Nanomechanics of graphene membranes

Frederik Grunnet Kristensen (s164003),* Christoffer Vendelbo Sørensen (s163965),† and Rasmus Kronborg Finnemann Wiuff (s163977)‡

Technical University of Denmark§

Supervisors: Mads Brandbyge and Tue Gunst

DTU Nanotech

(Dated: 10/1-2018)

Abstract: This paper analyses simulated vibrational modes in nanomembranes with diameters between 1 and 5 nm, using Atomistix Toolkit[1], with the purpose of characterising the frequency response of the membrane, along with atomic behaviour of a few localised modes as to obtain a correlation with findings of similar computational simulations of microscale membranes[2]. The nanomembranes in question are both a theoretical single layer idealised membrane and a more realistic double layered membrane, imitating a graphene layer upon a substrate. We hypothesised that the findings of such similar simulations will correlate in behaviour when moving to the nanoscale. As it turns out, the modes do scale, as the membrane sizes reduce to the nanoscale, and the frequencies and atomic movement are easily obtained using our developed analysis tools. Our findings motivates experimental tests on the nanoscale. Such membranes could be setup using nanomesh substrates with a graphene layer attached. These membranes could be produced using block copolymer lithography for the substrate and CVD for the graphene layer.

CONTENTS

I. INTRODUCING THE PROJECT

Since the isolation and characterisation of Graphene in 2004 by Andre Geim and Konstantin Novoselov, scientists have marvelled over the physical properties and potential application of Graphene. Being a relatively new material, many aspects and ideas are being investigated and researched at all times. Graphene yield extreme tensile strength as well as extreme electric conductivity, yet its structure is fairly simple. Graphene consists solely of carbon atoms thus making it easy to simulate using specialised software, since carbon atoms are greatly understood in terms of chemical bonding. As graphene is a very versatile material the possibilities for research in simulation environments are virtually limitless. Therefore it is basically possible to make experiments limited only by imagination, in order to discover new properties and possible applications of graphene. This saves resources before entering the lab, where the simulated reality is tested.

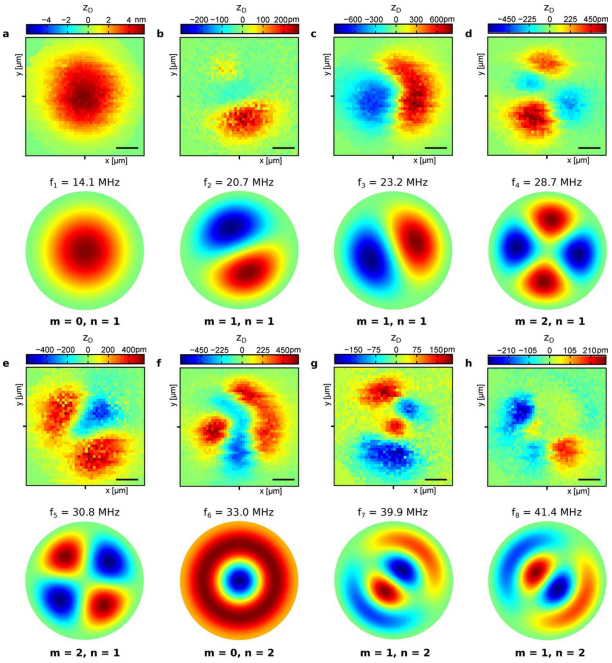

Figure 1: Visualizing resonant motion. (a–h) Top, experimental data; bottom, finite-element calculation. The modes predicted by the calculation are indexed by (m,n). Panels b and c show that the nanodrum hosts a split degenerate (1,1) mode, while also the (2,1) mode is split, as is shown in panels d and e. The displacement profile measured in panel f resembles a (0,2) mode, which is distorted due to an imperfection as discussed in the main text. Panels g and h reveal a degenerate (1,2) mode. Scale bars: 1 μm.[2]

In this rapport we will simulate and analyse the phonons of nanomembranes. In the article "Visualizing the Motion of Graphene Nanodrums"[2], nanomembranes on a microscale are simulated and experimentally tested. The article describes how phonons travel through these membranes. The results of the work in the article[2] (See Fig. 1) and many others suggests that the simulated phonon models holds true for systems on the micrometer scale when tested in the lab. We will examine if the same phenomenons are found at the nano-scale. To do this we start by considering a ideal system of just one sheet of carbon atoms. By simulating this ideal system in a virtual environment we will analyse the vibrational modes, and

frequencies in said modes. By simulating various sizes of membranes, we will examine how scaleable our analysis turns out to be. We will employ the software Atomistic ToolKit (ATK)[1] to carry out these simulations. The software will enable prompt setup of relevant structures with variying parameters. Afterwards in order to make a more realistic scenario we will investigate whether you can create the same membrane effects if you take a graphene layer and put it on top of a substrate with different sized holes, a nanomesh, to form a two layer membrane. The nanomesh can be fabricated as done in the paper "Graphene nanomesh"[3] where nanomeshes are created with holes of about 20 nm diameter, as shown on Fig. 2. It is expected possible to create such a nanomesh at the size of few tens of nanometers at DTU Nanotech with Block-copolymer lithography or TEM structuring of a substrate. We will simulate and analyse these two layer membranes and then compare them to the idealised membranes. We assume that the nanomembranes will have similar properties as the bigger membranes in the article "Visualizing the Motion of Graphene Nanodrums"[2] (Fig 3). The purpose of this project is to simulate phonons in the nanomembrane and find the optimal conditions for producing these phonons.

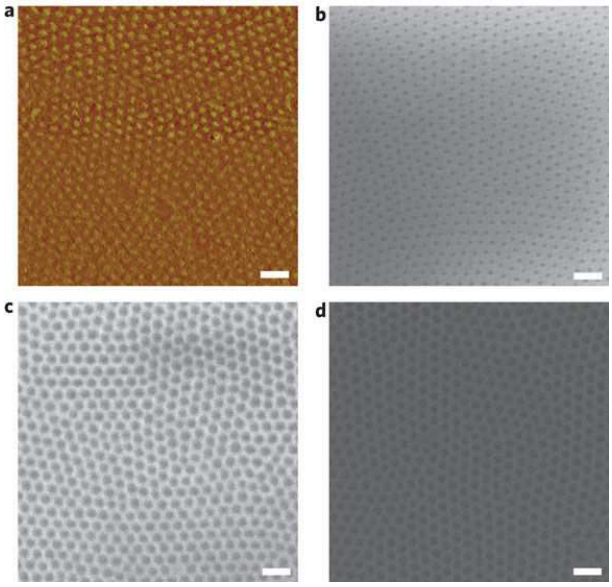

Figure 2: Images illustrating the steps of the nanomesh fabrication process. a, AFM phase contrast image of the annealed block-copolymer film on graphene, showing hexagonal-packed PMMA domains in the PS matrix. b, SEM image of a porous PS film obtained by selectively removing the PMMA domains. c, SEM image of the SiO_x nanomesh mask after reactive ion etching with the PS mask. d, SEM image of a GNM structure after removing the top SiO_x mesh mask. Scale bars, 100 nm.[3]

II. INTRODUCTION TO LATTICE GEOMETRY, NANOMECHANICS AND ATOMISTIX TOOLKIT

A. Lattice Geometry

In order to set up the simulation environment the basic geometry of system must be defined. Using generating vectors as well as unitcells the structure and holes of the

graphene sheet will be defined within this geometry.

1. *Bravais Lattice & Generating Vectors*

At first we want to specify a Bravais lattice for the graphene-layer. The Bravais lattice is a lattice that is invariant under translation which means that the lattice does not change when you translate it with the vector

$$\mathbf{T} = n\mathbf{a_1} + m\mathbf{a_2} \quad \mathbf{a_1}, \mathbf{a_2} \in \mathbb{R}^2 \tag{1}$$

Where $n, m \in \mathbb{Z}$ and $\mathbf{a_1}$, $\mathbf{a_2}$ are the generating vectors for the space lattice. Moreover, all the points in the lattice are given by

$$\mathbf{R} = (nd, md) \tag{2}$$

where d is the spacing between the lattice points and n, m can be both positive and negative integers.

As we are working with graphene which is carbon atoms arranged in a hexagonal structure, we want to choose a hexagonal Bravais lattice and define our generating vectors accordingly. Both generating vectors start at the center of a hexagon. This gives the generating vectors $\mathbf{a_1} = (d, 0)$ & $\mathbf{a_2} = \left(-\frac{d}{2}, \frac{\sqrt{3}d}{2}\right)$. In Fig. 3 a drawing of the lattice with its generating vectors is shown.

As the carbon-carbon bondlength in graphene is 1.42 Å, the distance between the lattice points in the Bravais lattice is $d = 2.46$ Å. This gives the numerical form

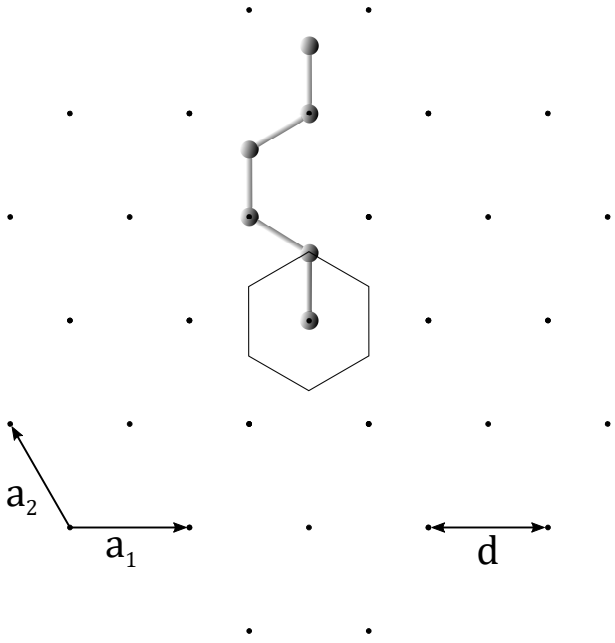

Figure 3: Hexagonal Bravais lattice structure with generating vectors $\mathbf{a_1}$ & $\mathbf{a_2}$. The Hexagon in the middle with the solid lines is unit cell and the shaded elements are the carbon atoms placed in the lattice.

of the analytical derived expression for the generating vectors $\mathbf{a_1}$ & $\mathbf{a_2}$.
$\mathbf{a_1} = (2.46\ \text{Å}, 0)$ & $\mathbf{a_2} = (-1.23\ \text{Å}, 2.13\ \text{Å})$.
These are the vectors that will be employed during simulation.

B. Nanomechanics

This following section will focus on the mechanics of the now defined geometry of the Graphene sheet. It will incorporate Hooke's law in three dimensions, the energy of the system the equation of motion and the normal modes of the lattice. Most of the ideas, formulas and the work-through in general, has its roots in the book [4]. We refer the

reader to this book for a more detailed describtion of nanomechanics. The very basics of the mechanics is described with an example using two atoms in a mass-spring system and can be reviewed in Appendix A.

1. *Dynamics in a three dimensional lattice*

As we are working in three dimensions the position of each atom has three coordinates and we will be working with bulk systems containing many atoms. To accommodate for these changes we will use vector notation. Otherwise the progression will be similar to the two atom system Appendix A, first defining the interaction potential with the harmonic approximation and a *Tensor* function, then the system of equations of motion which contains a *Dynamical Matrix*, an eigenvector-eigenvalue problem as well as the wavevector $\mathbf{q}$ and at last discussing the solutions to the system of equations.

To describe the atoms in the lattice and their vector field displacement $\mathbf{u}(\mathbf{r}, t)$ we are going to use classic mechanical theory as every atom in the lattice are seen as point masses, just like the system of two atoms Appendix A. As defined in Section II A 1 we have a lattice with $\mathbf{R}_j$ lattice points. In Section II A 1 we look at a two dimensional lattice but the following will describe a three dimensional lattice which basically means that $\mathbf{R}_j$ will have an additional coordinate. We want this system to be at equilibrium and therefore we choose an origin where the atom at the center of the j'th unitcell is placed on the $\mathbf{R}_j$ lattice point. Again looking at Fig. 3 that would be an example of such as system in two dimensions. A continuous displacement of the whole lattice $\mathbf{u}(\mathbf{r}, t)$ is now added to the lattice. The displacement is defined at every point $\mathbf{r}$ in the lattice which means that it is also defined in between the lattice points $\mathbf{R}_j$. The use of such displacement implicates that the atoms in the j'th unitcell will be displaced[4] (p.57) by $\mathbf{r}_j = \mathbf{R}_j + \mathbf{u}(\mathbf{r}_j, t)$.

We want to employ a harmonic approximation for the system and for that to be successful, one have to assume very small displacements, i.e. very small $\mathbf{u}$. Because of the assumption of very small displacements, the displacement field will be evaluated at the equilibrium position $\mathbf{R}_j$ instead of the instantaneous position $\mathbf{r}_j$. The displacement function can therefore be described as $\mathbf{u}(\mathbf{r}_j, t) \approx \mathbf{u}(\mathbf{R}_j, t)$. With displacements defined we can describe where the j'th atom is located at time t: $\mathbf{r}_j = \mathbf{R}_j + \mathbf{u}(\mathbf{R}_j, t) \rightarrow \mathbf{r}_j = \mathbf{R}_j + \mathbf{u}_j(t)$ using $\mathbf{u}_j(t) = \mathbf{u}(\mathbf{R}_j, t)$ as a short hand. The interaction potential energy of a two or three dimensional system is dependent on some displacement from equilibrium, why the total potential energy can be described with a function containing all atom positions at a time t. The potential energy takes the form $U = U(\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, ...\mathbf{r}_N)$. Now that we

have a function for the total potential energy, using a small displacement $\mathbf{u}_j(t)$, the harmonic approximation can be written up using a Taylor expansion[4] (eq.2.29).

$$U(\mathbf{r}_1, \mathbf{r}_2, ...\mathbf{r}_N) = U(\mathbf{R}_1, \mathbf{R}_2, ...\mathbf{R}_N) + \sum_{j=1}^{N}\sum_{\alpha=1}^{3} \frac{\partial U}{\partial r_{j\alpha}}\bigg|_{\mathbf{R}_j} u_{j\alpha} + \sum_{j,k=1}^{N}\sum_{\alpha,\beta=1}^{3} \frac{1}{2}\frac{\partial^2 U}{\partial r_{j\alpha}\partial r_{k\beta}}\bigg|_{\mathbf{R}_j} u_{j\alpha}u_{k\beta} + ... \quad (3)$$

The second term of the harmonic approximation is

$$\sum_{j,k=1}^{N}\sum_{\alpha=1}^{3} \frac{\partial U}{\partial r_{j\alpha}}\bigg|_{\mathbf{R}_j} u_{j\alpha} = 0$$

when evaluated at equilibrium, moreover the potential energy at the equilibrium positions $\mathbf{R}_j$ also equals 0. Dropping the higher order terms, this leaves us with the harmonic approximation[4] (eq.2.30)

$$U_{harm}(\mathbf{r}_1, ...\mathbf{r}_N) = \frac{1}{2}\sum_{jk=1}^{N}\sum_{\alpha\beta=1}^{3} \frac{\partial^2 U}{\partial r_{j\alpha}\partial r_{k\beta}}\bigg|_{\mathbf{R}_j} u_{j\alpha}u_{k\beta} \quad (4)$$

This potential energy is the equivalent to the elastic potential energy $U = \frac{1}{2}kx^2$ where $\frac{\partial^2 U}{\partial r_{j\alpha}\partial r_{k\beta}} = k$ and $x^2 = u_{j\alpha}u_{k\beta}$.
In order to simplify the notation when working with the dynamics of the system, we introduce a *tensor*[4] (p.58) which is a 3x3 matrix. The tensor does not change when translated to another coordinate system, but elements within the tensor does change however. The tensor[4] (eq.2.31) is defined by

$$\Phi_{\alpha\beta}(\mathbf{R}_i, \mathbf{R}_j) = \frac{\partial^2 U}{\partial r_{i\alpha}\partial r_{j\beta}} \quad (5)$$

and is in terms of the curvature of the total interaction energy $U(\mathbf{r}_1, ...\mathbf{r}_N)$[4] (p.58), evaluated at the equilibrium position. As the tensor are only dependent on the equilibrium positions $\mathbf{R}_i, \mathbf{R}_j$ the crystal itself does not change under translation. This means that the tensor only depend on the difference in the atoms positions. The tensor then becomes[4] (eq.2.32)

$$\Phi_{\alpha\beta}(\mathbf{R}_i, \mathbf{R}_j) = \Phi_{\alpha\beta}(\mathbf{R}_i - \mathbf{R}_j) \quad (6)$$

This will also change Eq. (4) to

$$U_{harm}(\mathbf{r}_1, ...\mathbf{r}_N) = \frac{1}{2}\sum_{jk=1}^{N}\sum_{\alpha\beta=1}^{3} u_{j\alpha}\Phi_{\alpha\beta}(\mathbf{R}_j - \mathbf{R}_k)u_{k\beta} \quad (7)$$

Now that an expression for the total potential energy has been described, we need an expression for the equation of motion. According to Newtons 2. Law $F = ma$ or Hooke's Law $F = -kx$. This means that $ma = -kx$. As we all ready have an expression which includes k and x (Eq. (4)), we can write the equation of motion as

$$M\frac{\partial^2}{\partial t^2}u_{j\alpha} = -\sum_{k=1}^{N}\sum_{\beta=1}^{3}\Phi_{\alpha\beta}(\mathbf{R}_j - \mathbf{R}_k)u_{k\beta} \quad (8)$$

Solving this system of equations requires finding the normal modes for the system which is the same as the general solution for the system of equations. This means that all solutions must have the same time dependence. As the general solution form a complete set, we can describe any motion

in the lattice as a superposition of normal modes. A guess to the harmonic solution is defined as.

$$\mathbf{u}_j(t) = \mathbf{A}e^{i\mathbf{q}\cdot\mathbf{R}_j}e^{i\omega t} \tag{9}$$

Here $\mathbf{A}$ is the displacement for the j'th atom in all three directions and $\mathbf{q}$ is the wavevector which has the frequency ω for a given mode. This frequency will be defined as $\omega_{\mathbf{q}}$.
Inserting this solution into Eq. (8) gives

$$\omega_{\mathbf{q}}^2 M A_\alpha = \sum_{k=1}^{N}\sum_{\beta=1}^{3}\Phi_{\alpha\beta}(\mathbf{R}_j - \mathbf{R}_k)e^{i\mathbf{q}\cdot(\mathbf{R}_k-\mathbf{R}_j)}A_\beta \tag{10}$$

