal of Physics - BRAZ J PHYS* **38,** 604–609 (Dec. 2008).
20. Yan, J.-A., Ruan, W. Y. & Chou, M. Y. Phonon dispersions and vibrational properties of monolayer, bilayer, and trilayer graphene: Density-functional perturbation theory. *Physical Review B* **77.** PRB, 125401. `https://link.aps.org/doi/10.1103/PhysRevB.77.125401` (2008).

21. Han, J., Xu, B., Hu, M., Lin, Y. H. & Liu, W. Analytical study on the size effect of phonon spectral energy density resolution. *Computational Materials Science* **132,** 6–9. ISSN: 0927-0256. `https://www.sciencedirect.com/science/article/pii/S0927025617300745` (2017).

22. Hossein-Nejad, H. & Scholes, G. D. Energy transfer, entanglement and decoherence in a molecular dimer interacting with a phonon bath. *New Journal of Physics* **12,** 065045. `https://doi.org/10.1088/1367-2630/12/6/065045` (June 2010).

23. Castro Neto, A. H., Guinea, F., Peres, N. M. R., Novoselov, K. S. & Geim, A. K. The electronic properties of graphene. *Reviews of modern physics* **81,** 109–162. ISSN: 0034-6861 (2009).

9 Acknowledgements

I'd like to thank Dr Erik Gauger and Dr Moritz Cygorek for their help with this research project. I'd also like to thank Dylan Ralph for his support through this project.

10 Appendix

10.1 Risk Assessment

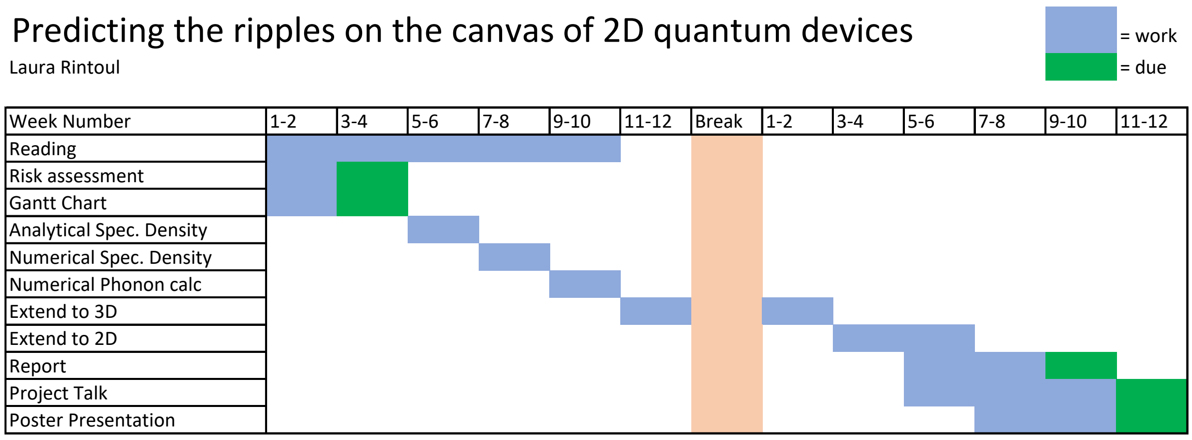
Risk Assessment

Activity title :	**Master's Project**
Location :	**My House**
Assessor :	**Laura Rintoul**
Date Assessed :	**20/9/21**

Risk Analysis Matrix Level of Risk						Likelihood	Severity
Likelihood	4	4	8	12	16	1. Unlikely	1. Insignificant/No Injury
	3	3	6	9	12	2. Possible	2. Minor Injury
	2	2	4	6	8	3. Likely	3. Moderate Injury
	1	1	2	3	4	4. Certain	4. Major Injury/Fatality
×		1	2	3	4	*Score likelihood*	*Score severity*
		Severity					

Hazard	Persons Affected	**Likelihood** of incident	Potential **Severity**	**Risk control already in place**	**RISK ASSESSMENT**	**Further Action required to control risk**
RSI	Me	1	3		3	Take regular breaks from coding and typing
Electrocution	Me	1	3	Not using faulty equipment	3	Check all equipment and cables for obvious faults: keep liquids away from equipment
Tripping	Me	1	3	Keep Workspace and cables tidy	3	
Back and neck strain	Me	2	2		4	Take regular breaks: pay attention to posture
Covid	Everyone	1	2	Vaccination	2	Social distancing, masks

10.2 Gantt Chart



Predicting the ripples on the canvas of 2D quantum devices

Laura Rintoul

🟦 = work

🟩 = due

Week Number	1-2	3-4	5-6	7-8	9-10	11-12	Break	1-2	3-4	5-6	7-8	9-10	11-12
Reading	🟦	🟦	🟦	🟦	🟦		🟧						
Risk assessment	🟦	🟩					🟧						
Gantt Chart	🟦	🟩					🟧						
Analytical Spec. Density			🟦				🟧						
Numerical Spec. Density				🟦			🟧						
Numerical Phonon calc					🟦		🟧						
Extend to 3D						🟦	🟧	🟦					
Extend to 2D							🟧		🟦	🟦			
Report							🟧			🟦	🟦	🟩	
Project Talk							🟧			🟦	🟦	🟦	🟩
Poster Presentation							🟧				🟦	🟦	🟩

10.3 Additional Phonon Spectral Density derivation

$$g(k) = \int \psi(x)(v_k(x+b) - v_k(x-b))dx$$

$$= \int Ae^{-1/2(\frac{x}{r})^2}(e^{i(k(x+b)-wt)} - e^{i(k(x-b)-wt)})dx$$

$$= Ae^{-iwt}(e^{ikb} - e^{-ikb}) \int e^{-1/2(\frac{x}{r})^2}(e^{ikx})dx$$

$$= Ae^{-iwt}(e^{ikb} - e^{-ikb}) \int e^{-1/2(\frac{x}{r})^2}(e^{ikx})dx$$

$$= Ae^{-iwt}(e^{ikb} - e^{-ikb}) \int e^{-1/2(\frac{x}{r})^2}(e^{ikx})dx$$

$$= Ae^{-iwt}sin(kb) \int e^{-1/2(\frac{x}{r})^2+ikx}dx$$

Complete the square for integral exponent.

$$-1/4(\frac{x}{r})^2 + ikx = -\frac{1}{2r^2}(x^2 - 2ikr^2x)$$

$$= -\frac{1}{2r^2}((x - ikr^2)^2 - (ikr^2)^2)$$

$$= -\frac{1}{2r^2}((x - 2ikr^2)^2 - (ikr^2)^2)$$

$$= -\frac{1}{2r^2}(x - 2ikr^2)^2) - \frac{1}{2}k^2r^2$$

$$g(k) = Be^{-iwt}sin(kb)e^{-\frac{1}{2}k^2r^2} * \int e^{-\frac{1}{2r^2}(x-2ikr^2)^2)}dx$$

$$= Ce^{-iwt}sin(kb)e^{-\frac{1}{2}k^2r^2}$$

$$\therefore |g(k)|^2 \propto \sin(kb)^2 e^{-k^2r^2}$$

10.4 Dynamical Matrix Calculations

10.4.1 Expanding potentials for given bond and coordinates

$U_{bb}^{ij} = U_{bb}^{ji}, U_{bs}^{ijk} = U_{bs}^{kji}$

$$U = \tfrac{1}{2} \sum_{i\in N_A} [\sum_{j\in nn(i)} U_{bs}^{ij} + \sum_{j,k\in nn(i)}^{k\neq j} U_{bb}^{jik}]$$

$$\tfrac{\partial U}{\partial r_m^i} = \tfrac{\partial}{\partial r_m^i} \tfrac{1}{2} \sum_{i\in N_A} [\sum_{j\in nn(i)} U_{bs}^{ij} + \sum_{j,k\in nn(i)}^{k\neq j} U_{bb}^{jik}]$$

$$= \tfrac{\partial}{\partial r_m^i} \tfrac{1}{2} [\sum_{j\in nn(i)} U_{bs}^{ij} + \sum_{j,k\in nn(i)}^{k\neq j} U_{bb}^{jik} + \sum_{j\in N_A}^{j\neq i} (\sum_{i\in nn(j)} U_{bs}^{ji} + \sum_{i,k\in nn(j)}^{k\neq i} U_{bb}^{ijk})]$$

$$= \tfrac{\partial}{\partial r_m^i} \tfrac{1}{2} [2 \sum_{j\in nn(i)} U_{bs}^{ij} + \sum_{j,k\in nn(i)}^{k\neq j} U_{bb}^{jik} + \sum_{j\in N_A}^{j\neq i} (\sum_{k\in nn(j)}^{k\neq i} U_{bb}^{ijk} + U_{bb}^{kji})]$$

$$\tfrac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \tfrac{\partial}{\partial r_m^i \partial r_n^j} \tfrac{1}{2} [2 \sum_{j\in nn(i)} U_{bs}^{ij} + \sum_{j,k\in nn(i)}^{k\neq j} U_{bb}^{jik} + \sum_{j\in nn(i)} (\sum_{k\in nn(j)}^{k\neq i} 2U_{bb}^{ijk})]$$

if $j \in nn(i)$,

$$\tfrac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \tfrac{\partial^2}{\partial r_m^i \partial r_n^j} \tfrac{1}{2} [2U_{bs}^{ij} + \sum_{k\in nn(i)}^{k\neq j} (U_{bb}^{jik} + U_{bb}^{kij}) + (\sum_{k\in nn(j)}^{k\neq i} 2U_{bb}^{ijk}) + \sum_{k\in nn(i),nn(j)} 2U_{bb}^{ikj}]$$

$$= \tfrac{\partial^2}{\partial r_m^i \partial r_n^j} [U_{bs}^{ij} + \sum_{k\in nn(i)}^{k\neq j} U_{bb}^{jik} + \sum_{k\in nn(j)}^{k\neq i} U_{bb}^{ijk} + \sum_{k\in nn(i),nn(j)} U_{bb}^{ikj}]$$

if $j \notin nn(i) \cup i$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \frac{\partial^2}{\partial r_m^i \partial r_n^j} \frac{1}{2} [\sum_{k\in nn(i),nn(j)} 2U_{bb}^{ikj}] = \frac{\partial^2}{\partial r_m^i \partial r_n^j} \sum_{k\in nn(i),nn(j)} U_{bb}^{ikj}$$

$$\tfrac{\partial^2 U}{\partial r_m^i \partial r_n^i} = \tfrac{\partial}{\partial r_m^i \partial r_n^i} \tfrac{1}{2} [2 \sum_{j\in nn(i)} U_{bs}^{ij} + \sum_{j,k\in nn(i)}^{k\neq j} U_{bb}^{jik} + \sum_{j\in nn(i)} (\sum_{k\in nn(j)}^{k\neq i} 2U_{bb}^{ijk})]$$

10.4.2 derivatives of individual bond potentials

$$U_{bs}^{ij} = \tfrac{3}{8}\alpha_{ij}\tfrac{(r_{ij}^2-d_{ij,0}^2)^2}{||d_{ij,0}^2||}$$

$$r_{ij}^2 = \sum_m (r_m^j - r_m^i)^2$$

$$\tfrac{\partial U_{bs}^{ij}}{\partial r_m^i} = \tfrac{3}{8}\alpha_{ij}\tfrac{2(r_{ij}^2-d_{ij,0}^2)(-2(r_m^j-r_m^i))}{||d_{ij,0}^2||}$$

for $i \neq j$:

$$\tfrac{\partial^2 U_{bs}^{ij}}{\partial r_m^i \partial r_m^j} = -\tfrac{3}{2}\alpha_{ij}\tfrac{(r_{ij}^2-d_{ij,0}^2)+2(r_m^j-r_m^i)(r_m^j-r_m^i)}{||d_{ij,0}^2||} = -\tfrac{3}{2}\alpha_{ij}\tfrac{(r_{ij}^2-d_{ij,0}^2)+2(r_m^j-r_m^i)^2}{||d_{ij,0}^2||}$$

$$\tfrac{\partial^2 U_{bs}^{ij}}{\partial r_m^i \partial r_n^j} = -\tfrac{3}{2}\alpha_{ij}\tfrac{2(r_n^j-r_n^i)(r_m^j-r_m^i)}{||d_{ij,0}^2||} = -3\alpha_{ij}\tfrac{(r_n^j-r_n^i)(r_m^j-r_m^i)}{||d_{ij,0}^2||} \text{ for } m \neq n$$

$$\tfrac{\partial^2 U_{bs}^{ij}}{(\partial r_m^i)^2} = \tfrac{3}{2}\alpha_{ij}\tfrac{(r_{ij}^2-d_{ij,0}^2)+2(r_m^j-r_m^i)^2}{||d_{ij,0}^2||}$$

$$\tfrac{\partial^2 U_{bs}^{ij}}{\partial r_m^i \partial r_n^i} = 3\alpha_{ij}\tfrac{(r_n^j-r_n^i)(r_m^j-r_m^i)}{||d_{ij,0}^2||} \text{ for } m \neq n$$

$$U_{bb}^{jik} = \tfrac{3}{8}\beta_{jik}\tfrac{(\Delta\theta_{jik})^2}{||d_{ij,0}||||d_{ik,0}||} = \tfrac{3}{8}\beta_{jik}\tfrac{(r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0})^2}{||d_{ij,0}||||d_{ik,0}||}$$

$$r_{ij}\cdot r_{ik} = \sum_m (r_m^j - r_m^i)(r_m^k - r_m^i)$$

$$\tfrac{\partial U_{bb}^{jik}}{\partial r_n^j} = \tfrac{3}{8}\beta_{jik}\tfrac{2(r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0})(r_n^k-r_n^i)}{||d_{ij,0}||||d_{ik,0}||} = \tfrac{3}{4}\beta_{jik}\tfrac{(r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0})(r_n^k-r_n^i)}{||d_{ij,0}||||d_{ik,0}||}$$

$$\tfrac{\partial^2 U_{bb}^{jik}}{\partial r_m^j \partial r_m^i} = \tfrac{3}{4}\beta_{jik}\tfrac{d_{ij,0}\cdot d_{ik,0}-r_{ij}\cdot r_{ik}+(r_m^k-r_m^i)(2r_m^i-r_m^j-r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$

$$\tfrac{\partial^2 U_{bb}^{jik}}{\partial r_m^i \partial r_n^j} = \tfrac{3}{4}\beta_{jik}\tfrac{(r_n^k-r_n^i)(2r_m^i-r_m^j-r_m^k)}{||d_{ij,0}||||d_{ik,0}||} \text{ for } m \neq n$$

$$\tfrac{\partial^2 U_{bb}^{jik}}{\partial r_m^j \partial r_m^k} = \tfrac{3}{4}\beta_{jik}\tfrac{r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0}+(r_m^k-r_m^i)(r_m^j-r_m^i)}{||d_{ij,0}||||d_{ik,0}||}$$

$$\tfrac{\partial^2 U_{bb}^{jik}}{\partial r_m^j \partial r_n^k} = \tfrac{3}{4}\beta_{jik}\tfrac{(r_m^k-r_m^i)(r_n^j-r_n^i)}{||d_{ij,0}||||d_{ik,0}||} \text{ for } m \neq n$$

$$\tfrac{\partial^2 U_{bb}^{jik}}{(\partial r_m^j)^2} = \tfrac{3}{4}\beta_{jik}\tfrac{r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0}+(r_m^k-r_m^i)^2}{||d_{ij,0}||||d_{ik,0}||}$$

$$\tfrac{\partial^2 U_{bb}^{jik}}{\partial r_m^j \partial r_n^j} = \tfrac{3}{4}\beta_{jik}\tfrac{(r_m^k-r_m^i)(r_n^k-r_n^i)}{||d_{ij,0}||||d_{ik,0}||} \text{ for } m \neq n$$

$$\tfrac{\partial U_{bb}^{jik}}{\partial r_m^i} = \tfrac{3}{4}\beta_{jik}\tfrac{(r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0})(2r_m^i-r_m^j-r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$

$$\tfrac{\partial^2 U_{bb}^{jik}}{(\partial r_m^i)^2} = \tfrac{3}{4}\beta_{jik}\tfrac{2((r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0})+(2r_m^i-r_m^j-r_m^k)^2}{||d_{ij,0}||||d_{ik,0}||}$$

$$\tfrac{\partial^2 U_{bb}^{jik}}{\partial r_m^i \partial r_n^i} = \tfrac{3}{4}\beta_{jik}\tfrac{(2r_n^i-r_n^j-r_n^k)(2r_m^i-r_m^j-r_m^k)}{||d_{ij,0}||||d_{ik,0}||} \text{ for } m \neq n$$

