Development of a Kinetic Monte Carlo Code

Daniel Pedersen

June 25, 2013

Abstract

A framework for constructing kinetic monte carlo (KMC) simulations of diffusive events on a lattice was developed. This code was then tested by running simulations of Fe adatom diffusion on graphene and graphene-boron nitride surfaces. The results from these simulations was then used to show that the modeled diffusion adheres to the laws of brownian motion and generates results similar to recent research findings.


Contents

1 Introduction 4

2 Theory 5
   2.1 Molecular Dynamics 5
   2.2 Monte Carlo Methodology 5
   2.3 Concepts Underlying the Kinetic Monte Carlo Method 6
   2.4 The Kinetic Monte Carlo Algorithm 6
   2.5 Calculating Rate Constants 8
      2.5.1 Transition State Theory 8
      2.5.2 Harmonic Transition State Theory 9
   2.6 The Graphene-Boron Nitride Lattice 9
   2.7 States for Fe Diffusion on Graphene/Graphene-Boron Nitride Surfaces 9

3 Implementation 11
   3.1 The Lattice 11
   3.2 KMC Algorithm Implementation 12
   3.3 From KMC to Lattice-KMC 13
   3.4 Adatom Interactions 14

4 Results and Discussion 16
   4.1 One Fe-atom on pure graphene 16
      4.1.1 Temperature Dependence 16
   4.2 Diffusion of Fe adatoms on the Graphene-Boron Nitride Lattice 17
   4.3 Limitations 19
      4.3.1 Adatom Clustering 19
   4.4 Improvements 19
      4.4.1 Dynamical Rate Catalogues 19
      4.4.2 Modelling Multi-Atom Clustering 19

5 References 25
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pictorial representation of KMC algorithm, in this case, the outcome $a \rightarrow b$ is illustrated.</td>
</tr>
<tr>
<td>2</td>
<td>Basic explanation of TST, see text for details.</td>
</tr>
<tr>
<td>3</td>
<td>Structure of the graphene-boron nitride lattice at the junction. The blue dots are saddle points and the shaded circles are equipotential curves near the potential minima that demonstrate the harmonic approximation used in HTST.</td>
</tr>
<tr>
<td>4</td>
<td>Demonstration of a neighbor basis vector, $v$. In this case the blue and orange shaded circles would be connected in the final lattice, as well as all other nodes separated by $v$.</td>
</tr>
<tr>
<td>5</td>
<td>In this example, the red curve is the static potential and the blue curve is the static potential together with an imagined interatomic potential that attracts the atom to the left (higher barrier to the right), here $\Delta V_{\text{atomic}} = V(\text{saddle}) + V(\text{atom})$.</td>
</tr>
<tr>
<td>6</td>
<td>Random walks of a single Fe-atom on a surface of graphene, both plots show 5000 steps. The path starts at the origin (green dot) and ends at the red dot. The axes are in the units of the transformed index coordinates (see sec 3.1).</td>
</tr>
<tr>
<td>7</td>
<td>Trajectory of the single Fe-atom for $T=300K$ and $T=350K$.</td>
</tr>
<tr>
<td>8</td>
<td>Mean square distance of 1000 simulations of a Fe-atom on a graphene surface for different temperatures.</td>
</tr>
<tr>
<td>9</td>
<td>Plot of the estimated diffusion coefficients with the corresponding regression line. See the text for details.</td>
</tr>
<tr>
<td>10</td>
<td>Snapshot from the graphical interface, shows two clustered Fe atoms (purple) that are stuck at the junction of graphene and boron-nitride (the black, pink and green dots are carbon, boron and nitrogen, respectively). The two atom cluster can only move vertically by doing rotations into the junction according to the red arrows.</td>
</tr>
</tbody>
</table>
1 Introduction

One of the more common methods of simulating atomic-scale systems is molecular dynamics (MD). This method has an wide number of applications in a variety of fields in science, it is for example used in biochemistry to simulate how polypeptide chains fold to generate 3d-structures, in computational biology it has even been used to simulate a complete virus [1]. In physics, MD has also been very useful particularly in materials science where some of its main applications include predicting physical properties of new materials and nanostructures and to give insight into how matter (ex polymers, metals/alloys, composite materials, ceramics, etc.) behaves under specific circumstances.