We make the equation independent of j by choosing another equilibrium point $\mathbf{R}_l = 0$, this changes Eq. (10) to

$$\omega_{\mathbf{q}}^2 M A_\alpha = \sum_{k=1}^{N}\sum_{\beta=1}^{3}\Phi_{\alpha\beta}(\mathbf{R}_k)e^{i\mathbf{q}\cdot\mathbf{R}_k}A_\beta \tag{11}$$

Which is an equivalent to Hooke's Law, but in three dimensions. If we take the Fourier transform of the tensor $\Phi(\mathbf{R}_j)$ which has been derived by symmetries of the interaction tensor in Eq. (5) (for a detailed workthrough, see [4] (eq.2.39-2.41)) we get a new tensor which is defined as $D_{\alpha\beta}(\mathbf{q})$ and has components

$$D_{\alpha\beta}(\mathbf{q}) = \sum_{k=1}^{N}\Phi_{\alpha\beta}(\mathbf{R}_k)e^{i\mathbf{q}\cdot\mathbf{R}_k} \tag{12}$$

Which is the same as the spring constant k. Inserting Eq. (12) into Eq. (11) gives

$$\omega_{\mathbf{q}}^2 M A_\alpha = \sum_{\beta=1}^{3} D_{\alpha\beta}(\mathbf{q})A_\beta \tag{13}$$

Where $\mathbf{q}$ is the wavevector. This new tensor is called the *Dynamical Matrix* and has big importance to the eigenvalue/eigenvector problem which is up next. The three equations that Eq. (13) consists of, can be written as an eigenvalue/eigenvector problem

$$M\omega_{\mathbf{q}}^2\begin{bmatrix}A_1\\A_2\\A_3\end{bmatrix} = \begin{bmatrix}D_{11} & D_{12} & D_{13}\\ D_{21} & D_{22} & D_{23}\\ D_{31} & D_{32} & D_{33}\end{bmatrix}\begin{bmatrix}A_1\\A_2\\A_3\end{bmatrix} \tag{14}$$

Here the eigenvector $\mathbf{A}$ and the tensor $D_{\alpha\beta}(\mathbf{q})$ are functions of the wavevector $\mathbf{q}$. There is three distinct eigenfrequencies ω for each wavevector $\mathbf{q}$. The eigenvector $\mathbf{A}$ determines mode polerizations and corresponds to each eigenfrequncy ω. Moreover the eigenfrequencies can be found by solving the equation

$$\mathrm{Det}(D_{\alpha\beta}(\mathbf{q})) = 0 \tag{15}$$

which then again can be used to find the eigenvectors by inserting the eigenfrequencies in Eq. (11).
Now that we have defined the eigenvalue/eigenvector problem, we basically have everything we need to calculate the normal modes of a system of three (or two) dimensions. Especially the result of the dynamical matrix is important when doing simulations as it contains information about the potential energy of the system. Because each normal mode for the discussed lattice system has quantized energy states we refer to them as *Phonons*. When working with systems in general the wavevector $\mathbf{q}$ is quantized and

therefore restricted to some specific set of values. Depending on the dimensions of the system the values may vary and may be described using *Periodic Boundary Conditions* depending of system of choice. The periodic boundary conditions for a two dimensional system has been worked out in Appendix B.

C. Atomistix ToolKit: ATKPython and Nanolanguage

In this project we will utilise the Atomistix ToolKit, developed by QuantumWise A/S[1], or ATK for short. The Toolkit takes Python scripts as inputs and simulates a completely custom lattice enviroment with various applied calculations. It is possible to simulate very specific scenarios, alter the simulation parameters and extract results for analysis easily and fast. ATKPython (the standalone simulation program) extends the Python language with Nanolanguage which tells ATKPython what to do when launched. One can set up specific bravais lattices and repeat them into large 2D structures. Then specific atoms can be tagged or altered and calculations can be set up. All of these features are available through QuantumWise VNL, a GUI which sets up the scripts and calculations, if need be. When working with VNL both the GUI and the CLI are used. In our case, the GUI serves mostly as a visual aid for understanding the modes with 3D graphics and easy graph extraction. The typical workflow when using VNL and ATK is depicted on Fig. 4

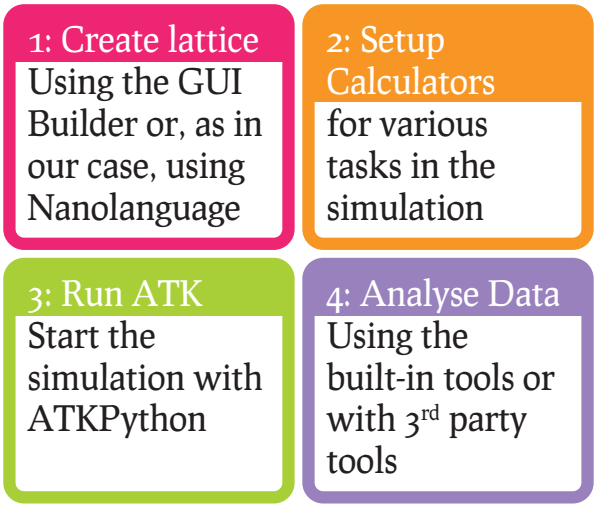

Figure 4: Diagram showing the typical workflow in VNL.

III. SIMULATING VIBRATING MEMBRANES

A. Our Workflow

In order to simulate the vibrational modes in a graphene lattice, the dynamical matrix needs to be, both calculated and postprocessed. Both of which needs substantial amounts of computational power and time. The workflow is therefore created with steps of scripts which can be sent to a HPC cluster[5] for computation. The workflow for simulating our graphene membrane is described as such:

1. Create the membrane with **NanoSheetCreator.py** or **NanoMembraneCreator.py**

(Section III C).

2. Use VNL Builder GUI to ensure correctly tagged holes. When creating two layer membranes, it is nessecary to remove the atoms in the hexagonal tag in one of the layers. Such tag can be seen on Fig. 5.

3. Use **00_FixConstraints.py** (Section III D) to fix the tagged atom indices[6].

4. Use **01_RelaxSheet.py** or **01_LennardJonesRelax.py** (Section III E) to optimise the bond length and geometry of the graphene sheet and apply inter-layer potentials.

5. Use **02_DynamicalMatrix.py** (Section III F) to compute the Dynamical Matrix for the sheet.

6. Use **03_SheetVibrations.py** (Section III G) to calculate the vibrational modes of the sheet.

7. Postprocessing and analysis of the created *.hdf5* files will then take place. (Section IV)

B. Code Repository and code compendium

All of the mentioned code and scripts exists as a repository available online: `https://github.com/rwiuff/Nanomechanics-for-graphene-membranes` Beware that ATKPython[1] is required to run the scripts. The code can also be found in the Code Compendium in Appendix D.

C. NanoSheetCreator.py and NanoMembraneCreator.py

The first script generates the graphene membrane with atoms tagged in a hexagonally shaped path. The output is the atomic configuration in the *.hdf5* format. The workings of the script is briefly described as such: First the script creates a bravais-lattice consisting of a hexagonal unit cell. Then two carbon atoms are placed in the unitcell and the unit cell is repeated to the users specifications. Listing 1 demonstrates, by example from *NanoSheetCreator.py*, how such lattice is created with Nanolanguage.

```
24 # Create hexagonal bravais lattice
25 lattice = Hexagonal(a=2.4612 * Angstrom, c=20 * Angstrom)
26
27 # Set atomic elements
28 elements = [Carbon] * 2
29
30 # Place carbon atoms in unit cell
31 coordinates = [(0, 0, 0), (0.33333, 0.66667, 0)]
32
33 # Create unit cell as "sheet"
34 sheet = BulkConfiguration(lattice, elements,
35                           fractional_coordinates=coordinates)
```

Listing 1: Lines 24-35 from the *NanoSheetCreator.py* script shows how Nanolanguage can be used to create a hexagonal bravais lattice

The user is then asked to input position and size of the hexagonal tag wanted for placement of the hole. Lastly the sheet can be repeated and information about the position and sizes of the tags are printed to a *.txt* file.

D. 00_FixConstraints.py

A historical bug in ATKPython demands consecutive indices for mapping constrained atoms, when using the method *phononEigensystem*. Thus a script was written to ensure that tags with these indices were consecutive. Furthermore, when working with two layer membranes, the script tags the individual layers as ”Layer1” and ”Layer2”.

```
28 # Set intralayer interaction.
29 potentialSet_layer1 = Tersoff_C_2010(tags='Layer1')
30 potentialSet_layer2 = Tersoff_C_2010(tags='Layer2')
31 # Interlayer interaction - Lennard-Jones type.
32 # Define a new potential for the interlayer interaction.
33 lj_interlayer_potential = TremoloXPotentialSet(
34     name="InterLayerPotential")
35 # Add particle type definitions for both types.
36 lj_interlayer_potential.addParticleType(ParticleType.fromElement(
37     Carbon, sigma=3.326 * Angstrom, epsilon=Zeta * 8.909 * meV))
38 # Add Lennard-Jones potentials between the carbon atoms
39 # of different layers.
40 lj_interlayer_potential.addPotential(
41     LennardJonesPotential('C', 'C', r_cut=10.0 * Angstrom))
42 lj_interlayer_potential.setTags(['Layer1', 'Layer2'])
43
44 # Combine all 3 potential sets in a single calculator.
45 calculator = TremoloXCalculator(
46     parameters=[potentialSet_layer1,
47                 potentialSet_layer2, lj_interlayer_potential])
48 calculator.setVerletListsDelta(0.25 * Angstrom)
```

Listing 2: Lines 28-48 from the *01_LennardJonesRelax.py* script shows how the 2 intralayer potentials are combined with the interlayer potential.

E. 01_RelaxSheet.py and 01_LennardJonesRelax.py

After creating a membrane configuration, said configuration needs a potential defined and to be relaxed so as to minimise bonding energies. These scripts does this for two different cases. *01_RelaxSheet.py* defines the potential for a single layered membrane with the Tersoff/Brenner Carbon Potential[7]. It then calculates the energies and adjust the geometry up to 200 times in order to find the optimal configuration. When handling a doubled layered membrane one needs to use *01_LennardJonesRelax.py*. This script uses a

simplified potential; the Lennard Jones potential, which is suitable for Van Der Waals structures. Normally the potential takes in, and accounts for, angles between atoms, but the Lennard Jones potential merely uses distance. For two particles i and j, the potential is defined as follows:

$$V_{ij}(r) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r}\right)^{12} - \left(\frac{\sigma_{ij}}{r}\right)^{6}\right] \quad (16)$$

Where ϵ and σ are material based parameters. They are given as:

$$\epsilon_{ij} = \sqrt{\epsilon_i \cdot \epsilon_j} \quad (17)$$

$$\sigma_{ij} = \frac{\sigma_i \cdot \sigma_j}{2} \quad (18)$$

The ϵ parameter is the depth of potential minimum for a given material. By altering the ϵ parameter we imitate different materials as to see the connection between substrate - graphene potential and vibrational modes. ATKPython allows combining potentials. Listing 2 shows how two intralayer Tersoff Brenner potentials are defined along with the interlayer potential. The potentials are then combined and further calculations now relies on these potentials.

F. 02_DynamicalMatrix.py

At this point in the workflow, the dynamical matrix is required to continue. The script *02_DynamicalMatrix.py* accomplishes this by using the function *DynamicalMatrix*. The function displaces atoms back and forth to calculate the finite difference of the forces in order to find the force constants. Each atom moves in 3 dimensions thus each atom has 3 degrees of freedom. Each displacement in every degree of freedom for each atom can be parallised as each displacement takes up one CPU. We used high performance computing clusters at DTU Nanotech to parallise the calculations. Usually we assigned 8 nodes and therefore 8 CPU = 8 displacement simultaneously calculated at a time thus reducing the calculation time.

G. 03_SheetVibrations.py

The last step before using more home concucted tools are that of the *VibrationalModes*. This function produces a file, from wich all information about a configurations vibrational modes can be extracted. However, this process demands large amounts of RAM, and without a server cluster, it is almost impossible to calculate holes larger than 4-5 nm in diameter. We used the HPC at DTU Nanotech to calculate the vibrational modes.

H. DataExtract.py

The first tool designed from scratch is the DataExtract utility. It loads a number of dynamical matrices and outputs pickled[8] datafiles containing information on the configurations phonon frequencies, relative projections and root-mean-square of atom displacement out of the plane. The script uses

the function *ProjectedPhononBandsDisplacement*, written by Tue Gunst. The function is described in Section III I. The function *ProjectedPhononBandsDisplacement* relies on the ATKPython method *phononEigensystem*. This function demands large amounts of RAM, and as with *03_SheetVibrations.py* (Section III G), a server cluster may be needed.

I. MyAnalysisFunction.py

This script were given to us by Tue Gunst, and was designed for, and used in the paper "Suppression of intrinsic roughness in encapsulated graphene"[9]. The function *MyAnalysisFunction.py* calculates the vibrational modes from a dynamical matrix and isolates the motion out of the lattice plane. It normalises and projects the motion unto a vector with each element $=< 1$. It then calculates the characteristic length of the modes, and from this length, the root-mean-square of the out of plane displacement is calculated and returned along with an array of mode frequencies and projections.

J. ProjectionPlotter.py

The *ProjectionPlotter.py* script is an analysis tool for analysing the out of plane projection for all modes in a series of membranes, varying in size. By loading pickled data created by the *DataExtract.py* tool, this script plots a scatter plot with a series of out of plane projection, and given frequencies for different sized membranes. The script then fits a reciprocal function on the frequencies of the two lowest modes. The function is given as:

$$f = a \cdot \frac{1}{x^n} \tag{19}$$

The tool can be used to analyse the scaling of frequencies as the nanomembrane grows in diameter.

K. ZetaPlotter.py

At this point in the workflow the idealised single layer membrane has been analysed and its modes, plottet. The next step is to simulate a double layer lattice. By optimising the geometry and setting a Lennard Jones potential with *01_LennardJonesRelax.py* (Section III E), with varying ϵ values and then calculate distinct dynamical matrices and vibrational modes with the described tools, the *ZetaPlotter.py* script can be used to compare the modes of varying double layered membranes with those of a known single layered idealised membrane. The script takes in pickled data and produces a scatter plot with varying series for variying ϵ parameters and then plots the single layer membrane's modes along with lines marking the single layered membrane's mode frequencies. This tool is usefull for comparing ϵ values.

L. 2DdataExtract.py

This script is used for the sole purpose of extracting data from *.hdf5* files containing vibrational modes and save it as pickled files to be loaded with *2DmodePlotter.py* (Section III M). It inputs files, each containing modes for various double layered membranes and a known single layered membrane for comparisson. Vibrational modes are saved as the class *VibrationalMode* in each file. ATKPython has a method for extracting trajectories from such class. The trajectory contains a series of configurations, each representing a time step in a full period of a vibrational mode. The function takes the configuration of atoms when the displacement is largest (at which point, the vibrational mode has its largest amplitude), and saves the coordinates for the atoms in the configuration in the pickled files. This is shown on Listing 3

```
66                 # Extract the trajectory from VibrationalMode at mode i and
67                 # temperature T
68             VM = vibrationalmodes.movie(
69                 mode_index=ModeIndex[i], temperature=T * Kelvin)
70                 # Extract configuration from image j,i in trajectory
71             ZModes[c] = VM.image(image_index=ImageIndex[j, i])
```

Listing 3: Lines 66-71 from the 2DdataExtract.py script shows how coordinates from a configuration of atoms can be easily obtained from the class *VibrationalMode* using the movie and image methods.

M. 2DmodePlotter.py

This script is designed to visualise and compare selected modes from a single layered membrane and various double layered membranes. By taking the z-coordinates of the configurations described in Section III L, the script finds the largest overall amplitudes of the membranes and creates a colorscale. It then plots contours of the modes, along with information on root-mean-squeare displacement of the moving atoms (manually inputted, but easily obtained from the pickled files constructed by the script *DataExtract.py* Section III H). The result is a series of contour plots showing the shape and amplitudes of the atoms in five vibrational modes for various membranes. This tool is valuable in comparing size, shape and root-mean-square displacement between the idealised single layered membrane and double layered membranes with varying ϵ values.

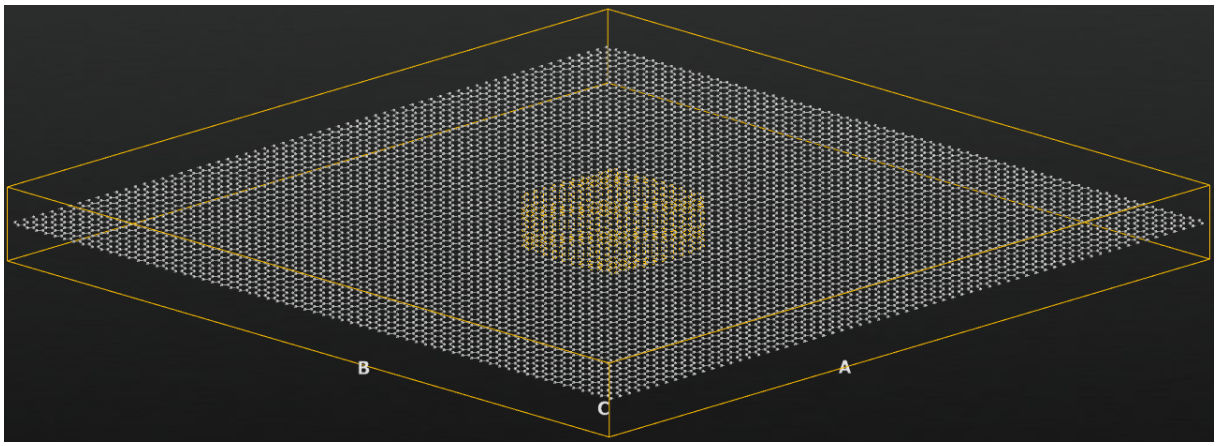

Figure 5: A snapshot from ATK[1] showing how the clamped system looks like when at rest. The hexagon in the middle is marked with tags and is the only part of the sheet that is not constrained i.e. the free standing membrane.

