Computer simulations of vibrations induced by the stretching of polycrystalline graphene

Master Thesis

Primary Supervisor
Dr. R. A. Duine
Institute for Theoretical Physics
Utrecht University

Secondary Supervisor
Dr. G. T. Barkema
Information and Computing Sciences
Utrecht University

Daily Supervisor
PhD candidate Federico D'Embrosio
Information and Computing Sciences
Utrecht University

Patrick van Dieten

November 29, 2019

Abstract

Computer simulations are used to generate realistic sample configurations of crystalline and polycrystalline graphene. This is done using an empirical potential, and structural evolution via bond transposition moves. In particular, the project focuses on efficient simulation techniques, such as the FIRE algorithm for energy minimization, parallelization, and further algorithmic improvements. With these improvements we studied the discontinuous evolution of the structure of stretching polycrystalline graphene: occasionally a small increase in stretching force induces an avalanche-like displacement. We experimented with techniques to study this behavior which could enable new ways of studying polycrystalline quasi-two-dimensional materials.

Contents

Introduction

Carbon is found in all known life forms and is all around us, from pencils to petroleum and plastics to steel. The versatility of carbon makes it a promising ingredient in nanomaterials. One of these nanomaterials is graphene, a single layer of carbon atoms arranged in a honeycomb lattice, see Figure 1.1, which has drawn a lot of attention with its extraordinary properties and proposed applications.

Graphene has many useful mechanical properties, from high friction resistance to extreme flexibility [2] and is said to be the strongest material in the world [3]. It is however most renowned for its electrical properties. Electrons in graphene exhibit linear dispersion relations allowing them to move through the material as if they are massless [4]. This ease of motion makes graphene and excellent electrical and thermal conductor, exhibiting ballistic transportation even at room temperature [5, 6]. Bilayered graphene has even been reported to be superconducting when the layers are twisted relative to each other by a small angle [7]. Its excellent thermal conductivity [8] and mechanical properties make it very friction resistant and an ideal material for high-pressure contacts.[9].

Extraordinary properties as described above are not exclusively for monocrystalline graphene. Topological defects ripple and buckle the graphene out-of-plane [10, 11], see Figure 1.1, changing its mechanical and electrical properties [12]. Controlled addition of defects or impurities can turn graphene into a semiconductor despite its excellent conductivity [13, 14]. The need for research into polycrystalline graphene is twofold: to explore the landscape of properties and applications, and to improve techniques for synthesizing graphene. Techniques for producing graphene are still developing and large samples are still polycrystalline [15]. Checking the quality of a graphene sample can be tedious as most experimental techniques detect lattice defects directly, such as scanning tunneling microscopy (STM) [16, 17], transmission electron microscopy (TEM) [18, 19] and atomic force microscopy (AFM) [20, 21]. Some indirect methods of detection are inelastic tunneling spectroscopy (IETS) [22, 23], neutron scattering[24], X-ray absorption spectroscopy [25, 26] and Raman spectroscopy [27, 28]. These indirect methods measure the phonon spectrum or vibrational density of states (VDOS) [29, 30]. By theoretically studying vibrations in graphene we hope to further improve the capability of these indirect techniques. In particular we will further investigate the vibrations that accompany the crackling of polycrystalline graphene while it is being stretched and relaxed [31].

Computer simulations are used to generate realistic sample configurations of crystalline and polycrystalline graphene and to simulate vibrations and calculate the vibrational density of states. The graphene sheet is modeled by a continuous random network (CRN) [32] on which Wooten, Winer and Weaire bond transpositions [33] are performed. The energy and molecular dynamics (MD) [34, 35] are calculated using the empirical potential obtained by Jain *et al.* [36], based on the Kirkwood and Keating potentials [37, 38]. The graphene sample is then minimized with the very effective FIRE algorithm [39]. Relaxed and minimized configurations are investigated by calculating their vibrational density of states and other properties.

Efficient simulation techniques and algorithmic improvements are investigated to which could lead to new methods of studying polycrystalline quasi-two-dimensional materials. An significant speedup is achieved by implementing improved data structures and program optimization techniques like parallelism using OpenMP [40, 41].

Chapter 1

Graphene

Graphene is a single layer of carbon atoms arranged in an hexagonal lattice, see Figure 1.1(a). It was first isolated in 2004 by Novoselov *et al.* [42] and the first two-dimensional crystal observed in nature [43]. The carbon atoms are connected by covalent sp^2 bonds, a hybridization of 2s and 2p orbitals. Each atom has three sp^2 orbitals and the corresponding bonds repel each other, leading to angles of $2\pi/3$ between them. The resulting hexagonal monocrystalline graphene has many useful properties such as excellent electrical and thermal conductivity [4, 5, 6]. Bilayered monocrystalline graphene has even been reported to be superconducting when the layers are twisted relative to each other by a small angle [7].

Techniques for synthesizing graphene are still developing and most larger samples are polycrystalline [15] i.e. contain topological defects and grains of different sizes. Topological defects lead to ripples which buckle graphene out-of-plane [10, 11], see Figure 1.1, changing its mechanical and electrical properties [12]. Controlled addition of defects and impurities can lead to new useful properties and applications. For instance graphene semiconductors, which are smaller and more heat resistant [13, 14]. To fully explore the possibilities, a theoretical framework is needed. However, most methods in theoretical solid-state physics rely on the crystalline periodicity to simplify the math and obtain any results, which is missing in polycrystalline materials. Lacking this periodicity and the associated methods we turn to computer models to simulate and calculate properties.

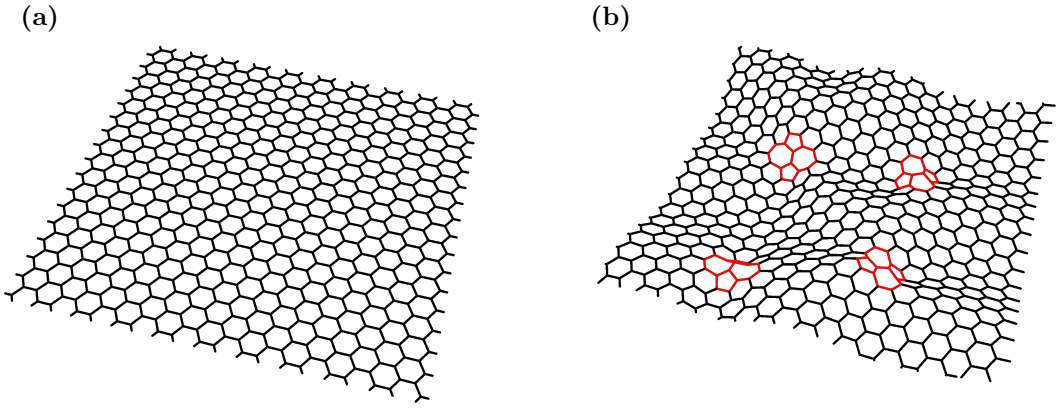


Figure 1.1: **(a)** A sample of pristine graphene. **(b)** A graphene sample with four point defects (red). The defects buckle the graphene out-of-plane.

1.1 Generation of polycrystalline graphene configurations

A good theoretical representation of polycrystalline graphene is a continuous random network (CRN) [32], first introduced by Zachariasen [44]. A graphene configuration of N atoms is represented by a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of N vertices $\mathcal{V}$ and $3N/2$ edges $\mathcal{E}$ where each vertex has three edges. A very powerful operation that obeys this requirement is the bond transposition proposed by Wooten, Winer and Weaire [33], see Figure 1.2. These transpositions can introduce and remove point defects in graphene, but also move defects and grain boundaries around. Such a small yet powerful operation is ideal for Monte-Carlo methods and allows for a more physically realistic exploration of graphene configurations, see section 2.2.

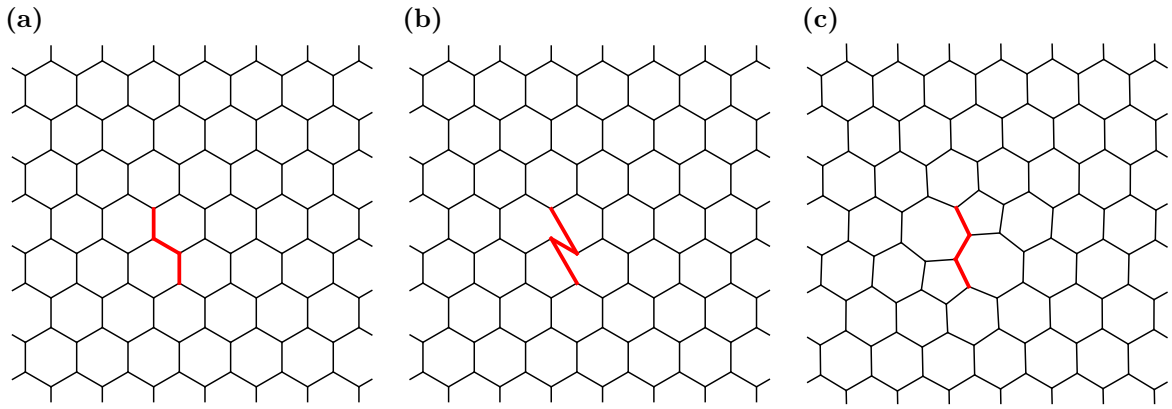


Figure 1.2: A Wooten-Winer-Weaire bond transposition forming a Stone-Wales defect. **(a)** Pristine graphene with consecutive bonds selected for transposition. **(b)** Selected bonds are rearranged according to the bond transposition. **(c)** After relaxation, the configuration contain a Stone-Wales defect.

Polycrystalline configurations can be generated by performing bond transposition on a pristine graphene configuration, see Figure 1.1. Performing random bond transpositions on a pristine hexagonal lattice however, does not necessarily generate a physically realistic configuration. For instance, grains do not form very naturally when starting from a pristine monocrystalline lattice. To generate unbiased random configurations that adhere to statistical physics, random two-dimensional periodic Voronoi diagrams [45] are generated using random seeds. Which are then structurally relaxed and subjected to random bond transpositions with a Boltzmann distributed Metropolis criterion 2.2 until thermal equilibrium is reached, see Figure 1.3.

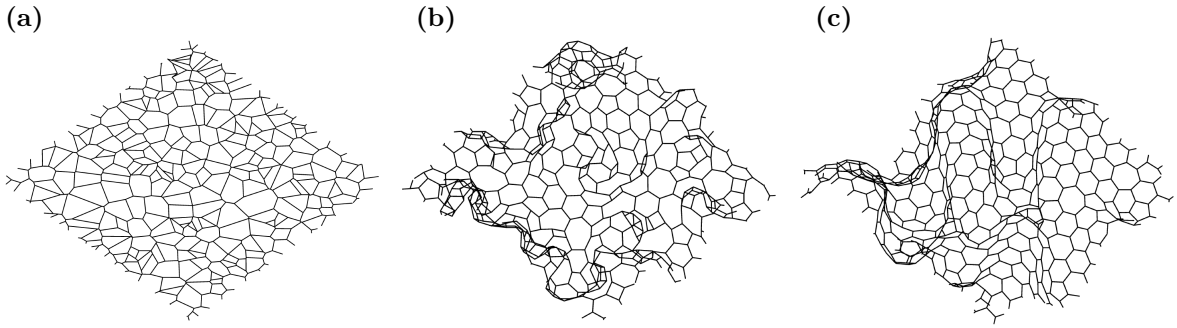


Figure 1.3: A polycrystalline grahpene configuration generated from a random Vonoroi diagram subjected to random WWW bond transpositions. **(a)** A random two-dimensional periodic Voronoi diagram with $N = 500$ vertices. **(b)** Configuration after structural relaxation of the Voronoi diagram by the FIRE algorithm, see section 2.1. **(c)** The configuration near thermal equilibrium at $T = 300$ K.

The internal energy of a configuration is calculated using an empirical potential (1.1) obtained by Jain *et al.* [36]. This potential is based on the Kirkwood and Keating potentials [37, 38] with an additional out-of-plane term to model three-dimensional deformations.

$$U^{\text{int}} = \frac{3}{16}\frac{\alpha}{d^2}\sum_{\langle i,j\rangle}\left(r_{ij}^2 - d^2\right)^2 + \frac{3}{8}\beta\, d^2 \sum_{\langle i,jk\rangle}\left(\theta_{i,jk} - \frac{2\pi}{3}\right)^2 + \gamma \sum_{\langle i,jkl\rangle} r_{i,jkl}^2 \tag{1.1}$$

Where r_{ij} is the length of the bond from atom i to atom j in Ångström, $\theta_{i,jk}$ the angle between the bonds connecting atom i to neighboring atoms j and k in radians (rad) and $r_{i,jkl}$ the distance from atom i to the plane spanned through the positions of its neighboring atoms j, k and l in Å. Parameters

$$\begin{aligned} d &= 1.420 \text{ Å} \\ \alpha &= 26.060 \text{ eV/Å}^2 \\ \beta &= 5.511 \text{ eV/Å}^2 \\ \gamma &= 0.517 \text{ eV/Å}^2 \end{aligned} \tag{1.2}$$

were empirically obtained by Jain *et al.* [36] using DFT with techniques from the Vienna ab initio Simulation Package (VASP) [46, 47] and a Van der Waals functional [48] for solids formulated by Dion *et al.* [49]. Properly fitting a classical potential to quantum mechanical calculations reduces computational costs and simplifies the mathematics while maintaining good accuracy. This particular potential has already been used in several papers [31, 50, 51, 52] and compared to other potentials [53, 54, 55].
The internal energy (1.1) has three distinct types of energy contributions

$$U^{\text{bond}} = \frac{3}{16}\frac{\alpha}{d^2}\sum_{\langle i,j\rangle}\left(r_{ij}^2 - d^2\right)^2 \quad U^{\text{angle}} = \frac{3}{8}\beta\, d^2 \sum_{\langle i,jk\rangle}\left(\theta_{i,jk} - \frac{2\pi}{3}\right)^2 \quad U^{\text{plane}} = \gamma \sum_{\langle i,jkl\rangle} r_{i,jkl}^2 \tag{1.3}$$

also known as two-, three- and four-body interactions respectively. The bond energy represents the covalent sp^2 bonds deviating from their ideal length of 1.42 Å, much like a spring. The angle energy represents the bonds repelling each other, leading to ideal angles of $2\pi/3$ between them. Notice that pulling an atom out of the plane with respect to its neighbors, makes the sum of the three angles smaller, but this effect is not enough to account for the energy levels observed in the DFT computations [36]. The plane energy represents the observed additional energy in out-of-plane deformations. To get an idea how these different energy contributions are distributed throughout the configuration, we assign each atom local energies

$$\begin{aligned} U_i^{\text{bond}} &= \frac{3}{32}\frac{\alpha}{d^2}\left[\left(r_{ij}^2 - d^2\right)^2 + \left(r_{ik}^2 - d^2\right)^2 + \left(r_{il}^2 - d^2\right)^2\right] \\ U_i^{\text{angle}} &= \frac{3}{8}\beta\, d^2\left[\left(\theta_{i,jk} - \frac{2\pi}{3}\right)^2 + \left(\theta_{i,kl} - \frac{2\pi}{3}\right)^2 + \left(\theta_{i,jl} - \frac{2\pi}{3}\right)^2\right] \\ U_i^{\text{plane}} &= \gamma\, r_{i,jkl}^2. \end{aligned} \tag{1.4}$$

where i, j, k are the neighbors of atom i. The positions of the carbon atoms in our model are restricted to $0 \le x < L_x$, $0 \le y < L_y$. As such, the atoms find themselves in a box with infinite height with periodic boundary conditions in both the x and y direction. The graphene configuration will be placed such that it connects with itself at the box's periodic boundaries. To stretch the graphene an additional strain energy term is added

$$U = \frac{3}{16}\frac{\alpha}{d^2}\sum_{\langle i,j\rangle}\left(r_{ij}^2 - d^2\right)^2 + \frac{3}{8}\beta\, d^2 \sum_{\langle i,jk\rangle}\left(\theta_{i,jk} - \frac{2\pi}{3}\right)^2 + \gamma \sum_{\langle i,jkl\rangle} r_{i,jkl}^2 - \varepsilon L_x L_y \tag{1.5}$$

where the strain parameter $\varepsilon\,(\text{eV/Å}^2)$ allows a stretching force to be applied. Note the atoms only interact with this strain energy indirectly, distributing the strain forces among the bonds.

1.2 Dynamic evolution

Dynamics lay at the heart of any physical theory. The ability to predict how a system evolves and behaves is what defines our physical understanding. To turn the graphene model described in the previous section into a physical model, requires forces to dictate the motion of the atoms. Since the energy potential (1.5) is a classical approximation, we turn to classical mechanics and the relation $\mathbf{F} = -\boldsymbol{\nabla} U$ to derive these forces.

Introduce vector $\boldsymbol{r}_{ij}$ pointing from atom i to atom j, and vector $\mathbf{n}_i$ normal to the plane through the neighbors of atom i, then with some geometry we have the following expressions

$$\begin{aligned} r_{ij}^2 = \|\mathbf{r}_{ij}\|^2 = \mathbf{r}_{ij}\cdot\mathbf{r}_{ij} \qquad \|\mathbf{r}_{ij}\| = \sqrt{\mathbf{r}_{ij}\cdot\mathbf{r}_{ij}} \qquad \mathbf{n}_i = (\mathbf{r}_{ik}-\mathbf{r}_{ij})\times(\mathbf{r}_{il}-\mathbf{r}_{ij}) \\ \theta_{i,jk} = \arccos\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right) \qquad r_{i,jkl}^2 = \frac{(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2}{\|\mathbf{n}_i\|^2} = \frac{(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2}{\mathbf{n}_i\cdot\mathbf{n}_i}. \end{aligned} \tag{1.6}$$

These expressions make the derivation of local forces easier, see Appendix A. The bond energy leads to forces parallel to the bond

$$\begin{aligned} \mathbf{F}_i^{\text{bond}} &= \frac{3\,\alpha}{4\,d^2}(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij}-d^2)\mathbf{r}_{ij} \\ \mathbf{F}_j^{\text{bond}} &= -\frac{3\,\alpha}{4\,d^2}(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij}-d^2)\mathbf{r}_{ij} \end{aligned} \tag{1.7}$$

visualized in Figure 1.4.

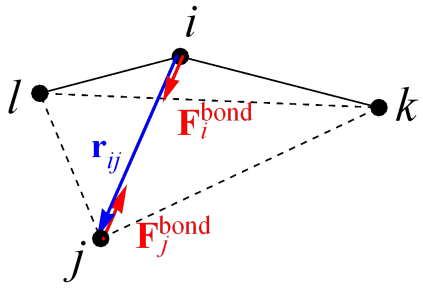


Figure 1.4: Visualization of forces due to two-body interactions between atoms i and j. Derived from the gradient of the bond energy (1.4).

The angle energy, representing bond repulsion, leads to forces in the plane spanned by the two bonds involved

$$\begin{aligned} \mathbf{F}_i^{\text{angle}} &= \frac{3\,\beta\,d^2}{4\,|\sin(\theta_{i,jk})|}\left(\theta_{i,jk}-\frac{2\pi}{3}\right)\left[\cos(\theta_{i,jk})\left(\frac{\mathbf{r}_{ij}}{\mathbf{r}_{ij}\cdot\mathbf{r}_{ij}}+\frac{\mathbf{r}_{ik}}{\mathbf{r}_{ik}\cdot\mathbf{r}_{ik}}\right)-\frac{\mathbf{r}_{ij}+\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right] \\ \mathbf{F}_j^{\text{angle}} &= -\frac{3\,\beta\,d^2}{4\,|\sin(\theta_{i,jk})|}\left(\theta_{i,jk}-\frac{2\pi}{3}\right)\left[\cos(\theta_{i,jk})\frac{\mathbf{r}_{ij}}{\mathbf{r}_{ij}\cdot\mathbf{r}_{ij}}-\frac{\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right] \\ \mathbf{F}_k^{\text{angle}} &= -\frac{3\,\beta\,d^2}{4\,|\sin(\theta_{i,jk})|}\left(\theta_{i,jk}-\frac{2\pi}{3}\right)\left[\cos(\theta_{i,jk})\frac{\mathbf{r}_{ik}}{\mathbf{r}_{ik}\cdot\mathbf{r}_{ik}}-\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right] \end{aligned} \tag{1.8}$$

as shown in Figure 1.5.

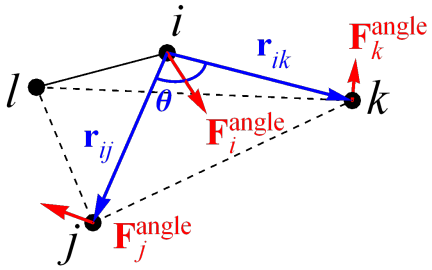


Figure 1.5: Visualization of forces due to three-body interactions between atoms i, j and k. Derived from the gradient of the angle energy (1.4).

The out-of-plane potential pushes the central atom straight down, normal to the plane through its neighbors, leading to forces

$$\begin{aligned}
\mathbf{F}_i^{\text{plane}} &= 2\gamma \frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\,\mathbf{n}_i \\
\mathbf{F}_j^{\text{plane}} &= 2\gamma \frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\left[(\mathbf{r}_{il}-\mathbf{r}_{ik})\times\mathbf{r}_{ij}-\mathbf{n}_i\right] + 2\gamma\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\right)^2(\mathbf{r}_{ik}-\mathbf{r}_{il})\times\mathbf{n}_i \\
\mathbf{F}_k^{\text{plane}} &= 2\gamma \frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}(\mathbf{r}_{ij}\times\mathbf{r}_{il}) + 2\gamma\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\right)^2(\mathbf{r}_{il}-\mathbf{r}_{ij})\times\mathbf{n}_i \\
\mathbf{F}_l^{\text{plane}} &= 2\gamma \frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}(\mathbf{r}_{ik}\times\mathbf{r}_{ij}) + 2\gamma\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\right)^2(\mathbf{r}_{ij}-\mathbf{r}_{ik})\times\mathbf{n}_i.
\end{aligned} \tag{1.9}$$

and its neighbors in the opposite direction, as shown in Figure 1.6. See Appendix A for full derivation.