Although molecular dynamics is very general and can be used to simulate many atomic systems, its not always particularly effective or practical. There are for example atomic systems such as surface diffusion where the phenomena of interest occurs on a time scale many orders of magnitude larger than the time scale of atomic motion. This means that if events occur every $10^{-9} - 10^{-6}$ s on average one will typically have to perform millions or even billions of timesteps with MD just to observe a single event, since a MD timestep typically can be on the order of $10^{-15} - 10^{-14}$ s [2]. But for a subset of these systems another method called Kinetic Monte Carlo (KMC) has been put forth, which exploits certain statistical properties of the system to be able to model only the events of interest. This means KMC can model these systems in a larger scope and also for substantially longer time when using the same amount resources as their MD counterparts.

The goal of this text is to present an implementation of such a Kinetic Monte Carlo simulation which can be applied to model diffusion events on general lattice structures. I also show that this implementation\textsuperscript{*} seems to produce feasible results, by constructing simulations of Fe adatoms diffusing on surfaces of graphene and graphene-boron nitride whose output is then compared with predictions from physical laws and closely related research findings [3].

\textsuperscript{*}The source code, code samples and precompiled binaries showing the graphical interface will be put up at [http://exjobb.danielpedersen.me/](http://exjobb.danielpedersen.me/)
2 Theory

This section gives a brief introduction to the subjects underlying the motivation and implementation of the code framework.

2.1 Molecular Dynamics

Molecular Dynamics is perhaps the most direct approach of simulating atomic systems. In MD one numerically solves Newton’s equations for all particles in the system in tiny discrete timesteps, usually in the order of femtoseconds [2]. This is done by calculating the forces acting on any one particle at a time and then move it under the influence of this constant net force during the chosen timestep. This process is then repeated to produce accurate trajectories of all particles in the system.

However, in some systems with a large number of particles \(N\), this process is time consuming since the computational work required for every time-step scales as \(N^2\) (for example if all forces in the simulation obey Newton’s third law.) Also when we are interested in the long-term behavior of a system, problems concerning the ill-conditioned (small numerical errors accumulate over time, eventually becoming significant) nature of MD become more apparent and extra work may be required to get accurate results for larger time scales. Sometimes there is of course also the problem with the studied phenomena occurring at time scales several orders of magnitude larger than what one can simulate in practice, as already mentioned in the introduction.

2.2 Monte Carlo Methodology

A common way to resolve these and other types of infeasibility issues in physical and mathematical problems is to randomly sample some relevant input domain and then do calculations for each sample. When all processed samples are aggregated we get an approximation to the real answer to the problem.

An example and common application is for numerical integration in higher dimensions [3]. Here one can escape the exponential increase in computational work with the number of dimensions by randomly sample points in a simple region \(R\) (whose length/area/volume/etc is known) containing the integration domain and calculating the value of the function at each sampled point (points outside the domain of integration are skipped or set to zero). The average of all these function values multiplied by the measure of \(R\) then approximates the true value of the integral, with uncertainty decreasing with number of sampled points \(N\) as \(N^{-1/2}\).
2.3 Concepts Underlying the Kinetic Monte Carlo Method

As pointed out above, Monte Carlo methods can be a very useful tool for problems/systems with a static character. Kinetic Monte Carlo (KMC) applies some of these ideas also to dynamical systems where the system evolves from one discrete state to another with time. If we have a system where we can clearly define particular states and know the intrinsic dynamics of the system, that is how the system changes state, we can in some cases employ KMC which in many cases is advantageous [5].

One fairly general case of such systems is infrequent-event systems, for which the time of overall state-changes is large compared to the motion of the constituent particles of the state. The key property of these systems that make them suitable for KMC is that the previous state of the system does not have a significant impact on what the next state will be, since it has enough time to regain thermal equilibrium between each state transition [5]. To put it in other words, the system is not significantly dependent on its own history, it "loses its memory" of how it got to the current state shortly after each transition.

For systems with these properties, we can meaningfully introduce a rate constant $c_{a \rightarrow b}$ that captures, per unit time, the probability of the system transitioning from the current state $a$ to some new state $b$. Now assuming we can evaluate all the rate constants for all possible transitions we can setup a stochastic, Monte-Carlo type procedure to simulate the "state trajectory" of the system starting from any initial state.