10.4.3 substituting evaluated derivatives

if $i \in nn(j)$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \frac{\partial^2}{\partial r_m^i \partial r_n^j}[U_{bs}^{ij} + \sum_{k \in nn(i)}^{k \neq j} U_{bb}^{jik} + \sum_{k \in nn(j)}^{k \neq i} U_{bb}^{ijk} + \sum_{k \in nn(i),nn(j)}^{k \neq i,j} U_{bb}^{ikj}]$$

$$\frac{\partial^2 U}{\partial r_m^i \partial r_m^j} = -\frac{3}{2}\alpha_{ij}\frac{(r_{ij}^2 - d_{ij,0}^2) + 2(r_m^j - r_m^i)(r_m^j - r_m^i)}{||d_{ij,0}^2||}$$

$$+ \sum_{k \in nn(i)}^{k \neq j} \frac{3}{4}\beta_{jik}\frac{d_{ij,0} \cdot d_{ik,0} - r_{ij} \cdot r_{ik} + (r_m^k - r_m^i)(2r_m^i - r_m^j - r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$

$$+ \sum_{k \in nn(j)}^{k \neq i} \frac{3}{4}\beta_{ijk}\frac{d_{ji,0} \cdot d_{jk,0} - r_{ji} \cdot r_{jk} + (r_m^k - r_m^j)(2r_m^j - r_m^i - r_m^k)}{||d_{ij,0}||||d_{jk,0}||}$$

$$+ \sum_{k \in nn(i),nn(j)}^{k \neq i,j} \frac{3}{4}\beta_{ikj}\frac{r_{ki} \cdot r_{kj} - d_{ki,0} \cdot d_{kj,0} + (r_m^j - r_m^k)(r_m^i - r_m^k)}{||d_{ki,0}||||d_{kj,0}||}$$

for $m \neq n$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = -3\alpha_{ij}\frac{(r_n^j - r_n^i)(r_m^j - r_m^i)}{||d_{ij,0}^2||}$$

$$+ \sum_{k \in nn(i)}^{k \neq j} \frac{3}{4}\beta_{jik}\frac{(r_n^k - r_n^i)(2r_m^i - r_m^j - r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$

$$+ \sum_{k \in nn(j)}^{k \neq i} \frac{3}{4}\beta_{ijk}\frac{(r_m^k - r_m^j)(2r_n^j - r_n^i - r_n^k)}{||d_{ij,0}||||d_{jk,0}||}$$

$$+ \sum_{k \in nn(i),nn(j)}^{k \neq i,j} \frac{3}{4}\beta_{ikj}\frac{(r_m^j - r_m^k)(r_n^i - r_n^k)}{||d_{kj,0}||||d_{ki,0}||}$$

if $j \notin nn(i) \cup i$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \frac{\partial^2}{\partial r_m^i \partial r_n^j} \sum_{k \in nn(i),nn(j)} U_{bb}^{ikj}$$

$$\frac{\partial^2 U}{\partial r_m^i \partial r_m^j} = \sum_{k \in nn(i),nn(j)}^{k \neq i,j} \frac{3}{4}\beta_{ikj}\frac{r_{ki} \cdot r_{kj} - d_{ki,0} \cdot d_{kj,0} + (r_m^j - r_m^k)(r_m^i - r_m^k)}{||d_{ki,0}||||d_{kj,0}||}$$

for $m \neq n$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \sum_{k \in nn(i),nn(j)}^{k \neq i,j} \frac{3}{4}\beta_{ikj}\frac{(r_m^j - r_m^k)(r_n^i - r_n^k)}{||d_{kj,0}||||d_{ki,0}||}$$

$$\frac{\partial^2 U}{(\partial r^i_m)^2} = \frac{\partial^2}{(\partial r^i_m)^2}[\sum_{j\in nn(i)} U^{ij}_{bs} + \frac{1}{2}\sum_{j,k\in nn(i)}^{k\neq j} U^{jik}_{bb} + \sum_{j\in nn(i)}(\sum_{k\in nn(j)}^{k\neq i} U^{ijk}_{bb})]$$

$$= \sum_{j\in nn(i)} \frac{3}{2}\alpha_{ij}\frac{(r^2_{ij} - d^2_{ij,0}) + 2(r^j_m - r^i_m)^2}{||d^2_{ij,0}||}$$

$$+ \sum_{j,k\in nn(i)}^{k\neq j} \frac{3}{8}\beta_{jik}\frac{2((r_{ij}\cdot r_{ik} - d_{ij,0}\cdot d_{ik,0}) + (2r^i_m - r^j_m - r^k_m)^2}{||d_{ij,0}||||d_{ik,0}||}$$

$$+ \sum_{j\in nn(i)}(\sum_{k\in nn(j)}^{k\neq i} \frac{3}{4}\beta_{ijk}\frac{r_{ji}\cdot r_{jk} - d_{ji,0}\cdot d_{jk,0} + (r^k_m - r^j_m)^2}{||d_{ji,0}||||d_{jk,0}||})$$

for $m \neq n$,

$$\frac{\partial^2 U}{\partial r^i_m \partial r^i_n} = \frac{\partial^2}{\partial r^i_m \partial r^i_n}[\sum_{j\in nn(i)} U^{ij}_{bs} + \frac{1}{2}\sum_{j,k\in nn(i)}^{k\neq j} U^{jik}_{bb} + \sum_{j\in nn(i)}(\sum_{k\in nn(j)}^{k\neq i} U^{ijk}_{bb})]$$

$$= \sum_{j\in nn(i)} 3\alpha_{ij}\frac{(r^j_n - r^i_n)(r^j_m - r^i_m)}{||d^2_{ij,0}||}$$

$$+ \sum_{j,k\in nn(i)}^{k\neq j} \frac{3}{8}\beta_{jik}\frac{(2r^i_n - r^j_n - r^k_n)(2r^i_m - r^j_m - r^k_m)}{||d_{ij,0}||||d_{ik,0}||}$$

$$+ \sum_{j\in nn(i)}(\sum_{k\in nn(j)}^{k\neq i} \frac{3}{4}\beta_{ijk}\frac{(r^k_m - r^j_m)(r^k_n - r^j_n)}{||d_{ji,0}||||d_{jk,0}||})]$$