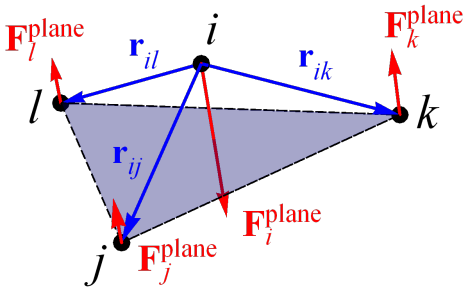


Figure 1.6: Visualization of forces due to four-body interactions involving atom i and its neighbors j, k and l. Derived from the gradient of the out-of-plane energy (1.4).

To stretch or relax the graphene configuration, the box's periodic faces are also allowed to move. Each is given the same mass as a single carbon atom and the forces acting on them arise from all the bonds crossing the periodic boundaries and the strain energy $-\varepsilon L_x L_y$ in (1.5). Note that moving the box's faces does not move the atoms, only their relative positions change. The forces $\mathbf{F}_i^{\text{bond}}$, $\mathbf{F}_i^{\text{angle}}$ and $\mathbf{F}_i^{\text{plane}}$ derived above will change, but do not pull on the box. Only forces acting on neighbors are contributing. If $\mathbf{r}_{ij}$ crosses the periodic boundary, then $\mathbf{F}_j^{\text{bond}}$, $\mathbf{F}_j^{\text{angle}}$ and $\mathbf{F}_j^{\text{plane}}$ act on that face, because from atom i's point of view it is atom j that moves when the face changes position. The total forces on the box's faces are

$$\begin{aligned}
F_x^{\text{box}} &= \sum_{\langle i,j\rangle} c_{ij,x}\left(\mathbf{F}_j^{\text{bond}} + \mathbf{F}_j^{\text{angle}} + \mathbf{F}_j^{\text{plane}}\right)\cdot\hat{\mathbf{x}} + \varepsilon L_y \\
F_y^{\text{box}} &= \sum_{\langle i,j\rangle} c_{ij,y}\left(\mathbf{F}_j^{\text{bond}} + \mathbf{F}_j^{\text{angle}} + \mathbf{F}_j^{\text{plane}}\right)\cdot\hat{\mathbf{y}} + \varepsilon L_x
\end{aligned} \tag{1.10}$$

where

$$c_{ij,x} = \begin{cases} 1 & \text{if } \mathbf{r}_{ij} \text{ crosses } x\text{-periodic boundary in the positive direction} \\ -1 & \text{if } \mathbf{r}_{ij} \text{ crosses } x\text{-periodic boundary in the negative direction} \\ 0 & \text{otherwise} \end{cases}$$

$$c_{ij,y} = \begin{cases} 1 & \text{if } \mathbf{r}_{ij} \text{ crosses } y\text{-periodic boundary in the positive direction} \\ -1 & \text{if } \mathbf{r}_{ij} \text{ crosses } y\text{-periodic boundary in the negative direction} \\ 0 & \text{otherwise.} \end{cases} \tag{1.11}$$

With all the forces formulated, Newton's equations of motion can be employed to evolve the graphene configuration. A graphene configuration of N atoms can be represented by a weighted undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of N vertices $\mathcal{V}$ and $3N/2$ edges $\mathcal{E}$ where each vertex has three edges. Vertices represent carbon atoms with a position, velocity and force acting on it. The weights of the edges represent bond lengths. Let us define the following global vectors

$$\begin{aligned} \mathbf{R} &= \big(r_{1,x} \, , \, r_{1,y} \, , \, r_{1,z} \, , \, r_{2,x} \, , \, r_{2,y} \, , \, r_{2,z} \, , \ldots \, , \, r_{N,y} \, , \, r_{N,z} \, , \, L_x \, , \, L_y \big) \\ \mathbf{V} &= \big(v_{1,x} \, , \, v_{1,y} \, , \, v_{1,z} \, , \, v_{2,x} \, , \, v_{2,y} \, , \, v_{2,z} \, , \ldots \, , \, v_{N,y} \, , \, v_{N,z} \, , \, V_x \, , \, V_y \big) \\ \mathbf{F} &= \big(F_{1,x} \, , F_{1,y} \, , F_{1,z} \, , F_{2,x} \, , F_{2,y} \, , F_{2,z} \, , \ldots , F_{N,y} \, , F_{N,z} \, , F_x^{\text{box}} \, , F_y^{\text{box}} \big) \end{aligned} \tag{1.12}$$

then for each graphene configuration $\mathcal{G}$, these three global vectors represent a physical state. All bodies in our model have the same mass and the equations of motion is therefore the same for every object. The whole system can thus be evolved dynamically by the following differential equation

$$\mathbf{F} = m \frac{d^2}{dt^2} \mathbf{R} \tag{1.13}$$

where m is the mass of a carbon atom 12.0107 u. There are no exact solutions for many-body systems like this, but the solution can be approximated by taking small time steps, see Algorithm 1 in section 2.1. Algorithms can now be designed to generate graphene samples and simulate their dynamics.

Chapter 2

Algorithm Design

Algorithms are recipes of instructions designed to obtain a result. Usually to find or calculate a solution to some kind of optimization problem or simulate a system for analysis. Algorithms are such an important part of society that multiple fields of study dedicate special attention to them. Algorithms in their most fundamental form are studied in mathematics and computer science. Analytic solutions are rare in physics and almost all physics is build upon perturbation theory, which is the mathematical study of approximations. There is even a branch of physics called computational physics, dedicated to numerical calculations and simulations.

In the previous chapter we introduced our computer model for graphene and its dynamics. This chapter will cover the various algorithms implemented to simulate the molecular dynamics and perform structural relaxation. Minimization of the energy will be achieved by the FIRE algorithm [39] and bond transpositions, see Figure 1.2, will be performed by the WWW algorithm [33]. Algorithmic improvements suggested by Barkema *et al.* [56] are discussed, followed by our own proposed improvements at the end of this chapter.

2.1 Molecular Dynamics and Structural Relaxation

Simulation of complex systems is often involves approximating the evolution with small consecutive increments. By taking small steps higher order corrections are relatively small, often simplifying the calculates. The equation of motion (1.13) derived from the potential (1.5) for instance, is too complex for polycrystalline configurations to be solved analytically. Taking small time steps Δt approximates the continuous change of the physical state

$$\begin{aligned}
\mathbf{R} &= \big(r_{1,x}\,,\, r_{1,y}\,,\, r_{1,z}\,,\, r_{2,x}\,,\, r_{2,y}\,,\, r_{2,z}\,,\ldots,\, r_{N,y}\,,\, r_{N,z}\,,\, L_x\,,\, L_y \big) \\
\mathbf{V} &= \big(v_{1,x}\,,\, v_{1,y}\,,\, v_{1,z}\,,\, v_{2,x}\,,\, v_{2,y}\,,\, v_{2,z}\,,\ldots,\, v_{N,y}\,,\, v_{N,z}\,,\, V_x\,,\, V_y \big) \\
\mathbf{F} &= \big(F_{1,x}\,, F_{1,y}\,, F_{1,z}\,, F_{2,x}\,, F_{2,y}\,, F_{2,z}\,,\ldots, F_{N,y}\,, F_{N,z}\,, F_x^{\text{box}}\,, F_y^{\text{box}} \big).
\end{aligned}$$

With every time step the latest forces are calculated and the positions and velocities updated, leading to Algorithm 1. Note that this algorithm uses the midpoint method, also known as the modified Euler method [57]. The calculation of the forces, part (3), is computationally the most expensive part of the molecular dynamics, using around 95% of the computation time. The molecular dynamics (MD) is then use in combination with the Fast Inertial Relaxation Engine (FIRE) developed by Bitzek *et al.* [39] to minimize the energy and structurally relax the configuration.

The FIRE algorithm, see Algorithm 2, is designed with simplicity in mind. Many minimization techniques are plagued by either getting stuck in local minima or overshooting the desired minimum. Attempts to remedy this situation often lead to complicated measures and checks. The FIRE algorithm uses a more physical approach. To void getting stuck in local minima FIRE uses the inertia already present in the molecular dynamics. Atoms moving down the potential can overcome small energy increases using their momentum. To avoid overshooting the minimum of the configuration as a whole, the algorithm stops all motion whenever the potential energy increases ($\mathbf{V} \cdot \mathbf{F} > 0$). Additionally the algorithm also mixes the velocities with the forces to steer the dynamics slightly more in the right direction, see part (5) of Algorithm 2.

Algorithm 1: Molecular Dynamics

input: starting positions $\mathbf{R}_{\text{init}}$ and velocities $\mathbf{V}_{\text{init}}$ of graphene configuration $\mathcal{G}$ at $t = t_{\text{init}}$
output: final positions $\mathbf{R}_{\text{final}}$ and velocities $\mathbf{V}_{\text{final}}$ of graphene configuration $\mathcal{G}$ at $t = t_{\text{final}}$

(1) $\mathbf{R} = \mathbf{R}_{\text{init}}$; $\mathbf{V} = \mathbf{V}_{\text{init}}$; $\mathbf{F} = -\nabla U(\mathbf{R}_{\text{init}})$;
$t = t_{\text{init}}$; $\Delta t = \Delta t_{\text{init}}$;

(2) **while** $t < t_{\text{final}} - \Delta t$ **do**
$\mathbf{R} = \mathbf{R} + \Delta t \mathbf{V} + \frac{\Delta t^2}{2m}\mathbf{F}$;
$\mathbf{F}_{\text{old}} = \mathbf{F}$;
(3) $\mathbf{F} = -\nabla U(\mathbf{R})$;
$\mathbf{V} = \mathbf{V} + \frac{\Delta t}{2m}(\mathbf{F} + \mathbf{F}_{\text{old}})$;
$t = t + \Delta t$
(4) $\mathbf{R}_{\text{final}} = \mathbf{R} + (t_{\text{final}} - t)\mathbf{V} + \frac{1}{2m}(t_{\text{final}} - t)^2\mathbf{F}$;
$\mathbf{F}_{\text{old}} = \mathbf{F}$;
$\mathbf{F} = -\nabla U(\mathbf{R}_{\text{final}})$;
$\mathbf{V}_{\text{final}} = \mathbf{V} + \frac{1}{2m}(t_{\text{final}} - t)(\mathbf{F} + \mathbf{F}_{\text{old}})$;

Algorithm 2: FIRE [39]

input: starting positions $\mathbf{R}_{\text{init}}$ of graphene configuration $\mathcal{G}$
output: relaxed positions $\mathbf{R}_{\text{relax}}$ of graphene configuration $\mathcal{G}$

(1) $\mathbf{R} = \mathbf{R}_{\text{init}}$; $\mathbf{V} = 0$; $\mathbf{F} = -\nabla U(\mathbf{R})$;
$\alpha = \alpha_{\text{init}}$; $\Delta t = \Delta t_{\text{init}}$; $n_{\text{step}} = 0$;

(2) **while** $F_{\max} > F_{\text{relax}}$ **do**
$\mathbf{R} = \mathbf{R} + \Delta t \mathbf{V} + \frac{\Delta t^2}{2m}\mathbf{F}$;
$\mathbf{F}_{\text{old}} = \mathbf{F}$;
(3) $\mathbf{F} = -\nabla U(\mathbf{R})$;
$\mathbf{V} = \mathbf{V} + \frac{\Delta t}{2m}(\mathbf{F} + \mathbf{F}_{\text{old}})$;
$n_{\text{step}} = n_{\text{step}} + 1$;
(4) **if** $\mathbf{V} \cdot \mathbf{F} < 0$ **then**
$\mathbf{V} = 0$;
$\alpha = \alpha_{\text{init}}$; $\Delta t = f_{\text{dec}}\,\Delta t$; $n_{\text{step}} = 0$;
else
(5) $\mathbf{V} = (1 - \alpha)\mathbf{V} + \alpha|\mathbf{V}|\,\hat{\mathbf{F}}$;
if $n_{\text{step}} > \min_{\text{step}}$ **then**
$\alpha = f_\alpha\,\alpha$;
(6) **if** $f_{\text{inc}}\,\Delta t > \Delta t_{\max}$ **then**
$\Delta t = \Delta t_{\max}$;
else
$\Delta t = f_{\text{inc}}\,\Delta t$;
$\mathbf{R}_{\text{relax}} = \mathbf{R}$;

Parameters $\Delta t_{\text{init}} = 0.1$, $f_\alpha = 0.99$, $f_{\text{inc}} = 1.1$, $f_{\text{dec}} = 0.5$ and $n_{\text{step}} = 5$ are obtained from [39]. As with the Molecular Dynamics algorithm, the computationally most expensive part of FIRE is the calculation of the forces, part (3) of Algorithm 2. A graphene configuration is considered relaxed when $F_{\max} \leq F_{\text{relax}}$, where $F_{\text{relax}} \in \{10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}\}$ depending on the required accuracy and

$$F_{\max} = \max\left\{|F_{1,x}|, |F_{1,y}|, |F_{1,z}|, |F_{2,x}|, |F_{2,y}|, |F_{2,z}|, \ldots, |F_{N,y}|, |F_{N,z}|, |F_x^{\text{box}}|, |F_y^{\text{box}}|\right\}. \tag{2.1}$$

2.2 Wooten-Winer-Weaire Algorithm

The algorithms in the previous section dynamically evolve and relax graphene configurations at fixed topology. Exploration of polycrystalline configurations requires this topology to evolve as well. These configurations are modelled by a continuous random network and represented by a graph $\mathcal{G}$ with the explicit requirement that each atom has three covalent bonds at all time. The Wooten-Winer-Weaire bond transposition [33], see Figure 1.2, obeys this condition. This simple yet powerful bond transposition is ideal for Monte-Carlo methods. Wooten, Winer and Weaire therefore proposed an algorithm attempting random WWW bond transpositions with a Boltzmann distributed Metropolis criterion. The idea is to propose a random WWW bond transposition, calculate how much the potential energy would be lowered by this transpositions, then accept or reject it with Metropolis probability

$$P_{\text{accept}} = \min\left\{1\,, \exp\left(\frac{U_{\text{pre}} - U_{\text{post}}}{k_{\text{B}}T}\right)\right\} \tag{2.2}$$

where k_{B} is the Boltzmann constant and T the temperature in Kelvin. This Metropolis criterion accepts any transposition that lowers the potential energy, but also allows for transpositions that increase the energy. The Boltzmann weight makes the occurrence of these unfavorable transpositions in accordance with thermal fluctuations and statistical physics. Any transposition forming a triangle however, is automatically rejected as they are considered unphysical. Let $T_{i_1i_2i_3i_4}$ denote the bond transposition that changes the bonds $(i_1, i_2), (i_2, i_3), (i_3, i_4)$ into $(i_1, i_3), (i_2, i_3), (i_2, i_4)$. Then $T_{i_1i_3i_2i_4}$ is the inverse of $T_{i_1i_2i_3i_4}$ and the complete WWW algorithm can be formulated by Algorithm 3.

Algorithm 3: WWW [33]

input: initial graphene configuration $\mathcal{G}_{\text{pre}}$ and its relaxed positions $\mathbf{R}_{\text{pre}}$
output: permuted graphene configuration $\mathcal{G}_{\text{post}}$ and its relaxed positions $\mathbf{R}_{\text{post}}$

```
    R = R_pre;
    U_pre = U(R);
    Accepted = False;
(1) while Accepted = False do
        NoTriangle = False;
(2)     while NoTriangle = False do
            NoTriangle = True;
            pick a random bond transpositions T_{i1 i2 i3 i4};
            perform T_{i1 i2 i3 i4};
            if T_{i1 i2 i3 i4} created a triangle then
                perform T_{i1 i3 i2 i4};
                NoTriangle = False;
(3)     relax the structure using FIRE (Algorithm 2);
        U_post = U(R);
(4)     P = min{1, exp((U_pre - U_post)/(k_B T))};
        pick a random number p ∈ [0, 1);
(5)     if p < P then
            Accepted = True;
        else
            R = R_pre;
            perform T_{i1 i3 i2 i4};
    R_post = R;
```

The while-loop, part (2) of Algorithm 3, searches for a transposition that does not introduce a triangle to the configuration. The found transposition is then performed and the new configuration is relaxed. Its energy is compared to the initial energy by the Metropolis criterion in parts (4) and (5). The relaxation in part (3) is unsurprisingly the most expensive part of the algorithm. Note that Algorithm 3 will continue attempting transpositions until it has successfully performed one.

2.3 Avoiding complete relaxation

The relaxation of the configuration after a transposition is computationally very expensive. Code optimization and parallelization can improve the situation, but it does not address the wasting of computational resources. Every rejected transposition is accompanied by a complete relaxation of the proposed configuration, which is then simply discarded. Worse still is that more and more transposition attempts will be rejected as the system gets closer to thermal equilibrium. To remedy this waste of computational resources, Barkema *et al.* [56] have proposed an improved FIRE algorithm, see Algorithm 4, where the relaxation is discontinued if it seems like the relaxation is not going to result in an accepted transposition. The improved algorithm inverts the Metropolis criterion by first picking a random number $p \in [0, 1)$ and then calculate for which values of U_{post} the criterion would accept. The highest energy that results in an accepted transposition is then used as a threshold

$$U_{\text{thres}} = U_{\text{pre}} - k_{\text{B}}T \ln(p). \tag{2.3}$$

If during relaxation the potential energy crosses the threshold energy U_{thres} then the transposition is going to be accepted. The objective is thus to detect as early as possible when it is not going to cross the threshold. The earlier this can be determined, the more resources will be saved. Barkema *et al.* proposed estimating the final potential energy by

$$U_{\text{post}} \approx U_{\text{approx}} = U(\mathbf{R}) - c_f|\mathbf{F}|^2 \tag{2.4}$$

where c_f is a fitted parameter. This estimation is however only valid when the configuration is already close enough to its minimum. What constitutes 'close enough' is hard to determine. Let F_{close} be the positive value such that when $|\mathbf{F}|^2 < F_{\text{close}}$ the approximation (2.5) is accurate. If F_{close} is small, then it will barely save any resources. When F_{close} in Algorithm 4 is too large, the approximation will no longer be valid which can lead to false negatives, introducing a bias to the decision making. A bias can significantly speedup the algorithm, but it destroys the Boltzmann statistics of the bond transpositions, rendering our algorithm unphysical.

Algorithm 4 does indeed speedup significantly by approximation 2.5, but further investigation also shows it becomes quite bias, only accepting transpositions that lowered the energy significantly. The bias can be countered by lowering F_{close}, but this forfeits the speedup which no longer comes anywhere near the speedup reported [56]. The failure of this method in our model is probably due to its differences in material. Barkema *et al.* applied the improved method to sillium, a three-dimensional amorphous material. This material does not have long-range effects like buckling graphene does, making the method less suitable for graphene. We propose lowering the approximated energy by a system-size dependent constant

$$U_{\text{approx}} = U(\mathbf{R}) - c_f|\mathbf{F}|^2 - \delta E_f N \tag{2.5}$$

to reduce the frequency of false negatives. This adjustment forces the algorithm to fully relax the configurations that lower the energy only by a small abound, while still allowing the early rejecting of obviously unfavorable transpositions. What speedup is achievable with this more cautious estimate is unfortunately not yet known and values for c_f and δE_f still have to be fitted. Another proposed algorithmic improved is to perform only a local relaxation around the attempted bond transposition [58], and based on the energy reduction decide to reject or continue with full relaxation.

Besides interrupting the relaxation one could also attempt to improve the transposition selection, part (2) of Algorithm 3. Analyzing the configuration beforehand might suggest which transpositions have a higher chance of lowering the energy. The introduction of a bias in the transposition proposal is current being investigated by Federico D'Ambrosio at Utrecht University.

Algorithm 4: FIRE with threshold [56]

input: starting positions $\mathbf{R}_{\text{init}}$ of graphene configuration $\mathcal{G}$ and threshold energy U_{thres}
output: relaxed positions $\mathbf{R}_{\text{relax}}$ of graphene configuration $\mathcal{G}$ and Boolean valued $pass$

(1) $\mathbf{R} = \mathbf{R}_{\text{init}}$ $\mathbf{V} = 0$; $\mathbf{F} = -\nabla U(\mathbf{R})$;
$\alpha = \alpha_{\text{init}}$; $\Delta t = \Delta t_{\text{init}}$; $n_{\text{step}} = 0$;

(2) **while** $F_{\max} > F_{\text{relax}}$ **do**
$\mathbf{R} = \mathbf{R} + \Delta t \mathbf{V} + \frac{\Delta t^2}{2m}\mathbf{F}$;
$\mathbf{F}_{\text{old}} = \mathbf{F}$;
(3) $\mathbf{F} = -\nabla U(\mathbf{R})$;
$\mathbf{V} = \mathbf{V} + \frac{\Delta t}{2m}(\mathbf{F} + \mathbf{F}_{\text{old}})$;
$n_{\text{step}} = n_{\text{step}} + 1$;
(4) **if** $|\mathbf{F}|^2 < F_{\text{close}}$ **and** $U_{\text{approx}} > U_{\text{thres}}$ **then**
break; (discontinue)
(5) **if** $\mathbf{V} \cdot \mathbf{F} < 0$ **then**
$\mathbf{V} = 0$;
$\alpha = \alpha_{\text{init}}$; $\Delta t = f_{\text{dec}}\, \Delta t$; $n_{\text{step}} = 0$;
(6) **else**
$\mathbf{V} = (1 - \alpha)\mathbf{V} + \alpha |\mathbf{V}|\, \hat{\mathbf{F}}$;
if $n_{\text{step}} > \min_{\text{step}}$ **then**
$\alpha = f_\alpha\, \alpha$;
(7) **if** $f_{\text{inc}}\, \Delta t > \Delta t_{\max}$ **then**
$\Delta t = \Delta t_{\max}$;
else
$\Delta t = f_{\text{inc}}\, \Delta t$;

(8) **if** $U(\mathbf{R}) < U_{\text{thres}}$ **then**
$\mathbf{R}_{\text{relax}} = \mathbf{R}$;
$pass$ = True;
else
$\mathbf{R}_{\text{relax}} = \mathbf{R}_{\text{init}}$;
$pass$ = False;

Chapter 3

Crackling Polycrystalline Graphene

Polycrystalline graphene naturally buckles out-of-plane forming an three-dimensional structure. On a membrane these deformations are severely limited, but when isolated graphene shows a landscape of structures and mechanical properties. One of these interesting properties is the discontinuous transitions in its structure when under continuously increasing or decreasing strain. A very small change in the stretching force can induce a significant change the preferred structural configuration. These sudden changes cause displacements which in turn create vibrations through the system, akin to avalanches. The discontinuous evolution of a polycrystalline graphene configurations recently published by D'Ambrosio *et al.* [31], are investigated further.