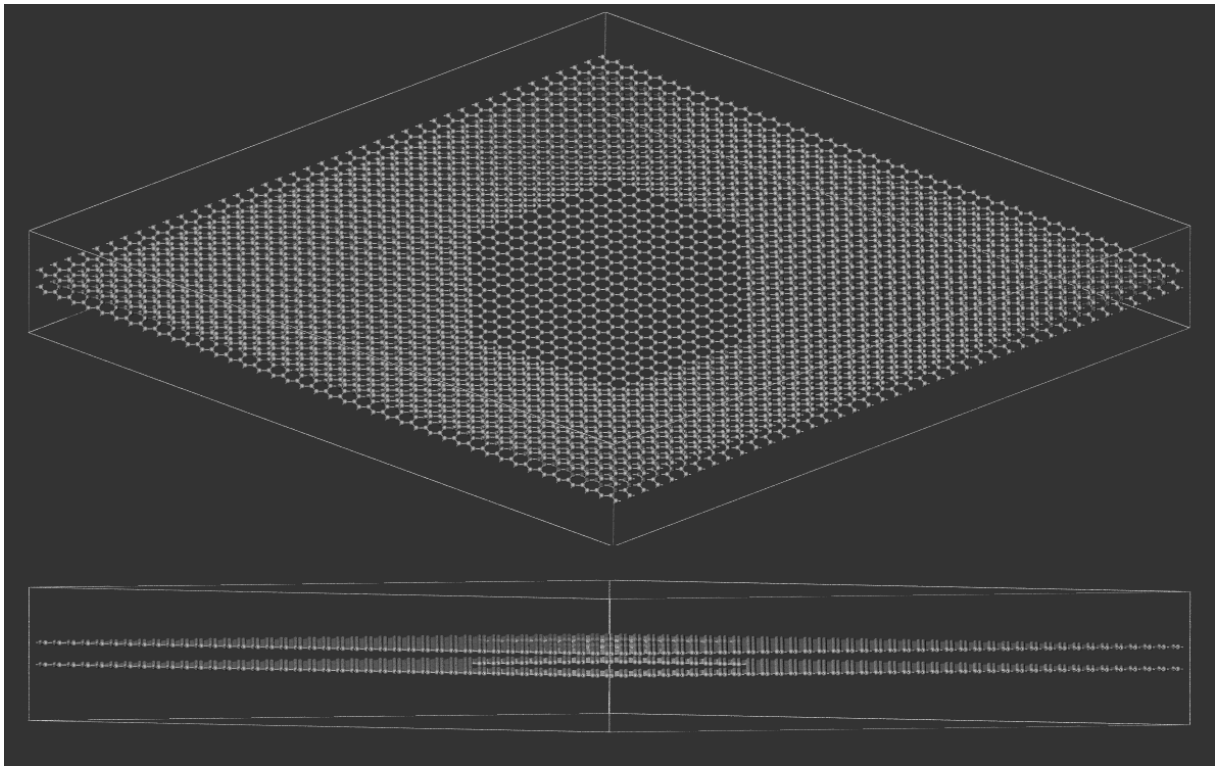

Figure 6: The picture on top shows the two layer system from above. The one on bottom shows the system from the side. Both snapshots have been taken in ATK[1].

IV. DATA EXTRACTION AND ANALYSIS

A. Clamped System

This following section focuses on data extraction and analysis for a system of a clamped graphene sheet with a free standing hexagon membrane in the middle of the sheet. See Fig. 5. The free standing hexagon will therefore be a simplified model compared to that of a more realistic model containing two graphene sheets: a constrained sheet with hexagonal holes and a free standing sheet on top. This model will be discussed later.

1. *Frequency vs. membrane size*

In order to find the correlation between frequency of the modes in the membrane and the membrane size, the frequency over each membrane, which varies in size, is extracted and plotted as a function of the size of the given membrane.

This is done by defining a sheet of a definite size in VNL and thereafter check the frequencies for the different membranes by inserting the different size membranes, one after the other. The distance from the edge of the membrane, to the edge of the sheet (or the next membrane) is called "neck" and has been chosen to be 5 nm for the biggest membrane which has a 10 nm diameter.

Furthermore, only the frequencies for the first couple of modes will be checked.

In this section the frequency will sometimes be noted in terms of energy for the modes [eV]. Other times just as the frequency in [Hz]. That is because the energy can be converted to a frequency and vice versa by the relation: $1\text{Hz} = 4.13\text{x}10^{-15}\text{eV}$.

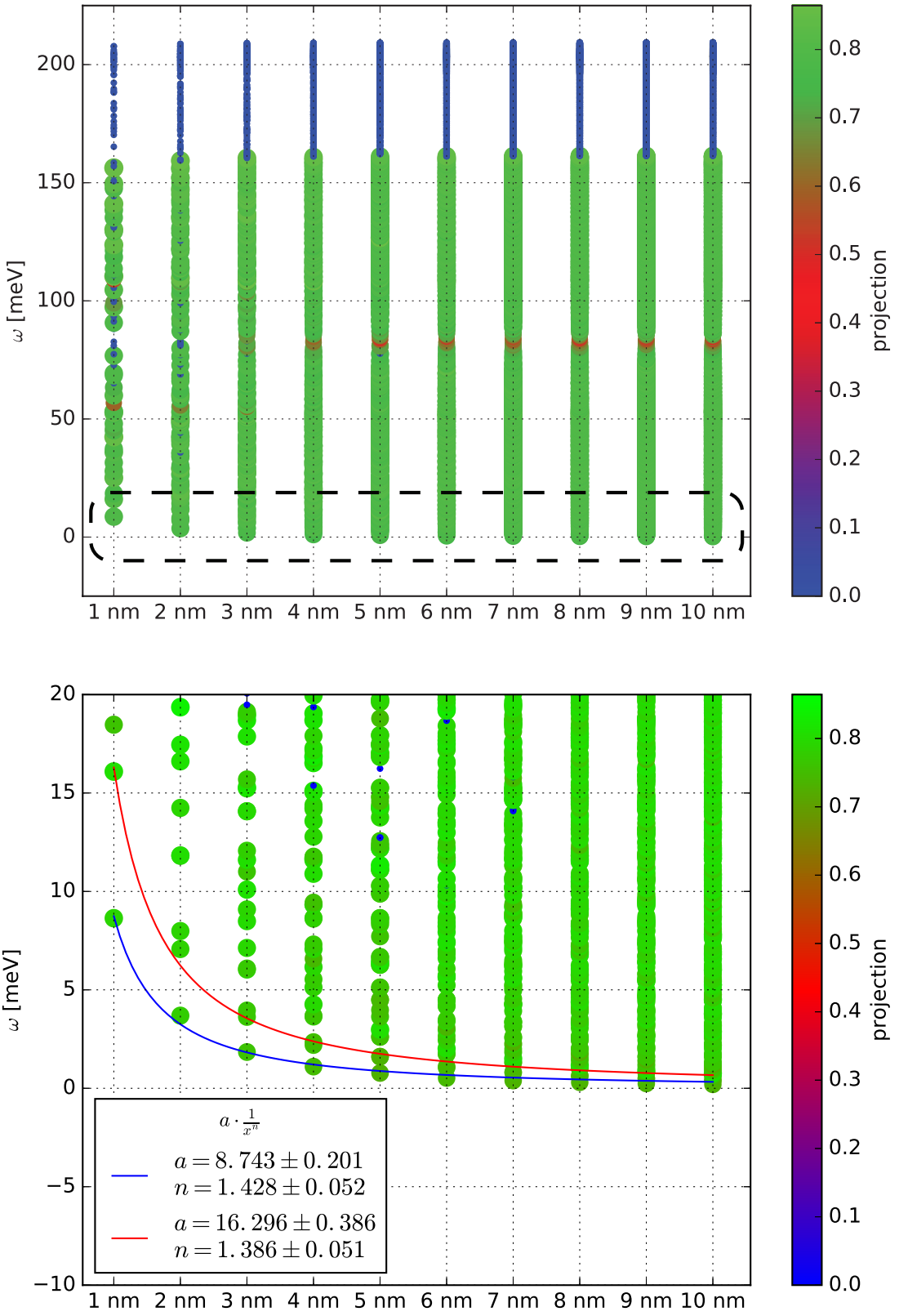

Figure 7: Top: Frequency plotted as a function of membrane size for a one-sheet clamped system. Bottom: Shows the same plot but zoomed in at first three modes. Every green dot being a specific mode, and the red and blue lines being trend curves for the first and second mode

Looking at the data it is clear to see that the frequency decreases when the size of the membrane increase and through our understanding of the physics behind the membrane, the frequency must have the following relation to the membrane size: $f(x) \to 0$ for $x \to \infty$. Using the regression for the first and second mode it is determined that the relation between frequency and membrane size can be described by:

$$f = a \cdot \frac{1}{x^n} \tag{20}$$

Where f is the frequency, x is the size of the membranes, and where a and n are constants. Comparing the constants for the first mode to the constants for the second it is seen that a relates to the frequency of the first membrane, while n is relatively the same. This indicates that even though the modes start at different frequencies they will conjugate towards zero at the same rate about $\frac{1}{x^{1.4}}$. This trend can also be used as a scaling factor to predict the frequency for even bigger membranes. For example: If you want a 20 nm membrane we can predict that the frequency of first mode for this hole will be

$$\begin{aligned} f =& \left(8.743 \cdot 10^{-3} \pm 2.012 \cdot 10^{-4}\right) \\ & \cdot \frac{1}{20^{1.428 \pm 5.166e-4} \text{ nm}} \\ =& 1.213 \cdot 10^{-4} \pm 1.898 \cdot 10^{-5} \text{ eV} \\ =& 29.3 \text{ GHz} \pm 4.59 \text{ GHz} \end{aligned} \tag{21}$$

This prediction fits with our understanding that the phonon in the membrane will decrease and eventually disappear as the membranes gets bigger, but never have a negative frequency.

B. Interlayer interaction

This section will focus on a simulation of a more realistic system that contains two graphene sheets. See Fig. 6. One sheet, containing holes, will be constrained and the other sheet will be standing free above. We will test what effect the interaction between the layers will have on the membranes of the free standing sheet. To to this we have chosen a specific hole size for the constrained sheet of 5nm and then varied the strength of the interaction potential between the layer. Three values were chosen: $\epsilon_{graphite} = 2.76$ meV[10], $\epsilon_{SiO_2} = 8.91$ meV[11] and $\epsilon_{strong} = 89.1$ meV. The values are the depth of the potential minimum for graphite, silicon dioxide and then a fictive and very strong potential. For the rest of this paper the ϵ values will be in relation to that of SiO_2, thus the values become: $\epsilon_{graphite} = 0.31$, $\epsilon_{SiO_2} = 1.00$ and $\epsilon_{strong} = 10.0$. The reason for including the very strong potential is to make a reference point for the two other substrate values and how well they correspond with the clamped system.

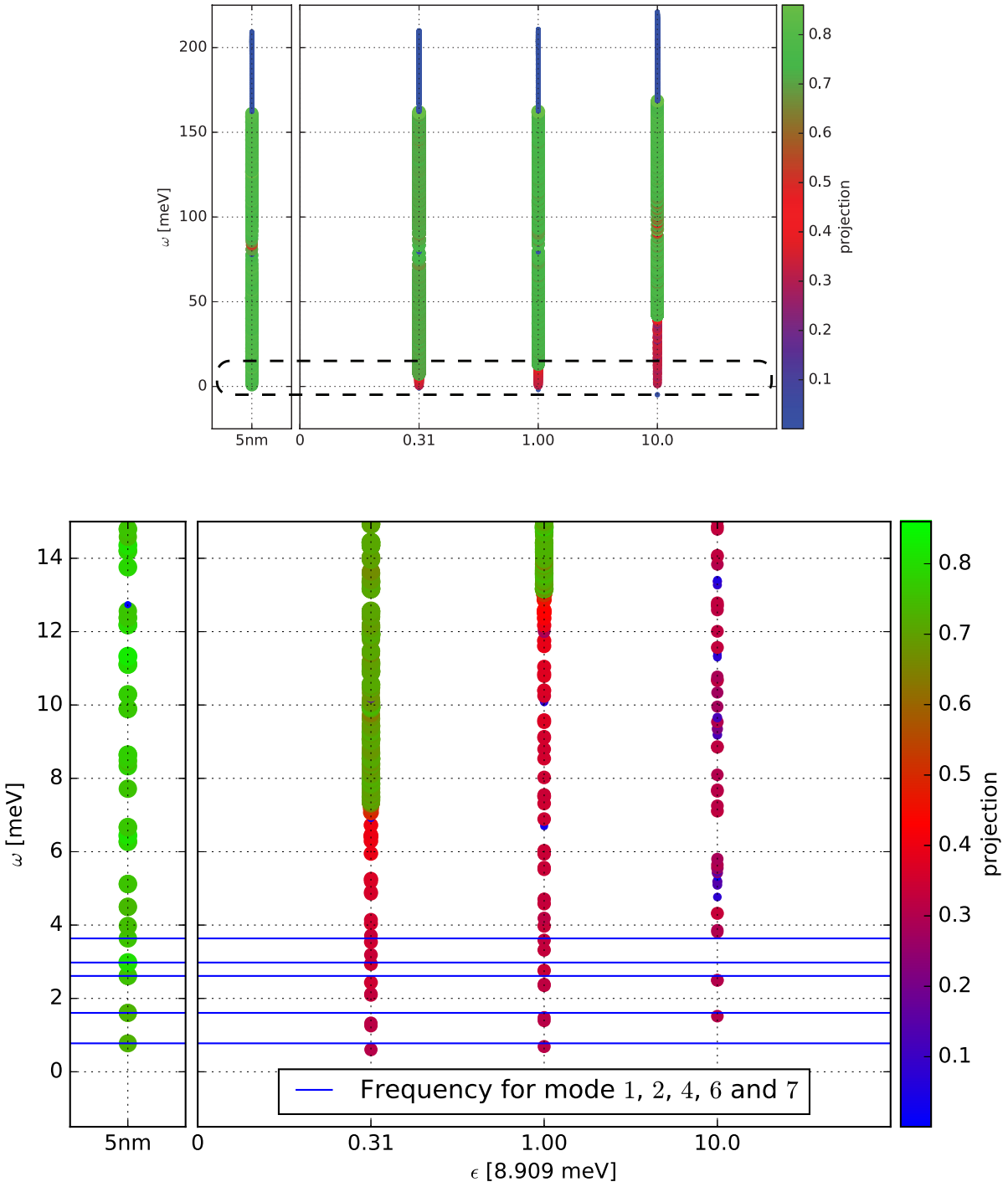

Figure 8: Top: The figure shows two plots: the small plot to the left shows the lowest modes and their frequencies for the single sheet with a 5nm membrane (shown as green dots). The plot to the right shows the corresponding modes for the system with two sheets again with a membrane of 5nm, but with three different interaction potentials between the layers: $\epsilon_{graphite} = 0.31$, $\epsilon_{SiO_2} = 1.00$, $\epsilon_{strong} = 10.0$ (from left to right). Bottom: Same plot but zoomed in at the bottom modes. The blue lines indicate the frequency values for the clamped system and shows how much they deviate from the system of two sheets depending on the strength of the potential between the layers.

Modes	Frequency [meV]	[THz]
1	0.78	0.19
2	1.61	0.39
4	2.61	0.63
6	2.98	0.72
7	3.64	0.88

Table I: Frequencies for modes 1,2,4,6,7 in the clamped system

As seen on Fig. 8 there is good correspondence in the frequency values of the clamped system and the two first interaction potentials in the two layer system. In Table II, a list of the frequencies for the two layer system and the deviation of the two layer system with respect to the clamped system can be seen. Table I shows the frequencies for the clamped system.

ϵ [8.91 meV]	Modes	Frequency [meV]	[THz]	Deviation [%]
0.31	1	0.61	0.15	21.8
	2	1.26	0.31	21.7
	4	2.10	0.51	19.5
	6	2.43	0.59	18.5
	7	2.94	0.71	19.2
1.00	1	0.69	0.17	11.5
	2	1.40	0.34	13.0
	4	2.35	0.57	10.0
	6	2.76	0.67	7.5
	7	3.32	0.80	8.8
10.0	1	1.52	0.37	94.9
	2	2.49	0.60	54.7
	4	3.81	0.92	46.0
	6	4.32	1.05	45.0
	7	4.76	1.15	30.8

Table II: Frequencies for modes 1,2,4,6,7 for the Two layer system with the three potentials $\epsilon_{graphite}, \epsilon_{\mathrm{SiO_2}}, \epsilon_{strong}$ and their deviation from the clamped system

ϵ [8.91 meV]	Average deviation [%]
0.31	20.2
1.00	10.1
10.0	54.2

Table III: Table showing the average deviation of the two layer systems with respect to the clamped system

However, as Fig. 8 also shows, the projection values does not have as much movement out of the plane as the clamped system. This can possibly be explained by the fact that there is some in-plane movement going on outside the area of the membrane. The calculation sums up movements over the whole sheet, why the out-of-plane movement over the membrane seems less because of the in-plane movement going on outside the mem-

brane. We will therefore make calculations for the RMS-amplitude of every atom only within the membrane in the clamped system and compare it to corresponding calculations for the two layer system with all three interlayer potentials. Fig. 9 shows the RMS calculations together with the out-of-plane displacements.

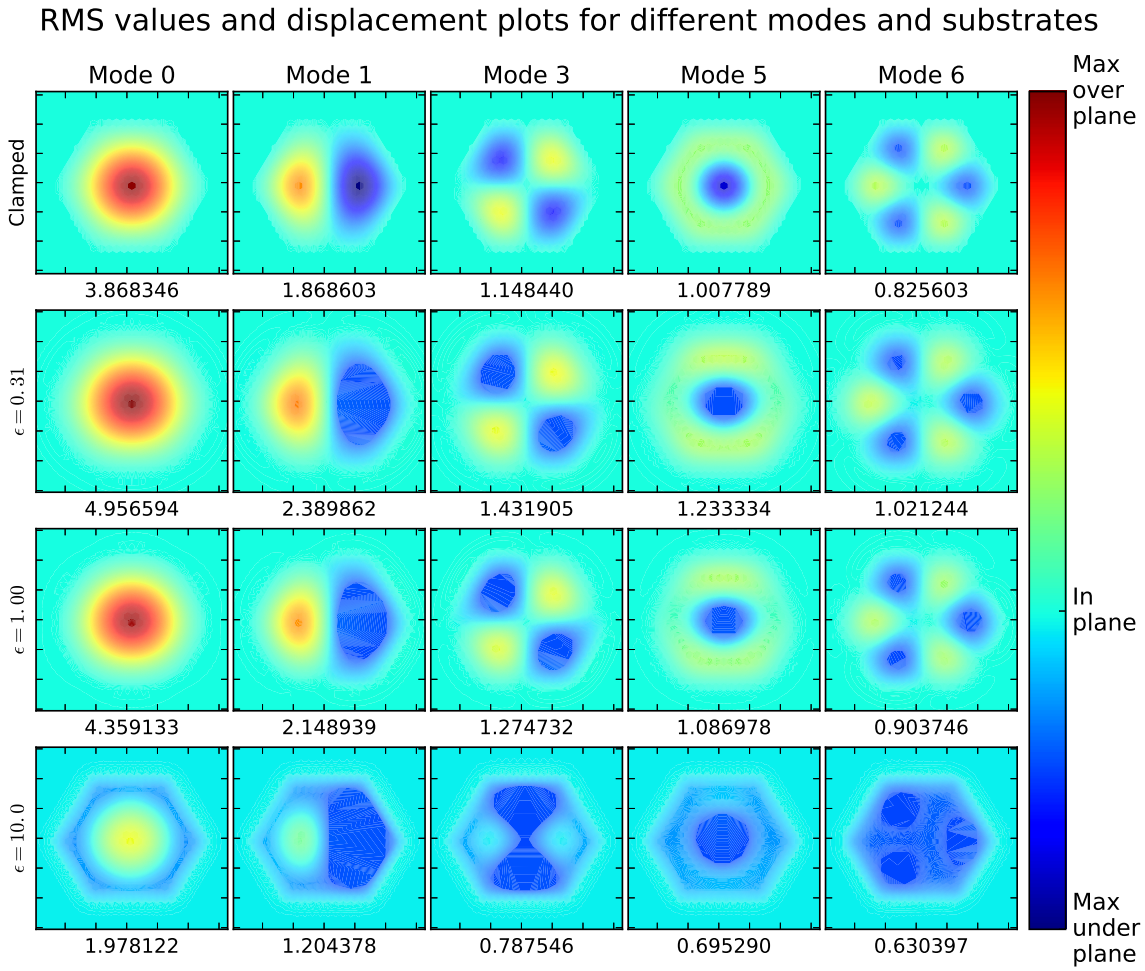

Figure 9: Figure showing the out of plane displacements with RMS-values for five different modes for the clamped system and the two-layer system with the three different interaction potentials respectively (Top to bottom). The reason why the modes has the values 0,1,3,5,6 is because the modes pairs $[2, 3]$, $[4, 5]$ are degenerated pairs, why we have only chosen mode 3 and 5.

