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

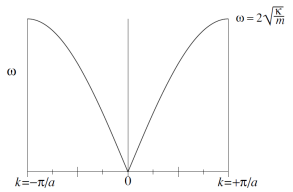
Practical quantum technologies rely on robust hardware. For many applications, semiconductor quantum nanostructures are ideal candidates, e.g. as physical implementations of quantum bits, or as emitters of non-classical light. However, phonons, i.e. vibrations in the underlying crystal structure, severely 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):

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

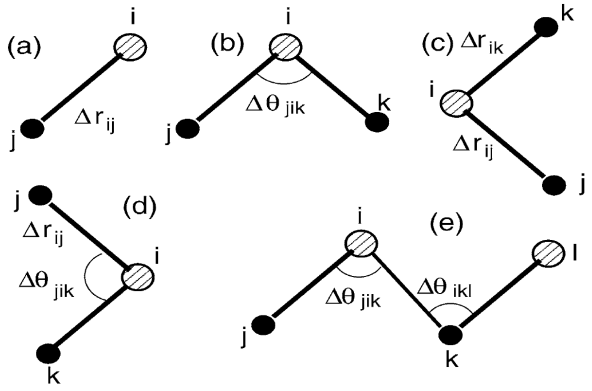


Figure 2: Bond bending types [10]

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. $(\Delta\theta_{jik} = (r_{ij}\cdot r_{ik} - d_{ij,0}\cdot d_{ik,0}))$.

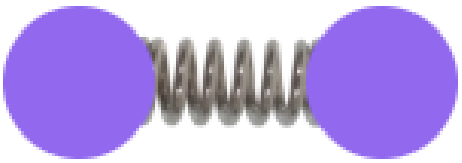
Prior to computational calculation, this potential has to be processed to give a form that can be described as a eigenvalue problem. This means calculating what is called the Dynamical Matrix:

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

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

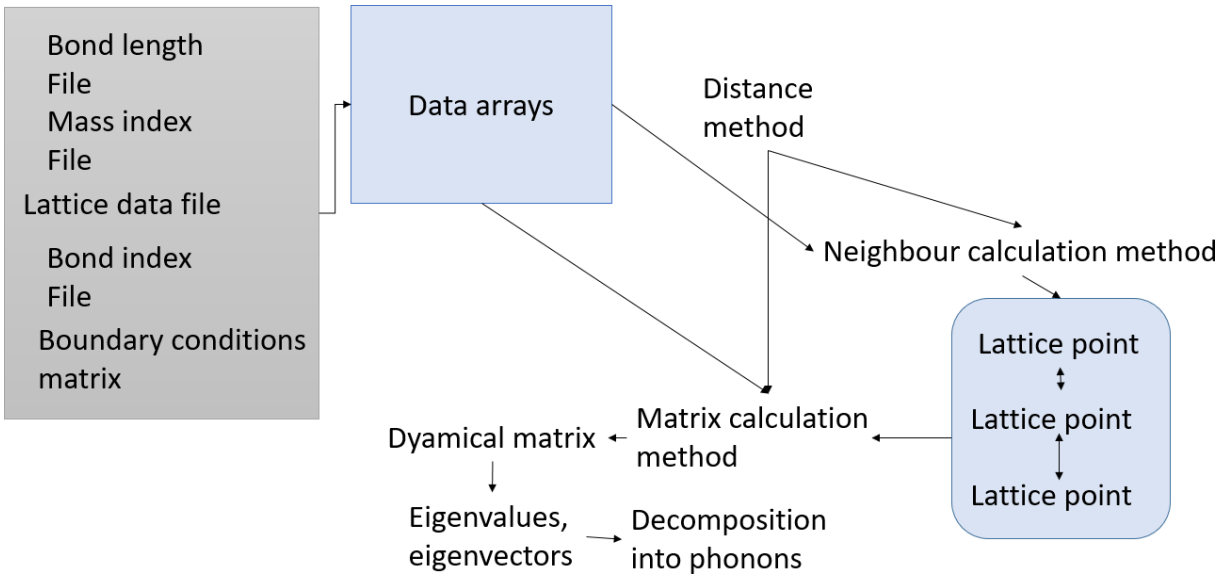


Figure 4: Program flowchart structure

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:

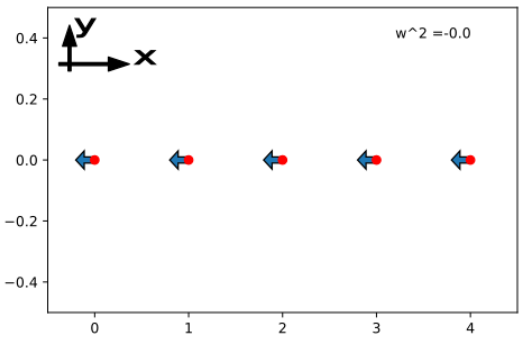


Figure 5: zero energy phonon for full translation

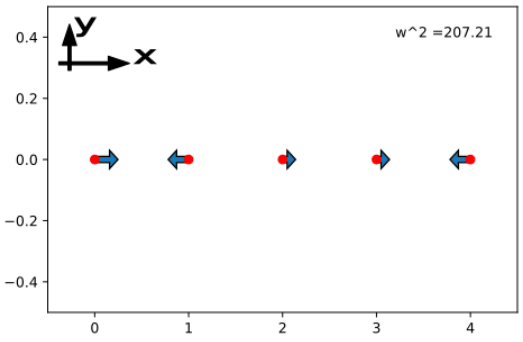


Figure 6: zero energy phonon for single orthogonal translation

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.