10.5 Python code

```python
import csv
import numpy as np
from numpy.linalg import matrix_power
import matplotlib.pyplot as plt
from numpy import linalg as LA
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDirectionArrows
import os
from scipy.optimize import leastsq
from scipy import sparse

class point:
    def __init__(self,x,y,z,ID):
        self.x = x
        self.y = y
        self.z = z
        self.v = np.array([x,y,z])
        self.ID = ID
        self.neighbours = []
        self.surfaceAtom = False

class lattice:



    @staticmethod
    def read(name):
        """
        reads a csv file and outputs a list of lists, holding data as strings
        """
        with open(name, newline='') as f:
            reader = csv.reader(f)
            return list(reader)


    @staticmethod
    def DeStringify(ListOfListOfStrings):
        """
        converts a list of list of strings to an array of floats
        """
        FloatArray= np.zeros([len(ListOfListOfStrings),len(ListOfListOfStrings[0])])
        for i in range(len(ListOfListOfStrings)):
            for j in range(len(ListOfListOfStrings[0])):
                FloatArray[i,j] = float(ListOfListOfStrings[i][j])
        return(FloatArray)

    @staticmethod
    def Indexer(InputArray):
        """
        converts an array, formatted as a set of ID's corresponding to a value in
        each column, to an array, with the number of dimensions matching the
        number of ID's
        """
        dim = len(InputArray[0]) - 1
        OutputArray = np.zeros(dim*[(1+int(np.max(InputArray[:,0:dim])))])

        if dim == 1:
            for element in InputArray:
                OutputArray[int(element[0])] = element[1]
        elif dim == 2:
            for element in InputArray:
                OutputArray[int(element[0]),int(element[1])] = element[2]
                OutputArray[int(element[1]),int(element[0])] = element[2]
        else:
            for element in InputArray:
                OutputArray[int(element[0]),int(element[1])
                            ,int(element[2])] = element[3]
                OutputArray[int(element[2]),int(element[1])
                            ,int(element[0])] = element[3]
        return(OutputArray)

    def ChainData(length,ID):
        """
        Produce data file for a chain of atoms
        """
        data = [[x,0,0,ID] for x in range(length)]
        with open('nChainData.csv', 'w', newline = '') as f:
            # create the csv writer
            writer = csv.writer(f)
            # write a row to the csv file
            writer.writerows(data)

    def GridData(length,dist,ID):
        """
        Produce data file for a grid of atoms
        dist is the distance between the atoms [x,y]
        length is the dimension [x,y]
        """
        data = []
        for x in range(length[0]):
            for y in range(length[1]):
                data.append([x*dist[0],y*dist[1],0,ID])

        with open('GridData.csv', 'w', newline = '') as f:
            # create the csv writer
            writer = csv.writer(f)
            # write a row to the csv file
            writer.writerows(data)

    def HoneycombData(length,a,ID):
        """
        Produce data file for a honeycomb of atoms
        dist is the distance between the atoms a
        length is the number of unit cells in each dimension [x,y]
        """
        data = []
        for x in range(length[0]):
                for y in range(length[1]):
                    data.append([a/2*(3*x + 3*y),
                                 a/2*(np.sqrt(3)*x - np.sqrt(3)*y),
                                  0,ID])
                    data.append([a/2*(3*x + 3*y + 1),
                                 a/2*(np.sqrt(3)*x - np.sqrt(3)*y + np.sqrt(3)),
                                  0,ID])


        with open('GridData.csv', 'w', newline = '') as f:
            # create the csv writer
            writer = csv.writer(f)
            # write a row to the csv file
            writer.writerows(data)

    def __init__(self,PointDataFileName,MassDataFileName,StretchDataFileName
                ,LengthDataFileName,BendDataFileName,AngleDataFileName,
                BoundaryConditions = None):
        """

        PointData file is formatted as first row x, second row y, third row z,
        last row ID: assign each to a lattice point object. ID is atomic number-1
        MassIndex: row of masses

        BondStretchData: Indexes bond stretching factor of BS(IJ).
        Format is ID I, ID J, length.

        IdealLengthData: Stores ideal bond length of BS(IJ).
        Format is ID I, ID J, length.

        IdealAngleData: Stores ideal angle of BB(JIK).
        Format is ID J, ID I, ID K, angle.

        BondBendingData: Stores bond bending factor of BB(JIK).
        Format is ID J, ID I, ID K, angle.

        Point Data is converted into point objects

        All other data types are converted to arrays of floats, and then
        an array of dimension equal to the number of ID's it has stores
        the data
        """

        data = self.read(PointDataFileName)

        self.points = [point(float(data[i][0]),float(data[i][1]),float(data[i][2]),
                             int(data[i][3])) for i in range(len(data))]
        self.DynamicalMatrix = np.zeros((3*len(self.points),3*len(self.points)))


        data = self.read(MassDataFileName)
        self.MassIndex= self.Indexer(self.DeStringify(data))

        data = self.read(LengthDataFileName)
        self.IdealLengthIndex= self.Indexer(self.DeStringify(data))

        data = self.read(StretchDataFileName)
        self.BondStretchingIndex= self.Indexer(self.DeStringify(data))

        data = self.read(BendDataFileName)
        self.BondBendingIndex= self.Indexer(self.DeStringify(data))

        data = self.read(AngleDataFileName)
        self.IdealAngleIndex= self.Indexer(self.DeStringify(data))

        self.BoundaryConditions = BoundaryConditions

    def neighbourscalc(self, bondlength):
        for PointToCheck in self.points:
            for neighbour in set(self.points) - {PointToCheck}: #check against
                #every other lattice point
                if self.DistanceObj(PointToCheck,neighbour,None)<=bondlength*1.001:
                    PointToCheck.neighbours.append(neighbour)
```

```python
def DistV(self, Point1, Point2,Axis = None):
    """
    calculates the vector between 2 points, as vectors
    """

    if self.BoundaryConditions is None:
        if Axis == None:
            return(Point2 - Point1)
        else:
            return(Point2[Axis]-Point1[Axis])
    else:
        if Axis == None:
            return([min(Point2[i]-Point1[i],
                        Point2[i]-Point1[i]-self.BoundaryConditions[i,1]
                        +self.BoundaryConditions[i,0],
                        Point2[i]-Point1[i]-self.BoundaryConditions[i,0]
                        +self.BoundaryConditions[i,1],
                    key = abs) for i in range(3)])
        else:
            return(min(Point2[Axis]-Point1[Axis]
                ,Point2[Axis]-Point1[Axis]-self.BoundaryConditions[Axis,1]
                +self.BoundaryConditions[Axis,0],
                Point2[Axis]-Point1[Axis]-self.BoundaryConditions[Axis,0]
                +self.BoundaryConditions[Axis,1],
                key = abs))

def Distance(self, Point1, Point2, Axis = None):
    """
    gets the Distance between 2 lattice points
    Point1 and Point2 are 2 lattice points / their indexes in the lattice
    object's list of points
    AreObjects is a boolean for whether the points are objects or their
    indexes(default)
    Axis defines whether the Distance is taken along a specific axis
    """

    return(np.linalg.norm(self.DistV(self.points[Point1].v ,
                                     self.points[Point2].v,Axis)))

def DistanceObj(self, Point1, Point2, Axis = None):
    """
    gets the Distance between 2 lattice points
    Point1 and Point2 are 2 lattice points / their indexes in the lattice
    object's list of points
    AreObjects is a boolean for whether the points are objects or their
    indexes(default)
    Axis defines whether the Distance is taken along a specific axis
    """

    return(np.linalg.norm(self.DistV(Point1.v,Point2.v,Axis)))
```

```
def innerObj(self,j,i,k):
    """
    Returns r(ij).r(ik)
    """
    return(np.inner(self.DistV(i.v,j.v)),self.DistV(i.v,k.v))

def inner(self,j,i,k, AreObjects = False):

    """
    Returns r(ij).r(ik)
    """

    return np.inner(self.DistV(
            self.points[i].v,self.points[j].v),
            self.DistV(self.points[i].v,self.points[k].v))

def matcalc(self):

    MassMatrix = lambda arg1: np.sqrt(arg1)*np.identity(3)

    #anon fn to generate mass matrix for given input mass

    for i in range(len(self.points)):
        pointI = self.points[i]
        for m in range(3):
                for n in range(3):
                    """
                    partial^2 U
                    /(partial r^i_m)(\partial r^i_n)
                    """
                    for pointJ in self.points[i].neighbours:
                        j = self.points.index(pointJ)

                        self.DynamicalMatrix[3*i+m,3*i+n]+=3*(
                            self.BondStretchingIndex[pointI.ID,pointJ.ID]
                            * self.Distance(i,j,m)
                            * self.Distance(i,j,n)
                            / self.IdealLengthIndex[pointI.ID,pointJ.ID]**2)

                        #BB term 1
                        for pointK in set(self.points[i].neighbours) - {pointJ}:
                            k = self.points.index(pointK)
                            self.DynamicalMatrix[3*i+m,3*i+n]+=3/8*(
                                self.BondBendingIndex[pointJ.ID,pointI.ID,
                                                      pointK.ID]
                                *(self.Distance(j,i,n)+self.Distance(k,i,n))
                                *(self.Distance(j,i,m)+self.Distance(k,i,m))
                                /self.IdealLengthIndex[pointI.ID,pointJ.ID]
                                /self.IdealLengthIndex[pointI.ID,pointK.ID])
                        #BB term 2
                        for pointK in set(self.points[j].neighbours) - {
                                pointI}:
                            k = self.points.index(pointK)
                            self.DynamicalMatrix[3*i+m,3*i+n]+=3/4*(
                                self.BondBendingIndex[pointI.ID,pointJ.ID,
                                                      pointK.ID]
                                *self.Distance(j,k,m)
                                *self.Distance(j,k,n)
                                /self.IdealLengthIndex[pointJ.ID,pointI.ID]
                                /self.IdealLengthIndex[pointJ.ID,pointK.ID])

        #incororate mass
        self.DynamicalMatrix[3*i:(3*i+3),3*i:(3*i+3)] = (
                    np.dot(np.dot(matrix_power(
                    MassMatrix(self.MassIndex[self.points[i].ID]),-1),
                    self.DynamicalMatrix[3*i:(3*i+3),3*i:(3*i+3)]),
                    matrix_power(MassMatrix(self.MassIndex[
                            self.points[i].ID]),-1)))

        for pointJ in pointI.neighbours:
            """
            nearest neighbour interaction terms
            """
            j = self.points.index(pointJ)

            #if already calculated symmetrically, reassign.
            if np.any(self.DynamicalMatrix[3*i:(3*i+3),3*j:(3*j+3)]):
                self.DynamicalMatrix[3*i:3*i+3,3*j:3*j+3] = np.transpose(
                        self.DynamicalMatrix[3*j:(3*j+3),3*i:(3*i+3)] )
            else:
                #looping for each dimension permutation:
                for m in range(3):
                    for n in range(3):
                        """
                        partial^2 U
                        /(partial r^i_m)(\partial r^j_n)
                        j in nn(i)
                        """
                        self.DynamicalMatrix[3*i+m,3*j+n] += -3 * (
                            self.BondStretchingIndex[pointI.ID, pointJ.ID]
                            * self.Distance(i,j,m)
                            * self.Distance(i,j,n)
                            / (self.IdealLengthIndex[pointI.ID,pointJ.ID])**2)

                        #Bond Bending #1
                        for pointK in set(pointI.neighbours) - {pointJ}:
                            k = self.points.index(pointK)
                            self.DynamicalMatrix[3*i+m,3*j+n] += -3/4* (
                                self.BondBendingIndex[pointJ.ID,
                                                      pointI.ID,pointK.ID]
                                *self.Distance(i,k,n)
                                *(self.Distance(i,k,m)+self.Distance(i,j,m))
                                /self.IdealLengthIndex[pointI.ID,pointJ.ID]
                                /self.IdealLengthIndex[pointI.ID,pointK.ID])

                        #Bond Bending #2
                        for pointK in set(pointJ.neighbours) - {pointI}:
                            k = self.points.index(pointK)
                            self.DynamicalMatrix[3*i+m,3*j+n] += -3/4* (
                                self.BondBendingIndex[pointI.ID,
                                                      pointJ.ID,pointK.ID]
                                *self.Distance(j,k,m)
                                *(self.Distance(j,k,n)+self.Distance(j,i,n))
                                /self.IdealLengthIndex[pointJ.ID,pointI.ID]
                                /self.IdealLengthIndex[pointJ.ID,pointK.ID])

                        #Bond bending #3
                        for pointK in set(pointI.neighbours).intersection(
                                set(pointJ.neighbours)):
                            k = self.points.index(pointK)
                            self.DynamicalMatrix[3*i+m,3*j+n] += 3/4*(
                                self.BondBendingIndex[pointI.ID,pointK.ID,
                                                      pointJ.ID]
                                *self.Distance(k,i,n)
                                *self.Distance(k,j,m)
                                /self.IdealLengthIndex[pointK.ID,pointI.ID]
                                /self.IdealLengthIndex[pointK.ID,pointJ.ID])

                #include mass in dynamical matrix
                self.DynamicalMatrix[3*i:(3*i+3),3*j:(3*j+3)
                    ] = np.dot(np.dot(matrix_power(MassMatrix(
                self.MassIndex[self.points[i].ID]),-1),
                    self.DynamicalMatrix[3*i:(3*i+3),
                    3*j:(3*j+3)]),
                    matrix_power(MassMatrix(self.MassIndex[self.points[j].ID]),-1))


        """
        partial^2 U
        /(partial r^i_m)(\partial r^j_m)
        m not in nn(i) U i
        """

        #work out set of points that are not nearest neighbours or i,
        #but neighbours of nearest neighbours of i
        pointJset = set([])
        for pointK in pointI.neighbours:
            for pointJ in set(pointK.neighbours)- {pointI
                             } - set(pointI.neighbours):
                pointJset.add(pointJ)
```

```
            for pointJ in pointJset:
                j = self.points.index(pointJ)
                #if already calculated symmetrically, reassign.
                if np.any(self.DynamicalMatrix[3*i:(3*i+3),3*j:(3*j+3)]):
                    self.DynamicalMatrix[3*i:3*i+3,3*j:3*j+3] = np.transpose(
                            self.DynamicalMatrix[3*j:(3*j+3),3*i:(3*i+3)] )
                else:
                    for pointK in set(pointJ.neighbours).intersection(
                            set(pointI.neighbours)):
                        k= self.points.index(pointK)
                        for m in range(3):
                                for n in range(3):
                                    self.DynamicalMatrix[3*i+m,3*j+n] += 3/4*(
                                        self.BondBendingIndex[pointI.ID,
                                                              pointK.ID,
                                                              pointJ.ID]
                                        *self.Distance(k,j,m)*self.Distance(
                                                k,i,n)
                                        /self.IdealLengthIndex[pointK.ID,
                                                               pointI.ID]
                                        /self.IdealLengthIndex[pointK.ID,
                                                               pointJ.ID])

                    #include mass in dynamical matrix
                    self.DynamicalMatrix[3*i:(3*i+3),3*j:(3*j+3)
                        ] = np.dot(np.dot(matrix_power(MassMatrix(
                            self.MassIndex[self.points[i].ID]),-1),
                        self.DynamicalMatrix[3*i:(3*i+3),
                        3*j:(3*j+3)]),
                        matrix_power(MassMatrix(self.MassIndex[
                                self.points[j].ID]),-1))

        self.DynamicalMatrixSparce = sparse.csr_matrix(self.DynamicalMatrix)

    def EigSolve(self):
        eigenvals, vectors = sparse.linalg.eigsh(self.DynamicalMatrixSparce,
                                                 k = len(self.points),
                                                 ncv = len(self.points)*2,
                                                 which = 'SM')
#         self.eigenvals = np.around(np.absolute(eigenvals),3)
        self.eigenvals = eigenvals
        self.w =np.sqrt(np.round(self.eigenvals,5))
#         self.vectors = np.around(np.absolute((vectors)),3)
        self.vectors = np.round(vectors,8)

    def EigPlot2D(self,ax,ValToPlot,param_dict,ArrowWidth = 0.1):
        #
        """
        graph for eigenvectors which are ALL orthogonal to one axis

        Parameters
```

```
----------
ax : Axes
    The axes to draw to

ValtoPlot: eigenvalue/eigenvectors to plot

param_dict : dict
   Dictionary of kwargs to pass to ax.plot

Returns
-------
out : list
    list of artists added
"""
CoordDict = {0: 'x', 1: 'y', 2: 'z'}
OrthAxis = None

#decide on orthogonal axis based on which has zero vals

for i in range(3):
    if np.any([self.vectors[i + 3*x,ValToPlot] for x in range(
            len(self.points))])== False:
        OrthAxis = i

if OrthAxis == None:
    raise TypeError("Eigenvectors are not orthogonal to x, y or z.")
    return

Axes = list({0,1,2} - {OrthAxis})

out = []
#plot arrows then points on top
for i in range(len(self.points)):
    pointI = self.points[i]
    #only plot arrow if there is non-zero eigenvector in the plane
    if np.any([self.vectors[[3*i + Axes[0],3*i + Axes[1]],
                           ValToPlot]])== True:
        out.append(plt.arrow(pointI.v[Axes[0]],
                  pointI.v[Axes[1]],
                  self.vectors[3*i + Axes[0],ValToPlot]/4,
                  self.vectors[3*i + Axes[1],ValToPlot]/4,
                  width = ArrowWidth))

    out.append(plt.scatter(pointI.v[Axes[0]], pointI.v[Axes[1]],color = 'r'))

simple_arrow = AnchoredDirectionArrows(ax.transAxes,
                                       CoordDict[Axes[0]],
                                       CoordDict[Axes[1]],
                                       color = 'black')

out.append(ax.add_artist(simple_arrow))
```

```
        textXLoc = (max([self.points[a].v[Axes[0]] for a in range(len(self.points))
                    ])) * 0.8
        textYLoc = (max([self.points[a].v[Axes[1]] for a in range(len(self.points))
                    ])) * 0.8
        out.append(plt.text(textXLoc,textYLoc,'w^2 =' + str(np.round(
                self.eigenvals[ValToPlot],2))))

        return out

    def EigPlotPhonon(self,ax,ValToPlot,param_dict,ArrowWidth = 0.1):
        #
        """
        graph for visualising phonons in single dimension

        Parameters
        ----------
        ax : Axes
            The axes to draw to

        ValtoPlot: eigenvalue/eigenvectors to plot

        param_dict : dict
           Dictionary of kwargs to pass to ax.plot

        Returns
        -------
        out : list
            list of artists added
        """
        #CoordDict = {0: 'x', 1: 'y', 2: 'z'}
        Axes = [0,1,2]

        #decide on orthogonal axis based on which has zero vals

        for i in range(3):
            if np.any([self.vectors[i + 3*x,ValToPlot] for x in range(
                    len(self.points))])== False:
                Axes.remove(i)

        if Axes[0] == None:
            raise TypeError("Eigenvectors are multidimensional.")
            return

        out = []
        #plot arrows then points on top

        #only plot arrow if there is non-zero eigenvector in the plane
        plt.plot([point.v[Axes[0]] for point in self.points],
                      [self.vectors[3*i + Axes[0],ValToPlot
                                  ] for i in range(len(self.points))])
    textXLoc = (max([self.points[a].v[0] for a in range(len(self.points))
    ])) * 0.8
    out.append(plt.text(textXLoc,0.4,'w^2 =' + str(
            np.round(self.eigenvals[ValToPlot],2))))

    return out


def PhononBasisX(self):
    '''
    creates PhononBasis attribute and Kx to go along with it
    assumes chain in x
    '''
    a = 1
    if self.BoundaryConditions is None:
        R = max([self.points[i].v[0] for i in range(len(self.points))]) - \
            min([self.points[i].v[0] for i in range(len(self.points))]) + a
    else:
        R = self.BoundaryConditions[0, 1]-self.BoundaryConditions[0, 0]
    nMax = int(R / 2 / a)
    self.Kx = [2 * np.pi * n / R for n in range(-nMax, nMax)]
    self.PhononBasis = np.zeros([2*nMax,len(self.vectors)],dtype = np.complex64)

    for n in range(2*nMax):
        for PointIndex in range(len(self.points)):
            self.PhononBasis[n,3*PointIndex] = np.exp(1j * self.Kx[n]
                *self.points[PointIndex].v[0])
        #normalise
        self.PhononBasis[n] = self.PhononBasis[n] / LA.norm(self.PhononBasis[n])

def EigenProjection(self):
    '''
    project eigenvectors onto basis
    '''
    self.projections = np.zeros([len(self.vectors),len(self.PhononBasis)])
    for index in range(len(self.vectors)):
        self.projections[index] = np.round(np.power
                        (np.abs(np.inner(self.PhononBasis,
                                        self.vectors[:,index])),2),3)

def kDecomposition(self, ax, index):
    '''
    create figure of intensity of eigenstate against k-value.

    Parameters
    ----------
    ax : Axes
        The axes to draw to
        index: index of eigenvalue to plot

        param_dict : dict
           Dictionary of kwargs to pass to ax.plot

        Returns
        -------
        out : list
            list of artists added
        '''
        out = [plt.scatter(self.Kx,self.projections[index])]
        out.append(plt.title('phonon '+ str(index)+', w = ' +str(
                np.round(self.w[index],2))))
        out.append(plt.xlabel('k'))
        out.append(plt.ylabel('I'))
        return out


    def eOverlap(self):
        '''
        get a k for each eigenvector, sums over each atom
        electron field is calculated by summing over each point's distance from centre
        '''
        D = 1

        # 3-Dimensional electric field parsing for each point. product of gaussians
        #in x, y and z
        self.electronField = [D * np.sqrt(1 /2/self.R0[0]/self.R0[1]/self.R0[2]) *(1/np.sqrt
                         np.exp(-1/4*(self.DistV(point.v,self.X0,0)/self.R0[0])**2)
                         *np.exp(-1/4*(self.DistV(point.v,self.X0,1)/self.R0[1])**2)
                         *np.exp(-1/4*(self.DistV(point.v,self.X0,2)/self.R0[2])**2)
                         for point in self.points]

        #self.dv is arranged as the first index corresponds to which k is being thought abou
        #and the second index is which atom is being looped over.
        self.dv = np.zeros([len(self.vectors[0]),len(self.points)])
        for v in range(len(self.vectors[0])):
            for point in range(len(self.points)):
                self.dv[v,point] = np.sum([2 * np.inner(
                       self.DistV(self.points[point].v,neighbour.v),
                       (3*self.vectors[3*self.points.index(neighbour):
                           3*self.points.index(neighbour)+3,v] -
                        self.vectors[3*point:3*point+3,v]))
                           for neighbour in self.points[point].neighbours])
#        self.electronField = []
#        for n in range(100):
#            for i in range(3): self.electronField.append(a[n])
#
#
        self.gk = [np.inner(v,self.electronField) for v in self.dv]
        self.absgk2 = [abs(np.inner(v,self.electronField))**2 for v in self.dv]
```

```
def gaussianRepresentation(self, n = 1000):
    '''
    takes gk^2 data and the frequencies, adds them as gaussians of width
    sigma
    '''
    #first, define the set of data points these gaussians are going to sit
    #on top of.
    self.wSpace = np.linspace(min(self.w),max(self.w),n)
    self.gk2Space = np.zeros(len(self.wSpace))

    #then, for every g(k)^2, add as a gaussian centred on w
    for i in range(len(self.w)):
        self.gk2Space = np.array([self.gk2Space[j] + self.absgk2[i]* np.exp(
                -1/2 * (self.wSpace[j] - self.w[i])**2/self.sigma**2)
        for j in range(len(self.gk2Space))])

def FittingGaussian(self,params):
    '''
    takes initial parameters and fits against guassian representation;
    creates variable of fitted parameters
    '''
    def AnalyticalGk2(params):
        return np.power(params[0] * np.sin(self.wSpace) *
                        np.exp(-params[1]**2*np.power(self.wSpace,2)),2)
    def FittingFunc(params):
        return self.gk2Space - AnalyticalGk2(params)


    self.paramsFitted = leastsq(FittingFunc, params)

def PLottingFitted(self,ax):
    #create plot
    def AnalyticalGk2(params):
        return np.power(params[0] * np.sin(self.wSpace) *
                        np.exp(-params[1]**2*np.power(self.wSpace,2)),2)
    items = []
    items.append(plt.plot(self.wSpace,self.gk2Space,
                          label = 'simulation data'))
    items.append(plt.plot(self.wSpace,AnalyticalGk2(self.paramsFitted[0]),
                          label = 'analytical fitted model'))
    items.append(plt.title('Fitting of spectral density'))
    items.append(plt.legend())
    items.append(plt.xlabel('w(k)'))
    items.append(plt.ylabel('g(k)**2'))
    return items

def RadiusFitting(self, radii):
    '''
    for each radius
    -calculate electron phonon interaction (eOverlap)
```

```
        -Calculate gaussian representation
        -calculate Fitting gaussian
        -store each parameter in a vector
        -plot each paramter against radius
        '''
        params = []
        for r in radii:
            self.R0 = r
            self.eOverlap()
            self.gaussianRepresentation()
            self.FittingGaussian([1,1])
            params.append(self.paramsFitted[0][1])
        return(params)

    def electronPhononInteraction(self, ax):
        '''
        create figure of abs(g(k))**2 against w(k).

        Parameters
        ----------
        ax : Axes
            The axes to draw to

        index: index of eigenvalue to plot

        param_dict : dict
           Dictionary of kwargs to pass to ax.plot

        Returns
        -------
        out : list
            list of artists added
        '''
        out = [plt.scatter(self.w,self.absgk2)]
        out.append(plt.title('electron phonon interaction, r = ' + str(np.round(self.R0, 2))
        out.append(plt.xlabel('w(k)'))
        out.append(plt.ylabel('abs{g(k)}^2'))
        return out

    def electronPhononInteractionGaussian(self, ax):
        '''
        create figure of abs(g(k))**2 against w(k), using gaussian added
        data.

        Parameters
        ----------
        ax : Axes
            The axes to draw to

        index: index of eigenvalue to plot
        param_dict : dict
           Dictionary of kwargs to pass to ax.plot

        Returns
        -------
        out : list
            list of artists added
        '''
        out = [plt.plot(self.wSpace,self.gk2Space)]
        out.append(plt.title('sum of electron phonon interaction, r = '
                             + str(np.round(self.R0, 2)) + ', sigma = '
                             +str(np.round(self.sigma,2))))
        out.append(plt.xlabel('w(k)'))
        out.append(plt.ylabel('abs{g(k)}^2'))
        return out

    def NeighboursPlot(self,ax):
        '''
        create figure of neighbours and bonds within them.

        Parameters
        ----------
        ax : Axes
            The axes to draw to

        index: index of eigenvalue to plot

        param_dict : dict
           Dictionary of kwargs to pass to ax.plot

        Returns
        -------
        out : list
            list of artists added
        '''
        out = []
        out.append(plt.scatter([point.v[0] for point in self.points],
                    [point.v[1] for point in self.points]))
#        for i in range(len(self.points)):
#            out.append(plt.annotate(i,
#                                    [self.points[i].v[0],self.points[i].v[1]],
#                                    textcoords = 'offset points',
#                                    xytext = [7,3]))
        for point in self.points:
            for neighbour in point.neighbours:out.append(plt.plot(
                    [point.v[0],neighbour.v[0]],
                    [point.v[1],neighbour.v[1]],color = 'red'))
        ax.set_aspect('equal', adjustable='box')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        out.append(plt.title("b: grid"))
```