In Fig. 9 we can see that there is good correspondence between the clamped system and the two systems with the low interaction potential. Even more so, when looking at the RMS-values we can see that system with the SiO_2 ($\epsilon = 1.00$) substrate potential is pretty close to the RMS-values of the clamped system, even closer than graphite.

The reason why the system with the strong interaction potential is blue is because the potential draws the upper sheet down towards the lower one. This causes the area around the rim to be blue no matter what mode we look at. Then again the system with strong interaction potential is only for reference.

C. Modes with varying membranes size, Clamped system

In this section will look at the clamped system only and investigate if and how shape of the modes in the membrane changes as we increase the size of the membrane. In Fig. 10 we can see that modes are similar in shape when increasing the membrane size. The size of the modes/membranes change as well as their orientation as some of them rotate around the axis at the center of the membrane going out of the plane. That they change in size, intuitively, makes good sense as we change the size of the membrane itself. When looking at the RMS-values it can also be seen that they increase as the membrane size increase i.e. there is more out of plane movement for larger membranes than small ones. Looking at the modes with respect to RMS, the values decrease as the modes increase. The reason why the RMS increase with the membrane size makes intuitively good sense, the more atoms there is in the membrane the higher the RMS-value. Why the RMS decreases with a mode increase can explained by the fact the an increase in mode number increases the number of nodes in the membrane. The atoms placed at the nodes do not move around that much why their RMS values are close to 0. Looking at Fig. 10 we can also see that there is more green space i.e. node space as the modes increase no matter the membrane size. In the section Appendix C there is a plot Fig. 12 showing how the RMS-values decrease with the increase of mode numbers for the different hole sizes.

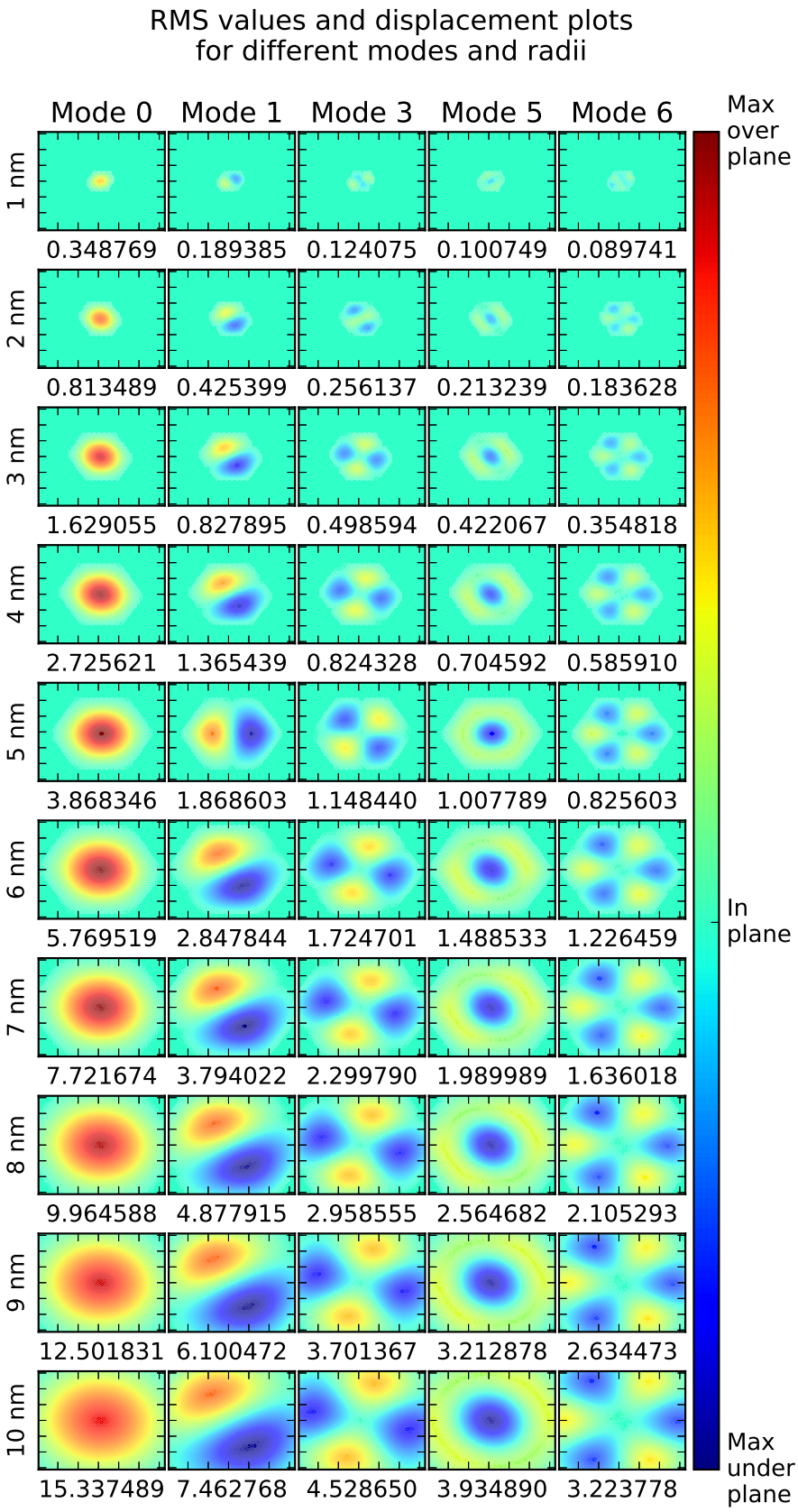

Figure 10: Plot showing how the modes of the membrane develops as the membrane size increases. Beneath each plot of the different modes are the corresponding RMS-values for that specific mode.

V. DISCUSSION

A. Size vs Frequency

In Section IV A 1 we achieved the result of $f = \frac{1}{x^{1.4}}$ when fitting to the data. The frequency for a classic membrane is proportional to the velocity of the wave divided by the radius of the membrane $\omega \propto \frac{v}{x}$. But for a layer of graphene the velocity of the wave is proportional to the frequency which then again is proportional to the inverse radius $v \propto \omega \propto \frac{1}{x}$. This means the frequency becomes proportional the the inverse radius squared $\omega \propto \frac{1/x}{x} = \frac{1}{x^2}$ In the real space of our lattice these two results corresponds to $\frac{1}{x}$ and $\frac{1}{x^2}$ This means that our value of the exponent $n = 1.4$ lies somewhere in between the frequency for a classic and a graphene memebrane. The reason for this is because of the very small membranes we have included in our data. The limit around the perimeter of the membrane is infinitely thin from a classic perspective, but for our system the limit consist of atoms which have non neglectible effects on the frequency, why we see a dependency of the radius which is in bewteen the classic and graphene models.

B. Interlayer interaction

In Section IV B we found that an ϵ corresponding to a substrate of silicon dioxide in the interlayer Lennard Jones potential, had an 10.1% average frequency deviation from the idealised clamped membrane. The average deviation are showed on Table III. Silicon dioxide turned out to be the best of the two candidates. As of such we now know that silicon dioxide is a suitable candidate as a substrate, if we were to simulate vibrational modes in a single layer membrane and then test the vibrational modes in the lab, where a substrate is needed. This is not a final result as other materials might be better suited and another research project could consist in determine the optimal theoretical ϵ value, and then find materials with such a value (or as close as).

C. Comparison of findings in other works

As mentioned in the introduction, we have seen that nanomembranes can contain localised modes on the microscale (see Fig. 1[2]). This tendency is clearly continued on the nanoscale as we find modes with similar behavior (Fig. 10). Comparing these results motivates experimental tests of membranes on the nanoscale in the lab.

D. Further analysis of the system

Following the analysis which has been worked through, the next natural step would be to look at the how the energy of a single mode disperses in a situation where mem-

brane is held at its maximum displacement for a given mode and then let go. After analysing such a situation we would follow up by expanding the model, adding another membrane to see how the coupling between them changes as distance between them is varied. The goal of this kind of analysis would be to find the distance between the membranes where there is no coupling between them.

E. Perspective: Going from theory to the lab

With the result from the analysis of the simulations, the next logical step would be to make tests in a lab to get measurements for comparison. Fig. 11 is a proposed experiment you could make to get this data. What is going to happen is that the laser will drive a oscillation in the membranes, which in turn will increase and decrease the distance between the graphene layer and the metal conductor. Because the substrate will act as a dielectric this creates a parallel-plate capacitor, which has a capacitance C that can be calculated as $C = \frac{\varepsilon \cdot A}{d}$ where ε where is the permittivity, A is the area that which the two plates share, and d is the distance between the plates. As we found out in the analysis of the two layered system the frequencies needed to drive the membranes were in the terahertz spectrum and therefore a terahertz laser would be needed to drive the membrane in the experimental setup. The calculated frequencies can be seen on Tables I and II Since the membranes effectively change the distance between the plates of the capacitor and thereby the capacitance they can be used to create an electric signal that match the analogue signal of the membrane. That is our proposal for getting data in the lab.

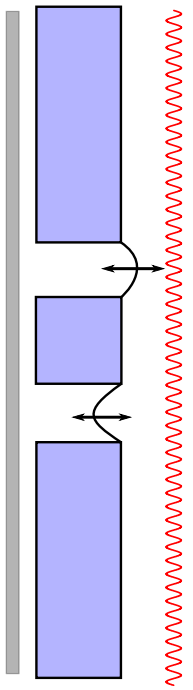

Figure 11: The setup of our proposed experiment, with the substrate (blue), the laser (red), the conductor (grey) and the membrane(black)

VI. CONCLUSION

Through our analysis of the both the clamped system and the two layer system we conclude that the membranes respond well to the frequencies at nano scale and that the

atomic behaviour of the modes correlated with the findings of similar simulations on micro scale membranes. The response frequencies for modes at nano scale were found to be in the Terahertz spectrum, which further motivates experimental tests. Because of these findings it reasonable to believe that a nano device which uses the membranes as gates, can be created.

ACKNOWLEDGMENTS

The authors would like to thank Tue Gunst and Mads Brandbyge at DTU Nanotech for their guidance and help. Furthermore we would like to thank QuantumWise for their swift technical support.

---

* E-mail at s164003@student.dtu.dk

† E-mail at s163965@student.dtu.dk

‡ E-mail at s163977@student.dtu.dk

§ Homepage of the Technical University of Denmark http://www.dtu.dk/english/;

Project Code Repository: https://github.com/rwiuff/Nanomechanics-for-graphene-membranes

[1] QuantumWise A/S, "Atomistix ToolKit version 2017.2," (2017).

[2] Dejan Davidovikj, Jesse J. Slim, Santiago J. Cartamil-Bueno, Herre S J Van Der Zant, Peter G. Steeneken, and Warner J. Venstra, "Visualizing the Motion of Graphene Nanodrums," Nano Letters **16**, 2768–2773 (2016), arXiv:1602.00135.

[3] Jingwei Bai, Xing Zhong, Shan Jiang, Yu Huang, and Xiangfeng Duan, "Graphene nanomesh," Nature Nanotechnology **5**, 190–194 (2010), arXiv:NIHMS150003.

[4] A. N. Cleland, *Springer* (2003) p. 440.

[5] See DTU HPC for more info: `http://www.hpc.dtu.dk/`.

[6] Due to a bug in ATKPython some numpy manipulation of the atomic bulk configuration was needed. See Section III D.

[7] L. Lindsay and D. A. Broido, "Optimized Tersoff and Brenner empirical potential parameters for lattice dynamics and phonon thermal transport in carbon nanotubes and graphene," Physical Review B - Condensed Matter and Materials Physics **81** (2010), 10.1103/PhysRevB.81.205441, arXiv:1003.2236.

[8] Binary datafile easily saved from or loaded into python via the Pickle package.

[9] Joachim Dahl Thomsen, Tue Gunst, Søren Schou Gregersen, Lene Gammelgaard, Bjarke Sørensen Jessen, David M.A. Mackenzie, Kenji Watanabe, Takashi Taniguchi, Peter Bøggild, and Timothy J. Booth, "Suppression of intrinsic roughness in encapsulated graphene," Physical Review B **96**, 014101 (2017).

[10] Irina V. Lebedeva, Andrey A. Knizhnik, Andrey M. Popov, Yurii E. Lozovik, and Boris V. Potapkin, "Interlayer interaction and relative vibrations of bilayer graphene," Physical Chemistry Chemical Physics **13**, 5687 (2011).

[11] Zhun-Yong Ong and Eric Pop, "Molecular dynamics simulation of thermal boundary conductance between carbon nanotubes and SiO2," Physical Review B **81**, 155408 (2010), arXiv:0910.2747.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

Appendices

Appendix A: Mechanics and dynamics of a simple system containing two atoms

To describe the dynamics of the system one must decompose the system into simpler elements in order to understand what happens for the system as a whole. Basically we want to describe the displacements each atom makes and then sum it up to some bigger form of system. The basis of a system containing only two atoms can be described a mass-spring system where each atom has a mass and where the *atomic interaction potential* works as the spring in between each atom. The interaction potential that we will be working with is called the *Lennard-Jones Potential*[4] (eq.1.3). The potential in its simplified an general form is given by

$$\phi(r) = -\frac{A}{r^6} + \frac{B}{r^{12}} \tag{A1}$$

where A is the strength of the attractive interaction , B is the strength of the repulsive interaction and r is the spacing between the atoms. The potential can be derived from the force between the atoms as the force is the derivative of the negative potential energy with respect to the atom spacing r[4] (eq.1.1).

$$f(r) \equiv -\frac{\mathrm{d}\phi}{\mathrm{d}r} \tag{A2}$$

However it is more convenient to work with the potential instead of the force.
The system is in a relaxed state when the distance between the atoms is equal to the *equilibrium distance* $r = r_0$. The equilibrium distance[4] (p.3) is given by

$$r_0 = \left(\frac{2B}{A}\right)^{\frac{1}{6}} \tag{A3}$$

Inserting (A3) in (A1) gives the minimum potential energy

$$\phi(r_0) = -\frac{A^2}{4B} \tag{A4}$$

When we work with larger systems we want the system to be relaxed i.e. in equilibrium first. However, for such systems it is not always the case hence why it must be relaxed with the required relaxation energy before we can apply external forces on it. Now that the small system has been described in its equilibrium state. We want to know what happens when we displace the atoms from their equilibrium position, that is to exert an external force on the atoms. As it is with all masses within classic mechanical theory, the atoms will start to

oscillate around their equilibrium position. Therefore a *harmonic potential approximation* is needed to describe the energy associated with such a displacement. This approximation only works for small displacements from equilibrium i.e. small external forces. This is because the approximation is made by using a *Taylor series* expansion[4] (eq.1.5) where only the first and second order terms are kept hence the approximation fails the further you get away from the equilibrium position. The expansion of the potential is given by

$$\begin{aligned}\phi(r) = \phi(r_0) + \left.\frac{\mathrm{d}\phi}{\mathrm{d}r}\right|_{r_0}(r-r_0) + \\ \frac{1}{2!}\left.\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2}\right|_{r_0}(r-r_0)^2 + \\ \frac{1}{3!}\left.\frac{\mathrm{d}^3\phi}{\mathrm{d}r^3}\right|_{r_0}(r-r_0)^3 + ...\end{aligned} \tag{A5}$$

Where the term $\left.\frac{\mathrm{d}\phi}{\mathrm{d}r}\right|_{r_0}$ equals 0 at equilibrium position. Dropping the higher order terms the series becomes the *harmonic potential approximation*[4] (eq.1.5)

$$\phi(r) \approx \phi(r_0) + \frac{1}{2}\left.\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2}\right|_{r_0}(r-r_0)^2 \tag{A6}$$

It can be seen that the potential energy depends on the displacement from equilibrium squared. When the atoms are in the presence of an external force, that is when they are displaced from equilibrium. The total potential energy[4] (p.4) becomes $U_{tot} = \phi(r) + \phi_{ext}(r)$, where $\phi_{ext}(r) = -f_{ext}r$. When a constant external force acts on the system, the point of equilibrium moves to a new location. At the location, the derivative of the total potential energy with respect to the displacement is zero $\frac{\mathrm{d}U_{tot}}{\mathrm{d}r} = 0$. Working on with this expression and using the result in (A6) the total energy in the point of the new equilibrium can described as

$$\begin{aligned}&\frac{\mathrm{d}U_{tot}}{\mathrm{d}r} = \frac{\mathrm{d}\phi(r)}{\mathrm{d}r} + \frac{\mathrm{d}\phi_{ext}(r)}{\mathrm{d}r} = 0 \\ &\frac{\mathrm{d}}{\mathrm{d}r}\left(\phi(r_0) + \frac{1}{2}\left.\frac{\mathrm{d}^2\phi(r)}{\mathrm{d}r^2}\right|_{r_0}(r-r_0)^2\right) - \\ &\frac{\mathrm{d}}{\mathrm{d}r}f_{ext}r = 0 \\ &\frac{\mathrm{d}}{\mathrm{d}r}\left(\frac{1}{2}\left.\frac{\mathrm{d}^2\phi(r)}{\mathrm{d}r^2}\right|_{r_0}(r-r_0)^2\right) - f_{ext} = 0\end{aligned} \tag{A7}$$

The square of the displacement gives the three terms $r^2 - 2r_0 r + r_0^2$ that only leaves $2r - 2r_0$ when derived with respect to r. This leaves

$$\begin{aligned} \frac{1}{2}\left.\frac{\mathrm{d}^2\phi(r)}{\mathrm{d}r^2}\right|_{r_0}(2r-2r_0) - f_{ext} &= 0 \\ \left.\frac{\mathrm{d}^2\phi(r)}{\mathrm{d}r^2}\right|_{r_0}(r-r_0) &= f_{ext} \\ r - r_0 &= \frac{1}{\mathrm{d}^2\phi(r)/\mathrm{d}r^2} f_{ext} \end{aligned} \tag{A8}$$

If the displacement is defined as $u \equiv r - r_0$ we can see that (A8) is an equivalent to Hooke's Law.

$$u \equiv r - r_0 = \frac{1}{\mathrm{d}^2\phi(r)/\mathrm{d}r^2} f_{ext} = \frac{1}{k} f_{ext} \tag{A9}$$

Where $k = \dfrac{\mathrm{d}^2\phi(r)}{\mathrm{d}r^2}$ acts as the spring constant. This last result emphasises that the system of two atoms can be viewed as a mass-spring system.