3.1 Discontinuous evolution indicators

The graphene configuration under investigation featuring both point and line defects is shown in Figure 3.1. The discontinuous transition in Figure 3.1 is one of the more visually striking, most transitions are barely noticeable by looking at the configurations directly. Several parameters are thus introduced to monitor the evolution closely and distinguish the different types of transitions. In analogy with crumpled a sheet of paper and for simplicity, the discontinuous transitions will be referred to as crackles from here on.

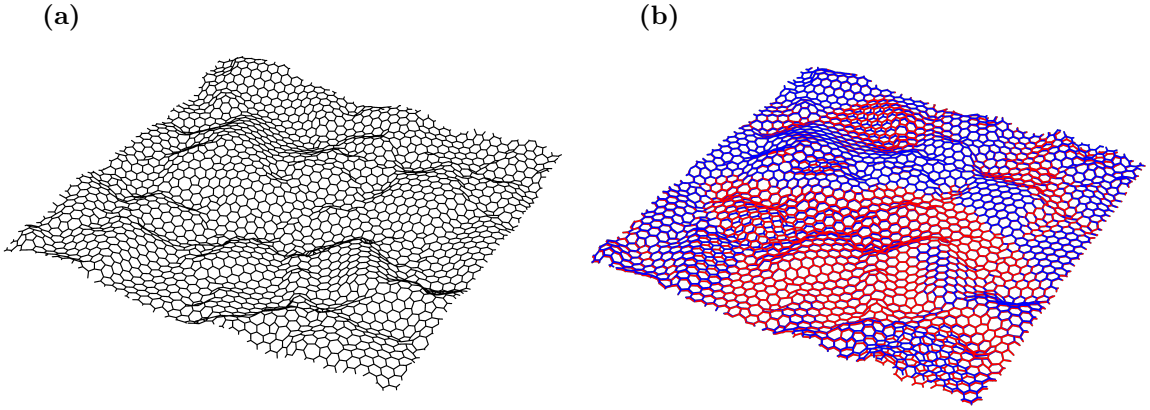


Figure 3.1: **(a)** The polycrystalline configuration from [31] when fully relaxed used for the generation of all figures in Appendices B and C. It consists of $N = 3200$ atoms hosting both point and line defects. **(b)** The lowest energy configuration just before (blue) and after (red) one of the discontinuous transitions. The structural difference is clearly visible, some regions moved up while others moved down. The energies of these configurations are -251.24 eV (blue) and -252.27 eV (red).

A stretching force is applied to the sample in all directions by increasing the strain parameter ε in the potential (1.5). A strain cycle consists of slowly increasing from $\varepsilon = 0\,\mathrm{eV/\AA^2}$ to $\varepsilon = 0.1\,\mathrm{eV/\AA^2}$

in steps of $\delta\varepsilon = 0.0001\,\mathrm{eV/\AA^2}$, fully relax and back to $\varepsilon = 0.1\,\mathrm{eV/\AA^2}$ again. With every step, 3000 in total, the FIRE algorithm finds the minimum energy configuration. All these configurations can then be investigated using energy calculations and structural parameters. We can monitor sudden shifts using the non-affinity parameter [56]

$$A = \frac{\langle (\mathbf{r}_i - \mathbf{r}_{i,A})^2 \rangle}{L_x L_y} \tag{3.1}$$

where $\mathbf{r}_{i,A}$ is the expected position of atom i if the configuration simply scaled with the expansion due to stretching. Any sudden dislocations will rapidly increase this parameter, as shown in Figure 3.2. The non-affinity parameter does not detect any sudden changes in area spanned by the sheet, nor does it indicated what kind of displacement took place. The area parameter $L = L_x L_y$ and out-of-plane deviation

$$\sigma_z = \sqrt{\langle (z_i - \bar{z})^2 \rangle} \tag{3.2}$$

partially balance these shortcomings. The hysteretic behavior is clearly visible in the distribution of the spikes. The first round of increasing strain show a distinct pattern compared to the second round of stretching after full relaxation.

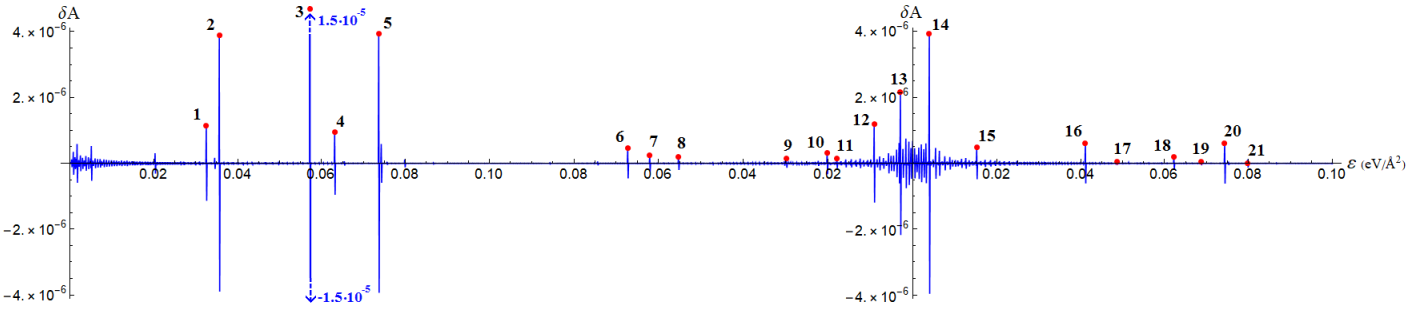


Figure 3.2: Evolution of the non-affinity parameter during a strain cycle at fixed topology. The stretching force on the sample of $N = 3200$ atoms, shown in Figure 3.1, is slowly increased and decreased. Small changes in the stretching force occasionally cause significant displacement. These displacements cause spikes in the non-affinity parameter differential δA, indicating crackles. The hysteretic behavior is also cleary visible, as the spikes are very different during the second round of increasing strain.

Another good indicator crackles is the potential energy (1.5) as shown in Figure 3.3. The size of the spike is a good indicator how energetic the crackle is. Note that a strong spike in the non-affinity parameter differential does not guarantee an energetic release of vibrations. The discontinuous evolution of the other parameters can be found in Appendix B. The various parameters together reveal substantial variety in crackle profiles: Some crackles show a lot of displacement, but release relatively little energy (crackle 14), while others do the reverse (crackle 4). Crackles can also convert energy between the different types (1.3), see for instance crackles 1 and 17 in Appendix B.

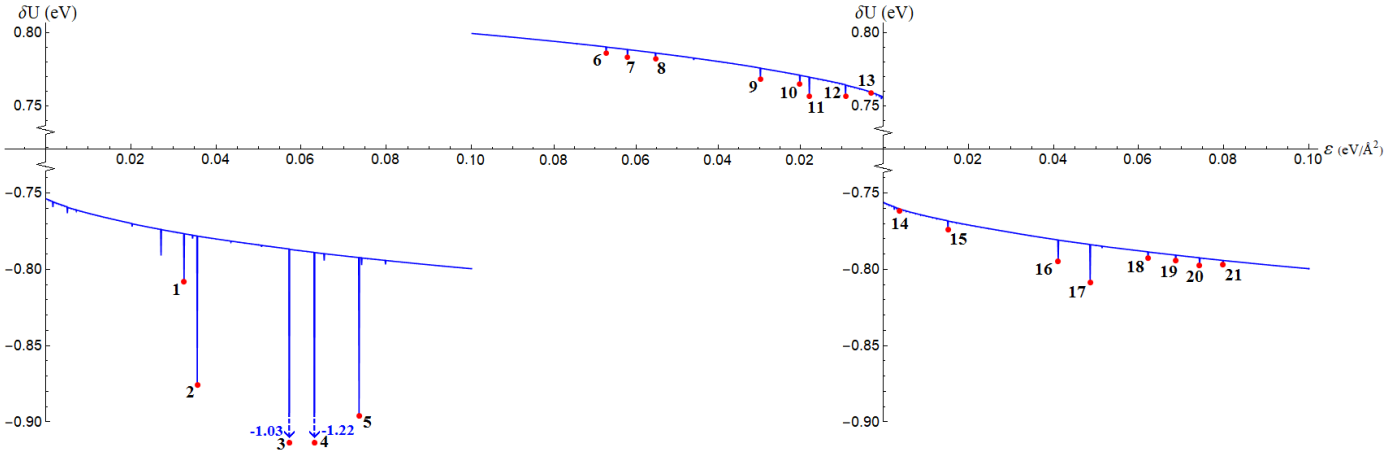


Figure 3.3: Evolution of the potential energy during a strain cycle at fixed topology. The stretching force on the sample of $N = 3200$ atoms is slowly increased and decreased. Small changes in the stretching force occasionally cause significant displacement. These displacements cause vibrations leading to spikes in the energy differential δU, as additional energy is lost to dissipation. These spikes can also indicate how energetic the crackle is.

3.2 Vibrational Density of States

The vibrational density of state is the spectrum of vibrations or phonons of a material, indicating how many excitation have roughly the same frequency. Experimental techniques like inelastic tunneling spectroscopy (IETS) [22, 23], neutron scattering[24], X-ray absorption spectroscopy [25, 26] and Raman spectroscopy [27, 28] measure the VDOS. These (spatial) frequencies are theoretically obtained by calculating the eigenvalues of the Hessian of the potential (1.5). These values are then converted to spatial frequency (cm^{-1}) and convoluted with a Gaussian of width $\sigma = 14\,\mathrm{cm}^{-1}$ to smoothen the plot.

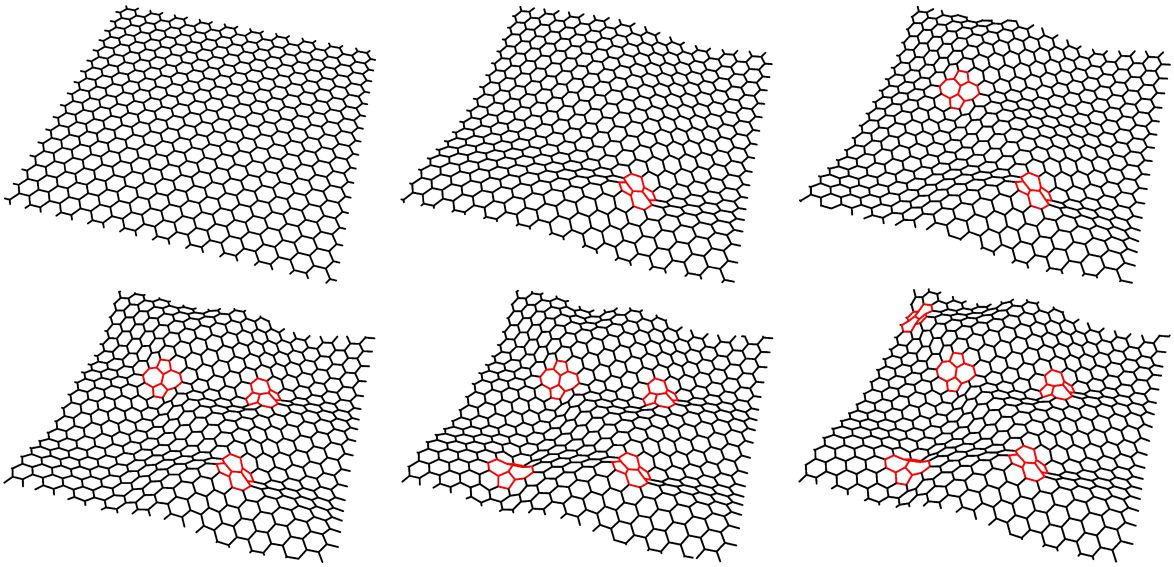


Figure 3.4: Small graphene configurations ($N = 680$) with increasing numbers of Stone-Wales defects (0-5). The buckling increases significantly with a higher density of defects. The VDOS of some of these samples are shown in Figure 3.5, demonstrating how defects influence the VDOS.

The VDOS of small samples with increasing numbers of Stone-Wales defects, see Figure 3.4, demonstrate that defects influence the VDOS significantly. The Stone-Wales defects lower and shift the L' and L peaks to higher frequencies, while red-shifting the Raman-active modes G. A new peak starts to form on the right-hand side of L, indicating a possible splitting of the L mode into separate modes. Defect detection in graphene by analyzing the VDOS has been demonstrated experimentally [59].

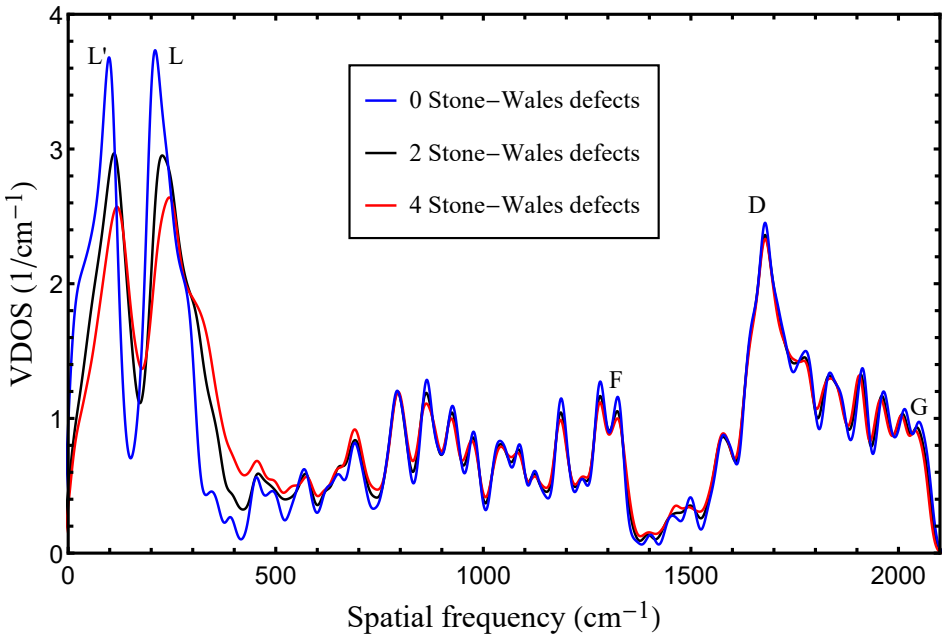


Figure 3.5: VDOS of graphene configurations of $N = 680$ atoms. Pristine graphene has two sharp peaks L' and L. Added Stone-Wales defects reduce the L and L' peak magnitudes and shifts them to higher frequencies, while red-shifting the G modes. A new peak starts to form on the right-hand side of L, indicating a possible splitting of the L mode into separate modes.

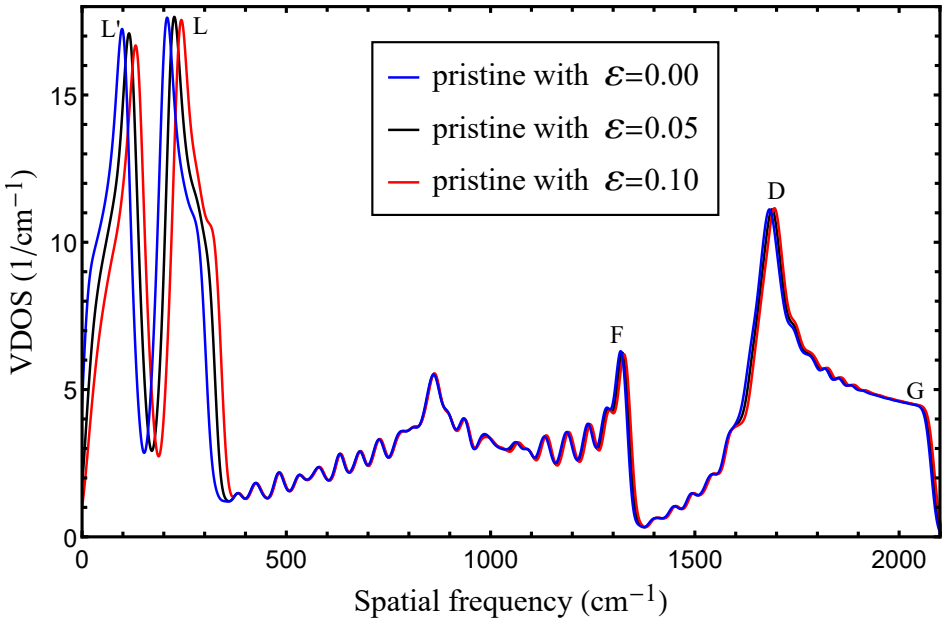


Figure 3.6: VDOS of pristine graphene ($N = 3200$) under increased strain ε (eV/Å^2). Stretching pristine graphene blue-shifts the peaks while barely reducing the peaks magnitude. The low frequency out-of-plane modes L' and L are most affected.

The stretching of graphene in general leads to higher frequency vibrations, see Figure 3.6. Note that the L and L' peaks are barely change in magnitude. While continuously stretching the polycrystalline configuration of Figure 3.1 from $\varepsilon = 0\,\mathrm{eV/\AA^2}$ to $\varepsilon = 0.1\,\mathrm{eV/\AA^2}$ leads to increased peak magnitude, see Figure 3.7. This result may seem counterintuitive at first, but it is actually the effect of Stone-Wales defects in reverse, see Figure 3.5. Stretching polycrystalline graphene flattens the configuration, see parameter σ_z Appendix B, which reduces the prevalence of defects and thus increases the magnitude. The individual crackles can also be investigated using the VDOS. Some crackles are accompanied by significant displacements, which can in turn change the VDOS. Appendix C shows how every numbered crackle affects the VDOS.

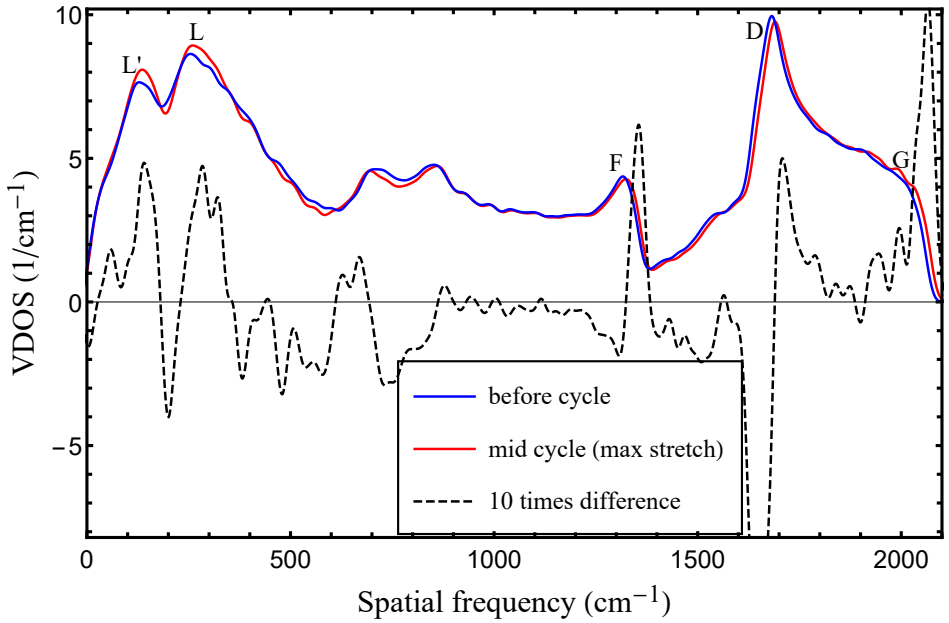


Figure 3.7: VDOS of a polycrystalline graphene configuration with $N = 3200$ atoms, during a strain cycle. Fully relaxed at the start (blue) and maximally stretched (red) at $\varepsilon = 0.1\,(\mathrm{eV/\AA^2})$. The out-of-plane modes L and L' are more prevalent in a strained polycrystalline sample.

Chapter 4

Program Optimization

Program optimization is the process of improving code to increase performance and use less resources. This can for instance be achieved by reducing needles flop operations, more efficiently structuring the data and code parallelization. Distinguishing between algorithmic improvements and implementation improvements can be difficult, as both involve lists of instructions to be executed. The subtle difference lays in the purpose of the instructions: An algorithm instructs one how to obtain a result, while the implementation instructs a computer how to perform the algorithm. A program performing the FIRE relaxations, see Algorithm 2, can be improved by calculating the forces $\mathbf{F} = -\nabla U(\mathbf{R})$ faster, but any changes to its parameters or if-statements are considered algorithmic improvements.