10.6 Previous work

Towards Predicting the Phonons of 2D Quantum Devices

Mphys. year 4 Research Project
Heriot-Watt University
Submission date: 16/07/2021
Student: Laura Rintoul(H00293952)
Supervisor: Dr E. Gauger

1 Heriot Watt University Physics Department – Own Work Declaration

I confirm that all this work is my own except where indicated, and that I have:
· Clearly referenced/listed all sources as appropriate
· Referenced and put in inverted commas all quoted text (from books, web, etc)
· Given the sources of all pictures, data etc. that are not my own
· Not made any use of the essay(s) of any other student(s) either past or present
· Not sought or used the help of any external professional agencies for the work
· Acknowledged in appropriate places any help that I have received from others
Name: Laura Rintoul
Student Number: H00293952
Course/Programme: B20AX, 4th Year Mathematical Physics
Title of work: Towards Predicting the Phonons of 2D Quantum Devices
Date: 21/06/21

Contents

2 Abstract

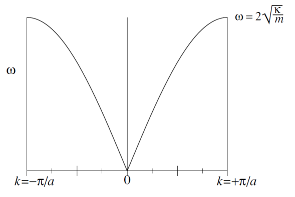
Practical quantum technologies rely on robust hardware. For many applications, semiconductor quantum nanostructures are ideal candidates, e.g. as physical implementations of quantum bits, or as emitters of non-classical light. However, phonons, i.e. vibrations in the underlying crystal structure, severely change the behaviour and operation of semiconductor quantum devices in ways that - in many cases - we do not yet fully understand. A novel class of promising nanostructures is based on two-dimensional materials such as graphene (the exploration of which awarded the Nobel Prize in 2010) and MoS2 (molybdenum disulfide). There are hopes of engineering specialist and novel materials by layering these materials. Surprisingly, the effects of phonons on quantum devices are largely determined by a single function, the so-called phonon spectral density. While analytic approximations to the spectral density are known for many conventional physical structures, the same is not the case for 2D material due to a number of additional and unusual features in their electron and phonon band structures. A modified valence force field (MVFF) model was simplified with the aims of testing and programming computation for phonon spectral density analysis for 1D structures, to compare with expected results for accuracy, with aims of then moving on to 2D materials. This was developed in python 3. This simplified MVFF model was successful in predicting the dispersions and effects of finite structures on periodic and non-periodic chain phonon structures.

3 Acknowledgements

I'd like to thank Dr Erik Gauger and Dr Moritz Cygorek for their help with this research project.

$$w = 2\sqrt{\frac{k}{m}} sin(\frac{ka}{2})$$

(a) solution equation



(b) dispersion plot

Figure 1: algebraic solution for 1-d chain

4 Introduction

Phonons are, in essence, vibrations in matter. While they can be described as quantum mechanical excitations and in terms of the Hamiltonian of a system, for the purposes of this project they will be considered classically: as the collective excited states of systems with regards to the particles that makes them up physically vibrating. In this classical description, these systems would be thought of as simple 'springs' connecting particles, giving rise to simple harmonic motion. However, even for more complicated valence fields, the 2nd order approximation makes sense for small displacements, so this relatively simple model of simple harmonic oscillators can be maintained. This results in certain stable conditions, or eigenstates, in the lattice being analysed. For example, in a 1-dimensional chain, the problem is generally simplified down to thinking purely about translational potentials and neglecting any bond bending. It can be classically depicted to obtain algebraically a function in the finite and infinite situations [1], as shown in figure 1.

Phonons are generally described in dispersion plots in terms of a frequency and a wavenumber. For periodic structures, phonons can be exactly decomposed onto a basis described by bloch functions [2], $\phi_k(\boldsymbol{r}) = exp(i\boldsymbol{k} \cdot \boldsymbol{r}) * u_k(\boldsymbol{r})$. Here, u is any function that is periodic with the structure. So, for the 1-dimensional chain of the same material, $u = 1$ is sufficient: $\phi_k(x) = exp(ikx)$ The k vectors described above are those that are used in dispersion plots. In a 1-dimensional case, the direction of that k-vector on one axis is enough to fully describe the basis of k-vectors. With periodic boundary conditions, the requirement that $\phi(x) = \phi(x + X)$ where X is the boundary condition distance means that k is discretised to $k = \frac{2\pi n}{X}$. For non-periodic boundary conditions, these functions are still a basis but the eigenstates will not project exactly, and phonon broadening is expected.

This project builds towards the existing work on 2D quantum devices and structures. There is a large amount of interest in use of 2D quantum devices for quantum computing, be it for qubits or for single photon emission [3]. There is particular interest in the combination of these structures to build new and novel materials, with the potential to custom build materials to fit a problem [4]. The problem of the impact of phonons in these materials is well known, such as interaction with electrons in structures [5] and their involvement in qubit relaxation processes [6]. Developing a tool that can accurately produce results for phonon spectral density based on the atomic arrangement of these structures could allow for computational testing without the requirement for the arduous process of building these devices in a lab. These atomised valence field models have been tested on GaP quantum dots to success [7], so general application to 2D structures is certainly worth persuing.

5 Theory and Approach

5.1 Model

The valence force field models are methods of analysing microscopic structures by treating them atomistically. This means that, instead of considering phonons in terms of waves from the onset, they are instead considered based on valence forces between the atoms in a structure. This set of methods has been used successfully to model phonon band structures, stress tensors and more [8] [9]. The specific method being used is called the Modified valence force method, and defines the potential of the bonds in the model using a few different types of terms: bond bending (a), bond stretching (b), bond stretching-bond stretching (c), bond stretching-bond bending (d) and bond bending-bond-bending (e):

$$U = \frac{1}{2}\sum_{i\in N_A}[\sum_{j\in nn(i)} U_{bs}^{ij} + \sum_{j,k\in nn(i)}^{k\neq j}(U_{bb}^{jik} + U_{bs-bs}^{jik} + U_{bs-bb}^{jik}) + \sum_{j,k,l\in COP_i}^{j\neq k\neq l} U_{bb-bb}^{jikl}]\ [10]$$

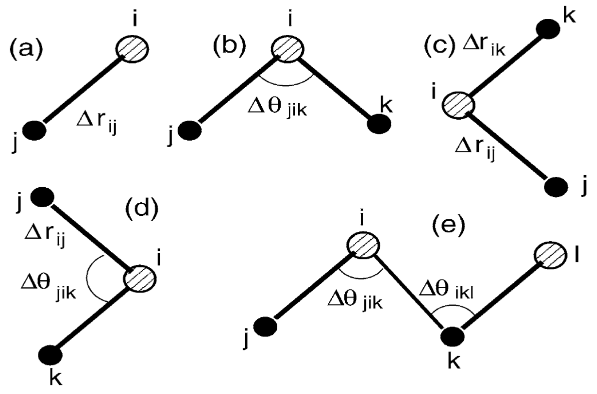


Figure 2: Bond bending types [10]

This large quantity of terms is useful for some results from valence field calculations, but for the purpose of phonon band structure extraction and computational limitation, only the bond stretching and bond bending terms were used. Due to the large symmetry of the structures being analysed, these terms can be dismissed as negligible. Therefore, the simplified form of:

$$U = \frac{1}{2}\sum_{i\in N_A}[\sum_{j\in nn(i)} U_{bs}^{ij} + \sum_{j,k\in nn(i)}^{k\neq j} U_{bb}^{jik}]$$

$$= \frac{1}{2}\sum_{i\in N_A}[\sum_{j\in nn(i)} \frac{3}{8}\alpha_{ij}\frac{(r_{ij}^2 - d_{ij,0}^2)^2}{||d_{ij,0}^2||} + \sum_{j,k\in nn(i)}^{k\neq j}\frac{3}{8}\beta_{jik}\frac{(\Delta\theta_{jik})^2}{||d_{ij,0}||||d_{ik,0}||}]$$

was used. here, the alpha and beta terms are constants analogous to spring constants, determining how high the potential increase over distance is. r_{ij} is the distance between points i and j, and $d_{ij,0}$ is the ideal bond length. $(\Delta\theta_{jik} = (r_{ij}\cdot r_{ik} - d_{ij,0}\cdot d_{ik,0}))$.

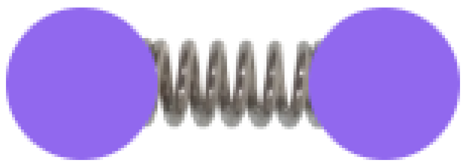
Prior to computational calculation, this potential has to be processed to give a form that can be described as a eigenvalue problem. This means calculating what is called the Dynamical Matrix:

$$D_{mn}^{ij} = \frac{\partial^2 U_{elastic}}{\partial r_m^i \partial r_m^j}$$

for each dimension and each pair of atoms. Therefore, the explicit partial derivative of each pair of interactions was calculated (8.1, page 12). This was explicitly calculated for all dimensions and possible permutations of which point is having the derivative taken so that the dynamical matrix calculation is applicable to all structures: while it would be easier to calculate explicitly for just the one dimensional chain being investigated, it would be less useful for the broader investigation of this method with regards to applications for other structures.

$$D = \begin{bmatrix} 3 & 0 & 0 & -3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) dynamical matrix



(b) how the model physically describes the pair of atoms

Figure 3: atom pair on x-axis, $\alpha = 1$, distance between atoms = 1

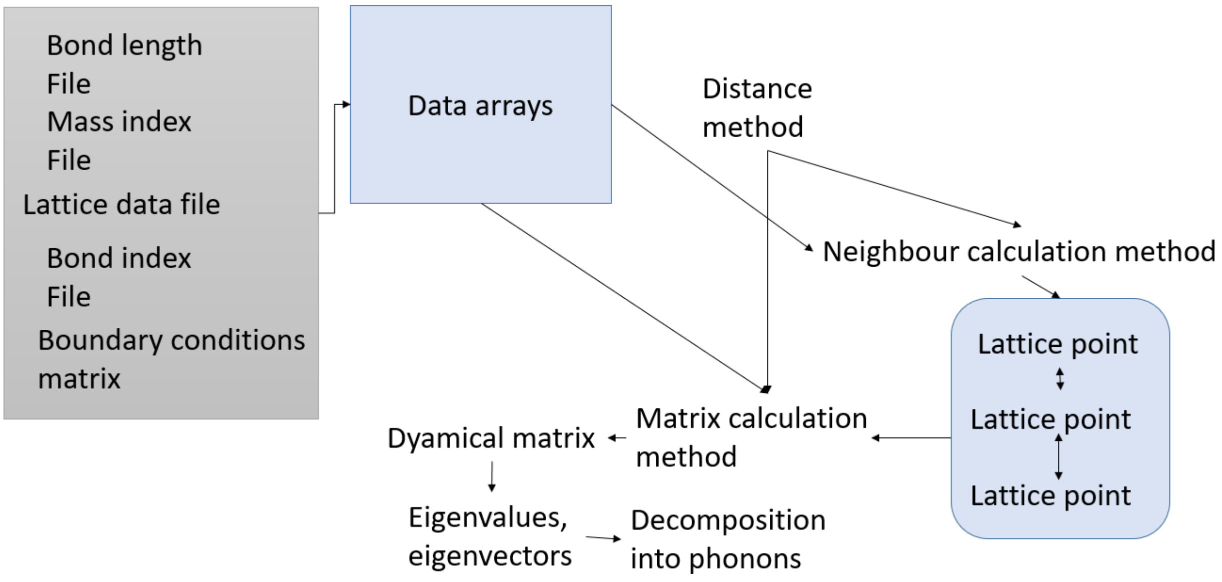


Figure 4: Program flowchart structure

This dynamical matrix is a method of deconstructing the behaviour of a lattice around the equilibrium through a 2nd order Taylor expansion of the force. This means that higher order terms will drop out entirely, and what results from the computation is a matrix that is $3n * 3n$ in dimension. For example, Figure 3b shows how the dynamical matrix for a pair of atoms would be arranged. The assignment of atom reference numbers is entirely arbitrary but must be maintained throughout the calculation; the central diagonal will always be the derivative of each atom with respect to its own coordinates.