(A8) can also be used to find the equation of motion. As the force equals mass times acceleration the equation must satisfy

$$f_{ext} = \left.\frac{\mathrm{d}^2\phi(r)}{\mathrm{d}r^2}\right|_{r_0} u = ma = m\frac{\mathrm{d}^2}{\mathrm{d}t^2}u \tag{A10}$$

For the specific case of the two atoms the equation becomes

$$\mu\frac{\mathrm{d}^2}{\mathrm{d}t^2}u = -\left.\frac{\mathrm{d}^2\phi(r)}{\mathrm{d}r^2}\right|_{r_0} u \tag{A11}$$

Where μ is the reduced mass of the two atoms $\dfrac{1}{\mu} = \dfrac{1}{m_1} + \dfrac{1}{m_2}$ and the sign in front of the right side of (A11) is indicating that the force is the restoring force opposing the external force f_{ext}. The normal mode solution[4] (eq.1.17) to this equation is

$$u(t) = u_0 \cos(\omega_0 t + \varphi) = \mathrm{Re}\left(u_0 e^{-i\omega_0 t}\right) \tag{A12}$$

and the resonance frequency ω_0[4] (eq.1.18) is given by

$$\omega_0 = \sqrt{\frac{1}{\mu}\frac{\mathrm{d}^2\phi}{\mathrm{d}r^2}} = \sqrt{\frac{k}{\mu}} \tag{A13}$$

This concludes the example of a system of two atoms. The next step is to scale up this theory to two and three dimensions.

Appendix B: Boundary Conditions in Two Dimensions

When describing the motion of the atoms in the lattice as well as the forces acting upon them, some boundary conditions must be defined. Because the system is very large compared to the characteristic wavelength in the displacement field, periodic boundary conditions (PBC) work well. The actual boundaries as well as the specific normal modes are not of significant importance due the same fact of size difference in the system vs. the characteristic wavelength.

An intuitive approach to imagining PBC is to take the plane of the system and fold it in to at torus. All the four edges has now been put together pairwise. This means that any force acting on one atom which causes some form of motion is now translated on to the atom next to it.

For an arbitrary system in two dimensions of size X (side lenght) and with atomic spacing r_0, each axis has $\mathcal{N} = \frac{X}{r_0}$ atoms. The total amount of atoms in the sheet naturally becomes $N = \mathcal{N}^2$. The Bloch wavevector can be described as

$$\mathbf{q} = (q_x, q_y) \tag{B1}$$

where

$$(q_x, q_y) = \frac{2\pi}{X}(m, n) \tag{B2}$$

Here m, n are integers in the interval $-\frac{\mathcal{N}}{2}, +\frac{\mathcal{N}}{2}$. The range of q_x, q_y becomes $-\frac{\pi}{r_0}, +\frac{\pi}{r_0}$.

In the specific case of this study we work with a fixed boundary, more specific, a clamped boundary which means that no bending is allowed in the plane. A clamped boundary results in the need for numerical approximations in calculations of the dynamics. As the tools used for calculations are on a"per atom" atom basis, there wont be any need for finite-element models nor any partial differential equations solver (PDE).

Appendix C: Plots of RMS-values with varying mode numbers

Appendix D: Code Compendium

1. NanoSheetCreator.py

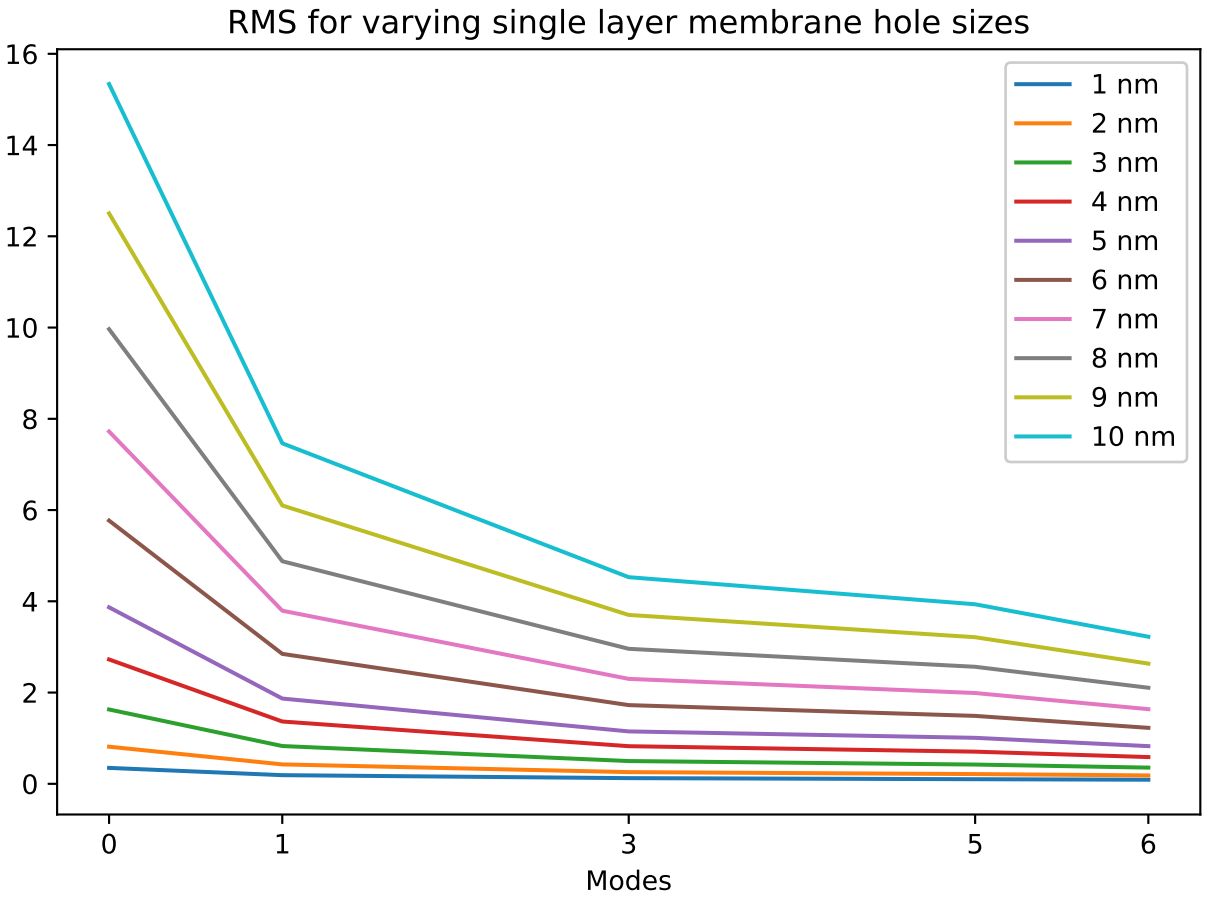

Figure 12: Figure showing how the RMS decreases as the modes increase. The y-axis are the RMS values for the 10 different membrane sizes as a function of the modes.

```
1  # ======================================================================
2  #
3  # By Frederik Grunnet Kristensen, Christoffer Vendelbo Sorensen
4  # & Rasmus Wiuff, s163977@student.dtu.dk
5  #
6  # Script for creating a 2 dimensional lattice with predefined
7  # hexagonal holes and sheet size, using ATKPython.
8  #
9  # ======================================================================
10
11 # ----------------------------------------------------------------
12 # Import Libraries
13 # ----------------------------------------------------------------
```

```
14 import numpy as np
15 from numpy import linalg as LA
16 from matplotlib import path
17 import matplotlib.pyplot as plt
18 import matplotlib.patches as patches
19 import math
20
21 # ------------------------------------------------------------
22 # Construct sheet
23 # ------------------------------------------------------------
24 # Input user for repetitions of generating vectors
25 nA = input("Repetitions along the A axis: ")
26 nB = input("Repetitions along the B axis: ")
27
28 # Create hexagonal bravais lattice
29 lattice = Hexagonal(a=2.4612 * Angstrom, c=20 * Angstrom)
30
31 # Set atomic elements
32 elements = [Carbon] * 2
33
34 # Place carbon atoms in unit cell
35 coordinates = [(0, 0, 0), (0.33333, 0.66667, 0)]
36
37 # Create unit cell as "sheet"
38 sheet = BulkConfiguration(lattice, elements,
39                           fractional_coordinates=coordinates)
40
41 # Repeat unit sheet along generating vectors
42 sheet = sheet.repeat(nA, nB, 1)
43 sheet = sheet.center()
44
45 # Extract cartesian sheet coordinates
```

```
46 sheetcoor = sheet.cartesianCoordinates()
47 # Remove units from coordinates
48 sheetcoor = sheetcoor / Ang
49
50 # Extract primitive vectors
51 pV = sheet.primitiveVectors()
52 pA = np.array(pV[0])
53 pB = np.array(pV[1])
54
55 # Print size of primitive vectors
56 sideA = float(LA.norm(pA) / Ang)
57 sideB = float(LA.norm(pB) / Ang)
58 size = "The graphene lattice is {:.3f} by {:.3f} Angstrom".format(sideA,
59                                                                  sideB)
60 print(size)
61
62 # Find center coordinates
63 pA1 = pA / 2
64 pB1 = pB / 2
65 cX0 = pA1[0] + pB1[0]
66 cY0 = pA1[1] + pB1[1]
67
68 # Convert sheet coordinates to Numpy array and delete z-coordinates
69 sheetcoor = np.array(sheetcoor)
70 sheetcoor = np.delete(sheetcoor, 2, 1)
71
72
73 # ------------------------------------------------------------
74 # Construct hexagonal tags
75 # ------------------------------------------------------------
76 # Loop variable and Information array
77 s = 1
```

```
78  info = np.array(
79      ["Tag name", "Center coordinate (x)", "Center coordinate (y)",
80       "Diameter", "Area"])
81
82  # Hexagon creation loop
83  while s == 1:
84      try:
85          # Prompt user for hexagon center offsets
86          cXI = float(
87              raw_input("Hexagon center X-offset (default: 0): "))
88      except ValueError:
89          cXI = 0
90      try:
91          cYI = float(
92              raw_input("Hexagon center Y-offset (default: 0): "))
93      except ValueError:
94          cYI = 0
95      # Prompt user for hexagon diameter (greatest)
96      d = float(input("Hexagon diameter: "))
97      # Find radius
98      r = d / 2
99      # Define center coordinates
100     cX = cX0 + cXI
101     cY = cY0 + cYI
102     # Calculate first point in hexagon
103     p1 = (cX + r, cY)
104     # Calculate second point in hexagon
105     p2 = (cX + math.cos(math.pi / 3) * r,
106           cY + math.sin(math.pi / 3) * r)
107     # Calculate third point in hexagon
108     p3 = (cX - math.cos(math.pi / 3) * r,
109           cY + math.sin(math.pi / 3) * r)
```

```
    # Calculate fourth point in hexagon
    p4 = (cX - r, cY)
    # Calculate fifth point in hexagon
    p5 = (cX - math.cos(math.pi / 3) * r,
          cY - math.sin(math.pi / 3) * r)
    # Calculate sixth point in hexagon
    p6 = (cX + math.cos(math.pi / 3) * r,
          cY - math.sin(math.pi / 3) * r)
    # Draw path connecting the six points
    p = path.Path([p1, p2, p3, p4, p5, p6, p1])

    # Find atoms inside hexagon path
        # and add them to Numpy array
    Test = p.contains_points(sheetcoor)
    atoms = np.array([])
    for i in range(Test.size):
        if Test[i] == True:
            atoms = np.append([atoms], [i])

    # Tag the found atoms
    atoms = atoms.astype(int)
    atoms = atoms.tolist()

    # Define sheet as a bulk configuration
    bulk_configuration = sheet

    # Calculate and print area of hexagon
    a = (cX + math.cos(math.pi / 3) * r) - (
        cX - math.cos(math.pi / 3) * r)
    A = (3 * math.sqrt(3) * a ** 2) / 2
    area = "Area of hexagon is {:.3f} Angstrom squared".format(A)
    print(area)
```

```
142     # Prompt user for tag name
143     tagname = raw_input("Input tag name: ")
144     # Add tag
145     bulk_configuration.addTags(tagname, atoms)
146     # Add tag information to Numpy array
147     instanceinfo = np.array(
148         [tagname, "{:.3f}".format(cX), "{:.3f}".format(cY),
149          "{:.3f}".format(d), "{:.3f}".format(A)])
150
151     info = np.vstack((info, instanceinfo))
152
153     # Ask user for figure graphic
154     savefig = raw_input("Save hole figure? [Y/N]: ")
155     if savefig == "Y":
156         # Create points with coordinates from the sheet
157         fig = plt.figure()
158         ax = fig.add_subplot(111)
159         xp = np.transpose(sheetcoor)[0]
160         yp = np.transpose(sheetcoor)[1]
161         plt.plot(xp, yp, ',')
162         # Create hexagonal figure
163         patch = patches.PathPatch(p, facecolor='orange', lw=2)
164         ax.add_patch(patch)
165         ax.set_xlim(0, np.amax(sheetcoor[:, 0]))
166         ax.set_ylim(0, np.amax(sheetcoor[:, 1]))
167         plt.axis('equal')
168         # Save figure with custom name
169         NM = int(d) / 10
170         NM = str(NM)
171         if len(NM) == 1:
172             NM = "".join(("0", NM))
173         savefigfilename = "{}{}".format(NM, "nm")
```

```
174         savefigformat = "pdf"
175         savefigfile = ".".join((savefigfilename, savefigformat))
176         plt.savefig(savefigfile, format='pdf')
177     elif savefig == "N":
178         savefig = "N"
179     # Prompt user to create more tags and restart loop
180     AH = raw_input("Add hole? [Y/N]: ")
181     if AH == "Y":
182         s = 1
183     elif AH == "N":
184         s = 0
185 
186 # ------------------------------------------------------------
187 # Repeat sheet
188 # ------------------------------------------------------------
189 R = raw_input("Repeat sheet? [Y/N]: ")
190 if R == "Y":
191     # Repetitions along first generating vector
192     Areps = int(input("A-vector repetitions: "))
193     sheet = sheet.repeat(Areps, 1, 1)
194     # Repetitions along second generating vector
195     Breps = int(input("B-vector repetitions: "))
196     sheet = sheet.repeat(1, Breps, 1)
197     # ---------------------- Process repeated tags ----------------------
198     # Calculate total repetitions
199     HI = Areps * Breps
200     # Create numpy array with tag names
201     tags = sheet.tags()
202     tags = np.array(list(tags))
203     # Create new tags for each original tag
204     for i in range(tags.size):
205         # Tag indices for given tag
```

```
206         tag = sheet.indicesFromTags(tags=tags[i])
207         tag = np.array(tag)
208         # Sort tag indices
209         tag = np.sort(tag, axis=None)
210         # Construct array with split tags
211         tagsize = np.array(tag.shape)
212         newtagsize = tagsize / HI
213         newtag = np.array([])
214         newtag = tag.reshape(HI, newtagsize)
215         newtag = newtag.astype(int)
216         # Add new tag to configuration
217         for j in range(HI):
218             newtagname = "".join((tags[i], str(j)))
219             sheet.addTags(newtagname, newtag[j, :])
220
221 elif R == "N":
222     R = "N"
223 # Prompt user for savename
224 savename = raw_input("Input nanosheet filename: ")
225 # Save bulk configuration as .hdf5 file
226 nlsave(savename, sheet)
227 txtformat = "txt"
228 savetxt = ".".join((savename, txtformat))
229 # Create textfile with created hexagon characteristics
230 np.savetxt(savetxt, info, fmt='%30s %30s %30s %30s %30s',
231            delimiter='|')
```