Numerical projects in computational physics can quickly grow out of proportion, driven by ambitious complexity and accuracy goals. Some physical phenomenon only appear at scales that require very large numbers of particles, such physical systems are usually many orders of magnitude larger than can be simulated in a reasonable amount of time. To approximate large systems and avoid boundary effects, simulations often employ periodic boundary conditions. This technique does however have limitations and simulations with large system sizes are still in high demand. Most simple algorithms in computational solid state physics feature only nearest neighbor interactions. This significantly lowers the algorithmic complexity, often growing linearly $\mathcal{O}(N)$ with the system size N. The empirical potential (1.5) used in our model is no exception. This chapter introduces concepts in computing that influence the performance of numerical code and discusses possible improvements. Introductions to parallel algorithms and OpenMP are included. It also shows that parallelization and source code optimization can lead to a speedup upto 7.5 times faster when run on a Intel Core i5-8250U.

4.1 Computer architecture

The most essential parts of a general purpose computer are its CPU's to perform logical and computational operations, memory devices to store data and instructions, and external devices to interact with other computers or humans. The CPU or central processing unit is a piece of hardware that takes in data words and performs basic operations like addition, multiplication or comparisons as instructed. These data words and instructions are fed to the CPU from memory. In general computers have their memory divided into three tiers: main memory drive, dynamic random access memory (DRAM) and cache memory (SRAM). The main memory drive, usually a hard disk drive (HDD) or solid state drive (SSD), is a very large but relatively slow memory drive where all programs and data are stored. The DRAM or working memory is a medium sized with relatively quick access, used to store the programs and data that are currently in use. From here the instructions and data are distributed to the caches, which are much smaller, but very fast, pieces of memory closer to the CPU's.

The cache memory is divided into three levels: L1, L2 and L3, with L1 closest to the CPU as shown in Figure 4.1. Note that L1 is explicitly (and physically) separated into memory for the instructions and memory for the data on which the CPU is to perform its operations. To get a sense of scale, in most personal computers the main memory is several hundreds of gigabytes while the working memory only 4 to 8 gigabytes. The L3 cache typically ranges from 4 upto 50 megabytes, with L2 only

a few megabytes and L1 usually 256 kilobytes or less.

The bottleneck for computation is the transfer of information to and from the different pieces of memory. Using the caches efficiently, filling them with the right data at the right time and reducing the communication between can result in significant speedup. Luckily the operating system and compiler take care of the details at the hardware level. However, the way a program and its data are organized can still have a large impact.

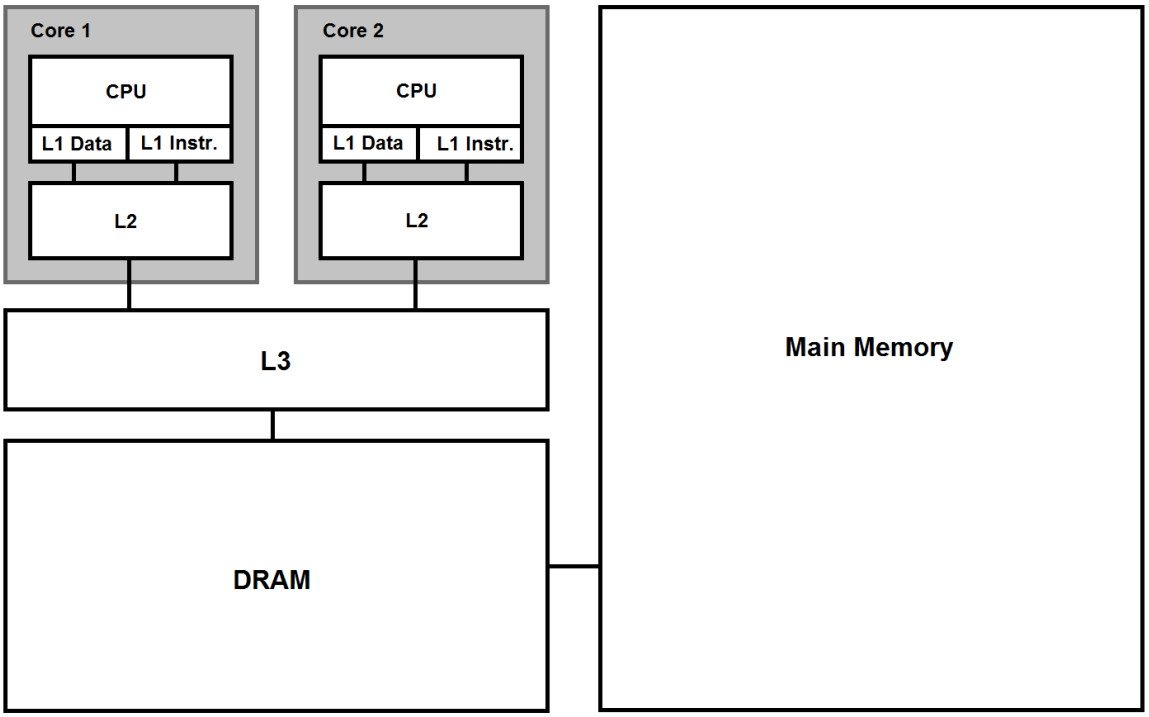


Figure 4.1: Schematic architecture of a general dual-core computer. L1, L2 and L3 are caches.

4.2 Data structure

Data structures are methods of organizing and managing the storage of information. Data can be represented abstractly and stored in many different ways. Depending on the application it might be important to have quick and easy access to data or have it compressed or encrypted for communication. Famous examples are hash functions and Lempel-Ziv-Welch compression [60, 61]. On the machine level any information is represented as a sequence of 1's and 0's called bits, stored somewhere in the memory. The memory address of such a sequence is also represented by a sequence of 1's and 0's. A data word is a fixed-sized piece of data that acts as the basic data unit for a computer. When a computer uses data words of k bits, it is called an k-bit computer. Historically computers were developed for all sorts of values of k, but over time the byte (8-bits) became standard. Computers have since grown from 8-bit to 64-bit. These binary sequences only have meaning when there is consensus on what they represent. An international standard has been developed by the IEEE to represent numbers by either 4 bytes (single precision) or 8 bytes (double precision). Natural language are represented using a character map, mapping sequences of bits to characters in human languages. Standards like ASCII and Unicode are designed to enable worldwide digital communication.

Most programming languages work with basic data types `int`, `float` and their double precision counterparts `long`, `double` to represent integers and real numbers respectively. With `bool` representing Boolean values (True or False) and `char` for natural characters. Additionally there is the `pointer` type to store memory addresses. These basics data types can be organized in sequences, a form of data structure, which are often treated like objects or types in and of themselves. A sequence of integers or floats is often called an array, while sequences of characters are known as strings. Note

that an array of pointers is also possible. All these data types are collectively called classes.

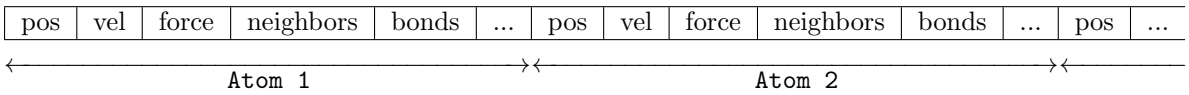
Object-oriented programming languages allow the declaration of custom classes. For instance one can declare a `complex` class to represent complex numbers or a `fraction` class to represent fractions. This makes coding very flexible and often easier to read, behind the scenes however, the data is still represented by the basic data types.

4.3 Source code improvements

Source code is a collection of instructions and statements, formulated in a programming language, that defines a program. Each programming language has its own syntax dictating how instructions should be written. Source code must adhere to the syntax, or the associated compiler cannot translate it to machine code. This machine code is a list of instructions written in machine language, the lowest level of software, telling the hardware what to do. Compilers have grown increasingly complex and smart to optimize the machine code while adhering to the source code. Despite these improvements behind the scenes it still follows the instructions in the source code, this obligates the programmer to write decent source codes. This section discusses several techniques from 'The C++ Programming Language' [62] and 'Using OpenMP' [41] that were used to improve our source code's performance.

4.3.1 Improved data structure

Data structures can significantly impact a code's performance. Simulations often use large amounts of data that need to be accessed and updated over many iterations. Quick access to data is therefore essential. The original code by Federico D'Ambrosio was written in an object-oriented style. The atoms and bonds in a graphene sample were represented by custom classes `Atom` and `Bond`. The `Atom` class contained all information concerning the atom: position, velocity, total force acting on it, which atoms are its neighbors, which bonds connect to its neighbors, etc. The `Bond` class similarly contained all information concerning the bond: direction, length, which atoms it connects and whether or not it crosses the periodic boundaries of the box. The graphene configuration is then represented by an array of N `Atom`'s and an array of $3N/2$ `Bond`'s. This object-oriented approach makes the code more readable and easily adjustable. If one wanted to add the property of spin to atoms, one could easily do so without having to reconsider the existing code. This data structure has however its downsides. The array of atoms has the following structure in the memory:



The algorithm however only needs a few physical quantities at each step, updating the positions only requires the current positions, velocities and forces. To perform calculations on these variables they have to be moved to the L1 caches near the processors. The required quantities are however not stored together, but distributed among the `Atom`'s. The computer will either have to gather the desired variables before sending them to the cache, or it simply moves the entire `Atom` to the cache, dragging along useless information. Where the latter is especially costly as the transfer of data between memory components is proportionally very slow. Much more efficient is bundling each physical quantity i.e. an array of positions, an array of velocities, etc. The computer can then easily fetch the required information and move it to the caches, reducing the traffic between the memory components.

4.3.2 Pointers

Pointers are data types containing memory addresses. Computers use them to keep track of where data is stored and receive instructions where to move data to. Most programming languages allow the use of pointers in source code, enabling the programmer to use them as well. Pointers can be used to quickly access a piece of memory repetitively, or easily refer to larger amounts of data without having to make a copy. They can be an elegant and powerful tool in coding. Improper use of pointers

however, can lead to overflow and memory leaks. In `C` and `C++` pointer syntax is as follows:

`datatype var;`	declares a variable of data type `datatype` named `var`.
`datatype *p;`	declares a pointer named `p` pointing to a variable (or list variables) of data type `datatype`.
`p = &var;`	sets the pointer equal to the address of variable `var`.
`*p`	gives the value of the variable stored in memory pointed to by `p`.

Code using pointers affluently can make it harder to comprehend and can also reduce its performance. Let us consider the following code

```
int var;
int *p;
p = &var;

var = 1;
*p = 1;
```

where the last two lines of code both assign the value 1 to integer variable `var`. The second line is slightly slower as it makes the computer fetch the address stored in `p` before it goes to the location of `var`.
The original code used lists of pointers containing the locations of values representing the positions, velocities and forces acting on the atoms. An understandable choice considering the data structure problem described in section 4.3.1. Updating positions and velocities was thus done similarly to the following example code

```
double *pos_pointers[n];
double *vel_pointers[n];
double *force_pointers[n];

for( int i = 0 ; i < n ; i++ ){
    *pos_pointers[i] += *vel_pointers[i] * dt + (1.0 / mass) * dt * dt * *force_pointers[i];
}
```

where `n` $= 3N + 2$, three directions for each atom plus two for the box's sides. With the improved data structure, where each physical quantity is stored in a designated array, the following code

```
double positions[n];
double velocities[n];
double forces[n];

for( int i = 0 ; i < n ; i++ ){
    positions[i] += velocities[i] * dt + (1.0 / mass) * dt * dt * forces[i];
}
```

is equivalent. Besides looking more clean-cut, it is also more readable for compilers. Over the years compilers have been developed that can optimize code behind the scenes without loss of generality. To achieve such improvements the compiler needs as much information about the code as possible. Pointers however obscure some of the information as they merely contain addresses and no information about the size of the object. The abstract nature of pointers limits the compiler's ability to optimize.

4.3.3 Cache misses

A processor requires data to process otherwise it will stand idle waiting for instructions. Cache misses are instances where the processors idles because it needs to fetch data from a higher level memory location. Cache misses are costly because the transfer of information is relatively slow and the CPU will meanwhile be idle. To reduce communication and the occurrence of cache misses, data in the cache needs to be used to their maximal potential.

Large arrays may not fit inside the limited memory of an L1 cache. Code looping over such large arrays moves consecutive portions one by one to the CPU for processing. Multiple such loops involving the same array may result in portions going needlessly back and forth multiple times. Merging these loops into a single loop can improve the situation and incidentally reduce overhead costs. For example

```
double positions[n];
double velocities[n];
double forces[n];

for( int i = 0 ; i < n ; i++ ){
    positions[i] += velocities[i] * dt + (1.0 / mass) * dt * dt * forces[i];
}
for( int i = 0 ; i < n ; i++ ){
    velocities[i] += forces[i] * dt;
}
```

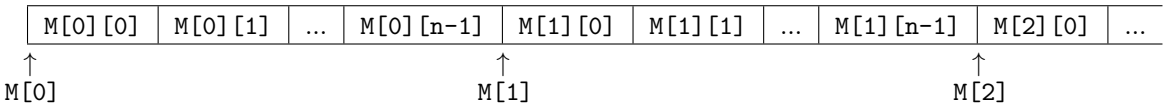
can be rewritten as

```
double positions[n];
double velocities[n];
double forces[n];

for( int i = 0 ; i < n ; i++ ){
    positions[i] += velocities[i] * dt + (1.0 / mass) * dt * dt * forces[i];
    velocities[i] += forces[i] * dt;
}
```

making sure the entries of `forces` are used for both updating the positions as well as updating the velocities, while they are still located in the cache.

Information is moved to and from the caches in bulk, whole lines of memory at once. When a variable is moved to the cache, its neighboring variables are moved along with it. Arrays are consecutively stored sequences of variables on the hardware level, making them ideal to take advantage of this mechanism. In the languages `C` and `C++` a matrix is merely an array with an additional structure: pointers referring to regular intervals of row length. A matrix with rows of length `n` is stored consecutively on a single line of memory in the following way:



Where `M[0]`, `M[1]`, `M[2]`, etc., are pointing to the start of each row. The order in which code loops over row or column indices can have significant impact, for example

```
double matrix[n][n];
double sum = 0.;

for( int i = 0 ; i < n ; i++ ){
    for( int j = 0 ; j < n ; j++ ){
     sum += matrix[i][j];
    }
}
```

is faster than

```
double matrix[n][n];
double sum = 0.;

for( int i = 0 ; i < n ; i++ ){
    for( int j = 0 ; j < n ; j++ ){
     sum += matrix[j][i];
    }
}
```

because the latter strides through the array. This drags neighboring variables along to the cache without using them once arrived.

4.3.4 Flop costs

Flops are a measure of computational cost, not in terms of physical time, but in relative operation cost. One flop is often defined as the computational cost of an addition of two `float`s on a CPU. For simplicity multiplications are also considered one flop each, while division cost four flops. Writing code in a flop reducing style is an easy yet effective way to achieve adequate performance. For example

```
double positions[n];
double velocities[n];
double forces[n];

for( int i = 0 ; i < n ; i++ ){
    positions[i] += velocities[i] * dt + (1.0 / mass) * dt * dt * forces[i];
    velocities[i] += forces[i] * dt;
}
```

can be rewritten as

```
double positions[n];
double velocities[n];
double forces[n];
double over_mass = 1.0 / mass;
double dt2 = dt * dt;

for( int i = 0 ; i < n ; i++ ){
    positions[i] += velocities[i] * dt + over_ mass * dt2 * forces[i];
    velocities[i] += forces[i] * dt;
}
```

ensuring the expensive division is not performed `n` times, while the introduction of `dt2` also saves a flop every iteration. More sophisticated flop cost reductions may employ mathematical equivalences like

$$\sin\big(\arccos(x)\big) = \sqrt{1 - x^2}$$

which can transform

```
double a,b;
double x,y,z;

x = a / b;
y = std::acos( x );
z = std::sin( y );
```

into

```
double a,b;
double x,y,z;

x = a / b;
y = std::acos( x );
z = std::sqrt( 1.0 - x * x );
```

where the `std::sin` costs about 14 flops, while `std::sqrt` costs 6 flops with an additional 2 flops for the calculation of its argument `1.0 - x * x` [63].

4.4 Parallelization

Parallelization is the art of running a program on multiple processors simultaneously to obtain maximal performance. Most computers nowadays have multiple processors, usually 2 or 4, while specialized servers can have hundreds or thousands. Huge variety in parallel computers and networks makes portability and algorithm design a challenge. Abstract parallel computer models were devised

to investigate parallel algorithms from a theoretical perspective. Special parallel libraries like MulticoreBSP and OpenMP have been developed to make parallel code exportable. Parallel programs are harder to optimize and more easily plagued by deadlock and nondeterministic behavior. Proficiency in parallel coding takes practice and regularly reading up on the latest developments. This section will introduce several concepts in parallel computing based on 'Parallel Scientific Computing' [40] and 'Using OpenMP' [41]. Short code segments demonstrate how one can remedy the subtle adversities that occur in parallel coding.

4.4.1 Parallel computer models

Parallel computers come in roughly two models: shared memory and distributed memory. In a shared memory parallel computer the CPU's are assumed to have quick access to a relatively large piece of memory with nearly equal access times, see Figure 4.2. The architecture in Figure 4.1 is also considered a shared memory parallel computer. Distributed memory parallel computers are assumed to have both instructions and data distributed over its cells or cores, see Figure 4.3, either due to the lack of a shared memory (networks) or relatively slow access (supercomputers). In the distributed memory model communication is considered so expensive that most of the algorithm design revolves around finding distributions that minimize the communication. Distributed memory algorithms are considered more general as any shared memory computer can be turned into a distributed memory computer by partitioning its memory.

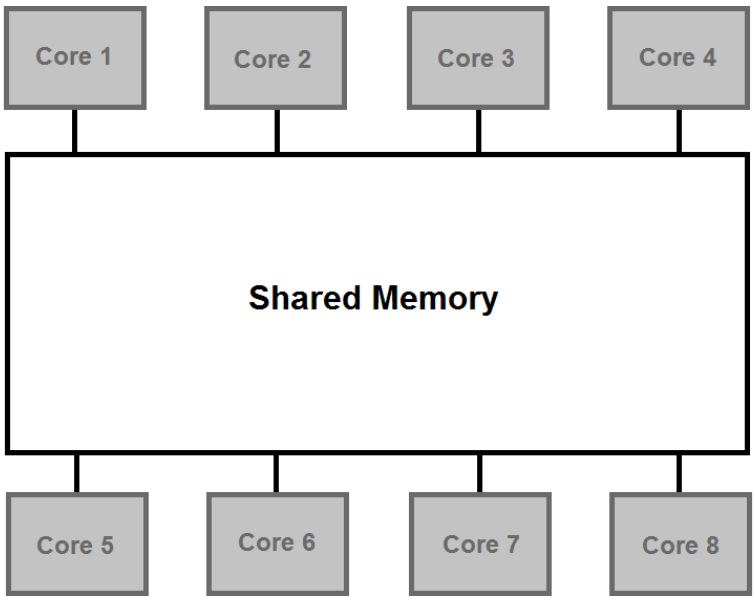


Figure 4.2: Schematic architecture of a shared memory parallel computer. All cores are expected to have very limited internal memory and have roughly the same access time to the shared memory.

A software thread is a sequence of instructions to be executed. Every piece of software has its own software thread, which are scheduled by the operating system for execution. The computer switches between all the software threads at high frequency, making it seem like multiple programs run simultaneously. The ability to handle multiple software threads is called multi-threading. A hardware thread is a CPU or processor where a software thread can be executed. A computer with multiple hardware threads can genuinely execute multiple software threads simultaneously. Some CPU's consist of multiple processors closely integrated, sharing resources and appearing to the operating system as multiple hardware threads. This close integration enables the threads to quickly use each other's resources when one idles, increasing the overall efficiency of the CPU. The ability for a single CPU to run multiple hardware threads is called hyper-threading.

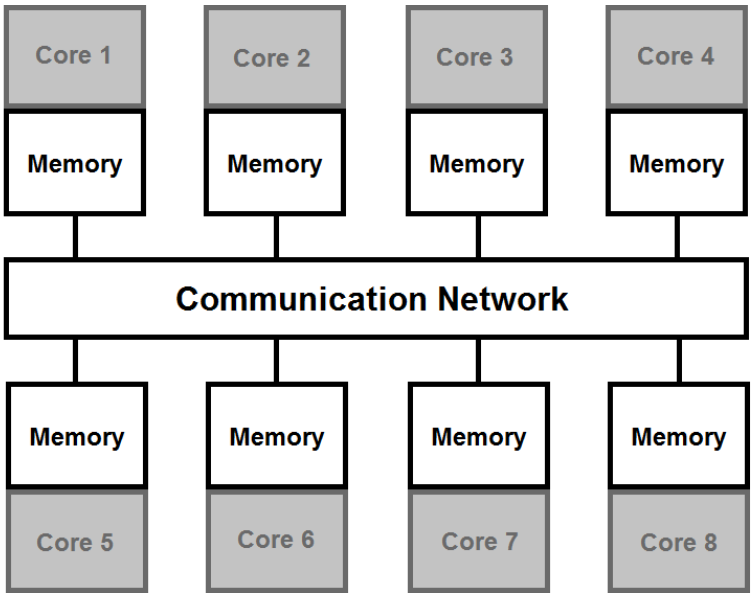


Figure 4.3: Schematic architecture of a distributed memory parallel computer or network of computers. Cores are assumed to have memory with a very fast access time nearby and very slow access to the other processors and their memory.

In parallel coding threads are organized in teams, each member with its own set of instructions, synchronizing with their team members to share information. Thread teams can be organized in two major styles: Bulk Synchronous Parallel (BSP) where all threads synchronize collectively for communication sessions, making it one team. Or nested threading, where threads can fork into teams of threads which only synchronize with their team members. Each team is headed by a master thread, which started the fork, and after execution the other team members will merge with the master thread again. Note that non-master threads can also fork and become masters of their own team. BSP is a distributed memory model, while most nested threading models are shared memory models.

Our parallel code uses OpenMP, an application programming interface (API) for shared memory parallelism. While OpenMP supports nested threading, our code uses a single team of threads that synchronize in bulk.