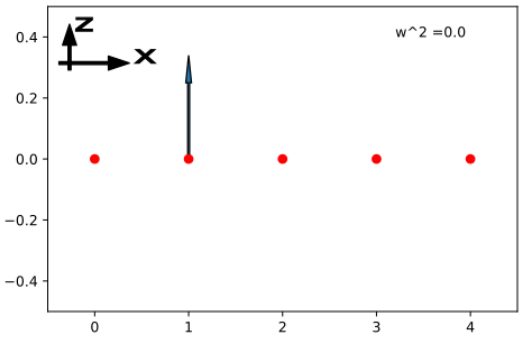


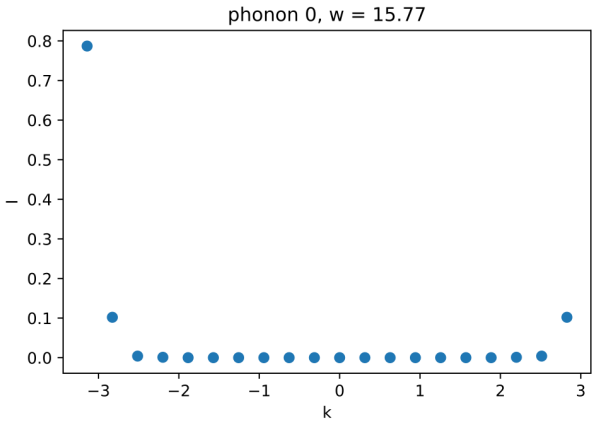
Figure 7: non-zero energy phonon

6.2 Phonon Distribution for Open and Closed Boundary Conditions

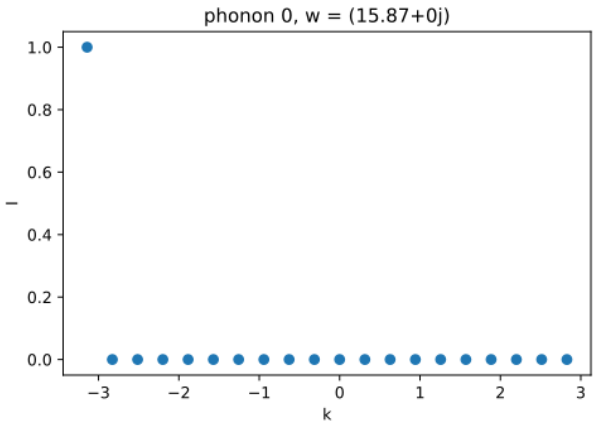
As previously stated, it is expected that for a linear chain that phonon eigenstates' spectral density should project directly onto bloch states, with some spectral broadening for non-periodic boundary conditions. For a 20 atom linear chain, some of these are compared in figures 8, 9 and 10. The expected 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

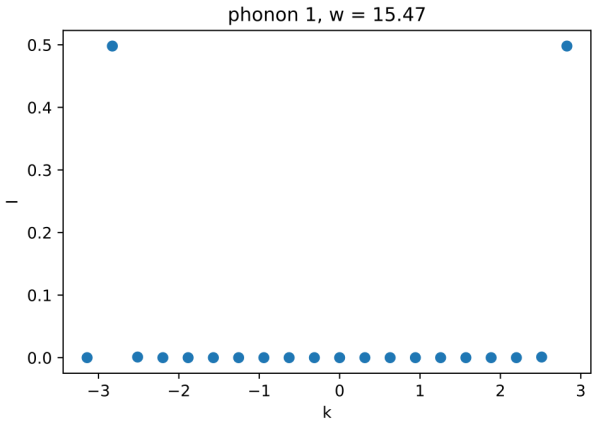


(b) periodic decomposition of an eigenstate

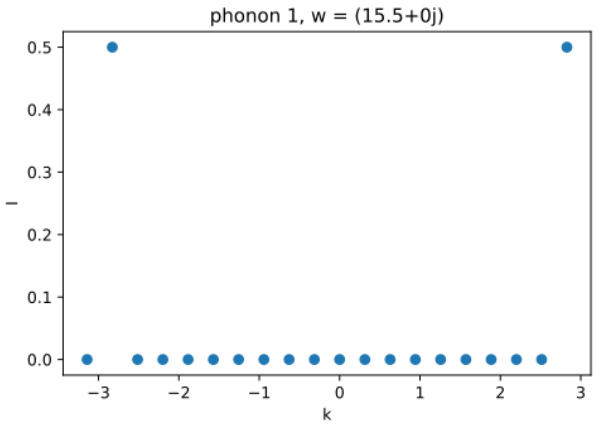
Figure 8: spectral densities of an eigenstate for a chain length of 20