2. NanoMembraneCreator.py

```
# ========================================================================
#
# By Frederik Grunnet Kristensen, Christoffer Vendelbo Sorensen
# & Rasmus Wiuff, s163977@student.dtu.dk
#
# Script for creating a 2 dimensional double layered membrane with
# predefined hexagonal holes and sheet size, using ATKPython.
#
# ========================================================================

# ------------------------------------------------------------
# Import Libraries
# ------------------------------------------------------------
import numpy as np
from numpy import linalg as LA
from matplotlib import path
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import math

# ------------------------------------------------------------
# Construct sheet
# ------------------------------------------------------------
# Input user for repetitions of generating vectors
nA = input("Repetitions along the A axis: ")
nB = input("Repetitions along the B axis: ")

# Create hexagonal bravais lattice
lattice = Hexagonal(a=2.4612 * Angstrom, c=3.2 * Angstrom)

# Set atomic elements
elements = [Carbon] * 2
```

```
33
34  # Place carbon atoms in unit cell
35  coordinates = [(0, 0, 0), (0.33333, 0.66667, 0)]
36
37  # Create unit cell as "sheet"
38  sheet = BulkConfiguration(lattice, elements,
39                            fractional_coordinates=coordinates)
40
41  # Repeat unit sheet along generating vectors and create substrate
42  sheet = sheet.repeat(nA, nB, 2)
43
44  # Increase Lattice Parameters
45  new_a = float(LA.norm(sheet.primitiveVectors()[0]))
46  new_lattice = Hexagonal(a=new_a * Angstrom, c=20 * Angstrom)
47  sheet.setBravaisLattice(bravais_lattice=new_lattice)
48  sheet = sheet.center()
49
50  # Extract cartesian sheet coordinates
51  sheetcoor = sheet.cartesianCoordinates()
52  # Remove units from coordinates
53  sheetcoor = sheetcoor / Ang
54
55  # Extract primitive vectors
56  pV = sheet.primitiveVectors()
57  pA = np.array(pV[0])
58  pB = np.array(pV[1])
59
60  # Print size of primitive vectors
61  sideA = float(LA.norm(pA) / Ang)
62  sideB = float(LA.norm(pB) / Ang)
63  size = "The graphene lattice is {:.3f} by {:.3f} Angstrom".format(sideA,
64                                                                   sideB)
```

```
65  print(size)
66
67  # Find center coordinates
68  pA1 = pA / 2
69  pB1 = pB / 2
70  cX0 = pA1[0] + pB1[0]
71  cY0 = pA1[1] + pB1[1]
72
73  # Convert sheet coordinates to Numpy array and delete z-coordinates
74  sheetcoor = np.array(sheetcoor)
75  sheetcoor = np.delete(sheetcoor, 2, 1)
76
77  # ------------------------------------------------------------
78  # Construct hexagonal tags
79  # ------------------------------------------------------------
80  # Loop variable and Information array
81  s = 1
82  info = np.array(
83      ["Tag name", "Center coordinate (x)", "Center coordinate (y)",
84       "Diameter", "Area"])
85
86  # Hexagon creation loop
87  while s == 1:
88      try:
89          # Prompt user for hexagon center offsets
90          cXI = float(
91              raw_input("Hexagon center X-offset (default: 0): "))
92      except ValueError:
93          cXI = 0
94      try:
95          cYI = float(
96              raw_input("Hexagon center Y-offset (default: 0): "))
```

```
97      except ValueError:
98          cYI = 0
99      # Prompt user for hexagon diameter (greatest)
100     d = float(input("Hexagon diameter: "))
101     # Find radius
102     r = d / 2
103     # Define center coordinates
104     cX = cX0 + cXI
105     cY = cY0 + cYI
106     # Calculate first point in hexagon
107     p1 = (cX + r, cY)
108     # Calculate second point in hexagon
109     p2 = (cX + math.cos(math.pi / 3) * r,
110           cY + math.sin(math.pi / 3) * r)
111     # Calculate third point in hexagon
112     p3 = (cX - math.cos(math.pi / 3) * r,
113           cY + math.sin(math.pi / 3) * r)
114     # Calculate fourth point in hexagon
115     p4 = (cX - r, cY)
116     # Calculate fifth point in hexagon
117     p5 = (cX - math.cos(math.pi / 3) * r,
118           cY - math.sin(math.pi / 3) * r)
119     # Calculate sixth point in hexagon
120     p6 = (cX + math.cos(math.pi / 3) * r,
121           cY - math.sin(math.pi / 3) * r)
122     # Draw path connecting the six points
123     p = path.Path([p1, p2, p3, p4, p5, p6, p1])
124
125     # Find atoms inside hexagon path
126         # and add them to Numpy array
127     Test = p.contains_points(sheetcoor)
128     atoms = np.array([])
```

```
129     for i in range(Test.size):
130         if Test[i] == True:
131             atoms = np.append([atoms], [i])
132             i = i + 2
133 
134     # Tag the found atoms
135     atoms = atoms.astype(int)
136     atoms = atoms.tolist()
137     print(atoms)
138 
139     # Calculate and print area of hexagon
140     a = (cX + math.cos(math.pi / 3) * r) - (
141         cX - math.cos(math.pi / 3) * r)
142     A = (3 * math.sqrt(3) * a ** 2) / 2
143     area = "Area of hexagon is {:.3f} Angstrom squared".format(A)
144     print(area)
145     # Prompt user for tag name
146     tagname = raw_input("Input tag name: ")
147     # Add tag
148     sheet.addTags(tagname, atoms)
149     # Add tag information to Numpy array
150     instanceinfo = np.array(
151         [tagname, "{:.3f}".format(cX), "{:.3f}".format(cY),
152          "{:.3f}".format(d), "{:.3f}".format(A)])
153 
154     info = np.vstack((info, instanceinfo))
155 
156     # Ask user for figure graphic
157     savefig = raw_input("Save hole figure? [Y/N]: ")
158     if savefig == "Y":
159         # Create points with coordinates from the sheet
160         fig = plt.figure()
```

```
161         ax = fig.add_subplot(111)
162         xp = np.transpose(sheetcoor)[0]
163         yp = np.transpose(sheetcoor)[1]
164         plt.plot(xp, yp, ',')
165         # Create hexagonal figure
166         patch = patches.PathPatch(p, facecolor='orange', lw=2)
167         ax.add_patch(patch)
168         ax.set_xlim(0, np.amax(sheetcoor[:, 0]))
169         ax.set_ylim(0, np.amax(sheetcoor[:, 1]))
170         plt.axis('equal')
171         # Save figure with custom name
172         NM = int(d) / 10
173         NM = str(NM)
174         if len(NM) == 1:
175             NM = "".join(("0", NM))
176         savefigfilename = "{}{}".format(NM, "nm")
177         savefigformat = "pdf"
178         savefigfile = ".".join((savefigfilename, savefigformat))
179         plt.savefig(savefigfile, format='pdf')
180     elif savefig == "N":
181         savefig = "N"
182     # Prompt user to create more tags and restart loop
183     AH = raw_input("Add hole? [Y/N]: ")
184     if AH == "Y":
185         s = 1
186     elif AH == "N":
187         s = 0
188 
189 # Prompt user for savename
190 savename = raw_input("Input nanosheet filename: ")
191 # Save bulk configuration as .hdf5 file
192 nlsave(savename, sheet)
```

```
193 txtformat = "txt"
194 savetxt = ".".join((savename, txtformat))
195 # Create textfile with created hexagon characteristics
196 np.savetxt(savetxt, info, fmt='%30s %30s %30s %30s %30s',
197            delimiter='|')
```

3. 00_FixConstraints.py

```
1  # ======================================================================
2  #
3  # By Tue Gunst & Rasmus Wiuff
4  #
5  # Script for correcting constrain tags to omit ATKPython bug
6  #
7  # ======================================================================
8
9  # ----------------------------------------------------------------
10 # Import Libraries
11 # ----------------------------------------------------------------
12 from NanoLanguage import *
13 import numpy as np
14
15 # ----------------------------------------------------------------
16 # Set layermode
17 # ----------------------------------------------------------------
18 lm = 1
19 #lm = 0
20
21 # ----------------------------------------------------------------
22 # Load File
23 # ----------------------------------------------------------------
```

```
path = 'Sheet.hdf5'
bulk_configuration = nlread(path, BulkConfiguration)[-1]


# ------------------------------------------------------------
# Create Substrate tag
# ------------------------------------------------------------
if lm == 1:
    # Get atom coordinates
    coor = bulk_configuration.fractionalCoordinates()
    sub = np.array([])
    # Sort coordinates for 1 layer
    for i in range(coor.shape[0]):
        if coor[i, 2] < 0.50:
            sub = np.append([sub], [i])
    # Convert indices to list and define tag with constraints
    sub = sub.astype(int)
    sub = sub.tolist()
    bulk_configuration.addTags('Substrate', sub)

    constraints = bulk_configuration.indicesFromTags('Substrate')
elif lm == 0:
    # Define tag with constraints
    constraints = bulk_configuration.indicesFromTags('Substrate')



# ------------------------------------------------------------
# Fix Constraints
# ------------------------------------------------------------
# Make indices of atoms in configuration and get constraints.
atomindices = numpy.arange(bulk_configuration.numberOfAtoms())
# Move constrained atoms to end of list.
atomindices_new = numpy.delete(atomindices, constraints)
```

```
atomindices_new = numpy.append(atomindices_new, constraints)
constraints_new = atomindices[len(atomindices) - len(constraints)::]
new_xyz = bulk_configuration.cartesianCoordinates().inUnitsOf(Angstrom)
new_xyz = new_xyz[atomindices_new]
# Setup new configuration.
bulk_configuration._changeAtoms(positions=new_xyz * Angstrom)
constraints = constraints_new
bulk_configuration.addTags('FixedFinal', constraints)


# ------------------------------------------------------------
# Make Layer tags
# ------------------------------------------------------------
if lm == 1:
    coor = bulk_configuration.fractionalCoordinates()
    l1 = np.array([])
    l2 = np.array([])
    for i in range(coor.shape[0]):
        if coor[i, 2] < 0.50:
            l2 = np.append([l2], [i])
        elif coor[i, 2] > 0.50:
            l1 = np.append([l1], [i])
    l1 = l1.astype(int)
    l2 = l2.astype(int)
    l1 = l1.tolist()
    l2 = l2.tolist()
    bulk_configuration.addTags('Layer1', l1)
    bulk_configuration.addTags('Layer2', l2)
# Save configuration with fixed tags
nlsave('SheetFixed.hdf5', bulk_configuration)
nlprint(bulk_configuration)
```

4. 01_RelaxSheet.py

```
# ======================================================================
#
# By Tue Gunst (edited by Rasmus Wiuff)
#
# Script for optimising the geometry of
# a single layered membrane structure
# and define the atomic potential
#
# ======================================================================


# ----------------------------------------------------------------
# Import Libraries
# ----------------------------------------------------------------

from NanoLanguage import *

# ----------------------------------------------------------------
# Load File
# ----------------------------------------------------------------
path = 'SheetFixed.hdf5'
bulk_configuration = nlread(path, BulkConfiguration)[-1]

# ----------------------------------------------------------------
# Calculator
# ----------------------------------------------------------------
# Define potential for the atomic structure
potentialSet = Tersoff_C_2010()
calculator = TremoloXCalculator(parameters=potentialSet)
calculator.setVerletListsDelta(0.25 * Angstrom)

```

```
bulk_configuration.setCalculator(calculator)
nlprint(bulk_configuration)
bulk_configuration.update()

# ------------------------------------------------------------
# Optimize Geometry
# ------------------------------------------------------------
# Optimise the geometry and positions of atoms according to the
# defined potential
bulk_configuration = OptimizeGeometry(
    bulk_configuration,
    max_forces=0.001 * eV / Ang,
    max_stress=0.001 * eV / Ang**3,
    max_steps=200,
    max_step_length=0.2 * Ang,
    trajectory_filename=None,
    optimizer_method=LBFGS(),
    constrain_bravais_lattice=True,
)
nlsave('SheetRelaxed.hdf5', bulk_configuration)
nlprint(bulk_configuration)

# ------------------------------------------------------------
# Optimize Geometry
# ------------------------------------------------------------
# Optimise the geometry and positions of atoms according to the
# defined potential
# and with constrained atoms
# ------------------------------------------------------------
myConstraints = bulk_configuration.indicesFromTags('FixedFinal')
bulk_configuration = OptimizeGeometry(
    bulk_configuration,
```

```
    max_forces=0.001 * eV / Ang,
    max_stress=0.001 * eV / Ang**3,
    max_steps=200,
    max_step_length=0.2 * Ang,
    constraints=myConstraints,
    trajectory_filename=None,
    optimizer_method=LBFGS(),
    constrain_bravais_lattice=True,
)
# Save configuration
nlsave('SheetRelaxed.hdf5', bulk_configuration)
nlprint(bulk_configuration)
```

5. 01_LennardJonesRelax.py

```
# ============================================================================
#
# By Tue Gunst (edited by Rasmus Wiuff)
#
# Script for optimising the geometry of
# a doubled layered membrane structure
# and define the atomic potential
#
# ============================================================================


# ----------------------------------------------------------------
# Import Libraries
# ----------------------------------------------------------------
from NanoLanguage import *


# ----------------------------------------------------------------
```

```
# Load File
# ------------------------------------------------------------
path = 'SheetFixed.hdf5'
bulk_configuration = nlread(path, BulkConfiguration)[-1]


# ------------------------------------------------------------
# Lennard-Jones Calculator
# ------------------------------------------------------------
# Set zeta parameter like 0.31, 1.0, 10 for graphite, Si coupling
# (In SiO2) or something extremely strongly interacting, respectively.
Zeta = 1.0
# Set intralayer interaction.
potentialSet_layer1 = Tersoff_C_2010(tags='Layer1')
potentialSet_layer2 = Tersoff_C_2010(tags='Layer2')
# Interlayer interaction - Lennard-Jones type.
# Define a new potential for the interlayer interaction.
lj_interlayer_potential = TremoloXPotentialSet(
    name="InterLayerPotential")
# Add particle type definitions for both types.
lj_interlayer_potential.addParticleType(ParticleType.fromElement(
    Carbon, sigma=3.326 * Angstrom, epsilon=Zeta * 8.909 * meV))
# Add Lennard-Jones potentials between the carbon atoms
# of different layers.
lj_interlayer_potential.addPotential(
    LennardJonesPotential('C', 'C', r_cut=10.0 * Angstrom))
lj_interlayer_potential.setTags(['Layer1', 'Layer2'])


# Combine all 3 potential sets in a single calculator.
calculator = TremoloXCalculator(
    parameters=[potentialSet_layer1,
                potentialSet_layer2, lj_interlayer_potential])
calculator.setVerletListsDelta(0.25 * Angstrom)
```

```
49
50
51  bulk_configuration.setCalculator(calculator)
52  nlprint(bulk_configuration)
53  bulk_configuration.update()
54  myConstraints = bulk_configuration.indicesFromTags('FixedFinal')
55  myConstraints = map(int, myConstraints)
56  print(len(myConstraints))
57  # ------------------------------------------------------------
58  # Optimize Geometry
59  # ------------------------------------------------------------
60  # Optimise the geometry and positions of atoms
61  # according to the defined potential
62  bulk_configuration = OptimizeGeometry(
63      bulk_configuration,
64      max_forces=0.001 * eV / Ang,
65      max_stress=0.001 * eV / Ang**3,
66      max_steps=200,
67      max_step_length=0.2 * Ang,
68      constraints=myConstraints,
69      trajectory_filename='RelaxTraj.hdf5',
70      optimizer_method=LBFGS(),
71      constrain_bravais_lattice=True,
72  )
73  # Save configuration
74  nlsave('SheetRelaxed.hdf5', bulk_configuration)
75  nlprint(bulk_configuration)
```

6. 02_DynamicalMatrix.py

```
# ======================================================================
#
# By Rasmus Wiuff
#
# Script for calculating the Dynamical Matrix
#
# ======================================================================


# ----------------------------------------------------------------
# Load File
# ----------------------------------------------------------------
path = 'SheetRelaxed.hdf5'
bulk_configuration = nlread(path, BulkConfiguration)[-1]


# ----------------------------------------------------------------
# Dynamical Matrix
# ----------------------------------------------------------------
# Calculate the dynamical matrix with constrained atoms
myConstraints = bulk_configuration.indicesFromTags('FixedFinal')
myConstraints = map(int, myConstraints)
print(len(myConstraints))
dynamical_matrix = DynamicalMatrix(
    configuration=bulk_configuration,
    repetitions=[3, 3, 1],
    atomic_displacement=0.01 * Angstrom,
    acoustic_sum_rule=True,
    symmetrize=True,
    finite_difference_method=Central,
    force_tolerance=1e-08 * Hartree / Bohr**2,
    processes_per_displacement=1,
```

```
    log_filename_prefix='displacement_',
    use_wigner_seitz_scheme=False,
    constraints=myConstraints,
)
# Save Dynamical Matrix and configuration
nlsave('DynamicalMatrix.hdf5', dynamical_matrix)
nlsave('DynamicalMatrix.hdf5', bulk_configuration)
```

7. 03_SheetVibrations.py

```
# ======================================================================
#
# By Rasmus Wiuff
#
# Script for calculating the vibrational modes for a membrane
#
# ======================================================================

# ----------------------------------------------------------------
# Analysis from File. Load the Dynamical Matrix
# ----------------------------------------------------------------
path = 'DynamicalMatrix.hdf5'
configuration = nlread(path, BulkConfiguration)[-1]
dynamical_matrix = nlread(path, DynamicalMatrix)[-1]

# ----------------------------------------------------------------
# Vibrational Mode
# ----------------------------------------------------------------
vibrational_mode = VibrationalMode(
    configuration=configuration,
    dynamical_matrix=dynamical_matrix,
```

```
    kpoint_fractional=[0.0001, 0.0001, 0],
    mode_indices=None,
)
# Save configuration
nlsave('SheetVib.hdf5', vibrational_mode)
```

8. DataExtract.py

```
# ------------------------------------------------------------
# Load libraries and ProjectedPhononBandsDisplacement
# ------------------------------------------------------------
from NanoLanguage import *
from pylab import *
from MyAnalysisFunctions import ProjectedPhononBandsDisplacement
import pickle

# ------------------------------------------------------------
# Create data arrays
# ------------------------------------------------------------
myfile = np.array([])
configuration = np.array([])
dynamical_matrix = np.array([])
n_modes = np.array([])

# ------------------------------------------------------------
# Load data from dynamical matrices
# ------------------------------------------------------------
nof = 3
for i in range(nof + 1):
    if i == nof:
        break
```

```
    elif i >= 9:
        # Choose file
        myfile = np.append(myfile,
                           '{}DynamicalMatrix.hdf5'.format(i + 1))
    else:
        myfile = np.append(myfile,
                           '0{}DynamicalMatrix.hdf5'.format(i + 1))
    # Load configuration with calculator
    configuration = np.append(configuration, nlread(
        myfile[i], BulkConfiguration)[-1])

    # Load DynamicalMatrix
    dynamical_matrix = np.append(
        dynamical_matrix, nlread(myfile[i], DynamicalMatrix)[-1])

    # Vibrational state to project onto
    n_modes = np.append(n_modes, (len(
        configuration[i]) -
        len(dynamical_matrix[i].constraints())) * 3)

# Display loaded matrices and modes
print(
    '+---------------------------------------------------------------------+')
for i in range(nof):
    pstring = "| The File {} is loaded and contains {:4d} modes |".format(
        myfile[i], int(n_modes[i]))
    print(pstring)
print(
    '+---------------------------------------------------------------------+')

# ----------------------------------------------------------
# Make projection vectors
```

```
56 # ------------------------------------------------------------
57 projectionmode = 0
58 projection_vibration = {}
59 for i in range(nof):
60     constrained = dynamical_matrix[i].constraints()
61     tmp = numpy.zeros((n_modes[i]), dtype=float)
62     tmp[2::3] = 1  # project on z-motion
63     projection_vibration[i] = tmp
64     # print 'Projecting on: ', projection_vibration[i], n_modes[i]
65
66 # ------------------------------------------------------------
67 # Set qpoint
68 # ------------------------------------------------------------
69 fractional_qpoints = [0.0, 0.0, 0.0]
70
71 # ------------------------------------------------------------
72 # Calculate projected phonon dispersion
73 # ------------------------------------------------------------
74 # Create data dictionaries
75 qpoints = {}
76 frequency_list = {}
77 projection = {}
78 anti_projection = {}
79 RMS = {}
80
81 for i in range(nof):
82     print(
83         "Calculating the projected phonon dispersion for {}".format(
84             myfile[i]))
85     T = 300. * Kelvin
86     qpoints[i], frequency_list[i],
87     projection[i], anti_projection[i],
```

```
88      RMS[i] = ProjectedPhononBandsDisplacement(
89          configuration[i], dynamical_matrix[i],
90          fractional_qpoints, projection_vibration[i], temperature=T)
91
92  print("+========================+")
93  print("|    Saving datafiles    |")
94  print("--------------------------")
95  with open('qpoints.pickle', 'wb') as handle:
96      pickle.dump(qpoints, handle,
97                  protocol=pickle.HIGHEST_PROTOCOL)
98  print("| (1/5): Q-points        |")
99  with open('frequency_list.pickle', 'wb') as handle:
100     pickle.dump(frequency_list, handle,
101                 protocol=pickle.HIGHEST_PROTOCOL)
102 print("| (2/5): Frequency list  |")
103 with open('projection.pickle', 'wb') as handle:
104     pickle.dump(projection, handle,
105                 protocol=pickle.HIGHEST_PROTOCOL)
106 print("| (3/5): Projection      |")
107 with open('anti_projection.pickle', 'wb') as handle:
108     pickle.dump(anti_projection, handle,
109                 protocol=pickle.HIGHEST_PROTOCOL)
110 print("| (4/5): Anti projection |")
111 with open('RMS.pickle', 'wb') as handle:
112     pickle.dump(RMS, handle,
113                 protocol=pickle.HIGHEST_PROTOCOL)
114 print("| (5/5): RMS             |")
115 print("+========================+")
116 quit()
```