For the linear chain, this model cannot accommodate for 180 degree bond bending potentials in orthogonal axes, so the bond bending parameter effectively acted as an additional bond stretching potential.

5.2 Implementation

This dynamical matrix system of computing phonon structures was computed using python 3. An object oriented approach was used, in which lattice data was stored in a file externally and then imported into the program and converted into a lattice object composed of lattice points. A flowchart of the program is shown in figure 4. The process of importing this data about lattice points, an ID and the coordinate points of the lattice point were used to describe each point.These ideas were used to transfer information about bond length, ideal bond angle and potential coefficients for bonds to the program. Then, the program could determine which lattice points were neighbours.

In order to calculate distances while implementing boundary conditions, a custom distance method had to be used. In one dimension, it is easy to implement boundary conditions: it's only one distance vector. But in higher dimensions, there are many combinations of boundary conditions possible. Therefore, developing a method for describing these combinations and checking which is shortest was a major challenge for the project. The solution arrived at was using ternary multiples of the boundary condition vectors to represent all possible combinations, and taking the minimum of their addition to the distance between any two points.

The equations calculated in 8.1 were directly coded into the program. The equations were derived by hand and checked using sympy for python.

As the number of atoms gets larger, the size of the dynamical matrix increases rapidly: $9n^2$ elements, for n being the number of atoms. Therefore, to reduce on computing time, the symmetrical nature of partial derivatives was taken advantage of to approximately half computing time: only an upper diagonal of the points need be calculated and the rest are symmetrical.

6 Results

6.1 Depicting Expected Behaviour

For the following calculations, a bend stretching coefficient of 1 and bond bending coefficient of 20 were used. Due to the unitless nature of these calculations and the bond-bending essentially acting as another bond stretching term, the actual values of these were unimpactful towards the actual results seen. The first checks for ensuring the model was working as expected were to see if the model was successfully generating eigenvectors that correspond to the expected eigenvalues: i.e. translation and other zero energy operations having zero energy. An unexpected side effect of the dynamical matrix being a Taylor expansion nature is that for 180 degree bond angles, orthogonal displacements had zero energy contribution. However, this would also be the case even for adding in the more complicated terms to the MVFF model. For example, in the 5 atom case, some of the eigenstates are depicted in the following figures:

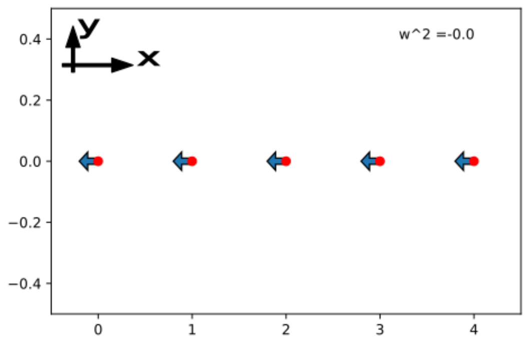


Figure 5: zero energy phonon for full translation

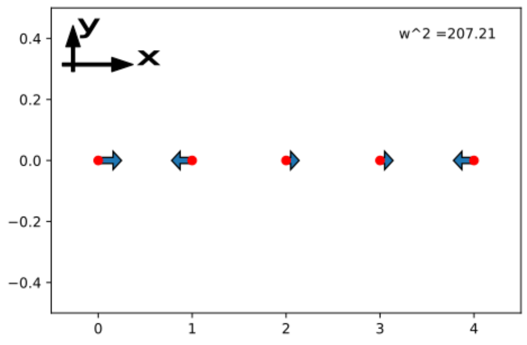


Figure 6: zero energy phonon for single orthogonal translation

In these diagrams, the red dots represent atoms and the arrows show the direction of movement in the eigenstate. w^2 in the top right corner is the angular frequency squared of the state. For non zero w^2, the implication is that there would be a restoring motion in the opposite direction of the arrow also. Figures 5 and 7 clearly show the expected zero energies for full system translation and orthogonal translation, and figure 6 shows an eigenvalue that looks promising. However, this format of viewing phonons as simply movements can only reveal so much about how accurate the system is.

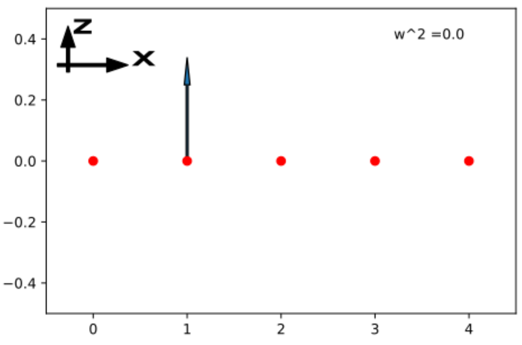


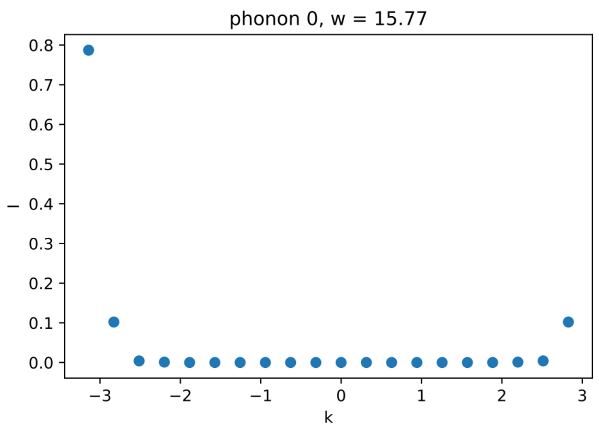
Figure 7: non-zero energy phonon

6.2 Phonon Distribution for Open and Closed Boundary Conditions

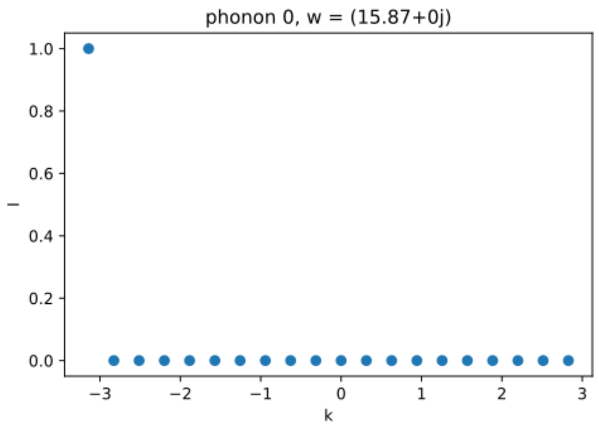
As previously stated, it is expected that for a linear chain that phonon eigenstates' spectral density should project directly onto bloch states, with some spectral broadening for non-periodic boundary conditions. For a 20 atom linear chain, some of these are compared in figures 8, 9 and 10. The expected decomposition is shown for periodic, which is full contribution from one energy state. For the non zero energy spectral densities 9b and 10b, the two states with 0.5 spectral density (the intensity on the y axis) have the same energy; are degenerate. For the finite decompositions, there is a range of spectral broadenings observed, determined by how close the phonon is to one of the discrete basis Bloch states. In 10a, a broader spectrum is observed than in 9. As the frequency increases, the boundary effects lessen. For higher numbers of points, it is harder to see this effect as the spectral broadenings narrow with the resolution (11). It should be noted that these spectral densities are shown in the first brillouin zone, so these plots are periodic in nature: hence why in 8a, there is a contribution from a high k value as well as a low.

6.3 Phonon Spectral Densities

The phonon spectral densities going from finite to infinite did not differ greatly, as is to be expected. They replicated the expected results for a 1 dimensional infinite chain, as was shown in 1. This spectral density is shown in 12.



(a) Finite decomposition of an eigenstate

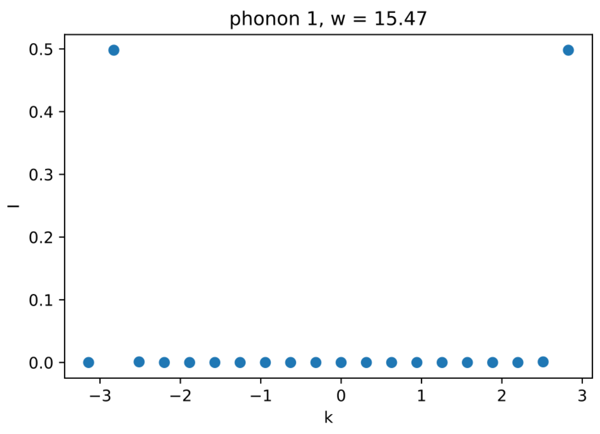


(b) periodic decomposition of an eigenstate

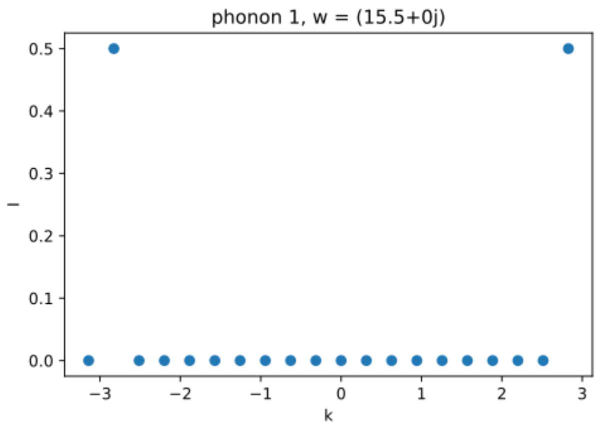
Figure 8: spectral densities of an eigenstate for a chain length of 20

7 Conclusions

In conclusion, the MVFF method for analysing phonon densities has proven successful in the 1 dimensional case. It successfully demonstrated phonon broadening for finite structures, giving expected results for physical behaviour of phonon eigenstates. Spectral densities for 1 dimensional chains returned the expected results. It has shown promise for the simplification of the MVFF method to just bond bending and bond stretching terms, and this project has laid the groundwork for further analysis on 3 dimensional and 2 dimensional structures.



(a) Finite decomposition of an eigenstate



(b) periodic decomposition of an eigenstate

Figure 9: spectral densities of an eigenstate for a chain length of 20

8 Appendix

8.1 Dynamical matrix calculations

8.1.1 Expanding potentials for given bond and coordinates

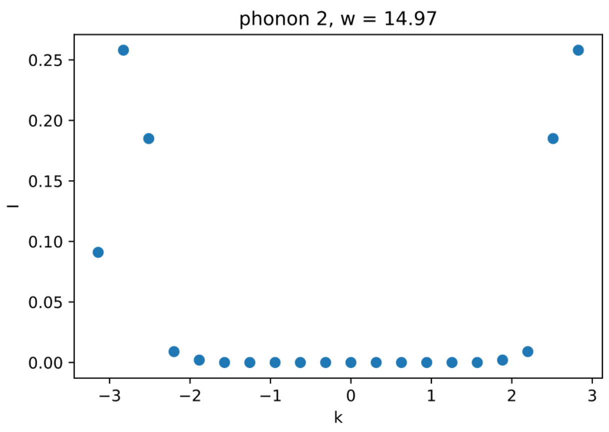
$U_{bb}^{ij} = U_{bb}^{ji}, U_{bs}^{ijk} = U_{bs}^{kji}$

$$U = \frac{1}{2} \sum_{i \in N_A} [\sum_{j \in nn(i)} U_{bs}^{ij} + \sum_{j,k \in nn(i)}^{k \neq j} U_{bb}^{jik}]$$

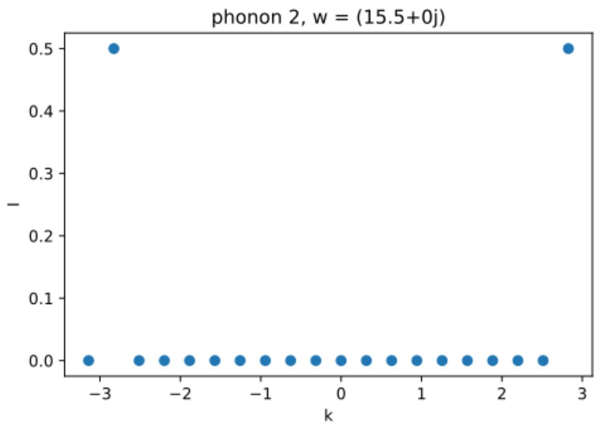
$$\frac{\partial U}{\partial r_m^i} = \frac{\partial}{\partial r_m^i} \frac{1}{2} \sum_{i \in N_A} [\sum_{j \in nn(i)} U_{bs}^{ij} + \sum_{j,k \in nn(i)}^{k \neq j} U_{bb}^{jik}]$$

$$= \frac{\partial}{\partial r_m^i} \frac{1}{2} [\sum_{j \in nn(i)} U_{bs}^{ij} + \sum_{j,k \in nn(i)}^{k \neq j} U_{bb}^{jik} + \sum_{j \in N_A}^{j \neq i} (\sum_{i \in nn(j)} U_{bs}^{ji} + \sum_{i,k \in nn(j)}^{k \neq i} U_{bb}^{ijk})]$$

$$= \frac{\partial}{\partial r_m^i} \frac{1}{2} [2 \sum_{j \in nn(i)} U_{bs}^{ij} + \sum_{j,k \in nn(i)}^{k \neq j} U_{bb}^{jik} + \sum_{j \in N_A}^{j \neq i} (\sum_{k \in nn(j)}^{k \neq i} U_{bb}^{ijk} + U_{bb}^{kji})]$$



(a) Finite decomposition of an eigenstate



(b) periodic decomposition of an eigenstate

Figure 10: spectral densities of an eigenstate for a chain length of 20

$$\frac{\partial^2 U}{\partial r^i_m \partial r^j_n} = \frac{\partial}{\partial r^i_m \partial r^j_n}\frac{1}{2}[2\sum_{j\in nn(i)} U^{ij}_{bs} + \sum_{j,k\in nn(i)}^{k\neq j} U^{jik}_{bb} + \sum_{j\in nn(i)} (\sum_{k\in nn(j)}^{k\neq i} 2U^{ijk}_{bb})]$$

if $j \in nn(i)$,

$$\frac{\partial^2 U}{\partial r^i_m \partial r^j_n} = \frac{\partial^2}{\partial r^i_m \partial r^j_n}\frac{1}{2}[2U^{ij}_{bs} + \sum_{k\in nn(i)}^{k\neq j} (U^{jik}_{bb} + U^{kij}_{bb}) + (\sum_{k\in nn(j)}^{k\neq i} 2U^{ijk}_{bb}) + \sum_{k\in nn(i),nn(j)} 2U^{ikj}_{bb}]$$

$$= \frac{\partial^2}{\partial r^i_m \partial r^j_n}[U^{ij}_{bs} + \sum_{k\in nn(i)}^{k\neq j} U^{jik}_{bb} + \sum_{k\in nn(j)}^{k\neq i} U^{ijk}_{bb} + \sum_{k\in nn(i),nn(j)} U^{ikj}_{bb}]$$

if $j \notin nn(i) \cup i$,

$$\frac{\partial^2 U}{\partial r^i_m \partial r^j_n} = \frac{\partial^2}{\partial r^i_m \partial r^j_n}\frac{1}{2}[\sum_{k\in nn(i),nn(j)} 2U^{ikj}_{bb}] = \frac{\partial^2}{\partial r^i_m \partial r^j_n}\sum_{k\in nn(i),nn(j)} U^{ikj}_{bb}$$

$$\frac{\partial^2 U}{\partial r^i_m \partial r^i_n} = \frac{\partial}{\partial r^i_m \partial r^i_n}\frac{1}{2}[2\sum_{j\in nn(i)} U^{ij}_{bs} + \sum_{j,k\in nn(i)}^{k\neq j} U^{jik}_{bb} + \sum_{j\in nn(i)} (\sum_{k\in nn(j)}^{k\neq i} 2U^{ijk}_{bb})]$$