4.4.2 Distributions

Distribution of work is an essential part of the parallel algorithm design. The goal is a distribution with a balanced workload and minimal communication both in amount and number of synchronizations. With such an distribution processors work efficiently with little idle time. Finding the optimal distribution is a field of study in and of itself. Algorithms containing large loops often distribute the loop iterations among threads. Let p be the number of parallel threads and n the number of iterations, then the block distribution maps iterations to threads using

$$i \mapsto \left\lfloor \frac{i}{b} \right\rfloor \quad \text{with} \quad b = \left\lceil \frac{n}{p} \right\rceil \quad , \quad 0 \leq i < n \tag{4.1}$$

where b is the block size, visualized in Figure 4.4. Other prevalent distributions are the cyclic distribution

$$i \mapsto i \bmod p \quad \text{with} \quad 0 \leq i < n \tag{4.2}$$

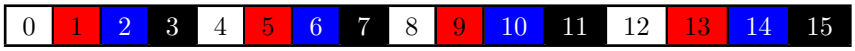
and the block-cyclic distribution

$$i \mapsto \left\lfloor \frac{i}{b} \right\rfloor \bmod p \quad \text{with} \quad b < \left\lceil \frac{n}{p} \right\rceil \quad , \quad 0 \leq i < n. \tag{4.3}$$

The block-cyclic distribution with block size $b = 2$ is also known as the bicyclic distribution.

Block distribution :

Cyclic distribution :



Block-cyclic distribution :

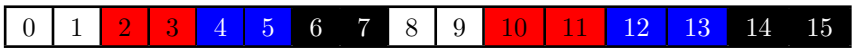


Figure 4.4: Three common distributions visualized for a loop of size $n = 16$ distributed over $p = 4$ threads.

The most optimal distribution often still requires communication between threads. To do this safely all threads should synchronize, share data, synchronize again and continue. Synchronizations however, lead to idle time and can be slow if processors are relatively far apart. It is therefore essential to keep the number of synchronizations to a minimum. Some parallel computers have limited bandwidth between the processors making the communication itself very expansive as well. Note however that the cost of a communication session is dictated by the maximum amount of data words received or send by any single processor, not the total amount of communication by all processors.

4.4.3 How OpenMP works

OpenMP is an application programming interface (API) that supports parallel programming in `Fortran`, `C` and `C++`, the latter will be used for our examples. It includes a collection of library routines and compiler directives to make regular code parallel. OpenMP is designed for the shared memory model and supports nested threading. The main syntax revolves around declaring parallel regions, sections of code to be executed by all members of a thread team. Parallel regions are stated by an directive and followed by clauses. The first thread to arrive at this directive forks into a team of threads. Each member of this team executes the code inside the parallel region and merges with the master thread at the end. Directives are preceded by `#pragma` a special kind of environment to provide information to the compiler. OpenMP has several directives, but our code examples only use three of them:

`omp parallel`	Starts a parallel region where every threads executes the block.
`omp master`	Creates a block that only the master thread will execute.
`omp barrier`	Creates a barrier where all team members wait for each other. This directive must be encountered by all team members or the others will wait forever resulting in deadlock.

The `omp parallel` directive needs clauses to function, in particular it needs to know which variables are 'shared' or 'private'. A shared variable will be accessible to all threads, the others will be affected if a thread changes it. Every processor will technically still work with a copy, but it will share changes to it with the other threads and fetch the latest version when necessary. Private variables are hidden from the other threads. Each thread will have to initialize its private variables before using them. Private variables are not automatically initialized, even if it was declared and initialized before the parallel region. The program will not check if the threads have the same private values and these values will not be available after the parallel region has been executed.

Clauses follow the directive to inform the compiler about variables and methods to be used in the parallel region. The status of variables is passed on to the compiler by the following clauses:

`default(none)`	Instructs the compiler not to automatically assume data-sharing. The programmer will have to provide explicitly which variables are private or shared.
`private(list)`	States that the variables in `list` are private.
`shared(list)`	States that the variables in `list` are shared.

OpenMP also provides essential built-in functions to manage the thread teams. Particularly useful functions are:

`int omp_get_max_threads()`	Returns the maximum number of threads available for the next parallel region. If called in the sequential part of the code, it will return the number of hardware threads.
`omp_set_num_threads(int)`	Sets the number of threads to be used in a parallel region to `int`. If this number exceeds the available hardware threads, it continues to create software threads.
`int omp_get_thread_num()`	Returns a number ranging from 0 to $m-1$ indicating the thread's identity within a team of m threads. Where thread number 0 is the team's master thread.

Parallel code is best structured in blocks, making it more readable and easier to adjust. Directives and build-in functions create these block as demonstrated in the following example code:

```
double data[n];
int i,j,k;
int tid;

//Initial sequential part of the program.

omp_set_num_threads(p);
#pragma omp parallel default(none) shared(n,data)\
    private(i,j,k,tid)
{
    tid = omp_get_thread_num();

    //This part is executed by every thread.

    if(tid == 3){
        //Only executed by thread 3.
    }
    #pragma barrier  //All threads wait, thread 3 may arrive later.

    #pragma omp master
    {
        //Only executed by the master thread (thread 0).
    }

    //This part is executed by all threads again, thread 0 may lag behind.
}

//Final sequential part of the program. Any changes to i,j,k inside
//the parallel region will not effect the i,j,k in this part. Changes
//to data[n] inside the parallel region will be present.
```

4.4.4 Racing conditions

Race conditions are situations where a program is sensitive to uncontrolled changes in the timing of instruction executions. Sequential programs rarely encounter racing conditions, but parallel programs need special care to avoid them. The most notorious race condition is the deadlock, where the program gets stuck because a thread is waiting for something that never arrives, or two threads are blocking each other's way. Synchronizations are essential to maintain coherence during execution.

The frequency at which processors operate fluctuates slightly. It may even slow down significantly when too much thermal energy builds up. As mentioned in section 4.4.1 processors quickly switch between software threads. The threads from other program are generally not equally distributed, which makes some processors take longer to perform their parallel tasks. Slight changes in execution time can lead to shared variables not being updated in time. These subtle uncontrollable changes are the main source of nondeterministic behavior, to remedy this behavior the code needs to be written in a robust style. Synchronizations are the most effective defense against this type of race condition. A variable may also be corrupted when updated by two threads simultaneously, or its copy gets corrupted because the variable was being updated by another thread during the copying process. The potential corruption of data in this way is called a data race.

As discussed in section 4.2, data types `float` and `double` represent real numbers in computing. These representations are inherently of finite precision, thus if two floating-point numbers differ too much, accuracy may be lost during computation. In particular, when very large and very small floating point are added or subtracted, it may simply return the largest one. While addition is mathematically commutative, in computing the result may depend on the order of summation. Consider for instance the parallel summation

```
double data[n];
double sum = 0.;
double localsum = 0.;
int i,tid;
int b = std::ceil( (double)n / (double)p );
int nstart,nstop;

omp_set_num_threads(p);
#pragma omp parallel default(none) shared(n,b,data,sum)\
    private(i,tid,nstart,nstop,localsum)
{
    tid = omp_get_thread_num();
    nstart = tid * b;
    nstop = std::min( n , nstart + b );
    localsum = 0.;
    for( i = nstart ; i < nstop ; i++ ){
        localsum += data[i];
    }
    sum += localsum;
}
```

where the local sum is added to the total sum whenever the thread finishes. The order of summation is thus not guaranteed and there is even risk of data race. To fix these issues and guarantee deterministic behavior, the local sums should be lined up before adding them together:

```
double data[n];
double localsums[p];
double sum = 0.;
double localsum;
int i,tid;
int b = std::ceil( (double)n / (double)p );
int nstart,nstop;
```

```
omp_set_num_threads(p);
#pragma omp parallel default(none) shared(n,b,data,sum,localsums)\
    private(i,tid,nstart,nstop,localsum)
{
    tid = omp_get_thread_num();
    nstart = tid * b;
    nstop = std::min( n , nstart + b );
    localsum = 0.;
    for( int i = nstart ; i < nstop ; i++ ){
        localsum += data[i];
    }
    localsums[tid] = localsum;
    #pragma omp barrier
    #pragma omp master
    {
        for( i = 0 ; i < p ; i++ ){
         sum += localsums[i];
        }
    }
}
```

Where the summation is always performed in the same order. All summations in our code are performed in this or similar ways to avoid racing conditions. The final sum however, may differ from the sequential version, as the order in parallel is different from the sequential order of summation.

4.4.5 Cache coherence

Some parallel computers have a special mechanism to avoid memory consistency problems, ensuring the L1 caches have the latest versions of variables. An example of such an cache coherence mechanism is the use of 'state bits' which flag a line of memory as outdated whenever something has been altered. The other processors are notified and will fetch the latest version of that line before using any variables from this memory line. Flagging the entire line however can lead to unnecessary fetches of variables that remain unchanged. This needless sharing and fetching of data is called false sharing. False sharing can be avoided by separating reading and writing operations. Variables can then safely be used without risk of using an outdated value. Let us first parallelize the code from section 4.3.4 using the block distribution (4.1).

```
double positions[n];
double velocities[n];
double forces[n];
double over_mass = 1.0 / mass;
double dt2 = dt * dt;

int i,tid;
int b = std::ceil( (double)n / (double)p );
int nstart,nstop;

omp_set_num_threads(p);
#pragma omp parallel default(none) private(i,tid,nstart,nstop)\
    shared(n,b,positions,velocities,forces,over_mass,dt,dt2)
{
    tid = omp_get_thread_num();
    nstart = tid * B;
    nstop = std::min( n , nstart + b );

    for( i = nstart ; i < nstop ; i++ ){
        positions[i] += velocities[i] * dt + over_ mass * dt2 * forces[i];
        velocities[i] += forces[i] * dt;
    }
}
```

Where the += operation reads the variable and writes the update value back to the same variable. This flags the variable and those sharing its line of memory as outdated for other processors. Reading and writing is separated by introducing temporary arrays `oldpositions` and `oldvelocities`.

```
double positions[n];
double oldpositions[n];
double velocities[n];
double oldvelocities[n];
double forces[n];
double over_mass = 1.0 / mass;
double dt2 = dt * dt;

int i,tid;
int b = std::ceil( (double)n / (double)p );
int Nstart,Nstop;

omp_set_num_threads(p);
#pragma omp parallel default(none) private(i,tid,Nstart,Nstop)\
    shared(n,b,positions,oldpositions,velocities,oldvelocities,forces,over_mass,dt,dt2)
{
    tid = omp_get_thread_num();
    nstart = tid * b;
    nstop = std::min( n , nstart + b );




    for( i = nstart ; i < nstop ; i++ ){
        oldpositions[i] = positions[i];
        oldvelocities[i] = velocities[i];
    }
    #pragma omp barrier
    for( i = nstart ; i < nstop ; i++ ){
        positions[i] = oldpositions + oldvelocities[i] * dt + over_ mass * dt2 * forces[i];
        velocities[i] = oldvelocities[i] + forces[i] * dt;
    }
}
```

Note the for-loops are separated by a synchronization for additional safety.

4.5 Results

The previous sections introduced various concepts in computing and discussed the issues that can occur. Example codes demonstrated bad practices in parallel coding and how to improve the various situations. This final section discusses the improved performance all these consideration have achieved in our parallel code. The success of parallelization in computer science is measured by the speedup gained and how scalable the algorithm is. Speedup and efficiency are defined by

$$S_p = \frac{T_{\text{seq}}}{T_p}, \qquad E_p = \frac{T_{\text{seq}}}{p\,T_p} \tag{4.4}$$

respectively, where p is the number of processors and T_{seq}, T_p the sequential and parallel computation times respectively. Note that the speedup compares multi-processor computation times with the sequential time, not $T_{p=1}$! Theoretically one expects

$$1 \leq S_p \leq p \tag{4.5}$$

but sometimes superlinear speedup is achieved by more efficient use of the caches. Efficiency reflects the parallel code's ability to scale with increasing numbers of processors p, compared to the ideal linear speedup.

The computation times are obtained by performing the FIRE algorithm, see Algorithm 2, on a graphene samples of different sizes. Each relaxation is performed 10 times to reduce the impact of fluctuations in the computers performance. Different numbers of processors lead to different orders of summation, which can change the number of iterations in the while-loop of part (2). The measured computation times reported in Table 4.1 are rescaled accordingly. All measurements are produced on a Intel Core i5-8250U at a clock speed of 1.60 GHz integrated in a HP laptop with 8 GB of DRAM with speed 2400 MHz and the 64-bit Windows 10 operating system. The Intel Core is an 64-bit quad-core with multi-threading and hosts a total of 8 processors. The measurements are assumed to be normally distributed random variable. Computing times can then be compared using the following approximation

$$\sigma_S^2 \approx \frac{\sigma_a^2}{\mu_b} + \frac{\mu_a^2\sigma_b^2}{\mu_b^4} - 2\rho\,\sigma_a\sigma_b\frac{\mu_a}{\mu_b^3} \quad \text{with} \quad S = \frac{T_a}{T_b} \quad \text{where} \quad T_a \sim \mathcal{N}\big(\mu_a, \sigma_a^2\big), \quad T_b \sim \mathcal{N}\big(\mu_b, \sigma_b^2\big) \tag{4.6}$$

where ρ is the correlation between T_a and T_b. The uncertainty in computation times T originates from random fluctuations in the hardware's performance, thus there are no correlations ($\rho = 0$).

The sequential speedup in Table 4.1 shows a slight increase with system size N, but not significantly. An weighted average

$$\overline{S}_{\text{seq}} = \frac{\sum_N S_{\text{seq}}^N \sigma_{S_{\text{seq}}^N}^{-2}}{\sum_N \sigma_{S_{\text{seq}}^N}^{-2}} \quad \text{with} \quad \sigma_{\overline{S}_{\text{seq}}}^2 = 1\Big/\sum_N \sigma_{S_{\text{seq}}^N}^{-2}\,. \tag{4.7}$$

suffices to extract the speedup achieved by source code improvements. The source code improvements discussed in section 4.3 lead to an average speedup of 1.6401(55), making our improved sequential code 64% faster!

The parallel speedup is extracted from the computation times in Table 4.1 using (4.6). The results are visualized in Figure 4.5 where the plumes are one standard deviation of uncertainty. Improved performance with increasing system size N and number of processors p in clearly visible. Note that for N=100 parallelization does not pay off, as the overhead costs are too high in proportion to the computational gain, this is also reflected in its low efficiency shown in Figure 4.6.

The main goal of this research project was to improve the code for maximal performance. A significant speedup opens the door to larger system sizes and generation of more data, enabling more future research. The total speedup achieved by the methods discussed in this chapter is visualized in Figure 4.7. Running on 8 processors, the improved code performs the FIRE algorithm upto 7.6 times faster than the original sequential code, accomplishing our main goal.

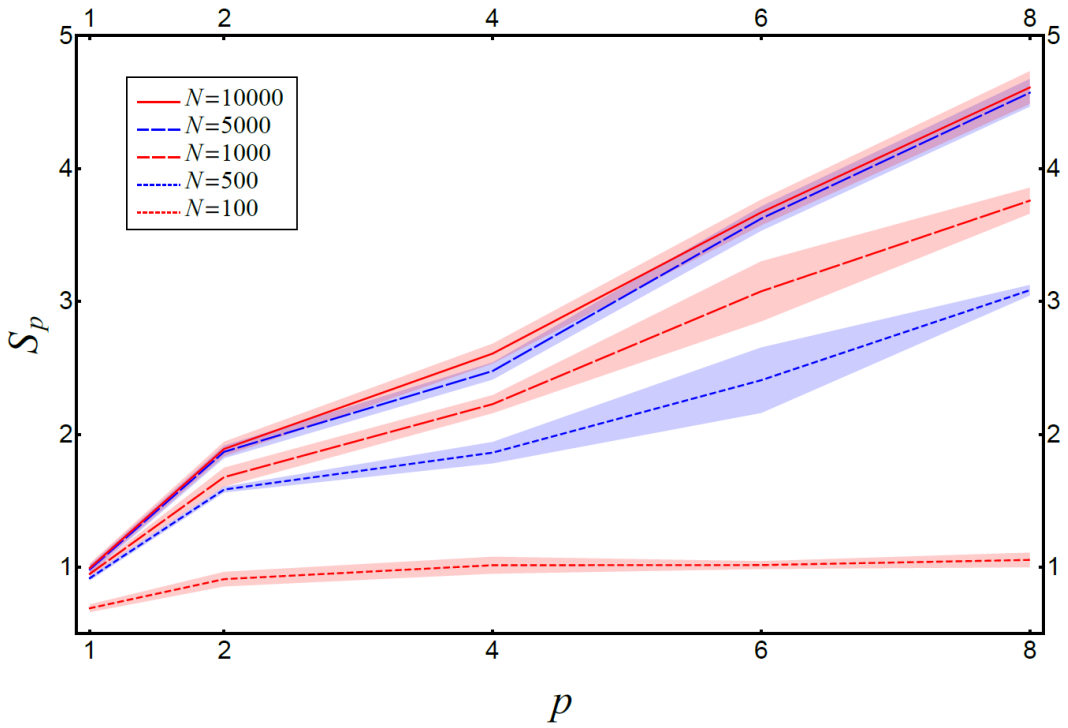


Figure 4.5: Speedup achieved by the parallel code with respect to our improved sequential code, based on the average rescaled computation times of 10 executions of the FIRE algorithm for samples with N atoms, see Table 4.1. The plumes are one standard deviation of uncertainty. The speedup barely improves due to parallelization for small system sizes, but scales well for large system sizes $N \geq 5000$.

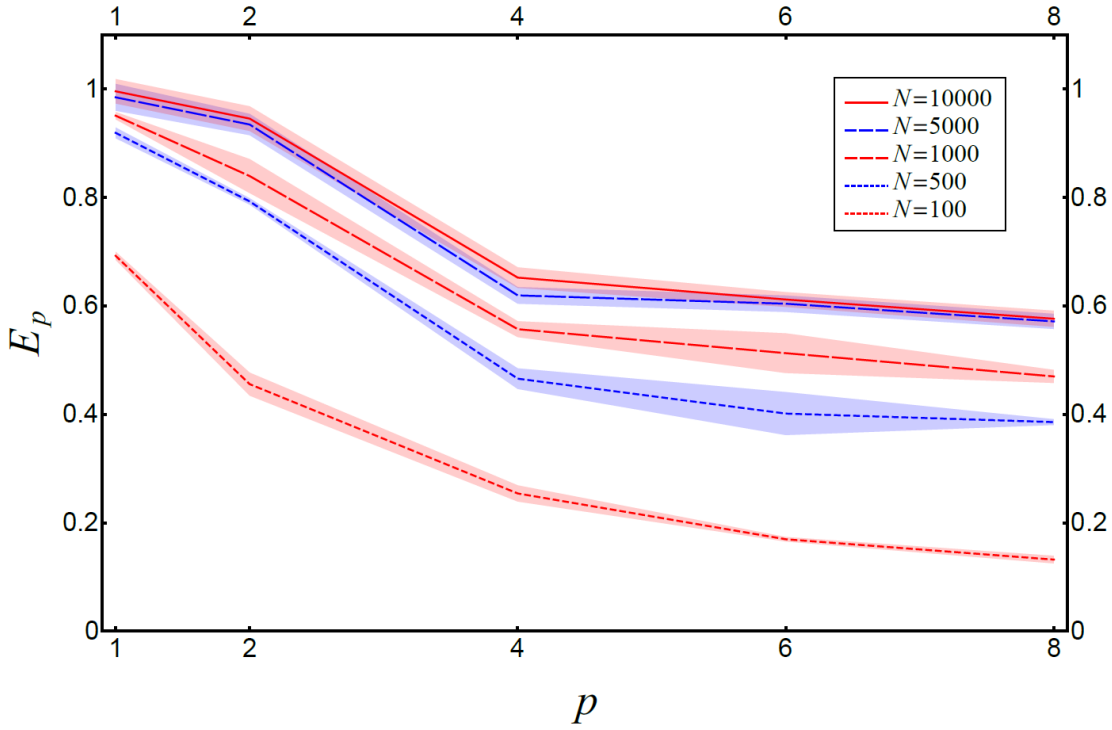


Figure 4.6: Efficiency of the parallel code for several system sizes N. Efficiency reflects how well the parallel code speeds up, with increasing numbers of processors p, compared to the ideal linear speedup. Results are based on the average rescaled computation times of 10 executions of the FIRE algorithm for samples with N atoms, see Table 4.1. The efficiency displayed here is very common in parallel computing. An efficiency above 0.5 is considered good scalability.