2.4 The Kinetic Monte Carlo Algorithm

The KMC algorithm takes as input all the rate constants $c_{a \rightarrow b}$ and outputs which state transition should be performed. This is done in a manner such that the probability of transition $a \rightarrow b$ happening is simply the relative size of it’s rate constant to the sum of all rate constants, $P(a \rightarrow b) = c_{a \rightarrow b}/C$, where $C$ is the sum of all rate constants. To understand how this is accomplished, one can visualize all the rates as line segments, with the length of the segment corresponding to the magnitude of the rate constant. Then all these line segments are combined and with the use of a pseudo random number generator a random spot on this combined line is chosen. The line segment under the selected spot has some corresponding rate constant $c_{a \rightarrow b}$ and so the transition performed is $a \rightarrow b$. This process is also depicted in figure[1]
Formally, we can say that we form the cumulative sums

\[ K_n = \sum_{i=0}^{n} k_i \]

where \( k_i \) is the \( i \)th rate constant (ordering doesn’t matter). Next, we draw a uniform random number \( r \) from the interval \((0, 1]\) and then return the integer \( j \) which makes Eq \([1]\) true.

\[ K_{j-1} < rK_N \leq K_j \tag{1} \]

Here, \( N \) is the total number of rate constants and thus \( K_N \) is the sum total of all rate constants.

At each step of the algorithm, the simulation time \( t \) is incremented by \( \Delta t \) which is given by

\[ \Delta t = -\frac{\ln(r')}{K_N} \]

where \( r' \) is once again a uniform random number drawn from \((0, 1]\). This effectively creates a exponentially distributed random number from the uniformly distributed \( r' \). Since the mean value of the function \( y(x) = -\ln x \) on
(0, 1] is unity, the average time interval between jumps is $\frac{1}{K_N}$. Thus one could use $\Delta t = \frac{1}{K_N}$ to get the same long-time average time, the random number is only used to capture the stochastic, Poisson-type, nature of the transitions.

### 2.5 Calculating Rate Constants

If one has knowledge of all possible paths the system can take from one state to another state, then there are in principle straight-forward methods of calculating rate constants for those transitions.

#### 2.5.1 Transition State Theory

Transition State Theory can in some cases give exact or at least gives good estimates of the rate constants by considering the equilibrium flux through an imaginary surface that separates the two states [6]. For example, consider the case in fig 2, we can define the states $L$ and $R$, the system is in state $L$ when some particle is to the left of the potential hill and in state $R$ when it is to the right. To put it in the language of TST, we can imagine a dividing surface (line) $S : x = 0$ that separates the states. This surface is called the transition state and the rate constant $c_{L \rightarrow R}$ is given by calculating the probability of the system being in the transition state multiplied by the flux out of the transition state ($\nu_\perp$ in fig 2).

![Figure 2: Basic explanation of TST, see text for details.](image)
2.5.2 Harmonic Transition State Theory

One can simplify these types of calculations by doing a second order taylor expansion (harmonic approximation) of the potential energy surface close to the local minima and saddle points (example in figure 3). If we also require that the trajectory of the transition pass close to one of the saddle points then it can be shown [5] that the rate constant is fairly well approximated by equation 2. This simplification to TST is called Harmonic Transition State Theory (HTST).

\[ e^{HTST}_{a \rightarrow b} = P \exp \left( -\frac{E_{static}^{a \rightarrow b}}{k_B T} \right) \]  

Here \( E_{static}^{a \rightarrow b} \) is the diffusion barrier seperating state \( a \) and state \( b \), also called the static barrier height. Aside from the Boltzmann constant \( (k_B) \) and the temperature of the ensemble \( T \), we have a preexponential factor \( P \), given by equation 3.

\[ P = \frac{\prod_{i}^{3N} \nu_{\min}^{i}}{\prod_{i}^{3N-1} \nu_{\text{saddle}}^{i}} \]  

In this expression \( \nu_{\min}^{i} \) and \( \nu_{\text{saddle}}^{i} \) is the normal mode frequencies at the minimum and saddle point, respectively. This preexponential factor is most often on the order of \( 10^{12} \) to \( 10^{13} \) Hz and can be in many cases taken to be constant [5] to save computational work.