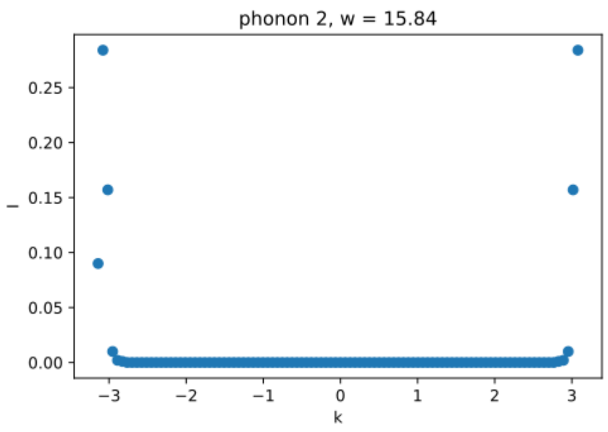


Figure 11: Finite decomposition of an eigenstate for a chain length of 100

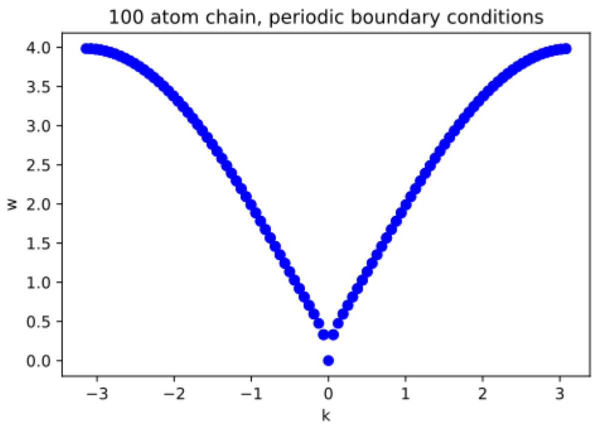


Figure 12: Spectral Density of phonons for a 1 dimensional 100 atom chain

8.1.2 derivatives of individual bond potentials

$$U_{bs}^{ij} = \frac{3}{8}\alpha_{ij}\frac{(r_{ij}^2 - d_{ij,0}^2)^2}{||d_{ij,0}^2||}$$

$$r_{ij}^2 = \sum_m (r_m^j - r_m^i)^2$$

$$\frac{\partial U_{bs}^{ij}}{\partial r_m^i} = \frac{3}{8}\alpha_{ij}\frac{2(r_{ij}^2 - d_{ij,0}^2)(-2(r_m^j - r_m^i))}{||d_{ij,0}^2||}$$

for $i \neq j$:

$$\frac{\partial^2 U_{bs}^{ij}}{\partial r_m^i \partial r_m^j} = -\frac{3}{2}\alpha_{ij}\frac{(r_{ij}^2 - d_{ij,0}^2) + 2(r_m^j - r_m^i)(r_m^j - r_m^i)}{||d_{ij,0}^2||} = -\frac{3}{2}\alpha_{ij}\frac{(r_{ij}^2 - d_{ij,0}^2) + 2(r_m^j - r_m^i)^2}{||d_{ij,0}^2||}$$

$$\frac{\partial^2 U_{bs}^{ij}}{\partial r_m^i \partial r_n^j} = -\frac{3}{2}\alpha_{ij}\frac{2(r_n^j - r_n^i)(r_m^j - r_m^i)}{||d_{ij,0}^2||} = -3\alpha_{ij}\frac{(r_n^j - r_n^i)(r_m^j - r_m^i)}{||d_{ij,0}^2||} \text{ for } m \neq n$$

$$\frac{\partial^2 U_{bs}^{ij}}{(\partial r_m^i)^2} = \frac{3}{2}\alpha_{ij}\frac{(r_{ij}^2 - d_{ij,0}^2) + 2(r_m^j - r_m^i)^2}{||d_{ij,0}^2||}$$

$$\frac{\partial^2 U_{bs}^{ij}}{\partial r_m^i \partial r_n^i} = 3\alpha_{ij}\frac{(r_n^j - r_n^i)(r_m^j - r_m^i)}{||d_{ij,0}^2||} \text{ for } m \neq n$$

$$U_{bb}^{jik} = \frac{3}{8}\beta_{jik}\frac{(\Delta\theta_{jik})^2}{||d_{ij,0}||||d_{ik,0}||} = \frac{3}{8}\beta_{jik}\frac{(r_{ij}\cdot r_{ik} - d_{ij,0}\cdot d_{ik,0})^2}{||d_{ij,0}||||d_{ik,0}||}$$

$$r_{ij}\cdot r_{ik} = \sum_m (r_m^j - r_m^i)(r_m^k - r_m^i)$$

$$\frac{\partial U_{bb}^{jik}}{\partial r_n^j} = \frac{3}{8}\beta_{jik}\frac{2(r_{ij}\cdot r_{ik} - d_{ij,0}\cdot d_{ik,0})(r_n^k - r_n^i)}{||d_{ij,0}||||d_{ik,0}||} = \frac{3}{4}\beta_{jik}\frac{(r_{ij}\cdot r_{ik} - d_{ij,0}\cdot d_{ik,0})(r_n^k - r_n^i)}{||d_{ij,0}||||d_{ik,0}||}$$

$$\frac{\partial^2 U_{bb}^{jik}}{\partial r_m^j \partial r_m^i} = \frac{3}{4}\beta_{jik}\frac{d_{ij,0}\cdot d_{ik,0}-r_{ij}\cdot r_{ik}+(r_m^k-r_m^i)(2r_m^i-r_m^j-r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$

$$\frac{\partial^2 U_{bb}^{jik}}{\partial r_m^i \partial r_n^j} = \frac{3}{4}\beta_{jik}\frac{(r_n^k-r_n^i)(2r_m^i-r_m^j-r_m^k)}{||d_{ij,0}||||d_{ik,0}||} \text{ for } m \neq n$$

$$\frac{\partial^2 U_{bb}^{jik}}{\partial r_m^j \partial r_m^k} = \frac{3}{4}\beta_{jik}\frac{r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0}+(r_m^k-r_m^i)(r_m^j-r_m^i)}{||d_{ij,0}||||d_{ik,0}||}$$

$$\frac{\partial^2 U_{bb}^{jik}}{\partial r_m^j \partial r_n^k} = \frac{3}{4}\beta_{jik}\frac{(r_m^k-r_m^i)(r_n^j-r_n^i)}{||d_{ij,0}||||d_{ik,0}||} \text{ for } m \neq n$$

$$\frac{\partial^2 U_{bb}^{jik}}{(\partial r_m^j)^2} = \frac{3}{4}\beta_{jik}\frac{r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0}+(r_m^k-r_m^i)^2}{||d_{ij,0}||||d_{ik,0}||}$$

$$\frac{\partial^2 U_{bb}^{jik}}{\partial r_m^j \partial r_n^j} = \frac{3}{4}\beta_{jik}\frac{(r_m^k-r_m^i)(r_n^k-r_n^i)}{||d_{ij,0}||||d_{ik,0}||} \text{ for } m \neq n$$

$$\frac{\partial U_{bb}^{jik}}{\partial r_m^i} = \frac{3}{4}\beta_{jik}\frac{(r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0})(2r_m^i-r_m^j-r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$

$$\frac{\partial^2 U_{bb}^{jik}}{(\partial r_m^i)^2} = \frac{3}{4}\beta_{jik}\frac{2((r_{ij}\cdot r_{ik}-d_{ij,0}\cdot d_{ik,0})+(2r_m^i-r_m^j-r_m^k)^2}{||d_{ij,0}||||d_{ik,0}||}$$

$$\frac{\partial^2 U_{bb}^{jik}}{\partial r_m^i \partial r_n^i} = \frac{3}{4}\beta_{jik}\frac{(2r_n^i-r_n^j-r_n^k)(2r_m^i-r_m^j-r_m^k)}{||d_{ij,0}||||d_{ik,0}||} \text{ for } m \neq n$$

8.1.3 substituting evaluated derivatives

if $i \in nn(j)$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \frac{\partial^2}{\partial r_m^i \partial r_n^j}[U_{bs}^{ij} + \sum_{k\in nn(i)}^{k\neq j} U_{bb}^{jik} + \sum_{k\in nn(j)}^{k\neq i} U_{bb}^{ijk} + \sum_{k\in nn(i),nn(j)}^{k\neq i,j} U_{bb}^{ikj}]$$

$$\frac{\partial^2 U}{\partial r_m^i \partial r_m^j} = -\frac{3}{2}\alpha_{ij}\frac{(r_{ij}^2 - d_{ij,0}^2) + 2(r_m^j - r_m^i)(r_m^j - r_m^i)}{||d_{ij,0}^2||}$$
$$+ \sum_{k\in nn(i)}^{k\neq j} \frac{3}{4}\beta_{jik}\frac{d_{ij,0}\cdot d_{ik,0} - r_{ij}\cdot r_{ik} + (r_m^k - r_m^i)(2r_m^i - r_m^j - r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$
$$+ \sum_{k\in nn(j)}^{k\neq i} \frac{3}{4}\beta_{ijk}\frac{d_{ji,0}\cdot d_{jk,0} - r_{ji}\cdot r_{jk} + (r_m^k - r_m^j)(2r_m^j - r_m^i - r_m^k)}{||d_{ij,0}||||d_{jk,0}||}$$
$$+ \sum_{k\in nn(i),nn(j)}^{k\neq i,j} \frac{3}{4}\beta_{ikj}\frac{r_{ki}\cdot r_{kj} - d_{ki,0}\cdot d_{kj,0} + (r_m^j - r_m^k)(r_m^i - r_m^k)}{||d_{ki,0}||||d_{kj,0}||}$$

for $m \neq n$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = -3\alpha_{ij}\frac{(r_n^j - r_n^i)(r_m^j - r_m^i)}{||d_{ij,0}^2||}$$
$$+ \sum_{k\in nn(i)}^{k\neq j} \frac{3}{4}\beta_{jik}\frac{(r_n^k - r_n^i)(2r_m^i - r_m^j - r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$
$$+ \sum_{k\in nn(j)}^{k\neq i} \frac{3}{4}\beta_{ijk}\frac{(r_m^k - r_m^j)(2r_n^j - r_n^i - r_n^k)}{||d_{ij,0}||||d_{jk,0}||}$$
$$+ \sum_{k\in nn(i),nn(j)}^{k\neq i,j} \frac{3}{4}\beta_{ikj}\frac{(r_m^j - r_m^k)(r_n^i - r_n^k)}{||d_{kj,0}||||d_{ki,0}||}$$

if $j \notin nn(i) \cup i$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \frac{\partial^2}{\partial r_m^i \partial r_n^j} \sum_{k\in nn(i),nn(j)} U_{bb}^{ikj}$$

$$\frac{\partial^2 U}{\partial r_m^i \partial r_m^j} = \sum_{k\in nn(i),nn(j)}^{k\neq i,j} \frac{3}{4}\beta_{ikj}\frac{r_{ki}\cdot r_{kj} - d_{ki,0}\cdot d_{kj,0} + (r_m^j - r_m^k)(r_m^i - r_m^k)}{||d_{ki,0}||||d_{kj,0}||}$$

for $m \neq n$,

$$\frac{\partial^2 U}{\partial r_m^i \partial r_n^j} = \sum_{k\in nn(i),nn(j)}^{k\neq i,j} \frac{3}{4}\beta_{ikj}\frac{(r_m^j - r_m^k)(r_n^i - r_n^k)}{||d_{kj,0}||||d_{ki,0}||}$$

$$\frac{\partial^2 U}{(\partial r^i_m)^2} = \frac{\partial^2}{(\partial r^i_m)^2}[\sum_{j\in nn(i)} U^{ij}_{bs} + \frac{1}{2}\sum_{j,k\in nn(i)}^{k\neq j} U^{jik}_{bb} + \sum_{j\in nn(i)}(\sum_{k\in nn(j)}^{k\neq i} U^{ijk}_{bb})]$$

$$= \sum_{j\in nn(i)} \frac{3}{2}\alpha_{ij}\frac{(r^2_{ij} - d^2_{ij,0}) + 2(r^j_m - r^i_m)^2}{||d^2_{ij,0}||}$$

$$+ \sum_{j,k\in nn(i)}^{k\neq j} \frac{3}{8}\beta_{jik}\frac{2((r_{ij}\cdot r_{ik} - d_{ij,0}\cdot d_{ik,0}) + (2r^i_m - r^j_m - r^k_m)^2}{||d_{ij,0}||||d_{ik,0}||}$$

$$+ \sum_{j\in nn(i)}(\sum_{k\in nn(j)}^{k\neq i}\frac{3}{4}\beta_{ijk}\frac{r_{ji}\cdot r_{jk} - d_{ji,0}\cdot d_{jk,0} + (r^k_m - r^j_m)^2}{||d_{ji,0}||||d_{jk,0}||})$$

for $m \neq n$,

$$\frac{\partial^2 U}{\partial r^i_m \partial r^i_n} = \frac{\partial^2}{\partial r^i_m \partial r^i_n}[\sum_{j\in nn(i)} U^{ij}_{bs} + \frac{1}{2}\sum_{j,k\in nn(i)}^{k\neq j} U^{jik}_{bb} + \sum_{j\in nn(i)}(\sum_{k\in nn(j)}^{k\neq i} U^{ijk}_{bb})]$$

$$= \sum_{j\in nn(i)} 3\alpha_{ij}\frac{(r^j_n - r^i_n)(r^j_m - r^i_m)}{||d^2_{ij,0}||}$$

$$+ \sum_{j,k\in nn(i)}^{k\neq j} \frac{3}{8}\beta_{jik}\frac{(2r^i_n - r^j_n - r^k_n)(2r^i_m - r^j_m - r^k_m)}{||d_{ij,0}||||d_{ik,0}||}$$

$$+ \sum_{j\in nn(i)}(\sum_{k\in nn(j)}^{k\neq i}\frac{3}{4}\beta_{ijk}\frac{(r^k_m - r^j_m)(r^k_n - r^j_n)}{||d_{ji,0}||||d_{jk,0}||})]$$