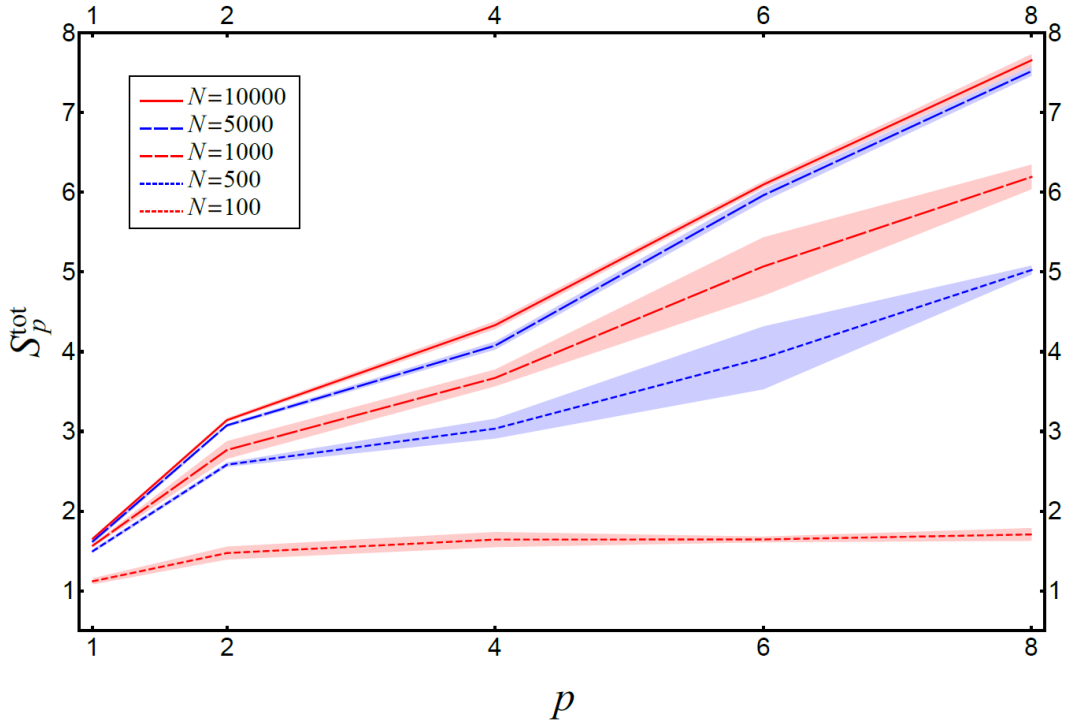


Figure 4.7: Speedup achieved by the parallel code with respect to the original code, based on the average rescaled computation times of 10 executions of the FIRE algorithm for samples with N atoms, see Table 4.1. The improved code is significantly faster and scales very well.

Computation times :

N	100	500	1000	5000	10000
T_{origin}	0.2128(20)	4.428(14)	16.999(55)	357.18(77)	622.9(23)
T_{seq}	0.1217(19)	2.718(12)	10.314(33)	217.0(43)	374.9(87)
$T_{p=1}$	0.1902(43)	2.957(21)	10.849(43)	220.4(28)	376.53(74)
$T_{p=2}$	0.1445(68)	1.7145(95)	6.14(22)	116.13(27)	198.30(46)
$T_{p=4}$	0.1296(67)	1.459(55)	4.63(19)	87.62(85)	143.7(12)
$T_{p=6}$	0.1295(16)	1.13(11)	3.35(23)	59.89(65)	102.15(38)
$T_{p=8}$	0.1247(50)	0.8810(72)	2.744(63)	47.49(29)	81.33(62)

Sequential speedup :

N	100	500	1000	5000	10000
S_{seq}	1.617(28)	1.6292(91)	1.6481(75)	1.646(32)	1.661(39)

Parallel speedup :

N	100	500	1000	5000	10000
$S_{p=1}$	0.692(19)	0.9190(78)	0.9507(48)	0.985(23)	0.996(23)
$S_{p=2}$	0.911(47)	1.585(11)	1.679(61)	1.869(37)	1.891(44)
$S_{p=4}$	1.016(54)	1.863(70)	2.228(58)	2.477(54)	2.608(64)
$S_{p=6}$	1.017(19)	2.41(23)	3.07(22)	3.624(81)	3.670(86)
$S_{p=8}$	1.056(45)	3.085(29)	3.759(87)	4.570(93)	4.61(11)

Table 4.1: Average rescaled computation times (in seconds) of 10 executions of the FIRE algorithm for samples with N atoms. For the computation times the programs from top to bottom are: the original code by Federico D'Ambrosio, our improved sequential code and the parallel code with increasing number of processors p. Produced on an Intel Core i5-8250U at clock speed 1.60 GHz.

Conclusions

Polycrystalline graphene naturally buckles out-of-plane forming an three-dimensional structure. Applying a continuously increasing or decreasing stretching force on a sheet of polycrystalline leads to discontinuous structural evolution. A very small change in the stretching force can induce a significant change the preferred structural configuration. These sudden changes cause displacements which in turn create vibrations through the system. We expect these vibrations to be experimentally observable and carry information about the structure of the material. Performing a cycle of stretching and relaxing on a sample can return it to a different state then before, clearly an hysteretic behavior. Continuous application of stress to polycrystalline graphene changes the shape of its ridges and vertices, which can be consequential for the electronic and mechanical properties.

This behavior originates from the embedding of a two-dimensional material in a three-dimensional space and should therefore be pertinent in other polycrystalline two-dimensional materials as well.

The program optimization through implementation improvements and parallelization was very successful. The improved code show good scaling and is upto 7.6 times faster on 8 processors when run on an Intel Core i5-8250U. This significant speedup will enable more research in the future.

Appendix A: Forces

Two-Body Interaction

The two-body interactions arise from the covalent bonds preferring an ideal length of $d = 1.42\,\text{Å}$. The potential energy associated with deviation from this ideal distance is

$$U_2 = \frac{3}{16}\frac{\alpha}{d^2}\left(r_{ij}^2 - d^2\right)^2 = \frac{3}{16}\frac{\alpha}{d^2}\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right)^2 \tag{A.8}$$

where r_{ij} is the bond length between atoms i and j in Å. Let us define the gradients

$$\boldsymbol{\nabla}_i = \left(\frac{d}{dr_i^x}, \frac{d}{dr_i^y}, \frac{d}{dr_i^z}\right) \qquad \mathbf{r}_i = (r_i^x, r_i^y, r_i^z) \tag{A.9}$$

where x_i, y_i, z_i are the spatial coordinates of atom i, to analyze how U_2 changes with variation in the position of the atoms involved, which leads directly to the forces acting on said atoms. Since $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ we find

$$\begin{aligned}
\boldsymbol{\nabla}_i(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij}) =& \boldsymbol{\nabla}_i\left[(\mathbf{r}_j - \mathbf{r}_i)\cdot(\mathbf{r}_j - \mathbf{r}_i)\right] \\
=& \boldsymbol{\nabla}_i(\mathbf{r}_i\cdot\mathbf{r}_i - 2\,\mathbf{r}_i\cdot\mathbf{r}_j + \mathbf{r}_j\cdot\mathbf{r}_j) \\
=& \boldsymbol{\nabla}_i(\mathbf{r}_i\cdot\mathbf{r}_i) - 2\boldsymbol{\nabla}_i(\mathbf{r}_i\cdot\mathbf{r}_j) \\
=& 2\,\mathbf{r}_i - 2\,\mathbf{r}_j \\
=& -2\,\mathbf{r}_{ij}
\end{aligned} \qquad
\begin{aligned}
\boldsymbol{\nabla}_j(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij}) =& \boldsymbol{\nabla}_j\left[(\mathbf{r}_j - \mathbf{r}_i)\cdot(\mathbf{r}_j - \mathbf{r}_i)\right] \\
=& \boldsymbol{\nabla}_j(\mathbf{r}_j\cdot\mathbf{r}_j - 2\,\mathbf{r}_j\cdot\mathbf{r}_i + \mathbf{r}_i\cdot\mathbf{r}_i) \\
=& \boldsymbol{\nabla}_j(\mathbf{r}_j\cdot\mathbf{r}_j) - 2\boldsymbol{\nabla}_j(\mathbf{r}_j\cdot\mathbf{r}_i) \\
=& 2\,\mathbf{r}_j - 2\,\mathbf{r}_i \\
=& 2\,\mathbf{r}_{ij}
\end{aligned} \tag{A.10}$$

$$\begin{aligned}
\mathbf{F}_i^{\text{bond}} =& -\boldsymbol{\nabla}_i U_2 \\
=& -\frac{3}{16}\frac{\alpha}{d^2}\boldsymbol{\nabla}_i\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right)^2 \\
=& -\frac{3}{8}\frac{\alpha}{d^2}\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right)\boldsymbol{\nabla}_i\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right) \\
=& \frac{3}{4}\frac{\alpha}{d^2}\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right)\mathbf{r}_{ij}
\end{aligned} \qquad
\begin{aligned}
\mathbf{F}_j^{\text{bond}} =& -\boldsymbol{\nabla}_j U_2 \\
=& -\frac{3}{16}\frac{\alpha}{d^2}\boldsymbol{\nabla}_j\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right)^2 \\
=& -\frac{3}{8}\frac{\alpha}{d^2}\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right)\boldsymbol{\nabla}_j\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right) \\
=& -\frac{3}{4}\frac{\alpha}{d^2}\left(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij} - d^2\right)\mathbf{r}_{ij}.
\end{aligned} \tag{A.11}$$

Three-Body Interaction

The three-body interactions arise from the atoms preferring their covalent bonds distributed equally with angles of $2\pi/3$ between them. The associated potential energy is

$$U_3 = \frac{3}{8}\beta\, d^2\left(\theta_{i,jk} - \frac{2\pi}{3}\right)^2 \tag{A.12}$$

where $\theta_{i,jk}$ is the angle between the two bonds going from atom i to neighboring atoms j and k in radians (rad), which can be expressed as

$$\theta_{i,jk} = \arccos\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right) = \arccos\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}}\right). \tag{A.13}$$

Since this expression is already quite complicated, we first calculate its gradients before we look at those of the whole U_3. The first step is to use the chain rule to circumvent the arccosine function

$$\begin{aligned}
\boldsymbol{\nabla}\theta_{i,jk} =& \boldsymbol{\nabla}\arccos\big(\cos(\theta_{i,jk})\big) \\
=& \frac{-1}{\sqrt{1-\cos(\theta_{i,jk})^2}}\boldsymbol{\nabla}\cos(\theta_{i,jk}) \\
=& \frac{-1}{|\sin(\theta_{i,jk})|}\boldsymbol{\nabla}\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right) \\
=& \frac{-1}{|\sin(\theta_{i,jk})|}\left(\frac{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|\boldsymbol{\nabla}(\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}) + (\mathbf{r}_{ij}\cdot\mathbf{r}_{ik})\boldsymbol{\nabla}\big(\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|\big)}{\|\mathbf{r}_{ij}\|^2\,\|\mathbf{r}_{ik}\|^2}\right)
\end{aligned} \tag{A.14}$$

then with the explicit gradients below, we can rewrite the three-body forces on the next page.

$$\begin{aligned}
\boldsymbol{\nabla}_i(\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}) =& \boldsymbol{\nabla}_i\left[(\mathbf{r}_j-\mathbf{r}_i)\cdot(\mathbf{r}_k-\mathbf{r}_i)\right] \\
=& \boldsymbol{\nabla}_i[\mathbf{r}_i\cdot\mathbf{r}_i-\mathbf{r}_i\cdot(\mathbf{r}_j+\mathbf{r}_k)+\mathbf{r}_j\cdot\mathbf{r}_k] \\
=& \boldsymbol{\nabla}_i(\mathbf{r}_i\cdot\mathbf{r}_i)-\boldsymbol{\nabla}_i[\mathbf{r}_i\cdot(\mathbf{r}_j+\mathbf{r}_k)] \\
=& 2\,\mathbf{r}_i-\mathbf{r}_j-\mathbf{r}_k \\
=& -\mathbf{r}_{ij}-\mathbf{r}_{ik}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\nabla}_i\big(\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|\big) =& \boldsymbol{\nabla}_i\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})} \\
=& \frac{\boldsymbol{\nabla}_i[(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})]}{2\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& \frac{(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})\boldsymbol{\nabla}_i(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})+(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})\boldsymbol{\nabla}_i(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}{2\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& -\frac{(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})\mathbf{r}_{ij}+(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})\mathbf{r}_{ik}}{\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& -\frac{\|\mathbf{r}_{ik}\|}{\|\mathbf{r}_{ij}\|}\mathbf{r}_{ij}-\frac{\|\mathbf{r}_{ij}\|}{\|\mathbf{r}_{ik}\|}\mathbf{r}_{ik}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\nabla}_j(\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}) =& \boldsymbol{\nabla}_j\left[(\mathbf{r}_j-\mathbf{r}_i)\cdot(\mathbf{r}_k-\mathbf{r}_i)\right] \\
=& \boldsymbol{\nabla}_j[\mathbf{r}_j\cdot(\mathbf{r}_k-\mathbf{r}_i)-\mathbf{r}_i\cdot(\mathbf{r}_k-\mathbf{r}_i)] \\
=& \boldsymbol{\nabla}_j[\mathbf{r}_j\cdot(\mathbf{r}_k-\mathbf{r}_i)] \\
=& \mathbf{r}_k-\mathbf{r}_i \\
=& \mathbf{r}_{ik}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\nabla}_j\big(\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|\big) =& \boldsymbol{\nabla}_j\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})} \\
=& \frac{\boldsymbol{\nabla}_j[(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})]}{2\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& \frac{(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})\boldsymbol{\nabla}_j(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})}{2\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& \frac{(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})\mathbf{r}_{ij}}{\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& \frac{\|\mathbf{r}_{ik}\|}{\|\mathbf{r}_{ij}\|}\mathbf{r}_{ij}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\nabla}_k(\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}) =& \boldsymbol{\nabla}_k\left[(\mathbf{r}_j-\mathbf{r}_i)\cdot(\mathbf{r}_k-\mathbf{r}_i)\right] \\
=& \boldsymbol{\nabla}_k[\mathbf{r}_k\cdot(\mathbf{r}_j-\mathbf{r}_i)-\mathbf{r}_i\cdot(\mathbf{r}_j-\mathbf{r}_i)] \\
=& \boldsymbol{\nabla}_k[\mathbf{r}_k\cdot(\mathbf{r}_j-\mathbf{r}_i)] \\
=& \mathbf{r}_j-\mathbf{r}_i \\
=& \mathbf{r}_{ij}
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\nabla}_k\big(\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|\big) =& \boldsymbol{\nabla}_k\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})} \\
=& \frac{\boldsymbol{\nabla}_k[(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})]}{2\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& \frac{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})\boldsymbol{\nabla}_k(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}{2\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& \frac{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})\mathbf{r}_{ik}}{\sqrt{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ij})(\mathbf{r}_{ik}\cdot\mathbf{r}_{ik})}} \\
=& \frac{\|\mathbf{r}_{ij}\|}{\|\mathbf{r}_{ik}\|}\mathbf{r}_{ik}
\end{aligned}$$

(A.15)

$$
\begin{aligned}
\mathbf{F}_i^{\text{angle}} =& -\boldsymbol{\nabla}_i U_3 \\
=& -\frac{3}{8}\beta\, d^2 \boldsymbol{\nabla}_i \left(\theta_{i,jk} - \frac{2\pi}{3}\right)^2 \\
=& -\frac{3}{4}\beta\, d^2 \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \boldsymbol{\nabla}_i \theta_{i,jk} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \boldsymbol{\nabla}_i \left(\frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right) \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \frac{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\| \boldsymbol{\nabla}_i(\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}) - (\mathbf{r}_{ij}\cdot\mathbf{r}_{ik})\boldsymbol{\nabla}_i(\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|)}{\|\mathbf{r}_{ij}\|^2\|\mathbf{r}_{ik}\|^2} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \frac{(\mathbf{r}_{ij}\cdot\mathbf{r}_{ik})\left(\frac{\|\mathbf{r}_{ik}\|}{\|\mathbf{r}_{ij}\|}\mathbf{r}_{ij} + \frac{\|\mathbf{r}_{ij}\|}{\|\mathbf{r}_{ik}\|}\mathbf{r}_{ik}\right) - \|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|(\mathbf{r}_{ij}+\mathbf{r}_{ik})}{\|\mathbf{r}_{ij}\|^2\|\mathbf{r}_{ik}\|^2} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \left[\frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\left(\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^2} + \frac{\mathbf{r}_{ik}}{\|\mathbf{r}_{ik}\|^2}\right) - \frac{\mathbf{r}_{ij}+\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right] \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \left[\cos(\theta_{i,jk})\left(\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^2} + \frac{\mathbf{r}_{ik}}{\|\mathbf{r}_{ik}\|^2}\right) - \frac{\mathbf{r}_{ij}+\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right] \\
\mathbf{F}_j^{\text{angle}} =& -\boldsymbol{\nabla}_j U_3 \\
=& -\frac{3}{8}\beta\, d^2 \boldsymbol{\nabla}_j \left(\theta_{i,jk} - \frac{2\pi}{3}\right)^2 \\
=& -\frac{3}{4}\beta\, d^2 \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \boldsymbol{\nabla}_j \theta_{i,jk} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \boldsymbol{\nabla}_j \left(\frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right) \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \frac{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\| \boldsymbol{\nabla}_j(\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}) - (\mathbf{r}_{ij}\cdot\mathbf{r}_{ik})\boldsymbol{\nabla}_j(\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|)}{\|\mathbf{r}_{ij}\|^2\|\mathbf{r}_{ik}\|^2} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \frac{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|\mathbf{r}_{ik} - (\mathbf{r}_{ij}\cdot\mathbf{r}_{ik})\frac{\|\mathbf{r}_{ik}\|}{\|\mathbf{r}_{ij}\|}\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^2\|\mathbf{r}_{ik}\|^2} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \left[\frac{\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|} - \frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^2}\right] \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \left[\frac{\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|} - \cos(\theta_{i,jk})\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^2}\right] \\
\mathbf{F}_k^{\text{angle}} =& -\boldsymbol{\nabla}_k U_3 \\
=& -\frac{3}{8}\beta\, d^2 \boldsymbol{\nabla}_k \left(\theta_{i,jk} - \frac{2\pi}{3}\right)^2 \\
=& -\frac{3}{4}\beta\, d^2 \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \boldsymbol{\nabla}_k \theta_{i,jk} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \boldsymbol{\nabla}_k \left(\frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\right) \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \frac{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\| \boldsymbol{\nabla}_k(\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}) - (\mathbf{r}_{ij}\cdot\mathbf{r}_{ik})\boldsymbol{\nabla}_k(\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|)}{\|\mathbf{r}_{ij}\|^2\|\mathbf{r}_{ik}\|^2} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \frac{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|\mathbf{r}_{ij} - (\mathbf{r}_{ij}\cdot\mathbf{r}_{ik})\frac{\|\mathbf{r}_{ij}\|}{\|\mathbf{r}_{ik}\|}\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|^2\|\mathbf{r}_{ik}\|^2} \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \left[\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|} - \frac{\mathbf{r}_{ij}\cdot\mathbf{r}_{ik}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|}\frac{\mathbf{r}_{ik}}{\|\mathbf{r}_{ik}\|^2}\right] \\
=& \frac{3\,\beta\, d^2}{4\,|\sin(\theta_{i,jk})|} \left(\theta_{i,jk} - \frac{2\pi}{3}\right) \left[\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|\,\|\mathbf{r}_{ik}\|} - \cos(\theta_{i,jk})\frac{\mathbf{r}_{ik}}{\|\mathbf{r}_{ik}\|^2}\right].
\end{aligned}
\tag{A.16}
$$

Four-Body Interaction

The four-body interactions arise from the covalent bonds being of type sp^2, which are planar, leading to an additional energy penalty for an atom sticking out of plane compared to its three neighbors. This additional energy penalty is expressed by the potential energy

$$U_4 = \gamma\, r_{i,jkl}^2 \tag{A.17}$$

where $r_{i,jkl}$ is the distance between atom i and the plane spanned through the three neighbors j, k, l. This distance squared can be expressed as

$$r_{i,jkl}^2 = \frac{(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2}{\|\mathbf{n}_i\|^2} = \frac{(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2}{\mathbf{n}_i\cdot\mathbf{n}_i} \qquad \mathbf{n}_i = (\mathbf{r}_{ik}-\mathbf{r}_{ij})\times(\mathbf{r}_{il}-\mathbf{r}_{ij}) \tag{A.18}$$

where $\mathbf{n}_i$ is the normal vector to the plane through the neighbors of atom i. First we observe that $\mathbf{n}_i$ is independent of $\mathbf{r}_i$ and remind ourselves of the following mathematical identities

$$\begin{aligned}
\boldsymbol{\nabla}_i[\mathbf{r}_i\cdot(\mathbf{r}_j\times\mathbf{r}_k)] &= \mathbf{r}_j\times\mathbf{r}_k \\
\boldsymbol{\nabla}_j[\mathbf{r}_i\cdot(\mathbf{r}_j\times\mathbf{r}_k)] &= \mathbf{r}_k\times\mathbf{r}_i \\
\boldsymbol{\nabla}_k[\mathbf{r}_i\cdot(\mathbf{r}_j\times\mathbf{r}_k)] &= \mathbf{r}_i\times\mathbf{r}_j
\end{aligned} \tag{A.19}$$

leading to the following gradients

$$\begin{aligned}
\boldsymbol{\nabla}_i(\mathbf{n}_i\cdot\mathbf{n}_i) &= 0 & \boldsymbol{\nabla}_i(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2 &= -2(\mathbf{r}_{ij}\cdot\mathbf{n}_i)\mathbf{n}_i \\
\boldsymbol{\nabla}_j(\mathbf{n}_i\cdot\mathbf{n}_i) &= 2(\mathbf{r}_{ik}-\mathbf{r}_{il})\times\mathbf{n}_i & \boldsymbol{\nabla}_j(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2 &= 2(\mathbf{r}_{ij}\cdot\mathbf{n}_i)[\mathbf{n}_i+(\mathbf{r}_{ik}-\mathbf{r}_{il})\times\mathbf{r}_{ij}] \\
\boldsymbol{\nabla}_k(\mathbf{n}_i\cdot\mathbf{n}_i) &= 2(\mathbf{r}_{il}-\mathbf{r}_{ij})\times\mathbf{n}_i & \boldsymbol{\nabla}_k(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2 &= 2(\mathbf{r}_{ij}\cdot\mathbf{n}_i)(\mathbf{r}_{il}\times\mathbf{r}_{ij}) \\
\boldsymbol{\nabla}_l(\mathbf{n}_i\cdot\mathbf{n}_i) &= 2(\mathbf{r}_{ij}-\mathbf{r}_{ik})\times\mathbf{n}_i & \boldsymbol{\nabla}_l(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2 &= 2(\mathbf{r}_{ij}\cdot\mathbf{n}_i)(\mathbf{r}_{ij}\times\mathbf{r}_{ik})
\end{aligned} \tag{A.20}$$