2.6 The Graphene-Boron Nitride Lattice

Graphene is a single layer of carbon atoms arranged in a pattern of hexagons, with the atoms placed at the vertices of these hexagons. The compound boron nitride (BN) can also form a similar structure but in an alternating pattern where the boron atoms only bond to nitrogen atoms and vice versa. One surface that is modeled in this report is a graphene surface connected to a boron nitride surface, forming a junction of bonded carbon-nitrogen pairs in the middle (see fig 3).

2.7 States for Fe Diffusion on Graphene/Graphene-Boron Nitride Surfaces

Here the notion of a state for Fe adatom diffusion on a hexagonal lattice is formalized. The state is taken to be the result of relaxing all atoms to their position of locally minimized energy, using for example a steepest decent
algorithm. Thus at any particular time all Fe atoms can be seen to be in some definite state, located in the middle of a hexagonal cell created by the atoms at the vertices of the hexagonal lattice. The transition states are the six planes that intersect the saddle points between the lattice atoms and is oriented along the line connecting the lattice atoms (for example the black lines in fig. 3, but as 3d surfaces extending into/out of the page).
3 Implementation

The code is purely written in C++, it was designed to be able to be used as a framework for constructing general (lattice-) KMC simulations and is not inherently restricted to the Graphene and Graphene-Boron nitride simulations considered in this text.

3.1 The Lattice

The first part was to write the internal memory representation of the lattice. This was done by writing a LatticeNode class that contain all the information needed for each node. This information consists of the node’s index coordinates, integers which determine the location of the node object internally in a N-dimensional matrix, as well as pointers to all adjacent nodes. The Lattice class then handles the creation and linking of the LatticeNode objects.

In the Lattice class, the LatticeNode objects are held in a data structure that I call a Static N-dimensional Array or the SNArray class, which is basically a N-dimensional matrix that makes it more practical to address the nodes compared to using a linear array. It also allows the code to be able to scale better and represent lattices in any number of dimensions.

The Lattice class also contains basis vectors for the lattice being modeled, so that we are able to switch from index coordinates to the real coordinates. Lastly, the Lattice contains what I call the neighbor basis. The neighbor basis is a collection of vectors of integers that describe in units of the basis vectors, where each neighbor is located with respect to a specific node. For example the neighbor basis vector $(2, -1)$ informs us that the node $N$ at $\mathbf{v} = 2\hat{u}_1 - \hat{u}_2$ relative to the current node $M$ is to be considered a neighbor to $N$, where in this case $\hat{u}_1$ and $\hat{u}_2$ are the lattice basis vectors, see fig 4. This generalized notion of neighbors is what eventually allows the code to represent and store just about any lattice.

So to summarize, to get a lattice in the code one creates an instance of the Lattice class which initializes some number of nodes and connects them (by making them point at their respective locations in memory) according to the provided neighbor basis. The basis vectors for the lattice itself are just stored and is only used when doing calculations involving distances or to transform from the index coordinates.

In the Lattice class I also implemented a convenient function for iterating over all nodes, apply(). This function takes a function pointer as an argument and then applies this function $f()$ to each node in the lattice. This function $f()$ therefore takes a pointer to a LatticeNode (or other node ob-
Figure 4: Demonstration of a neighbor basis vector, \( \mathbf{v} \). In this case the blue and orange shaded circles would be connected in the final lattice, as well as all other nodes separated by \( \mathbf{v} \).

ject) as its argument and then performs some operation on that node. This \texttt{apply()} function was considered useful since it enables us to very flexibly initialize all lattice nodes, as well as do other bulk operations in a compact fashion. If we for example want a 3d diamond lattice which can be seen as a specialized case of a face-centered cubic (FCC) lattice, we could easily create the FCC directly by just creating a \texttt{Lattice} object, then by using \texttt{apply()} we can perform the adjustments that make the FCC into the diamond lattice.

Also to note is that all these lattice classes are templates. This means that for example \texttt{Lattice} can have any class as its lattice nodes. In C++ code this is written as \texttt{Lattice<T>} which denotes that the \texttt{Lattice} template should use the class \( T \) for its constituent nodes. This notion of templates was also used to generalize the space that the objects are embedded in. Since the lattice considered in this report is best represented in cartesian coordinates the class \texttt{Cartesian2D} are used as a template argument to the lattice and many of the other lattice-related classes. So for example \texttt{Lattice<Cartesian2D,LatticeNode>} describes a lattice in planar cartesian coordinates with \texttt{LatticeNode} serving as the lattice nodes. Although this might make the code itself harder to follow for people unfamiliar with polymorphism in programming, it pays off by making the code highly reusable and modular — and by allowing for a more logical code structure.