9. MyAnalysisFunctions.py

```
from NanoLanguage import *
from NL.Analysis.Mobility import fermiDistribution
from pylab import *
import itertools



def ProjectedPhononBandsDisplacement(configuration, dynamical_matrix, qpoints,
                                     projection_vibration,
                                     normalize_by_projection=True,
                                     temperature=300. * Kelvin):
    """ """
    """
    Private function to calculate projected phonon dispersions.
    Projects |eigenmode vectors| on input projection_vibration's
    (NB: displacement signs are removed).

    @param configuration            : The BulkConfiguration for which to
                                      calculate the mobility.
    @type                           : BulkConfiguration

    @param dynamical_matrix         : The dynamical matrix object.
    @type                           : DynamicalMatrix.

    @param qpoints                  : fractional q-points.

    @param projection_vibration     : The vibrational state to project onto.
    @type                           : array with length equal to the number
                                      of phonon modes.
    @default                        : No unit

```

```
    @normalize_by_projection        : Whether or not to normalize by norm
                                        of the projection vibrational state.
    @type                           : Boolean.
    @default                        : True

    # Tue Gunst, 2017
    """

    # Get dimensions
    n_modes = (len(configuration) - len(dynamical_matrix.constraints())) * 3
    n_qpoints = 1
    # Antiprojection, 1-P:
    anti_projection_vibration = numpy.ones(
        (n_modes), dtype=float) - projection_vibration

    # Calculate norm of projection vector:
    print("Calculating norm of projection vector")
    if normalize_by_projection == True:
        projection_norm = numpy.linalg.norm(projection_vibration) + 1e-12
        anti_projection_norm = numpy.linalg.norm(
            anti_projection_vibration) + 1e-12
    else:
        projection_norm = 1.0
        anti_projection_norm = 1.0

    # Masses in units of electron_mass.
    print("Calculating masses in units of electron mass")
    masses_atoms = numpy.array([e.atomicMass().inUnitsOf(
        electron_mass) for e in configuration.elements()])
    masses = numpy.repeat(masses_atoms, 3)
    masses_diagonal_matrix = numpy.diag(masses)
    # Unit factor.
```

```
63      unit_factor = (1.0 * (hbar**1 * electron_mass **
64                            (-0.5) * eV**(-0.5))).inUnitsOf(Ang)
65
66      # Thermal smearing.
67      thermal_smearing = (boltzmann_constant * temperature).inUnitsOf(eV)
68
69      def coth(x):
70          """
71          coth(x)
72          Uses numpy's tanh(x).
73          """
74          return 1. / tanh(x)
75
76      # Find constraints.
77      if len(dynamical_matrix.constraints()) > 0:
78          constrained_displacements = numpy.sort(
79              list(3 * dynamical_matrix.constraints()) + list(
80                  3 * dynamical_matrix.constraints() + 1) + list(
81                  3 * dynamical_matrix.constraints() + 2)).astype(int)
82      # Calculate projections for each mode and q.
83      print("Calculating projections for each mode")
84      frequency_list = numpy.zeros((n_qpoints, n_modes), dtype=float)
85      projection = numpy.zeros((n_qpoints, n_modes), dtype=float)
86      anti_projection = numpy.zeros((n_qpoints, n_modes), dtype=float)
87      RMS = numpy.zeros((n_qpoints, n_modes), dtype=float)
88      qpoint = qpoints[:]
89
90      # Calculate eigenvalues and eigenvectors from dynamical matrix.
91      print("Calculating eigenvalues and eigenvectors")
92      eigenvalues, eigenvectors = dynamical_matrix.phononEigensystem(qpoint)
93      frequency_list[0, :] = eigenvalues.inUnitsOf(eV)
94      print("Done")
```

```
95      for mode in numpy.arange(n_modes):
96          # Remove constrained displacements.
97          if len(dynamical_matrix.constraints()) > 0:
98              eigenvector = numpy.delete(
99                  eigenvectors[:, mode], constrained_displacements)
100         else:
101             eigenvector = eigenvectors[:, mode]
102
103         # Get characteristic length and mean-square mode displacement.
104         eigenvalues_ph_mode = eigenvalues[mode].inUnitsOf(eV)
105         inverse_characteristic_length = numpy.sqrt(2) * numpy.sqrt(
106             abs(eigenvalues_ph_mode)) * numpy.sqrt(numpy.real(
107                 numpy.dot(numpy.conj(eigenvectors[:, mode].T), numpy.dot(
108                     masses_diagonal_matrix,
109                     eigenvectors[:, mode])))) / unit_factor  # 1/Ang
110         ratio_thermal_frequency = abs(
111             eigenvalues_ph_mode) / (2 * thermal_smearing)
112         # 1./(inverse_characteristic_length**2)#
113         x_squared = coth(ratio_thermal_frequency) /
114             (inverse_characteristic_length**2)
115
116         CP = "({}/{}) Calculating projections for mode".format(mode + 1,
117                                                                n_modes)
118         print(CP)
119         projection[0, mode] = numpy.abs(
120             numpy.dot(numpy.abs(eigenvector),
121                       projection_vibration)) / projection_norm
122         anti_projection[0, mode] = numpy.abs(numpy.dot(
123             numpy.abs(eigenvector),
124             anti_projection_vibration)) / anti_projection_norm
125         RMS[0, mode] = numpy.sqrt(x_squared)
126
```

```
127     # Return result
128     return qpoints, frequency_list * eV, projection, anti_projection, RMS
```

10. ProjectionPlotter.py

```
1   # ======================================================================
2   #
3   # By Frederik Grunnet Kristensen, Christoffer Vendelbo Sorensen
4   # & Rasmus Wiuff, s163977@student.dtu.dk
5   #
6   # Script for creating projected mode plots and trendlines
7   #
8   # ======================================================================
9
10  # ----------------------------------------------------------------
11  # Load libraries, ProjectedPhononBandsDisplacement and switch
12  # matplotlib backend for HPC compatibility
13  # ----------------------------------------------------------------
14  import matplotlib.pyplot as plt
15  from NanoLanguage import *
16  from pylab import *
17  import pickle
18  import numpy as np
19  import numpy.polynomial.polynomial as poly
20  from scipy.optimize import curve_fit
21
22  # ----------------------------------------------------------------
23  # Create data arrays
24  # ----------------------------------------------------------------
25  myfile = np.array([])
26  configuration = np.array([])
```

```
dynamical_matrix = np.array([])
n_modes = np.array([])


# -----------------------------------------------------------
# Load data from dynamical matrices
# -----------------------------------------------------------
nof = 10
for i in range(nof + 1):
    if i == nof:
        break
    elif i >= 9:
        # Choose file
        myfile = np.append(myfile,
                           '{}nmDynamicalMatrix.hdf5'.format(i + 1))
    else:
        myfile = np.append(myfile,
                           '0{}nmDynamicalMatrix.hdf5'.format(i + 1))
    # Load configuration with calculator
    configuration = np.append(configuration, nlread(
        myfile[i], BulkConfiguration)[-1])

    # Load DynamicalMatrix
    dynamical_matrix = np.append(
        dynamical_matrix, nlread(myfile[i], DynamicalMatrix)[-1])

    # Vibrational state to project onto
    n_modes = np.append(n_modes,
                        (len(configuration[i]) -
                         len(dynamical_matrix[i].constraints())) * 3)
    # projection_vibration = numpy.zeros((n_modes),dtype=float)

# Display loaded matrices and modes
```

```
59 print('+--------------------------------------------------------+')
60 for i in range(nof):
61     pstring = "| The File {} contains {:4d} modes |".format(
62         myfile[i], int(n_modes[i]))
63     print(pstring)
64 print('+--------------------------------------------------------+')
65
66 # ------------------------------------------------------------
67 # Make projection vectors
68 # ------------------------------------------------------------
69 projection_vibration = {}
70 for i in range(nof):
71     constrained = dynamical_matrix[i].constraints()
72     tmp = numpy.zeros((n_modes[i]), dtype=float)
73     tmp[2::3] = 1  # project on z-motion
74     projection_vibration[i] = tmp
75     # print 'Projecting on: ', projection_vibration[i], n_modes[i]
76
77 # ------------------------------------------------------------
78 # Set qpoint
79 # ------------------------------------------------------------
80 fractional_qpoints = [0.0, 0.0, 0.0]
81
82 # ------------------------------------------------------------
83 # Load pickled data
84 # ------------------------------------------------------------
85 # Create data dictionaries
86 qpoints = {}
87 frequency_list = {}
88 projection = {}
89 anti_projection = {}
90 RMS = {}
```

```
# Load data into dictionaries
print("+=========================+")
print("|    Loading datafiles    |")
print("|-------------------------|")
with open('qpoints.pickle', 'rb') as handle:
    qpoints = pickle.load(handle)
print("| (1/5): Q-points         |")
with open('frequency_list.pickle', 'rb') as handle:
    frequency_list = pickle.load(handle)
print("| (2/5): Frequency list   |")
with open('projection.pickle', 'rb') as handle:
    projection = pickle.load(handle)
print("| (3/5): Projection       |")
with open('anti_projection.pickle', 'rb') as handle:
    anti_projection = pickle.load(handle)
print("| (4/5): Anti projection  |")
with open('RMS.pickle', 'rb') as handle:
    RMS = pickle.load(handle)
print("| (5/5): RMS              |")
print("+=========================+")
# ------------------------------------------------------------
# Define colormaps
# ------------------------------------------------------------
cmap, norm = cm.get_cmap('brg'), None  # 'hot','brg','seismic'


# ------------------------------------------------------------
# Plot projections
# ------------------------------------------------------------
figure()
# plot with color and without/with variable point size
myscale = {}
for i in range(nof):
```

```
123     myscale[i] = projection[i]  # /numpy.max(projection)
124
125 for i in range(nof):
126     # print numpy.max(projection[i])
127     plotmode = 1
128     if plotmode == 0:
129         scatter(numpy.repeat(np.array([i + 1]), n_modes[i]),
130                 frequency_list[i].inUnitsOf(
131             eV).flatten() * 1000, c=myscale[i], s=150, marker='o',
132             edgecolor='none', cmap=cmap, norm=norm)
133     elif plotmode == 1:
134         scatter(numpy.repeat(np.array([i + 1]), n_modes[i]),
135                 frequency_list[i].inUnitsOf(eV).flatten() * 1000,
136                 c=myscale[i], s=15 + myscale[i] * 120, marker='o',
137                 edgecolor='none', cmap=cmap, norm=norm)
138 # colorbar
139 cb = colorbar()  # colorbar(ticks=[-1, 0, 1], orientation='vertical')
140 cb.set_label('projection', fontsize=12)
141 tick_locator = matplotlib.ticker.MaxNLocator(nbins=10, prune=None)
142 cb.locator = tick_locator
143 cb.update_ticks()
144 # Set x-ticks
145 kticks = [w * 1 for w in range(nof + 2)]
146 ticklabels = ['%i nm' % w for w in range(nof + 1)]
147 xticks(kticks, ticklabels)
148 grid(kticks)
149 plt.subplots_adjust(left=None, bottom=None, right=0.97, top=None,
150                     wspace=None, hspace=None)
151
152 # ------------------------------------------------------------
153 # Fit 1/r^2 plot for mode 0
154 # ------------------------------------------------------------
```

```
155 x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
156 y = np.zeros(10)
157 for i in range(nof):
158     y[i] = np.sort(frequency_list[i].inUnitsOf(
159         eV).flatten() * 1000)[0]
160
161 # Define reciprocal function
162
163
164 def reciproc(x, a, n):
165     return a * (1 / (x**n))
166
167
168 # Fit reciprocal curve
169 popt, pcov = curve_fit(reciproc, x, y, p0=(1, 2))
170 perr = np.sqrt(np.diag(pcov))
171 x_rec0 = np.linspace(x[0], x[-1], num=len(x) * 10)
172 y_rec0 = reciproc(x_rec0, *popt)
173
174 # Create legend label
175 rec0label = r'$a={:.3f}\pm{:.3f}$' '\n' r'$n={:.3f}\pm{:.3f}$'.format(
176     popt[0], perr[0], popt[1], perr[1])
177
178 # ------------------------------------------------------------
179 # Fit 1/r^2 plot for mode 1
180 # ------------------------------------------------------------
181 x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
182 y = np.zeros(10)
183 for i in range(nof):
184     y[i] = np.sort(frequency_list[i].inUnitsOf(
185         eV).flatten() * 1000)[1]
186
```

```
187 # Define reciprocal function
188
189
190 def reciproc(x, a, n):
191     return a * (1 / (x**n))
192
193
194 # Fit reciprocal curve
195 popt, pcov = curve_fit(reciproc, x, y, p0=(1, 2))
196 perr = np.sqrt(np.diag(pcov))
197 x_rec1 = np.linspace(x[0], x[-1], num=len(x) * 10)
198 y_rec1 = reciproc(x_rec1, *popt)
199
200 # Create legend label
201 rec1label = r'$a={:.3f}\pm{:.3f}$' '\n' r'$n={:.3f}\pm{:.3f}$'.format(
202     popt[0], perr[0], popt[1], perr[1])
203
204 # ----------------------------------------------------------------
205 # Show or save plots
206 # ----------------------------------------------------------------
207 # Print menu
208 rec = r'$a \cdot \frac{1}{x^n}$'
209 showsave = 0
210 menu = np.array(["Show plot                 ",
211                  "Show zoomed and fitted plot",
212                  "Save plot                 ",
213                  "Save zoomed and fitted plot"])
214 while showsave == 0:
215     print("+================================+")
216     print("|       Show or save plot?       |")
217     print("|--------------------------------|")
218     for i in range(len(menu)):
```

```
219         print("| {:d}. {:s} |".format(i + 1, menu[i]))
220     print("+================================+")
221     while not (np.any(showsave == np.arange(len(menu)) + 1)):
222         while True:
223             try:
224                 showsave = float(input("Choose an option: "))
225                 break
226             except ValueError:
227                 print("Only integers accepted")
228                 pass
229             except NameError:
230                 print("Only integers accepted")
231             except TypeError:
232                 print("Only integers accepted")
233 # Define graph ranges
234 ymin, ymax = -10, 20
235 xmin, xmax = 0.5326636405016747, 10.565891401841617
236 xlim(xmin, xmax)
237 # Show plot
238 if showsave == 1:
239     ylabel('$\omega$ [meV]')
240     ymin, ymax = -25, 225
241     ylim(ymin, ymax)
242     plt.show()
243 # Show fitted plot
244 elif showsave == 2:
245     ylim(ymin, ymax)
246     ylabel('$\omega$ [meV]')
247     ax = plt.gca()
248     ax.plot(x_rec0, y_rec0, 'b-', label=rec0label)
249     ax.plot(x_rec1, y_rec1, 'r-', label=rec1label)
250     ax.legend(loc=3, title=rec)
```

```
251     plt.show()
252 # Save plot
253 elif showsave == 3:
254     ylabel('$\omega$ [meV]')
255     print("+====================================+")
256     print("|              Saving plots          |")
257     print("|------------------------------------|")
258     print("| (1/2): FrequencyModeProjections.eps |")
259     savefig('FrequencyModeProjections.eps')
260     print("| (2/2): FrequencyModeProjections.png |")
261     savefig('FrequencyModeProjections.png')
262     print("+====================================+")
263 # Save fitted plot
264 elif showsave == 4:
265     ylim(ymin, ymax)
266     ylabel('$\omega$ [meV]')
267     ax = plt.gca()
268     ax.plot(x_rec0, y_rec0, 'b-', label=rec0label)
269     ax.plot(x_rec1, y_rec1, 'r-', label=rec1label)
270     ax.legend(loc=3, title=rec)
271     print("+============================================+")
272     print("|                 Saving plots               |")
273     print("|--------------------------------------------|")
274     print("| (1/2): FrequencyModeProjectionsZoomFit.eps |")
275     savefig('FrequencyModeProjectionsZoomFit.eps')
276     print("| (2/2): FrequencyModeProjectionsZoomFit.png |")
277     savefig('FrequencyModeProjectionsZoomFit.png')
278     print("+============================================+")
```