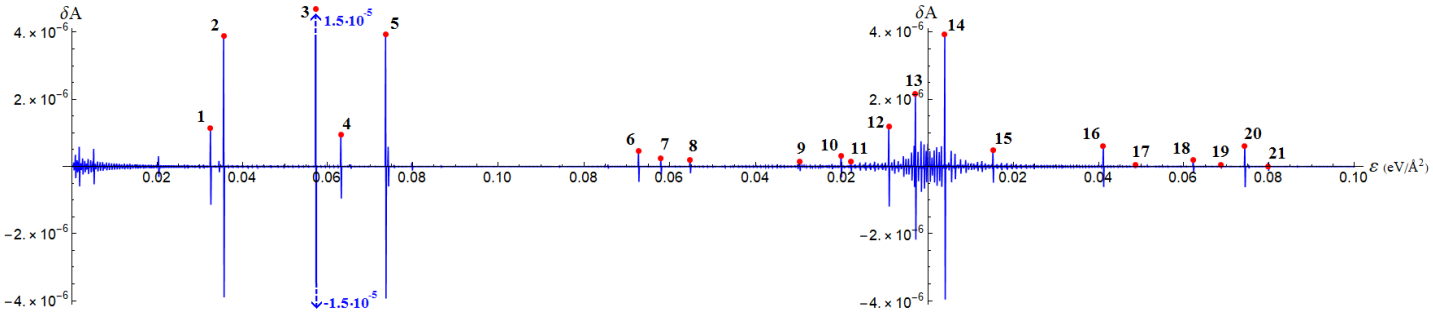
from which we derive the four-body forces

$$\begin{aligned}
\mathbf{F}_i^{\text{plane}} &= -\boldsymbol{\nabla}_i U_4 \\
&= -\gamma\boldsymbol{\nabla}_i\frac{(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2}{\mathbf{n}_i\cdot\mathbf{n}_i} \\
&= -\gamma\frac{(\mathbf{n}_i\cdot\mathbf{n}_i)\boldsymbol{\nabla}_i(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2-(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2\boldsymbol{\nabla}_i(\mathbf{n}_i\cdot\mathbf{n}_i)}{(\mathbf{n}_i\cdot\mathbf{n}_i)^2} \\
&= 2\gamma\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\mathbf{n}_i \\
\mathbf{F}_j^{\text{plane}} &= -\boldsymbol{\nabla}_j U_4 \\
&= -\gamma\boldsymbol{\nabla}_j\frac{(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2}{\mathbf{n}_j\cdot\mathbf{n}_i} \\
&= -\gamma\frac{(\mathbf{n}_i\cdot\mathbf{n}_i)\boldsymbol{\nabla}_j(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2-(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2\boldsymbol{\nabla}_j(\mathbf{n}_i\cdot\mathbf{n}_i)}{(\mathbf{n}_i\cdot\mathbf{n}_i)^2} \\
&= 2\gamma\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}[(\mathbf{r}_{il}-\mathbf{r}_{ik})\times\mathbf{r}_{ij}-\mathbf{n}_i]+2\gamma\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\right)^2(\mathbf{r}_{ik}-\mathbf{r}_{il})\times\mathbf{n}_i \\
\mathbf{F}_k^{\text{plane}} &= -\boldsymbol{\nabla}_k U_4 \\
&= -\gamma\boldsymbol{\nabla}_k\frac{(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2}{\mathbf{n}_j\cdot\mathbf{n}_i} \\
&= -\gamma\frac{(\mathbf{n}_i\cdot\mathbf{n}_i)\boldsymbol{\nabla}_k(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2-(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2\boldsymbol{\nabla}_k(\mathbf{n}_i\cdot\mathbf{n}_i)}{(\mathbf{n}_i\cdot\mathbf{n}_i)^2} \\
&= 2\gamma\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}(\mathbf{r}_{ij}\times\mathbf{r}_{il})+2\gamma\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\right)^2(\mathbf{r}_{il}-\mathbf{r}_{ij})\times\mathbf{n}_i
\end{aligned} \tag{A.21}$$

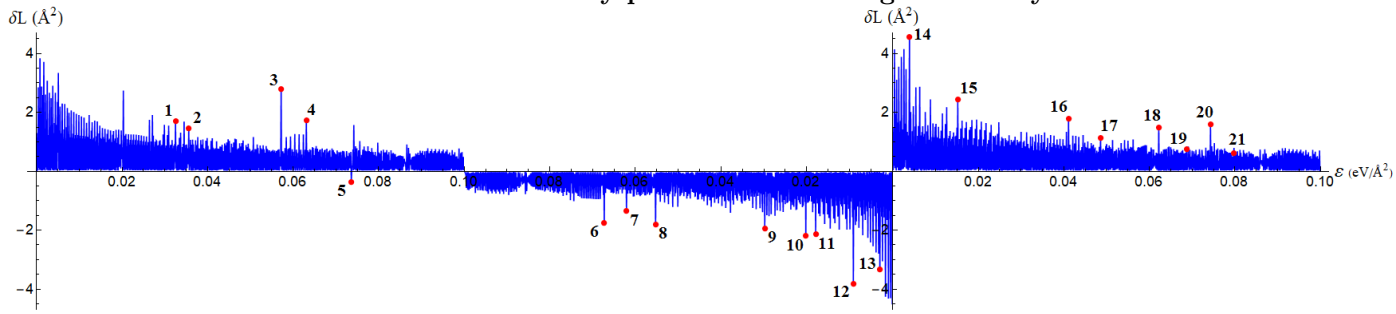
$$
\begin{aligned}
\mathbf{F}_l^{\text{plane}} &= -\boldsymbol{\nabla}_l U_4 \\
&= -\gamma \boldsymbol{\nabla}_l \frac{(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2}{\mathbf{n}_j\cdot\mathbf{n}_i} \\
&= -\gamma \frac{(\mathbf{n}_i\cdot\mathbf{n}_i)\boldsymbol{\nabla}_l(\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2 - (\mathbf{r}_{ij}\cdot\mathbf{n}_i)^2\boldsymbol{\nabla}_l(\mathbf{n}_i\cdot\mathbf{n}_i)}{(\mathbf{n}_i\cdot\mathbf{n}_i)^2} \\
&= 2\gamma\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}(\mathbf{r}_{ik}\times\mathbf{r}_{ij}) + 2\gamma\left(\frac{\mathbf{r}_{ij}\cdot\mathbf{n}_i}{\mathbf{n}_i\cdot\mathbf{n}_i}\right)^2(\mathbf{r}_{ij}-\mathbf{r}_{ik})\times\mathbf{n}_i
\end{aligned}
$$

Appendix B: Strain Cycle

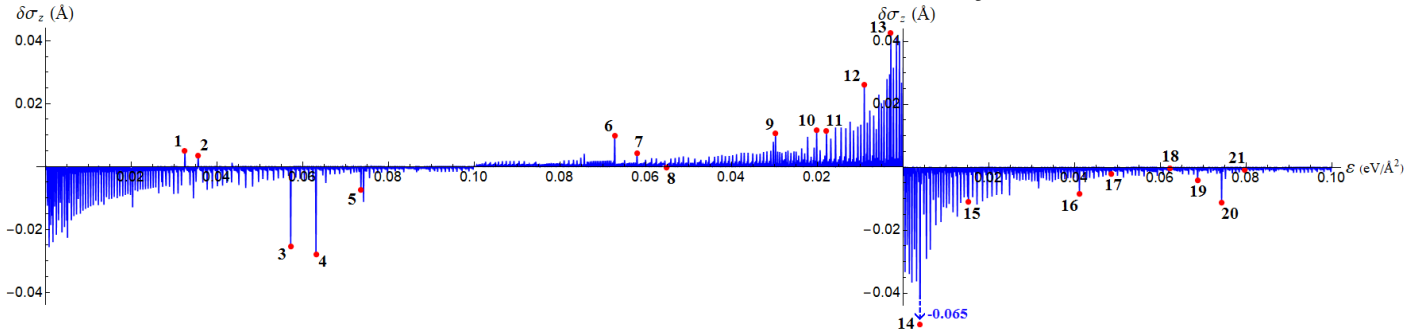
The discontinuous evolution of stretching polycrystalline graphene can be traced using various parameters. This appendix shows how the internal energy and its three components (1.3) evolve during a strain cycle. The evolution of structural parameters non-affinity A, area L and out-of-plane deviation σ_z discussed in Chapter 3 are displayed below. A graphene sample of $N = 3200$ atoms is stretched, fully relaxed and stretched again in small steps $\delta\varepsilon = 0.0001\,\mathrm{eV/\AA^2}$ where δP is the change in parameter P each step. The non-affinity parameter clearly shows sharp spikes that indicate discontinuous changes in the structure of the graphene, as earlier demonstrated by D'Ambrosio *et al.* [31]. The other two structural parameters show more 'noise' but still feature spikes. Note that a large δA spike does not guarantee an equally large spike in the other two parameters.



Discontinuous evolution of non-affinity parameter A during a strain cycle.

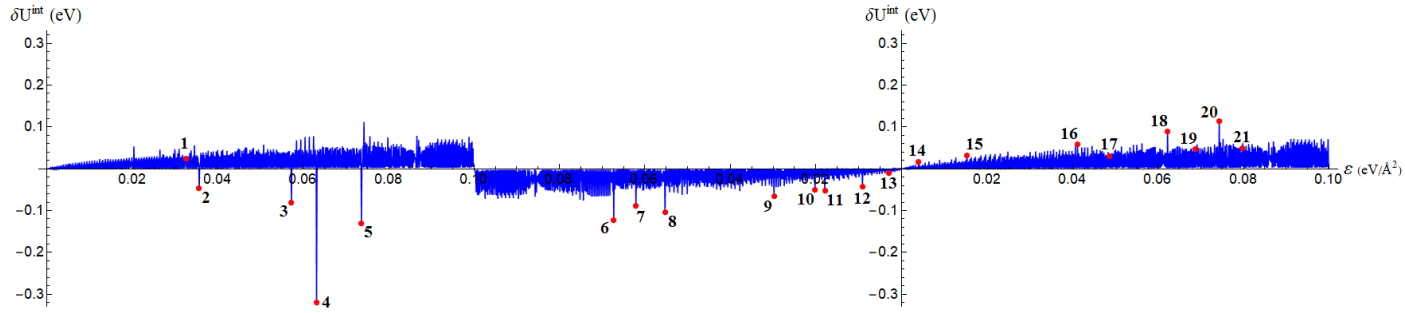


Discontinuous evolution of area spanned by box's sides L_x and L_y during a strain cycle.

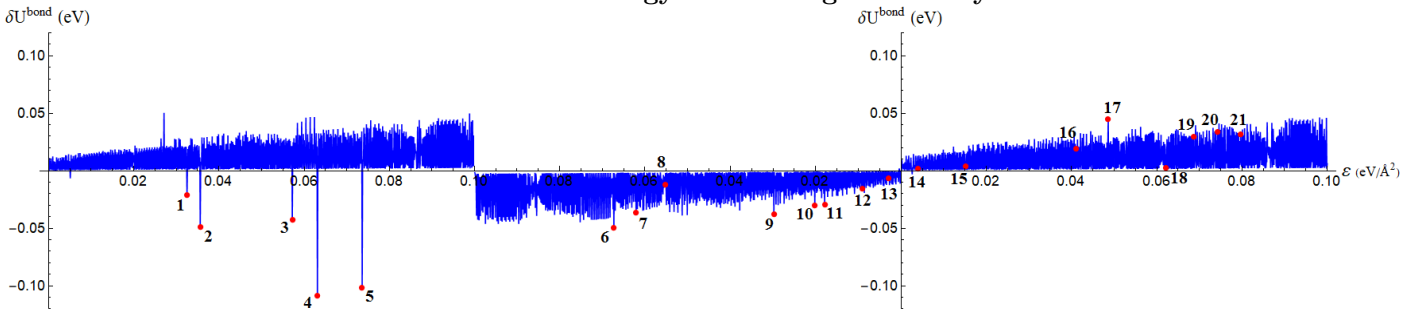


Out-of-plane deviation of atoms with respect to average z during a strain cycle.

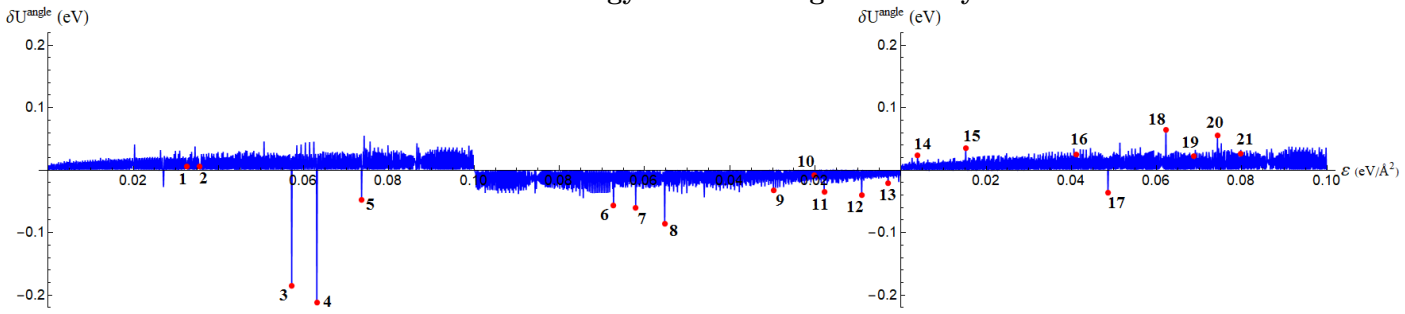
The discontinuous evolution of the internal energy and its components during the stretching are shown below. The 'noise plumes' are bigger for the bond energy indicating that most of the energy is stored in bonds during stretching. Notice that crackles have large differences between the three energy components. Crackle 4 for instance, has a very small spike in plane energy despite its large spike the total internal energy. Some crackles are not very energetic in themselves, but transform energy between the different types, for example crackles 1 and 17.



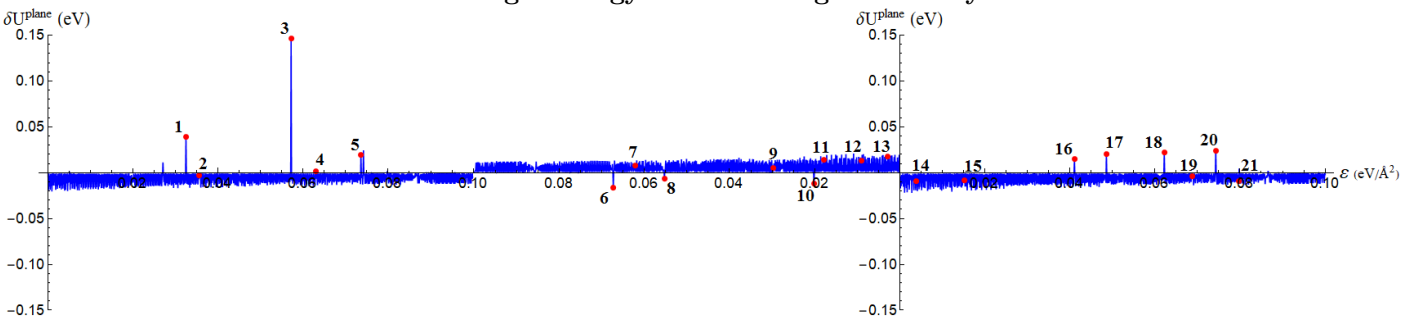
Discontinuous evolution of internal energy U^{int} during a strain cycle.



Discontinuous evolution of bond energy U^{bond} during a strain cycle.



Discontinuous evolution of angle energy U^{angle} during a strain cycle.

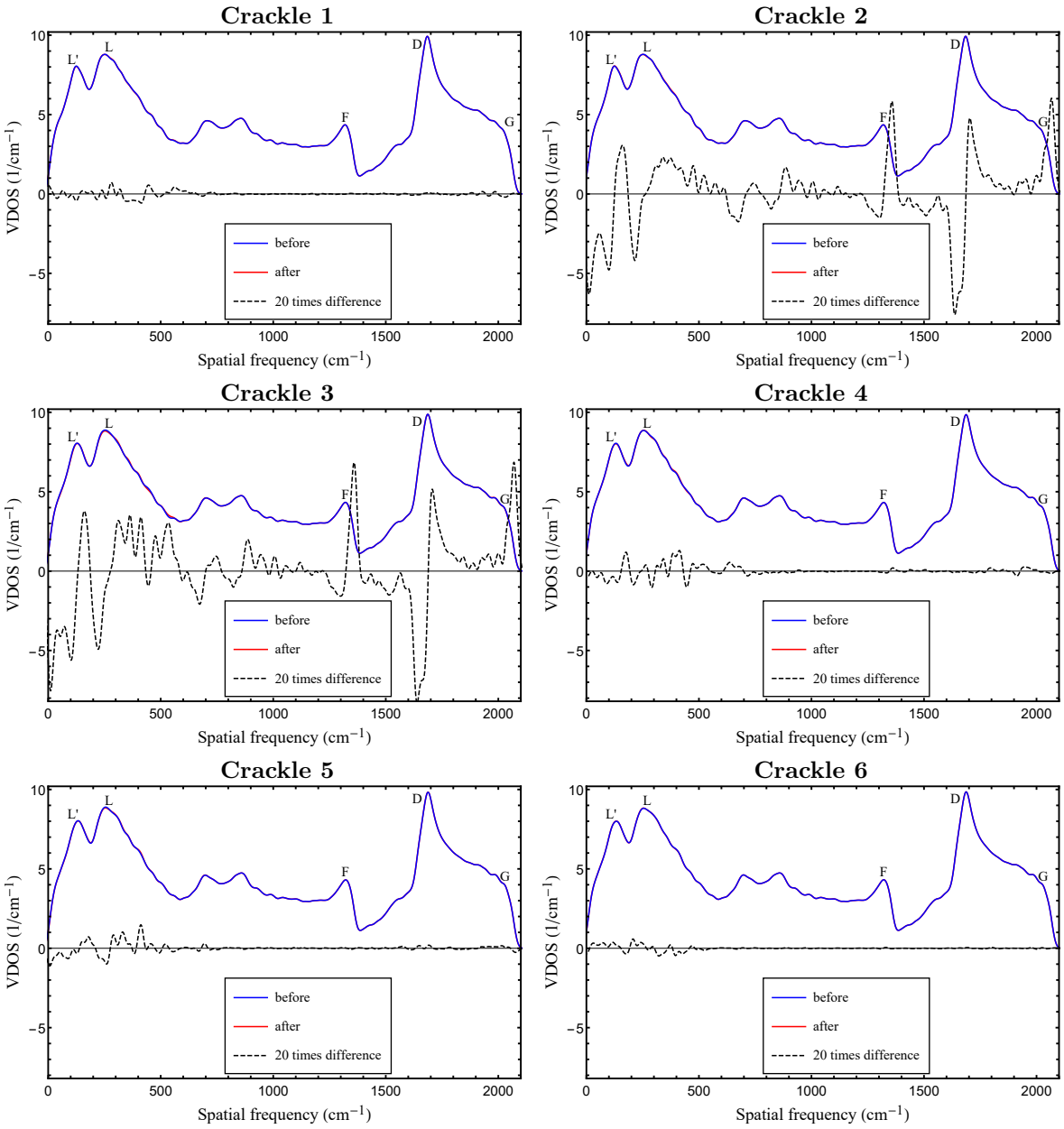


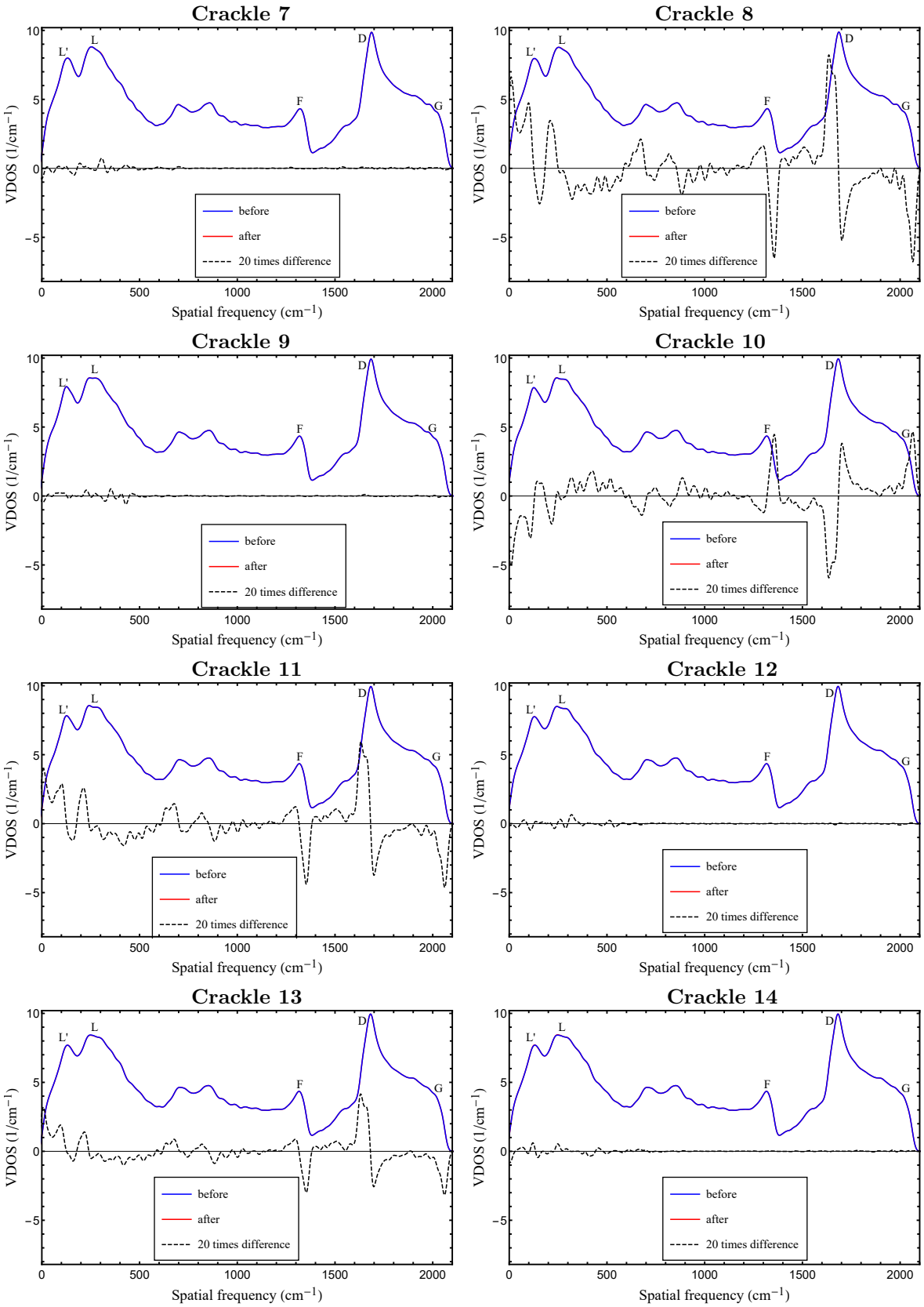
Discontinuous evolution of out-of-plane energy U^{plane} during a strain cycle.