3.2 KMC Algorithm Implementation

The actual KMC implementation was fairly straight-forward. Everything is managed through the \texttt{kmc} class. This class holds a vector containing the current rate constants and a floating-point value representing the current time.
The most important function in this class is the `nextState()` function, which uses the current rate constants to carry out a step in the KMC algorithm as described in section 2.4. Since the rates are allowed dynamically change between state transitions the `KMC` class also has functions for managing the vector of rates.

The rate vector is only conceptually a vector, internally, the vector of rates is not stored in an array/vector but in a binary search tree (BST) to make the process of finding the index corresponding to the randomly chosen transition \( j \) in eq 1 significantly more efficient when the simulation has a large number of rate contents \( n \). Compared to the naive linear search algorithm, which has an average case performance of \( \mathcal{O}(n) \), the binary search has a significantly better average case that instead scale as \( \mathcal{O}(\log n) \), while using no significant amount of extra memory.

### 3.3 From KMC to Lattice-KMC

To be able to construct the particular simulation we had in mind, a subclass to the `LatticeNode` class was created called `LKMCNode`, which also stores a rate constant for each of it’s neighbors. A `LKMC` class was also needed, whose primary duties consist of the construction of the intended lattice and the coordination between instantiated objects, for example between the `Lattice` and `KMC` objects. This means that it is the `LKMC` class that adds the relevant rate constants from the lattice to the `KMC` object and then keeps track of the transition indices \( j \) in eq 1 so that it can execute the correct movement of the adatom on the lattice after `KMC.nextState()` has been called.

In actuality however, the `Lattice` object in `LKMC` does not correspond to the real lattice (i.e. graphene/ graphene-boron nitride) at all, but is instead the lattice of allowable states of the adatoms. Since these allowed states are exactly the middle points of the hexagonal cells of the real lattice (see sec 2.7), they create a triangular lattice of states directly related to the underlying hexagonal lattice of atoms.

The adatoms themselves are held in the `Adatom` class, in which the only thing stored is the position of the adatom in the index coordinates.

So in practice, when a `LKMC` object is created the 2d-triangular lattice is setup with the aid of the `Lattice` class, then using the `Lattice.apply()` function the rate constants corresponding to every static barrier height is precalculated for all neighbors of all nodes in the lattice. After this initialization of the `LKMC` object, one can for example put adatoms into the simulation by calling `LKMC.newAdatom()` and then the function `LKMC.step()` can be called, which executes the movement of a single adatom on the lattice according to the KMC algorithm. Raw output such as the current position
of the adatoms can then be read out from the LKMC-object using for example LKMC.outputStatus(), which can later be used to analyze the simulation quantitatively.

### 3.4 Adatom Interactions

Modelling of adatom interactions was realized by considering the effect of the interatomic potential on the static barrier heights. Thus the interactions was modeled by temporarily and locally changing the barrier heights (which are directly related to the rate constants in HTST) for a given atom. So calling the atom whose barrier heights we wish to modify \( A \), we change each of its barriers by calculating the interpotential generated from all other atoms at the saddle point and then subtract the same interpotential evaluated at the position of atom \( A \) (fig 5). If we call this modification \( \Delta V_{i\rightarrow j}^{atomic} \), we get the new barrier heights

\[
E_{i\rightarrow j} = E_{i\rightarrow j}^{static} + \Delta V_{i\rightarrow j}^{atomic}
\]

Figure 5: In this example, the red curve is the static potential and the blue curve is the static potential together with an imagined interatomic potential that attracts the atom to the left (higher barrier to the right), here \( \Delta V_{i\rightarrow j}^{atomic} = V(saddle) + V(atom) \).
When this modified barrier height is put into the rate equation given by HTST (eq 2), we see that we can factor out this new term

\[ c'_{i \rightarrow j} = P \exp \left( -\frac{E_{i \rightarrow j}}{k_B T} \right) \]

\[ = P \exp \left( -\frac{E_{i \rightarrow j}^{\text{static}} + \Delta V_{i \rightarrow j}^{\text{atomic}}}{k_B T} \right) \]

\[ = c_{i \rightarrow j} \exp \left( -\frac{\Delta V_{i \rightarrow j}^{\text{atomic}}}{k_B T} \right). \]