8.2 Python code

```
#Modelling interactions for string of atoms
import csv
import numpy as np
from numpy.linalg import matrix_power
import matplotlib.pyplot as plt
from numpy import linalg as LA
from mpl_toolkits.axes_grid1.anchored_artists import AnchoredDirectionArrows
import os
from scipy.optimize import leastsq

class point:
    def __init__(self,x,y,z,ID):
        self.x = x
        self.y = y
        self.z = z
        self.v = np.array([x,y,z])
        self.ID = ID
        self.neighbours = []
        self.surfaceAtom = False

class lattice:


    @staticmethod
    def read(name):
        """
        reads a csv file and outputs a list of lists, holding data as strings
        """
        with open(name, newline='') as f:
            reader = csv.reader(f)
            return list(reader)


    @staticmethod
    def DeStringify(ListOfListOfStrings):
        """
        converts a list of list of strings to an array of floats
        """
        FloatArray= np.zeros([len(ListOfListOfStrings),len(ListOfListOfStrings[0])])
        for i in range(len(ListOfListOfStrings)):
            for j in range(len(ListOfListOfStrings[0])):
                FloatArray[i,j] = float(ListOfListOfStrings[i][j])
        return(FloatArray)

    @staticmethod
    def Indexer(InputArray):
        """
        converts an array, formatted as a set of ID's corresponding to a value in
        each column, to an array, with the number of dimensions matching the
        number of ID's
        """
        dim = len(InputArray[0]) - 1
        OutputArray = np.zeros(dim*[(1+int(np.max(InputArray[:,0:dim])))])

        if dim == 1:
            for element in InputArray:
                OutputArray[int(element[0])] = element[1]
        elif dim == 2:
            for element in InputArray:
                OutputArray[int(element[0]),int(element[1])] = element[2]
                OutputArray[int(element[1]),int(element[0])] = element[2]
        else:
            for element in InputArray:
                OutputArray[int(element[0]),int(element[1])
                            ,int(element[2])] = element[3]
                OutputArray[int(element[2]),int(element[1])
                            ,int(element[0])] = element[3]
        return(OutputArray)

    def ChainData(length,ID):
        """
        Produce data file for a chain of atoms
        """
        data = [[x,0,0,ID] for x in range(length)]
        with open('nChainData.csv', 'w', newline = '') as f:
            # create the csv writer
            writer = csv.writer(f)
        # write a row to the csv file
        writer.writerows(data)

def __init__(self,PointDataFileName,MassDataFileName,StretchDataFileName
             ,LengthDataFileName,BendDataFileName,AngleDataFileName,
             BoundaryConditions = None):
    """

    PointData file is formatted as first row x, second row y, third row z,
    last row ID: assign each to a lattice point object. ID is atomic number-1

    MassIndex: row of masses

    BondStretchData: Indexes bond stretching factor of BS(IJ).
    Format is ID I, ID J, length.

    IdealLengthData: Stores ideal bond length of BS(IJ).
    Format is ID I, ID J, length.

    IdealAngleData: Stores ideal angle of BB(JIK).
    Format is ID J, ID I, ID K, angle.

    BondBendingData: Stores bond bending factor of BB(JIK).
    Format is ID J, ID I, ID K, angle.

    Point Data is converted into point objects

    All other data types are converted to arrays of floats, and then
    an array of dimension equal to the number of ID's it has stores
    the data
    """

    data = self.read(PointDataFileName)

    self.points = [point(float(data[i][0]),float(data[i][1]),float(data[i][2]),
                         int(data[i][3])) for i in range(len(data))]
    self.DynamicalMatrix = np.zeros((3*len(self.points),3*len(self.points)))


    data = self.read(MassDataFileName)
    self.MassIndex= self.Indexer(self.DeStringify(data))

    data = self.read(LengthDataFileName)
    self.IdealLengthIndex= self.Indexer(self.DeStringify(data))

    data = self.read(StretchDataFileName)
    self.BondStretchingIndex= self.Indexer(self.DeStringify(data))

    data = self.read(BendDataFileName)
    self.BondBendingIndex= self.Indexer(self.DeStringify(data))
```

```
        data = self.read(AngleDataFileName)
        self.IdealAngleIndex= self.Indexer(self.DeStringify(data))

        self.BoundaryConditions = BoundaryConditions

    def neighbourscalc(self, bondlength):
        for PointToCheck in self.points:
            for neighbour in set(self.points) - {PointToCheck}: #check against
                #every other lattice point
                if self.DistanceObj(PointToCheck,neighbour,None)<=bondlength:
                    PointToCheck.neighbours.append(neighbour)

    def DistV(self, Point1, Point2,Axis = None):
        """
        calculates the vector between 2 points, as vectors
        """

        if self.BoundaryConditions is None:
            if Axis == None:
                return(Point2 - Point1)
            else:
                return(Point2[Axis]-Point1[Axis])
        else:
            if Axis == None:
                return([min(Point2[i]-Point1[i],
                            Point2[i]-Point1[i]-self.BoundaryConditions[i,1]
                            +self.BoundaryConditions[i,0],
                            Point2[i]-Point1[i]-self.BoundaryConditions[i,0]
                            +self.BoundaryConditions[i,1],
                    key = abs) for i in range(3)])
            else:
                return(min(Point2[Axis]-Point1[Axis]
                    ,Point2[Axis]-Point1[Axis]-self.BoundaryConditions[Axis,1]
                    +self.BoundaryConditions[Axis,0],
                    Point2[Axis]-Point1[Axis]-self.BoundaryConditions[Axis,0]
                    +self.BoundaryConditions[Axis,1],
                    key = abs))

    def Distance(self, Point1, Point2, Axis = None):
        """
        gets the Distance between 2 lattice points
        Point1 and Point2 are 2 lattice points / their indexes in the lattice
        object's list of points
        AreObjects is a boolean for whether the points are objects or their
        indexes(default)
        Axis defines whether the Distance is taken along a specific axis
        """

        return(np.linalg.norm(self.DistV(self.points[Point1].v ,
                                         self.points[Point2].v,Axis)))
```

```
def DistanceObj(self, Point1, Point2, Axis = None):
    """
    gets the Distance between 2 lattice points
    Point1 and Point2 are 2 lattice points / their indexes in the lattice
    object's list of points
    AreObjects is a boolean for whether the points are objects or their
    indexes(default)
    Axis defines whether the Distance is taken along a specific axis
    """

    return(np.linalg.norm(self.DistV(Point1.v,Point2.v,Axis)))


def innerObj(self,j,i,k):
    """
    Returns r(ij).r(ik)
    """
    return(np.inner(self.DistV(i.v,j.v)),self.DistV(i.v,k.v))

def inner(self,j,i,k, AreObjects = False):

    """
    Returns r(ij).r(ik)
    """

    return np.inner(self.DistV(
            self.points[i].v,self.points[j].v),
            self.DistV(self.points[i].v,self.points[k].v))

def matcalc(self):

    MassMatrix = lambda arg1: np.sqrt(arg1)*np.identity(3)

    #anon fn to generate mass matrix for given input mass

    for i in range(len(self.points)):
        pointI = self.points[i]
        for m in range(3):
                for n in range(3):
                    if m == n:

                        """
                        partial^2 U
                        /(partial r^i_m)^2
                        """
                        for pointJ in pointI.neighbours:
                            #BS term
                            j = self.points.index(pointJ)
                            self.DynamicalMatrix[3*i+m,3*i+n]+=3/2 *(
                                self.BondStretchingIndex[pointI.ID,pointJ.ID]
                                * (self.Distance(i,j)**2
                                - self.IdealLengthIndex[pointI.ID,pointJ.ID]**2
                                + 2 * self.Distance(i,j,m)**2
                                ) / (self.IdealLengthIndex[pointI.ID,pointJ.ID])**2)

                            #BB term 1
                            for pointK in set(pointI.neighbours) - {pointJ}:

                                k = self.points.index(pointK)
                                self.DynamicalMatrix[3*i+m,3*i+n]+=3/8*(
                                    self.BondBendingIndex[pointJ.ID,pointI.ID
                                                          ,pointK.ID]*
                                    (-2*(np.cos(np.pi/180*self.IdealAngleIndex[
                                            pointJ.ID,pointI.ID,pointK.ID]))
                                    +(2*self.inner(j,i,k)
                                    +(self.Distance(j,i,m)+self.Distance(k,i,m))**2)
                                    /self.IdealLengthIndex[pointI.ID,pointJ.ID]
                                    /self.IdealLengthIndex[pointI.ID,pointK.ID]))

                            #BB term 2
                            for pointK in set(pointJ.neighbours) - {pointI}:
                                k = self.points.index(pointK)
                                self.DynamicalMatrix[3*i+m,3*i+n]+=3/4*(
                                    self.BondBendingIndex[pointI.ID,pointJ.ID
                                                          ,pointK.ID]*
                                    (-1*(np.cos(np.pi/180*(self.IdealAngleIndex[
                                            pointI.ID,pointJ.ID,pointK.ID])))
                                    +(self.inner(i,j,k)
                                    +self.Distance(j,k,m)**2)
                                    /self.IdealLengthIndex[pointJ.ID,pointI.ID]
                                    /self.IdealLengthIndex[pointJ.ID,pointK.ID]))


                    else:
                        """
                        partial^2 U
                        /(partial r^i_m)(\partial r^i_n)
                        m=/=n
                        """
                        for neighbour in self.points[i].neighbours:
                            j = self.points.index(neighbour)

                            self.DynamicalMatrix[3*i+m,3*i+n]+=3*(
                                self.BondStretchingIndex[pointI.ID,pointJ.ID]
                                * self.Distance(i,j,m)
                                * self.Distance(i,j,n)
                                / self.IdealLengthIndex[pointI.ID,pointJ.ID]**2)

                            #BB term 1
                            for pointK in set(self.points[i].neighbours) - {pointJ}:
                                k = self.points.index(pointK)
                                self.DynamicalMatrix[3*i+m,3*i+n]+=3/8*(
                                                self.BondBendingIndex[pointJ.ID,pointI.ID,
                                                                      pointK.ID]
                                                *(self.Distance(j,i,n)+self.Distance(k,i,n))
                                                *(self.Distance(j,i,m)+self.Distance(k,i,m))
                                                /self.IdealLengthIndex[pointI.ID,pointJ.ID]
                                                /self.IdealLengthIndex[pointI.ID,pointK.ID])

                                        #BB term 2
                                        for pointK in set(self.points[j].neighbours) - {
                                                pointI}:
                                            k = self.points.index(pointK)
                                            self.DynamicalMatrix[3*i+m,3*i+n]+=3/4*(
                                                self.BondBendingIndex[pointI.ID,pointJ.ID,
                                                                      pointK.ID]
                                                *self.Distance(j,k,m)
                                                *self.Distance(j,k,n)
                                                /self.IdealLengthIndex[pointJ.ID,pointI.ID]
                                                /self.IdealLengthIndex[pointJ.ID,pointK.ID])

                    #incororate mass
                    self.DynamicalMatrix[3*i:(3*i+3),3*i:(3*i+3)] = (
                                np.dot(np.dot(matrix_power(
                                MassMatrix(self.MassIndex[self.points[i].ID]),-1),
                                self.DynamicalMatrix[3*i:(3*i+3),3*i:(3*i+3)]),
                                matrix_power(MassMatrix(self.MassIndex[
                                        self.points[i].ID]),-1)))

                    for pointJ in pointI.neighbours:
                        """
                        nearest neighbour interaction terms
                        """
                        j = self.points.index(pointJ)

                        #if already calculated symmetrically, reassign.
                        if np.any(self.DynamicalMatrix[3*i:(3*i+3),3*j:(3*j+3)]):
                            self.DynamicalMatrix[3*i:3*i+3,3*j:3*j+3] = np.transpose(
                                    self.DynamicalMatrix[3*j:(3*j+3),3*i:(3*i+3)] )
                        else:
                            #looping for each dimension permutation:
                            for m in range(3):
                                for n in range(3):
                                    if m == n:
                                        #calculation for same dimension; will add on each term
                                        #for readability and ease of looping
                                        """
                                        partial^2 U
                                        /(partial r^i_m)(\partial r^j_m)
                                        j in nn(i)
                                        """

                                        #Bond Stretching
                            self.DynamicalMatrix[3*i+m,3*j+n] += -3/2 * (
                                self.BondStretchingIndex[pointI.ID,pointJ.ID]
                                * (self.Distance(i,j)**2 -
                                (self.IdealLengthIndex[pointI.ID,pointJ.ID])**2
                                + 2 * self.Distance(i,j,m)**2)
                                / (self.IdealLengthIndex[pointI.ID,pointJ.ID])**2)

                            #Bond Bending #1
                            for pointK in set(pointI.neighbours) - {pointJ}:
                                k = self.points.index(pointK)
                                self.DynamicalMatrix[3*i+m,3*j+n] += 3/4* (
                                    self.BondBendingIndex[pointJ.ID,
                                                          pointI.ID,pointK.ID]
                                    *(np.cos(np.pi/180*self.IdealAngleIndex[
                                            pointJ.ID,pointI.ID,pointK.ID])
                                    +(-self.inner(j,i,k)-self.Distance(i,k,m)
                                      *(self.Distance(i,k,m)+self.Distance(i,j,m)))
                                    /self.IdealLengthIndex[pointI.ID,pointJ.ID]
                                    /self.IdealLengthIndex[pointI.ID,pointK.ID]))

                            #Bond Bending #2
                            for pointK in set(pointJ.neighbours) - {pointI}:
                                k = self.points.index(pointK)
                                self.DynamicalMatrix[3*i+m,3*j+n] += 3/4* (
                                    self.BondBendingIndex[pointI.ID,
                                                          pointJ.ID,pointK.ID]
                                    *(np.cos(np.pi/180*self.IdealAngleIndex[
                                            pointI.ID,pointJ.ID,pointK.ID])
                                    +(-self.inner(i,j,k)-self.Distance(j,k,m)
                                      *(self.Distance(j,k,m)+self.Distance(j,i,m)))
                                    /self.IdealLengthIndex[pointJ.ID,pointI.ID]
                                    /self.IdealLengthIndex[pointJ.ID,pointK.ID]))

                            #Bond bending #3
                            for pointK in set(pointI.neighbours
                                              ).intersection(set(pointJ.neighbours)):
                                k = self.points.index(pointK)
                                self.DynamicalMatrix[3*i+m,3*j+n] += 3/4*(
                                    self.BondBendingIndex[pointI.ID,
                                                          pointK.ID,pointJ.ID]
                                    *(-np.cos(np.pi/180*self.IdealAngleIndex[
                                            pointI.ID,pointK.ID,pointJ.ID])
                                    +(self.inner(i,j,k)+self.Distance(k,i,m)
                                      *self.Distance(k,j,m))
                                    /self.IdealLengthIndex[pointK.ID,pointI.ID]
                                    /self.IdealLengthIndex[pointK.ID,pointJ.ID]))
                        else:
                            #calculation for different dimension
                            """
                            partial^2 U
                            /(partial r^i_m)(\partial r^j_n)
```

```
                j in nn(i)
                m=/=n
                """
                self.DynamicalMatrix[3*i+m,3*j+n] += -3 * (
                    self.BondStretchingIndex[pointI.ID, pointJ.ID]
                    * self.Distance(i,j,m)
                    * self.Distance(i,j,n)
                    / (self.IdealLengthIndex[pointI.ID,pointJ.ID])**2)

                #Bond Bending #1
                for pointK in set(pointI.neighbours) - {pointJ}:
                    k = self.points.index(pointK)
                    self.DynamicalMatrix[3*i+m,3*j+n] += -3/4* (
                        self.BondBendingIndex[pointJ.ID,
                                              pointI.ID,pointK.ID]
                        *self.Distance(i,k,n)
                        *(self.Distance(i,k,m)+self.Distance(i,j,m))
                        /self.IdealLengthIndex[pointI.ID,pointJ.ID]
                        /self.IdealLengthIndex[pointI.ID,pointK.ID])

                #Bond Bending #2
                for pointK in set(pointJ.neighbours) - {pointI}:
                    k = self.points.index(pointK)
                    self.DynamicalMatrix[3*i+m,3*j+n] += -3/4* (
                        self.BondBendingIndex[pointI.ID,
                                              pointJ.ID,pointK.ID]
                        *self.Distance(j,k,m)
                        *(self.Distance(j,k,n)+self.Distance(j,i,n))
                        /self.IdealLengthIndex[pointJ.ID,pointI.ID]
                        /self.IdealLengthIndex[pointJ.ID,pointK.ID])

                #Bond bending #3
                for pointK in set(pointI.neighbours).intersection(
                        set(pointJ.neighbours)):
                    k = self.points.index(pointK)
                    self.DynamicalMatrix[3*i+m,3*j+n] += 3/4*(
                        self.BondBendingIndex[pointI.ID,pointK.ID,
                                              pointJ.ID]
                        *self.Distance(k,i,n)
                        *self.Distance(k,j,m)
                        /self.IdealLengthIndex[pointK.ID,pointI.ID]
                        /self.IdealLengthIndex[pointK.ID,pointJ.ID])

        #include mass in dynamical matrix
        self.DynamicalMatrix[3*i:(3*i+3),3*j:(3*j+3)
            ] = np.dot(np.dot(matrix_power(MassMatrix(
        self.MassIndex[self.points[i].ID]),-1),
            self.DynamicalMatrix[3*i:(3*i+3),
            3*j:(3*j+3)]),
            matrix_power(MassMatrix(self.MassIndex[self.points[j].ID]),-1))
```

```
"""
partial^2 U
/(partial r^i_m)(\partial r^j_m)
m not in nn(i) U i
"""

#work out set of points that are not nearest neighbours or i,
#but neighbours of nearest neighbours of i
pointJset = set([])
for pointK in pointI.neighbours:
    for pointJ in set(pointK.neighbours)- {pointI
                     } - set(pointI.neighbours):
        pointJset.add(pointJ)
for pointJ in pointJset:
    j = self.points.index(pointJ)
    #if already calculated symmetrically, reassign.
    if np.any(self.DynamicalMatrix[3*i:(3*i+3),3*j:(3*j+3)]):
        self.DynamicalMatrix[3*i:3*i+3,3*j:3*j+3] = np.transpose(
                self.DynamicalMatrix[3*j:(3*j+3),3*i:(3*i+3)] )
    else:
        for pointK in set(pointJ.neighbours).intersection(
                set(pointI.neighbours)):
            k= self.points.index(pointK)
            for m in range(3):
                    for n in range(3):
                        if m == n:
                            self.DynamicalMatrix[3*i+m,3*j+n] += 3/4*(
                                self.BondBendingIndex[pointI.ID,
                                                      pointK.ID,pointJ.ID]
                                *(-(np.cos(np.pi/180*self.IdealAngleIndex[
                                        pointI.ID,pointK.ID,pointJ.ID]))
                                +(self.inner(i,k,j)
                                + self.Distance(k,j,m)*self.Distance(k,i,m)
                                )/self.IdealLengthIndex[pointK.ID,
                                pointI.ID]
                                /self.IdealLengthIndex[pointK.ID,
                                                       pointJ.ID]))

                        else:
                            self.DynamicalMatrix[3*i+m,3*j+n] += 3/4*(
                                self.BondBendingIndex[pointI.ID,
                                                      pointK.ID,
                                                      pointJ.ID]
                                *self.Distance(k,j,m)*self.Distance(
                                        k,i,n)
                                /self.IdealLengthIndex[pointK.ID,
                                                       pointI.ID]
                                /self.IdealLengthIndex[pointK.ID,
                                                       pointJ.ID])
                    #include mass in dynamical matrix
                    self.DynamicalMatrix[3*i:(3*i+3),3*j:(3*j+3)
                        ] = np.dot(np.dot(matrix_power(MassMatrix(
                            self.MassIndex[self.points[i].ID]),-1),
                        self.DynamicalMatrix[3*i:(3*i+3),
                        3*j:(3*j+3)]),
                        matrix_power(MassMatrix(self.MassIndex[
                                self.points[j].ID]),-1))

    def EigSolve(self):
        eigenvals, vectors = LA.eig(self.DynamicalMatrix)
#         self.eigenvals = np.around(np.absolute(eigenvals),3)
        self.eigenvals = eigenvals
        self.w = np.sqrt(np.real(self.eigenvals))
#         self.vectors = np.around(np.absolute((vectors)),3)
        self.vectors = vectors

    def EigPlot2D(self,ax,ValToPlot,param_dict,ArrowWidth = 0.1):
        #
        """
        graph for eigenvectors which are ALL orthogonal to one axis

        Parameters
        ----------
        ax : Axes
            The axes to draw to

        ValtoPlot: eigenvalue/eigenvectors to plot

        param_dict : dict
           Dictionary of kwargs to pass to ax.plot

        Returns
        -------
        out : list
            list of artists added
        """
        CoordDict = {0: 'x', 1: 'y', 2: 'z'}
        OrthAxis = None

        #decide on orthogonal axis based on which has zero vals

        for i in range(3):
            if np.any([self.vectors[i + 3*x,ValToPlot] for x in range(
                    len(self.points))])== False:
                OrthAxis = i

        if OrthAxis == None:
            raise TypeError("Eigenvectors are not orthogonal to x, y or z.")
            return
    Axes = list({0,1,2} - {OrthAxis})

    out = []
    #plot arrows then points on top
    for i in range(len(self.points)):
        pointI = self.points[i]
        #only plot arrow if there is non-zero eigenvector in the plane
        if np.any([self.vectors[[3*i + Axes[0],3*i + Axes[1]],
                                ValToPlot]])== True:
            out.append(plt.arrow(pointI.v[Axes[0]],
                       pointI.v[Axes[1]],
                       self.vectors[3*i + Axes[0],ValToPlot]/4,
                       self.vectors[3*i + Axes[1],ValToPlot]/4,
                       width = ArrowWidth))

        out.append(plt.plot(pointI.v[Axes[0]], pointI.v[Axes[1]],
                 **param_dict))

    simple_arrow = AnchoredDirectionArrows(ax.transAxes,
                                           CoordDict[Axes[0]],
                                           CoordDict[Axes[1]],
                                           color = 'black')

    out.append(ax.add_artist(simple_arrow))

    textXLoc = (max([chain.points[a].v[0] for a in range(len(chain.points))
    ])) * 0.8
    out.append(plt.text(textXLoc,0.4,'w^2 =' + str(np.round(
            self.eigenvals[ValToPlot],2))))

    return out

def EigPlotPhonon(self,ax,ValToPlot,param_dict,ArrowWidth = 0.1):
    #
    """
    graph for visualising phonons in single dimension

    Parameters
    ----------
    ax : Axes
        The axes to draw to

    ValtoPlot: eigenvalue/eigenvectors to plot

    param_dict : dict
       Dictionary of kwargs to pass to ax.plot

    Returns
    -------
    out : list
        list of artists added
    """
    #CoordDict = {0: 'x', 1: 'y', 2: 'z'}
    Axes = [0,1,2]

    #decide on orthogonal axis based on which has zero vals

    for i in range(3):
        if np.any([self.vectors[i + 3*x,ValToPlot] for x in range(
                len(self.points))])== False:
            Axes.remove(i)

    if Axes[0] == None:
        raise TypeError("Eigenvectors are multidimensional.")
        return

    out = []
    #plot arrows then points on top

    #only plot arrow if there is non-zero eigenvector in the plane
    plt.plot([point.v[Axes[0]] for point in self.points],
                  [self.vectors[3*i + Axes[0],ValToPlot
                                ] for i in range(len(self.points))])


    textXLoc = (max([chain.points[a].v[0] for a in range(len(chain.points))
    ])) * 0.8
    out.append(plt.text(textXLoc,0.4,'w^2 =' + str(
            np.round(self.eigenvals[ValToPlot],2))))

    return out


def PhononBasisX(self):
    '''
    creates PhononBasis attribute and Kx to go along with it
    assumes chain in x
    '''
    a = 1
    if self.BoundaryConditions == None:
        R = max([self.points[i].v[0] for i in range(len(self.points))]) - \
            min([self.points[i].v[0] for i in range(len(self.points))]) + a
    else:
        R = self.BoundaryConditions[0, 1]-self.BoundaryConditions[0, 0]
    nMax = int(R / 2 / a)
    self.Kx = [2 * np.pi * n / R for n in range(-nMax, nMax)]
    self.PhononBasis = np.zeros([2*nMax,len(self.vectors)],dtype = np.complex64)

    for n in range(2*nMax):
        for PointIndex in range(len(self.points)):
            self.PhononBasis[n,3*PointIndex] = np.exp(1j * self.Kx[n]
                *self.points[PointIndex].v[0])
            #normalise
            self.PhononBasis[n] = self.PhononBasis[n] / LA.norm(self.PhononBasis[n])

def EigenProjection(self):
    '''
    project eigenvectors onto basis
    '''
    self.projections = np.zeros([len(self.vectors),len(self.PhononBasis)])
    for index in range(len(self.vectors)):
        self.projections[index] = np.round(np.power
                             (np.abs(np.inner(self.PhononBasis,
                                     self.vectors[:,index])),2),3)

def kDecomposition(self, ax, index):
    '''
    create figure of intensity of eigenstate against k-value.

    Parameters
    ----------
    ax : Axes
        The axes to draw to

    index: index of eigenvalue to plot

    param_dict : dict
       Dictionary of kwargs to pass to ax.plot

    Returns
    -------
    out : list
        list of artists added
    '''
    out = [plt.scatter(self.Kx,self.projections[index])]
    out.append(plt.title('phonon '+ str(index)+', w = ' +str(
            np.round(self.w[index],2))))
    out.append(plt.xlabel('k'))
    out.append(plt.ylabel('I'))
    return out

if __name__ == "__main__":

chainlength = 20
lattice.ChainData(chainlength,12)
BC = np.array([[0,chainlength],[-10,10],[-10,10]])
BC = None
chain = lattice('nChainData.csv','MassIndex.csv',
                      'BondStretchingData.csv','BondLengthData.csv',
                      'BondBendingData.csv','IdealAngleData.csv',
                      BC)
chain.neighbourscalc(1)
chain.matcalc()
chain.EigSolve()

#define directory for storing plots
#first, get directory
real_path = os.path.realpath(__file__)
dir_path = os.path.dirname(real_path)
if chain.BoundaryConditions is None:
chaintype = 'chainFin'
else:
chaintype = 'chainPeriodic'

#create directory
ImgDir = dir_path +  '\\' + str(chainlength) + chaintype + '\\'
if os.path.exists(ImgDir) == False:
os.mkdir(ImgDir)


#store dynamical matrix to file
mat_path = ImgDir+'\\'+"DynamicalMatrix"+ '.csv'

if os.path.exists(mat_path) == False:
    open(mat_path,'x')

with open(mat_path, 'w',newline = '') as f:
# create the csv writer
writer = csv.writer(f)

# write a row to the csv file
writer.writerows(np.ndarray.tolist(chain.DynamicalMatrix))


chain.PhononBasisX()
chain.EigenProjection()
"""
plotting data
"""
fig, ax = plt.subplots(1, 1)
for i in range(len(chain.projections)):
for k in range(len(chain.projections[0])):
    if chain.projections[i,k] >0.4:
 #       plt.scatter(chain.Kx[k],chain.eigenvals[i],color = 'b')
        plt.scatter(chain.Kx[k],abs(chain.w[i]),color = 'b')
plt.xlabel('k')
plt.ylabel('w')
plt.title(str(chainlength)+' atom chain, periodic boundary conditions')
graph_path_phonon = ImgDir+'\\'+'phonons.svg'
if os.path.exists(graph_path_phonon) == False:
open(graph_path_phonon,'x')
plt.savefig(graph_path_phonon)
plt.close()
```

```
#eigenstate decomposition
fig, ax = plt.subplots(1,1)
for i in range(chainlength):
    fig, ax = plt.subplots(1, 1)
    chain.kDecomposition(ax,i)
    graph_path_eigenstateDecomp = ImgDir+'\\'+str(i)+ 'eigenstateDecomp.svg'
    if os.path.exists(graph_path_eigenstateDecomp) == False:
        open(graph_path_eigenstateDecomp,'x')
    plt.savefig(graph_path_eigenstateDecomp)
    plt.close()

plt.close()
#motion plot
for i in range(3*chainlength):
    fig, ax = plt.subplots(1, 1)
    chain.EigPlot2D(ax,i,{'marker': 'o','color' : 'r'},0.02)
    plt.ylim([-0.5,0.5])
    plt.xlim([-0.5,max([chain.points[a].v[0] for a in range(
        #len(chain.points))])+0.5])
    graph_path_motion = ImgDir+'\\'+str(i)+ 'motion.svg'
    if os.path.exists(graph_path_motion) == False:
        open(graph_path_motion,'x')
    plt.savefig(graph_path_motion)
    plt.close()

#eigenstate plot
for i in range(3*chainlength):
    fig, ax = plt.subplots(1, 1)
    chain.EigPlotPhonon(ax,i,{'marker': 'o','color' : 'r'},0.02)
    plt.ylim([-1,1])
    plt.xlim([-0.5,max([chain.points[a].v[0] for a in range(
            len(chain.points))])+0.5])
    graph_path_phonon = ImgDir+'\\'+str(i)+ 'phonon.svg'
    if os.path.exists(graph_path_phonon) == False:
        open(graph_path_phonon,'x')
    plt.savefig(graph_path_phonon)
    plt.close()
```