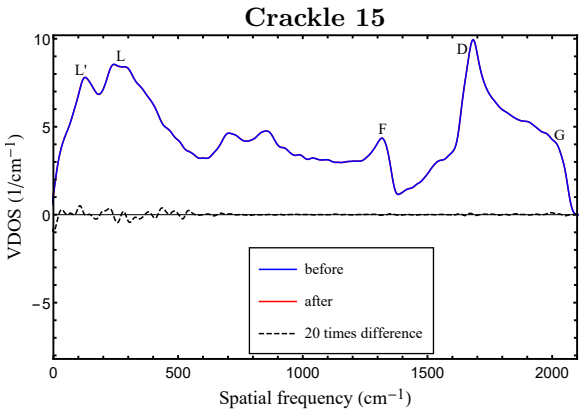
All parameters clearly show hysteretic behavior in their discontinuous evolution: The spikes in the first stretching are always distinct from those in the second. Notice how crackle 14 stands out in the structural parameters A, L and σ_z while barely noticeable in the energy evolutions.

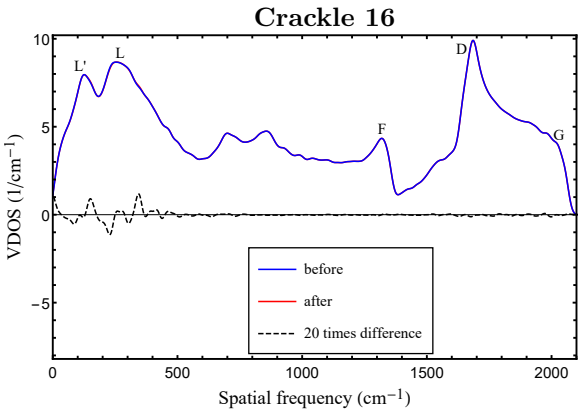
Appendix C: VDOS of Crackles

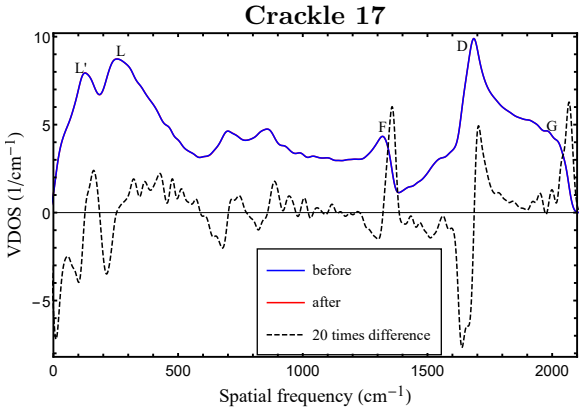
The discontinuous evolution during a straining cycle [31] can be interpreted as the crackling of graphene. Several of these 'crackles' are discussed in Chapter 3. This appendix features figures comparing the VDOS just before and after a crackle has occurred. Note that some crackles shift the peaks to the left or right, while others barely change the VDOS at all.

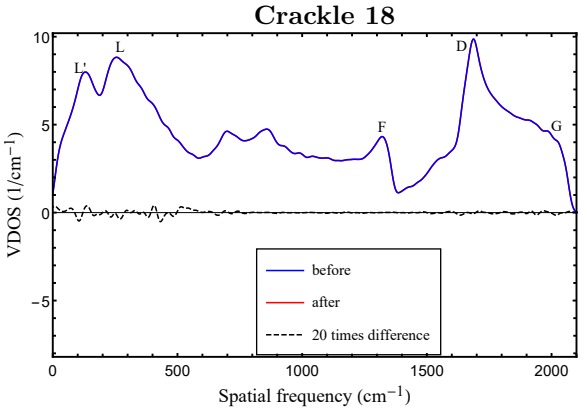


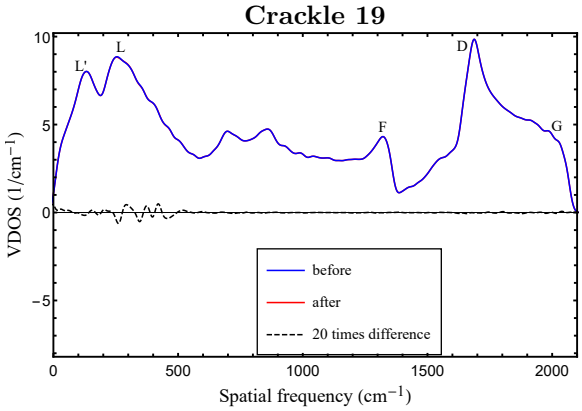

Crackle 7
Crackle 8
Crackle 9
Crackle 10
Crackle 11
Crackle 12
Crackle 13
Crackle 14
VDOS (1/cm^{-1})
Spatial frequency (cm^{-1})
L'
L
D
F
G
before
after
20 times difference

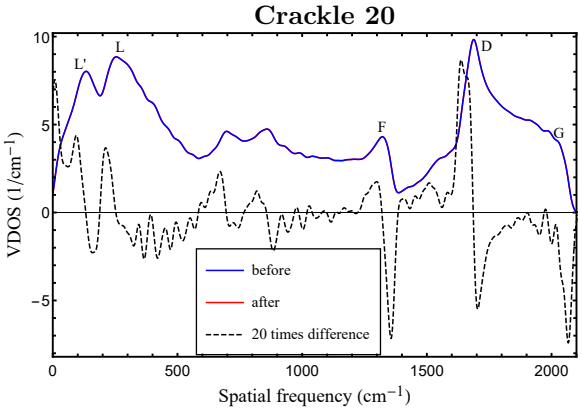

Crackle 15
VDOS (1/cm^{-1})
Spatial frequency (cm^{-1})
L'
L
D
F
G
before
after
20 times difference

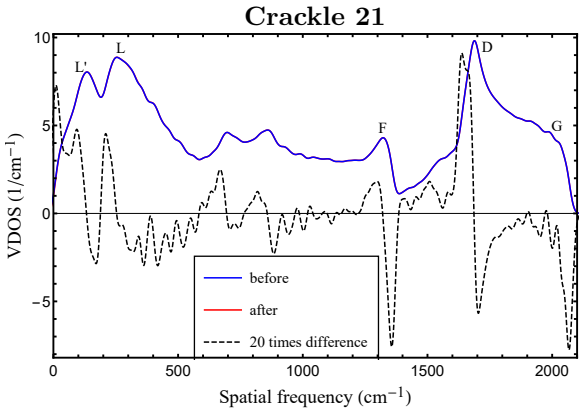

Crackle 16
VDOS (1/cm^{-1})
Spatial frequency (cm^{-1})
L'
L
D
F
G
before
after
20 times difference


Crackle 17
VDOS (1/cm^{-1})
Spatial frequency (cm^{-1})
L'
L
D
F
G
before
after
20 times difference


Crackle 18
VDOS (1/cm^{-1})
Spatial frequency (cm^{-1})
L'
L
D
F
G
before
after
20 times difference


Crackle 19
VDOS (1/cm^{-1})
Spatial frequency (cm^{-1})
L'
L
D
F
G
before
after
20 times difference


Crackle 20
VDOS (1/cm^{-1})
Spatial frequency (cm^{-1})
L'
L
D
F
G
before
after
20 times difference


Crackle 21
VDOS (1/cm^{-1})
Spatial frequency (cm^{-1})
L'
L
D
F
G
before
after
20 times difference

References

[1] Sandeep Kumar Jain, *Simulation of Graphene Mechanics*, PhD Thesis, Utrecht University (2017)

[2] C. Lee, X. Wei, J. W. Kysar, J. Hone, *Measurement of the elastic properties and intrinsic strength of monolayer graphene*, Science, **321**, 385 (2008)

[3] Y. Liu, B. Xie, Z. Zhang, Q. Zheng, Z. Xu, *Mechanical properties of graphene papers*, J. Mech. Phys. Solids, **60**, 591 (2012)

[4] K. S. Novoselov, A. K. Geim, S. V. Morozov, D. Jiang, M. I. Katsnelson, I. V. Grigorieva, S. V. Dubonos, A. A. Firsov, *Two-dimensional gas of massless Dirac fermions in graphene*, Nature, **438**, 197 (2005)

[5] X. Du, I. Skachko, A. Barker, E. Y. Andrei, *Approaching ballistic transport in suspended graphene*, Nat. Nano., **3**, 491 (2008)

[6] K. I. Bolotin, K. J. Strikes, Z. Jiang, M. Klima, G. Fudenberg, J. Hone, P. Kim, H. L. Stormer, *Ultrahigh electron mobility in suspended graphene*, Solid State Comm., **146** 351 (2008)

[7] Y. Cao, V. Fatemi, S. Fang, K. Watanabe, T. Taniguchi, E. Kaxiras, P. Jarillo-Herrero, *Unconventional superconductivity in magic angle graphene superlattices*, Nature, **556**, 43-50 (2018)

[8] A. A. Balandin, S. Ghosh, W. Bao, I. Calizo, D. Teweldebrhan, F. Miao, C. N. Lau, *Superiro thermal conductivity of single-layer grahpene*, Nano Lett., **8**, 902 (2008)

[9] S. Li, Q. Li, R. W. Carpick, P. Gumbsch, X. Z. Liu, X. Ding, J. Sun, J. Li, *The evolving quality of frictional contact with graphene*, Nature, **539**, 541 (2016)

[10] A. Fasolino, J. H. Los, M. I. Katsnelson, *Intrinsic ripples in graphene*, Nat. Mat., **6**, 858 (2007)

[11] H.-Shen Shen, Y.-Mou Xu, C.-Li Zhang, *Graphene: Why buckling occurs?*, Appl. Phys. Lett., **102**, 131905 (2013)

[12] S. Deng, V. Berry, *Wrinkled, rippled and crumpled graphene: an overview of formation mechanism, electronic properties, and applications*, Materials Today, **19**(4), 197-212 (2016)

[13] C. Soldano, A. Mahmood, E. Dujardin, *Production, properties and potential of graphene*, Carbon, **48**, 2127 (2010)

[14] C. Tan, D. Y. H. Ho, L. Wang, J.I. A. Lie, I. Yudhistira, D. A. Rhodes, T. Taniguchi, K. Watanabe, K. Shepard, P. L. McEuen, C. R. Dean, S. Adam, J. Hone, *Realization of a universal hydrodynamic semiconductor in ultra-clean dual-gated bilayer graphene*, arXiv:1908.10921 (2019)

[15] F. Banhart, J. Kotakoski, A. V. Krasheninnikov, *Structural defects in graphene*, ACS Nano, **5**, 26 (2011)

[16] M. M. Ugeda, I. Brihuega, F. Guinea, J. M. Gomez-Rodriguez, *Missing atom as a source of carbon magnetism*, Phys. Rev. Lett., **104**, 096804 (2010)

[17] E. Y. Andrei, G. Li, X. Du, *Electronic properties of graphene: A perspective from scanning tunneling microscopy and magnetotransport*, Rep. Prog. Phys., **75**, 056501 (2012)

[18] J. C. Meyer, C. Kisielowski, R. Erni, M. D. Rossell, M. F. Crommie, A. Zettl, *Direct imaging of lattice atoms and topological defects in graphene membranes*, Nano Lett., **8**, 3582 (2008)

[19] A. Hashimoto, K. Suenaga, A. Gloter, K. Urita, S. Iijima, *Direct evidence for atomic defects in graphene layers*, Nature, **430**, 870 (2004)

[20] M. P. Boneschanscher, S. K. Hamalainen, P. Liljeroth, I. Swart, *Sample corrugation affects the apparent bond length in atomic force microscopy*, ACS Nano, **8**, 3006 (2014)

[21] Y. Dedkov, E. Voloshina, *Multichannel scanning probe microscopy and spectroscopy of graphene Moire structures*, Phys. Chem. Chem. Phys., **16**, 3894 (2014)

[22] J. Cervenka, K. van de Ruit, C. F. J. Flipse, *Gaint inelastic tunneling in epitaxial graphene mediated by localized states*, Phys. Rev. B, **81**, 205403 (2010)

[23] M. L. N. Palsgaard, N. P. Andersen, M. Brandbyge, *Unravelling the role of inelastic tunneling into pristine and defected graphene*, Phys. Rev. B, **91**, 121403(R) (2015)

[24] C. Cavallari, D. Pontiroli, M. Jimanez-Ruiz, A. Ivanov, M. Mazzani, M. Gaboardi, M. Aramini, M. Brunelli, M. Ricce, S. Rols, *Hydrogen on graphene investigated by inelastic neutron scattering*, J. Phys.: Cong. Ser., **554**, 012009 (2014)

[25] V. Lee, C. Park, C. Jaye, D. A. Fischer, Q. Yu, W. Wu, Z. Liu, J. Bao, S.-Shem Pei, C. Smith, P. Lysaght, S. Banerjee, *Substrate hybridization and rippling of graphene evidenced by near-edge X-ray absorption fine structe spectroscopy*, J. Phys. Chem. Lett., **1**, 1247 (2010)

[26] L. R. De Jesus, R. V. Dennis, S. W. Depner, C. Jaye, D. A. Fischer, S. Banerjee, *Inside and outside: X-ray absorption spectroscopy mapping of chemical domains in graphene oxide*, J. Phys. Chem. Lett., **4**, 3144 (2013)

[27] K. N. Kudin, B. Ozbas, H. C. Schniepp, R. K. Prudhomme, I. A. Aksay, R. Car, *Raman spectra of graphite oxide and functionalized graphene sheets*, Nano Lett., **8**, 36 (2008)

[28] A. C. Ferrari, D. M. Basko, *Raman spectroscopy as a versatile tool for studying the properties of graphene*, Nat. Nanotech., **8**, 235 (2013)

[29] D. L. Matz, H. Sojoudi, S. Graham, J. E. Pemberton, *Signature vibrational bands for detects in CVD single-layer graphene by surface-enhanced Raman spectroscopy*, J. Phys. Chem. Lett., **6**, 964 (2015)

[30] S. K. Jain, V. Juričić, G. T. Barkema, *Probing crystallinity of graphene samples via the vibrational density of states*, J. Phys. Chem. Lett., **6**(19), 3897-3902 (2015)

[31] F. D'Ambrosio, V. Juričić, G. T. Barkema, *Discontinuous evolution of the structure of stretching polycrystalline graphene*, Phys. Rev. B, **100** , 161402 (2019)

[32] Richard Zallen, *The Physics of Amorphous Solids*, Wiley-VCH Verlag (1998)

[33] F. Wooten, K. Winer, D. Weaire, *Computer generation of structural models of amorphous Si and Ge*, Phys. Rev. Lett., **54**, 1392 (1985)

[34] E. Kim, Y. H. Lee, *Structural, electronic, and vibrational properties of liquid and amorphous silicon: Tight-binding molecular-dynamics approach*, Phys. Rev. B, **49**, 1743 (1994)

[35] S. Goedecker, M. Teter, *Tight-binding electronic-structure calculations and tight-binding molecular dynamics with localized orbital*, Phys. Rev. B, **51**, 9455 (1995)

[36] S. K. Jain, G. T. Barkema, N. Mousseau, C.-M. Fang, van Huis, M. A. Strong, *Strong long-range relaxations of structural defects in graphene simulated using a new semi-empirical potential*, J. Phys. Chem. C, **119**, 96469655 (2015)

[37] J. G. Kirkwood, *The Skeletal Modes of Vibration of Long Chain Molecules*, J. Chem. Phys., **7**, 506-509 (1939)

[38] P. N. Keating, *Effect of Invariance Requirements on the Elastic Strain Energy of Crystals with Application to the Diamond Structure*, Phys. Rev., **145**, 637 (1966)

[39] E. Bitzek, P. Koskinen, F. Gähler, M. Moseler, P. Gumbsch, *Structural Relaxation Made Simple*, Phys. Rev. Lett., **97**, 170201 (2006)

[40] R. H. Bisseling, *Parallel Scientific Computation*, Oxford University Press (2004)

[41] B. Chapman, G. Jost, R. van der Pas, *Using OpenMP*, The MIT Press (2008)

[42] K. S. Novoselov, A. K. Geim, S. V. Morozov, D. Jiang, Y. Zhang, S. V. Dubonos, I. V. Grigorieva, A. A. Firsov, *Electric field effect in atomically thin carbon films*, Science, **306**, 666 (2004)

[43] A. K. Geim, *Graphene: Status and prospects*, Science, **324**, 1530 (2009)

[44] W. H. Zachariasen, J. Am, Chem. Soc., *The atomic arrangement in glass*, **54**, 3841 (1932)

[45] G. Voronoi, *Nouvelles applications des paramtres continus à la théorie des formes quadratiques*, Journal für die Reine und Angewandte Mathematik, **133**, 97-178 (1908)

[46] G. Kresse, J. Hafner, *Ab initio molecular dynamics for liquid metals*, Phys. Rev. B, **47**, 558 (1993)

[47] G. Kresse, J. Furthmuller, *Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set*, Phys. Rev. B, **54**, 11169 (1996)

[48] J. Klimes, D. R. Bowler, A. Michaelides, *Chemical accuracy for the Van der Waals density functional*, J. Phys.: Condens. Matter, **22**, 022201 (2009)

[49] M. Dion, H. Rydber, E. Schröder, D. C. Langreth, B. I. Lundqvist, *Van der Waals density functional for general geometries*, Phys. Rev. Lett., **92**, 246401 (2004)

[50] Sandeep. K. Jain, Vladimir Juričić, Gerard. T. Barkema, *Structure of twisted and buckled bilayer graphene*, arXiv:1611.01000v1, (2016)

[51] S. K. Jain, V. Juričić, G. T. Barkema, *Probing the shape of a graphene nanobubble*, Phys. Chem. Chem. Phys., **19**(11), 7465-7470 (2017)

[52] A. J. Pool, S. K. Jain, G. T. Barkema, *structural characterization of carbon nanotubes via the vibrational density of states*, Carbon, **118**, 58-65 (2017)

[53] F. Arca, J. P. Mendez, M. Ortiz, P. Ariza, *Steric Interference in Bilayer Graphene with Point Dislocations*, Nanomaterials, **9**(7), 1012 (2019)

[54] J. H. Los, A. Fasolino, *Intrinsic long-range bond-order potential for carbon: Performance in Monte Carlo simulations of graphitization*, Phys. Rev. B, **68**, 024107 (2003)

[55] J. H. Los, L. M. Ghiringhelli, E. J. Meijer, A. Fasolino, *Improved long-range reactive bond-order potential for carbon. I. Construction*, Phys. Rev. B., **72**, 214102 (2006)

[56] G. T. Barkema, N. Mousseau, *High-quality continuous random networks*, Phys. Rev. B, **62**, 8 (2000)

[57] E. Sli, D. Mayers, *An Introduction to Numerical Analysis*, Cambridge University Press, ISBN 0-521-00794-1 (2003)

[58] M. Hejna, P. J. Steinhardt, S. Torquato, *Nearly hyperuniform network models of amorphous silicon*, Phys. Rev. B, **87**, 245204 (2013)

[59] A. Eckmann, A. Felten, A. Mishchenko, L. Britnell, R. Krupke, K. S. Novoselov, C. Casiraghi, *Probing the Nature of Defects in Graphene by Raman Spectroscopy*, Nano Letters, **12**(8), 3925-30 (2012)

[60] J. Ziv, A. Lempel, *Compression of Individual Sequences via Variable-Rate Coding*, IEEE Transactions on Information Theory, **24**(5), 530-536 (1978)

[61] T. A. Welch, *A Technique for High-Performance Data Compression*, Computer, **17**(6), 8-19 (1984)

[62] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Longman Inc., Third Edition (1997)

[63] `https://latkin.org/blog/2014/11/09/a-simple-benchmark-of-various-math-operations` consulted on 03-11-2019.

[64] W. Tian, W. Li, W. Yu, X. Liu, *A Review on Lattice Defects in Graphene: Types, Generation, Effects and Regulation*, MDPI (2017)

[65] J. Ma, D. Alfè, A. Michaelides, E. Wang, *Stone-Wales defects in graphene and other planar sp^2-bonded materials*, Phys. Rev. B, **80**, 033407 (2009)

[66] J. Alsayednoor, P. Harrison, *Evaluating the performance of microstructure generation algorithms for 2-d foam-like representative volume elements*, Mech. Mat., **98**, 44-58 (2016)