11. ZetaPlotter.py

```
# ------------------------------------------------------------
# Load libraries, ProjectedPhononBandsDisplacement and switch
# matplotlib backend
# ------------------------------------------------------------
import matplotlib.pyplot as plt
from matplotlib import gridspec
import matplotlib.lines as mlines
from NanoLanguage import *
from pylab import *
import pickle
import numpy as np
import numpy.polynomial.polynomial as poly
from scipy.optimize import curve_fit


# ------------------------------------------------------------
# Create data arrays
# ------------------------------------------------------------
myfile = np.array([])
configuration = np.array([])
dynamical_matrix = np.array([])
n_modes = np.array([])


# ------------------------------------------------------------
# Load data from dynamical matrices
# ------------------------------------------------------------
nof = 3
for i in range(nof + 1):
    if i == nof:
        break
    elif i >= 9:
```

```
31          # Choose file
32          myfile = np.append(myfile,
33                             '{}DynamicalMatrix.hdf5'.format(i + 1))
34      else:
35          myfile = np.append(myfile,
36                             '0{}DynamicalMatrix.hdf5'.format(i + 1))
37      # Load configuration with calculator
38      configuration = np.append(configuration, nlread(
39          myfile[i], BulkConfiguration)[-1])
40
41      # Load DynamicalMatrix
42      dynamical_matrix = np.append(
43          dynamical_matrix, nlread(myfile[i], DynamicalMatrix)[-1])
44
45      # Vibrational state to project onto
46      n_modes = np.append(n_modes, (len(configuration[i]) - len(
47          dynamical_matrix[i].constraints())) * 3)
48
49  # ------------------------------------------------------------
50  # Load data from clamped reference dynamical matrix
51  # ------------------------------------------------------------
52  RFDM = '05nmDynamicalMatrix.hdf5'
53  RFconfiguration = nlread(RFDM, BulkConfiguration)[-1]
54  RFdynamical_matrix = nlread(RFDM, DynamicalMatrix)[-1]
55  RFn_modes = ((len(RFconfiguration) - len(
56      RFdynamical_matrix.constraints())) * 3)
57
58  # Display loaded matrices and modes
59  print('+-----------------------------------------------------+')
60  for i in range(nof):
61      pstring = "| The File {} contains {:4d} modes |".format(
62          myfile[i], int(n_modes[i]))
```

```
63      print(pstring)
64  print('+-----------------------------------------------------+')
65
66  print('+-----------------------------------------------------+')
67  pstring = "| The File {} contains {:4d} modes |".format(
68      RFDM, int(RFn_modes))
69  print(pstring)
70  print('+-----------------------------------------------------+')
71
72  # quit()
73  # -----------------------------------------------------------
74  # Make projection vectors
75  # -----------------------------------------------------------
76  projection_vibration = {}
77  for i in range(nof):
78      constrained = dynamical_matrix[i].constraints()
79      tmp = numpy.zeros((n_modes[i]), dtype=float)
80      tmp[2::3] = 1  # project on z-motion
81      projection_vibration[i] = tmp
82      # print 'Projecting on: ', projection_vibration[i], n_modes[i]
83
84  # -----------------------------------------------------------
85  # Set qpoint
86  # -----------------------------------------------------------
87  fractional_qpoints = [0.0, 0.0, 0.0]
88
89  # -----------------------------------------------------------
90  # Create data dictionaries
91  # -----------------------------------------------------------
92  qpoints = {}
93  frequency_list = {}
94  projection = {}
```

```
95  anti_projection = {}
96  RMS = {}
97  RFfrequency_list = {}
98  RFprojection = {}
99
100 print("+=========================+")
101 print("|    Loading datafiles    |")
102 print("|-------------------------|")
103 with open('qpoints.pickle', 'rb') as handle:
104     qpoints = pickle.load(handle)
105 print("| (1/5): Q-points         |")
106 with open('frequency_list.pickle', 'rb') as handle:
107     frequency_list = pickle.load(handle)
108 print("| (2/5): Frequency list   |")
109 with open('projection.pickle', 'rb') as handle:
110     projection = pickle.load(handle)
111 print("| (3/5): Projection       |")
112 with open('anti_projection.pickle', 'rb') as handle:
113     anti_projection = pickle.load(handle)
114 print("| (4/5): Anti projection  |")
115 with open('RMS.pickle', 'rb') as handle:
116     RMS = pickle.load(handle)
117 print("| (5/5): RMS              |")
118 print("+=========================+")
119
120 print("+=========================+")
121 print("|    Loading datafiles    |")
122 print("|-------------------------|")
123 with open('Referencefrequency_list.pickle', 'rb') as handle:
124     RFfrequency_list = pickle.load(handle)
125 print("| (1/2): Frequency list   |")
126 with open('ReferenceProjection.pickle', 'rb') as handle:
```

```
127     RFprojection = pickle.load(handle)
128 print("| (2/2): Projection       |")
129 print("+=========================+")
130 # ----------------------------------------------------------------
131 # Define colormaps
132 # ----------------------------------------------------------------
133 cmap, norm = cm.get_cmap('brg'), None  # 'hot','brg','seismic'
134 
135 # ----------------------------------------------------------------
136 # Plot projections
137 # ----------------------------------------------------------------
138 Index = np.array([0, 1, 3, 5, 6])
139 ymin, ymax = -1.5, 15
140 fig = plt.figure(figsize=(8, 6))
141 gs = gridspec.GridSpec(1, 2, width_ratios=[1, 6])
142 gs.update(wspace=0.03)
143 
144 ax1 = plt.subplot(gs[0])
145 myscale = RFprojection[4]  # /numpy.max(projection)
146 plt.scatter(numpy.repeat(np.array([1]), RFn_modes),
147             RFfrequency_list[4].inUnitsOf(eV).flatten() * 1000, c=myscale,
148             s=15 + myscale * 120, marker='o', edgecolor='none',
149             cmap=cmap, norm=norm)
150 for i in range(5):
151     j = Index[i]
152     plt.axhline(y=RFfrequency_list[4].inUnitsOf(eV).flatten()[j] * 1000)
153     print(RFfrequency_list[4].inUnitsOf(eV).flatten()[j])
154 # Set x-ticks = the symmetry points
155 kticks = [1]
156 ticklabels = ['5nm']
157 xticks(kticks, ticklabels)
158 grid(kticks)
```

```
ylabel('$\omega$ [meV]')
ylim(ymin, ymax)


ax2 = plt.subplot(gs[1])
myscale = {}
for i in range(nof):
    myscale[i] = projection[i]  # /numpy.max(projection)
for i in range(nof):
    # print numpy.max(projection[i])
    plt.scatter(numpy.repeat(np.array([i + 1]), n_modes[i]),
                frequency_list[i].inUnitsOf(eV).flatten(
    ) * 1000, c=myscale[i], s=15 + myscale[i] * 120, marker='o',
        edgecolor='none', cmap=cmap, norm=norm)
for i in range(5):
    j = Index[i]
    plt.axhline(y=RFfrequency_list[4].inUnitsOf(eV).flatten()[j] * 1000)
# Set x-ticks = the symmetry points
kticks = [w * 1 for w in range(nof + 2)]
ticklabels = ['0', '0.31', '1.00', '10.0']
xticks(kticks, ticklabels)
grid(kticks)
xlabel('$\epsilon$ [8.909 meV]')
ylim(ymin, ymax)
plt.setp(ax2.get_yticklabels(), visible=False)
plt.subplots_adjust(left=0.07, bottom=None, right=0.88, top=None,
                    wspace=None, hspace=None)
blue_line = mlines.Line2D(
    [], [], color='blue',
    label=r'Frequency for mode $1$, $2$, $4$, $6$ and $7$')
plt.legend(handles=[blue_line], loc=8)
# colorbar
# colorbar(ticks=[-1, 0, 1], orientation='vertical')
```

```
191 cax = plt.axes([0.89, 0.1, 0.03, 0.8])
192 cb = plt.colorbar(cax=cax)
193 cb.set_label('projection', fontsize=12)
194 # ------------------------------------------------------------
195 # Show or save plots
196 # ------------------------------------------------------------
197 showsave = 0
198 menu = np.array(["Show plot                    ",
199                  "Save plot                    "])
200 while showsave == 0:
201     print("+================================+")
202     print("|       Show or save plot?       |")
203     print("|--------------------------------|")
204     for i in range(len(menu)):
205         print("| {:d}. {:s} |".format(i + 1, menu[i]))
206     print("+================================+")
207     while not (np.any(showsave == np.arange(len(menu)) + 1)):
208         while True:
209             try:
210                 showsave = float(input("Choose an option: "))
211                 break
212             except ValueError:
213                 print("Only integers accepted")
214                 pass
215             except NameError:
216                 print("Only integers accepted")
217             except TypeError:
218                 print("Only integers accepted")
219 if showsave == 1:
220     plt.show()
221 elif showsave == 2:
222     ylim(ymin, ymax)
```

```
223     print("+====================================+")
224     print("|              Saving plots          |")
225     print("|------------------------------------|")
226     print("| (1/2): ZetaModeProjections.eps |")
227     savefig('ZetaModeProjections.eps')
228     print("| (2/2): ZetaModeProjections.png |")
229     savefig('ZetaModeProjections.png')
230     print("+====================================+")
```

12. 2DdataExtract.py

```
1  # ------------------------------------------------------------
2  # Load libraries
3  # ------------------------------------------------------------
4  import matplotlib.pyplot as plt
5  from NanoLanguage import *
6  from pylab import *
7  import pickle
8  import numpy as np
9
10 # ------------------------------------------------------------
11 # Create data arrays
12 # ------------------------------------------------------------
13 configuration = np.array([])
14 vibrationalmodes = np.array([])
15
16 # ------------------------------------------------------------
17 # Load data from clamped reference dynamical matrix
18 # ------------------------------------------------------------
19 nof = 3
20 myfile = np.array([])
```

```
21 for i in range(nof + 1):
22     if i == nof:
23         break
24     elif i >= 9:
25         # Choose file
26         myfile = np.append(myfile,
27                            '{}SheetVib.hdf5'.format(i + 1))
28     else:
29         myfile = np.append(myfile,
30                            '0{}SheetVib.hdf5'.format(i + 1))
31
32 RFSV = '05nmSheetVib.hdf5'
33 RFvibrationalmodes = nlread(RFSV, VibrationalMode)[-1]
34
35 ClampedModes = {}
36 ZModes = {}
37 T = 300000
38 Index = [0, 1, 3, 5, 6]
39 coor = np.array([])
40 for i in range(5):
41     print('+---------------------------------+')
42     pstring = "| Extracting clamped mode {:2d} of {:2d} |".format(
43         int(1 + i), int('5'))
44     print(pstring)
45     print('+---------------------------------+')
46     VM = RFvibrationalmodes.movie(
47         mode_index=Index[i], temperature=T * Kelvin)
48     ClampedModes[i] = VM.lastImage()
49     VM = np.array([])
50 RFvibrationalmodes = np.array([])
51 ModeIndex = [2, 3, 5, 7, 8]
52 ImageIndex = np.array([[9, 19, 9, 9, 19],
```

```
                        [9, 9, 19, 19, 9],
                        [19, 19, 19, 19, 9]])
coor = np.array([])
c = 0
for j in range(3):
    vibrationalmodes = nlread(myfile[j], VibrationalMode)[-1]
    for i in range(5):
        c = c + 1
        print('+---------------------------------+')
        pstring = "| Extracting mode         {:2d} of {:2d} |".format(
            int(c), int('15'))
        print(pstring)
        print('+---------------------------------+')
                # Extract the trajectory from VibrationalMode at mode i and
                # temperature T
        VM = vibrationalmodes.movie(
            mode_index=ModeIndex[i], temperature=T * Kelvin)
                # Extract configuration from image j,i in trajectory
        ZModes[c] = VM.image(image_index=ImageIndex[j, i])
        VM = np.array([])



print("+======================+")
print("|   Saving datafiles   |")
print("------------------------")
with open('ClampedModes.pickle', 'wb') as handle:
    pickle.dump(ClampedModes, handle,
                protocol=pickle.HIGHEST_PROTOCOL)
print("| (1/2): ClampedModes  |")
with open('ZModes.pickle', 'wb') as handle:
    pickle.dump(ZModes, handle,
                protocol=pickle.HIGHEST_PROTOCOL)
```

```
85 print("| (2/2): ZModes        |")
86 print("+======================+")
```

13. 2DmodePlotter.py

```
1  # ------------------------------------------------------------
2  # Load libraries, ProjectedPhononBandsDisplacement and switch
3  # matplotlib backend
4  # ------------------------------------------------------------
5  import matplotlib.pyplot as plt
6  from matplotlib import gridspec
7  import matplotlib.colors as colors
8  from NanoLanguage import *
9  from pylab import *
10 import pickle
11 import numpy as np
12
13 # ------------------------------------------------------------
14 # Create data arrays
15 # ------------------------------------------------------------
16 configuration = np.array([])
17 vibrationalmodes = np.array([])
18
19 # ------------------------------------------------------------
20 # Load data from pickle files
21 # ------------------------------------------------------------
22 ZModes = {}
23 ClampedModes = {}
24
25 print("+======================+")
26 print("|   Loading datafiles   |")
```

```
print("|----------------------|")
with open('ClampedModes.pickle', 'rb') as handle:
    ClampedModes = pickle.load(handle)
print("| (1/2): Clamped Modes  |")
with open('ZModes.pickle', 'rb') as handle:
    ZModes = pickle.load(handle)
print("| (2/2): ZModes         |")
print("+======================+")


# ------------------------------------------------------------
# Define colormaps
# ------------------------------------------------------------
cmap = cm.get_cmap('brg')
v_min = 0
v_max = 0
for i in range(len(ClampedModes)):
    if v_min > np.min((
            ClampedModes[i].cartesianCoordinates().inUnitsOf(Ang)[:, 2]) - 10):
        v_min = np.min(
            (ClampedModes[i].cartesianCoordinates().inUnitsOf(Ang)[:, 2]) - 10)
    else:
        v_min = v_min
for i in range(1, len(ZModes)):
    coor = ZModes[i]
    coor = coor.cartesianCoordinates().inUnitsOf(Ang)
    rmindex = np.array([])
    for i in range(coor.shape[0]):
        if coor[i, 2] < 10:
            rmindex = np.append(rmindex, [i])
    coor = np.delete(coor, rmindex, axis=0)
    z = (coor[:, 2]) - 11.8747
    if v_min > np.min(z):
```

```
59          v_min = np.min(z)
60      else:
61          v_min = v_min
62  for i in range(len(ClampedModes)):
63      if v_max < np.max((
64              ClampedModes[i].cartesianCoordinates().inUnitsOf(Ang)[:, 2]) - 10):
65          v_max = np.max(
66              (ClampedModes[i].cartesianCoordinates().inUnitsOf(Ang)[:, 2]) - 10)
67      else:
68          v_max = v_max
69  for i in range(1, len(ZModes)):
70      if v_max < np.max((
71              ZModes[i].cartesianCoordinates().inUnitsOf(Ang)[:, 2]) - 11.8747):
72          v_max = np.max(
73              (ZModes[i].cartesianCoordinates().inUnitsOf(Ang)[:, 2]) - 11.8747)
74      else:
75          v_max = v_max
76  levels = np.linspace(v_min, v_max, 1000)
77  # ------------------------------------------------------------
78  # Plot RMS
79  # ------------------------------------------------------------
80  Index = [0, 1, 3, 5, 6]
81  np.set_printoptions(threshold='nan')
82  fig = plt.figure(figsize=(10.15, 8.8))
83  gs = gridspec.GridSpec(4, 5)
84  gs.update(wspace=0.03, hspace=0.2)
85  ax1 = {}
86  ax2 = {}
87  coor = np.array([])
88  for i in range(5):
89      print('+--------------------------------+')
90      pstring = "| Plotting clamped plot {:2d} of {:2d} |".format(
```

```
91          int(1 + i), int('5'))
92      print(pstring)
93      print('+--------------------------------+')
94      ax1[i] = plt.subplot(gs[0, i])
95      coor = ClampedModes[i]
96      coor = coor.cartesianCoordinates().inUnitsOf(Ang)
97      x = coor[:, 0]
98      y = coor[:, 1]
99      z = coor[:, 2]
100     z = (coor[:, 2]) - 10
101     plt.tricontourf(x, y, z, vmin=v_min, vmax=v_max, levels=levels)
102     plt.title('Mode {}'.format(Index[i]))
103     plt.ylim(-31.1505, 31.1505)
104     plt.xlim(50.464, 112.765)
105
106 RMS = [3.86834624,  1.86860316, 1.14843955, 1.0077887,  0.82560262]
107 for i in range(5):
108     plt.setp(ax1[i].get_yticklabels(), visible=False)
109 for i in range(5):
110     plt.setp(ax1[i].get_xticklabels(), visible=False)
111     ax1[i].set_xlabel('{:.6f}'.format(RMS[i]))
112 ax1[0].set_ylabel('Clamped')
113
114 coor = np.array([])
115 c = 0
116 for j in range(3):
117     for i in range(5):
118         c = c + 1
119         print('+--------------------------------+')
120         pstring = "| Plotting mode plot    {:2d} of {:2d} |".format(
121             int(c), int('15'))
122         print(pstring)
```

```
123         print('+--------------------------------+')
124         ax2[c - 1] = plt.subplot(gs[j + 1, i])
125         coor = ZModes[c]
126         coor = coor.cartesianCoordinates().inUnitsOf(Ang)
127         rmindex = np.array([])
128         for i in range(coor.shape[0]):
129             if coor[i, 2] < 10:
130                 rmindex = np.append(rmindex, [i])
131         coor = np.delete(coor, rmindex, axis=0)
132         x = coor[:, 0]
133         y = coor[:, 1]
134         z = (coor[:, 2]) - 11.8747
135         plt.tricontourf(x, y, z, vmin=v_min, vmax=v_max, levels=levels)
136         plt.ylim(-30.5601, 30.5601)
137         plt.xlim(20.5441, 82.8171)
138 
139 RMS = [4.95659411, 2.38986229, 1.43190479, 1.23333372,  1.0212438,
140        4.35913308, 2.14893905, 1.27473199, 1.08697788,  0.90374589,
141        1.97812165, 1.20437783, 0.78754608, 0.69529033,  0.63039666]
142 yl = ['$\epsilon = 0.31$', '$\epsilon = 1.00$', '$\epsilon = 10.0$']
143 rangeindex = [0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 10, 0, 0, 0, 0]
144 c = 0
145 for i in range(15):
146     if i != rangeindex[i]:
147         plt.setp(ax2[i].get_yticklabels(), visible=False)
148     else:
149         plt.setp(ax2[i].get_yticklabels(), visible=False)
150         ax2[i].set_ylabel('{}'.format(yl[c]))
151         c = c + 1
152 for i in range(15):
153     plt.setp(ax2[i].get_xticklabels(), visible=False)
154     ax2[i].set_xlabel('{:.6f}'.format(RMS[i]))
```

```
155
156 plt.subplots_adjust(left=0.06, bottom=None, right=0.87, top=None,
157                     wspace=None, hspace=None)
158
159 cax = plt.axes([0.88, 0.1, 0.03, 0.8])
160 cb = plt.colorbar(ticks=[v_min, 0, v_max], cax=cax)
161 #cb.set_label('Out of plane displacement', fontsize=12)
162 cb.ax.set_yticklabels(
163     ['Max\nunder\nplane', 'In\nplane', 'Max\nover\nplane'], fontsize=14)
164 fig.suptitle(
165     'RMS values and displacement plots for different modes and substrates',
166     size=18)
167
168 plt.show()
```