9 References

References

1. Yu, P. Y. & Cardona, M. *Fundamentals of Semiconductors: Physics and Materials Properties* ISBN: 1868-4513 (Berlin, Heidelberg: Springer Berlin Heidelberg, Berlin, Heidelberg).
2. Simon, S. H. *The Oxford solid state basics / Steven H. Simon* (Oxford : OUP, Oxford, 2013).
3. Liu, X. & Hersam, M. C. 2D materials for quantum information science. *Nature Reviews. Materials* **4,** 669–684. `https://www.proquest.com/scholarly-journals/2d-materials-quantum-information-science/docview/2300955815/se-2?accountid=16064%20http://hw-primo.hosted.exlibrisgroup.com/openurl/44HWA/44HWA_SP??url_ver=Z39.88-2004&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&genre=article&sid=ProQ:ProQ%3Amaterialsscijournals&atitle=2D+materials+for+quantum+information+science&title=Nature+Reviews.+Materials&issn=&date=2019-10-01&volume=4&issue=10&spage=669&au=Liu%2C+Xiaolong%3BHersam%2C+Mark+C&isbn=&jtitle=Nature+Reviews.+Materials&btitle=&rft_id=info:eric/&rft_id=info:doi/10.1038%2Fs41578-019-0136-x` (2019).
4. Geim, A. K. & Grigorieva, I. V. Van der Waals heterostructures. *Nature* **499,** 419–425. ISSN: 1476-4687. `https://doi.org/10.1038/nature12385` (2013).
5. Buin, A. K., Verma, A. & Anantram, M. P. Carrier-phonon interaction in small cross-sectional silicon nanowires. *Journal of Applied Physics* **104,** 053716-053716–9. ISSN: 0021-8979 (2008).
6. Yu, C.-J. *et al.* Spin and Phonon Design in Modular Arrays of Molecular Qubits. *Chemistry of materials* **32,** 10200–10206. ISSN: 0897-4756 (2020).
7. Fu, H., Ozoliņš, V. & Zunger, A. Phonons in GaP quantum dots. *Physical Review B* **59,** 2881–2887. `https://link.aps.org/doi/10.1103/PhysRevB.59.2881` (1999).
8. Perebeinos, V. & Tersoff, J. Valence force model for phonons in graphene and carbon nanotubes. *Physical review. B, Condensed matter and materials physics* **79.** ISSN: 1098-0121 (2009).
9. Keating, P. N. Effect of invariance requirements on the elastic strain energy of crystals with application to the diamond structure. *Physical review* **145,** 637–645. ISSN: 0031-899X (1966).
10. Paul, A., Luisier, M. & Klimeck, G. Modified valence force field approach for phonon dispersion: from zinc-blende bulk to nanowires. *Journal of Computational Electronics* (2010).