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

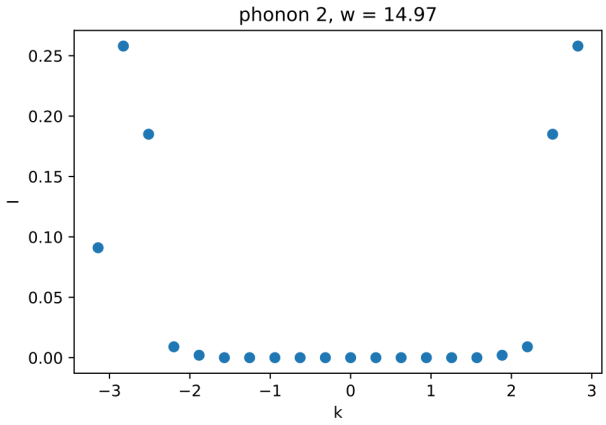
$U_{bb}^{ij} = U_{bb}^{ji}, U_{bs}^{ijk} = U_{bs}^{kji}$

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

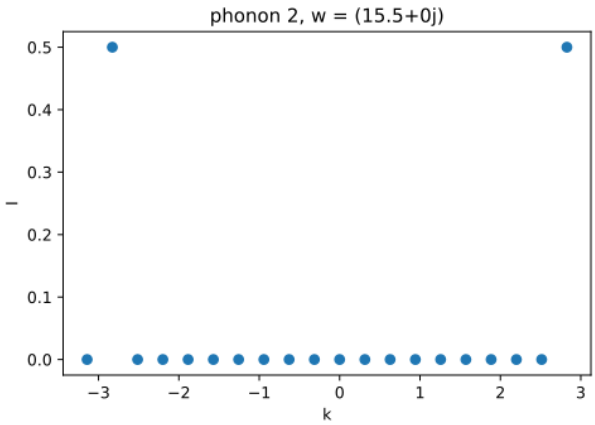
$$\frac{\partial U}{\partial r_m^i} = \frac{\partial}{\partial r_m^i} \frac{1}{2} \sum_{i \in N_A} [\sum_{j \in nn(i)} U_{bs}^{ij} + \sum_{j,k \in nn(i)}^{k \neq j} U_{bb}^{jik}]$$

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

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



(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

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

if $j \in nn(i)$,

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

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

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

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

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

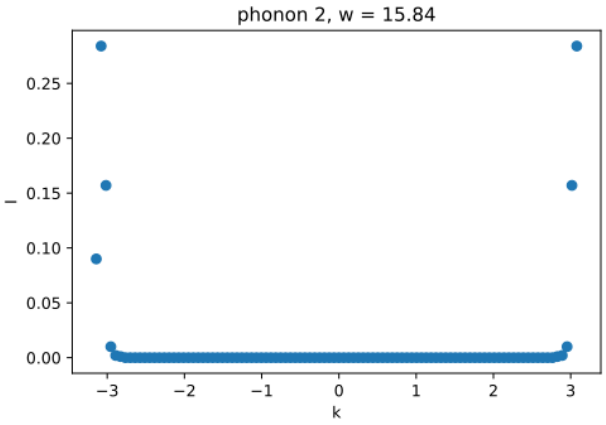


Figure 11: Finite decomposition of an eigenstate for a chain length of 100

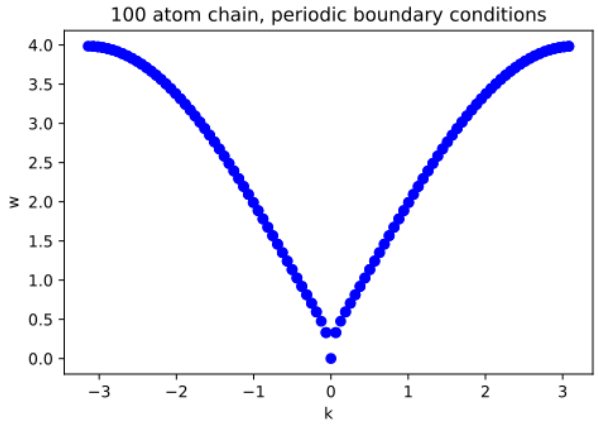


Figure 12: Spectral Density of phonons for a 1 dimensional 100 atom chain

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 $m \neq n$,

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

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

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

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

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

for $m \neq n$,

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

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

$$+ \sum_{j,k\in nn(i)}^{k\neq j} \frac{3}{8}\beta_{jik}\frac{(2r_n^i - r_n^j - r_n^k)(2r_m^i - r_m^j - r_m^k)}{||d_{ij,0}||||d_{ik,0}||}$$

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

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).