This means that to model the adatom interactions we can use the original static rates and then during the simulation calculate the modifications to each of the barrier heights, \( \Delta V_{i \rightarrow j}^{\text{atomic}} \), then we use this as prescribed in the formula above to get the new, temporary rates for each atom that serve to model both the interatomic interactions as well as the underlying lattice dynamics.
4 Results and Discussion

4.1 One Fe-atom on pure graphene

The first simulation was that of a single Fe-atom on a graphene surface, with a diffusion barrier set to 0.3 eV \[3\]. This simple system was important for debugging the core framework and is the most easily recreated\[†\]. Since this system has a high degree of spatial symmetry, the pathway of the Fe-atom is expected to be a 2 dimensional random-walk on the hexagonal surface. In other words, the atom has no directional preference so it just moves about randomly. Two runs that serve as examples are shown in figure 6.

4.1.1 Temperature Dependence

The effect of changing the temperature of the simulation on the motion of the single Fe-atom was also tested. This was also done on the graphene surface, but the simulation was set to run for a predetermined amount of time (simulation time, not real time) instead of a fixed number of algorithmic steps. As can be seen in figure 7, the rate of diffusion increases with temperature, as expected.

The temperature dependence of diffusion was examined further since it is easily tested against predictions from well-established physical laws and can therefore serve as a simple check of the correctness of the simulation. To get reasonable data, a total of 1000 simulations of at least one microsecond each was done for evenly spaced temperatures in the interval 250-350K. For each simulation, the square displacement from the origin was calculated at each time step. This data was then sampled at uniform timesteps and averaged, resulting in 1-10 thousand datapoints for each temperature. The results, shown in figure 8, indicates that the mean square displacement depends linearly on time, and that the temperature increases the slope of this line. Einstein’s theory of Brownian motion (this system exhibits brownian-type motion but on a discrete triangular lattice) predicts that the displacement from the origin of a particle undergoing Brownian motion is proportional to the square root of time, so the square displacement is expected to scale linearly with time. Einstein’s theory also tells us that the slope of this line is directly related to the diffusion coefficient \(D\). For diffusion in two dimensions equation \[4\] gives the correct form of this relationship. Furthermore, the expression in equation \[5\] predicts the diffusion coefficient when the temperature \(T\) and activation energy \(E_A\) (potential barrier) are known. The diffusion

\[†\]See http://exjobb.danielpedersen.me/
coefficient in eq [5] converges as the temperature goes to infinity and $D_0$ is that limit, the maximum diffusion coefficient.

$$\langle r(t)^2 \rangle = 4Dt \quad (4)$$

$$D = D_0 \exp \left( -\frac{E_A}{k_BT} \right) \quad (5)$$

Using simple linear regression, slopes were assigned to each of the data sets plotted in figure 8. As can be seen from eq 4 these slopes give an estimate of the diffusion coefficients (scaled by a factor of 4). The estimated diffusion coefficients was used together with eq 5 to find $D_0$ and $E_A$ by first linearizing eq 5, yielding eq 6.

$$\ln(D) = -\frac{E_A}{k_BT} + \ln(D_0) \quad (6)$$

Now if we take $\frac{1}{k_BT}$ to be the dependent variable in eq 6, then it is obvious that we have a linear relationship between $\ln(D)$ and $\frac{1}{k_BT}$. Thus $\ln(D_0)$ and $E_A$ can be estimated by again using regression analysis but on these derived quantities. These results are summarized in figure 9, where also high temperature cases for 1000K and 2000K are included (50 simulations of 1 ns each), which shows that the trend also seems to hold as $T \to \infty$.

Since the diffusion barrier, $E_A$, is known to be 0.3 eV we see that the calculated diffusion barrier agrees with this value with a margin of error of <2%. Furthermore, if we disregard the 1000K and 2000K data points which are of significantly lower quality (fewer simulations were done), the margin of error drops to <.02%. In light of this we can draw the conclusion that the Fe atoms in this model diffuses as expected by Einstein’s theory of Brownian motion.

### 4.2 Diffusion of Fe adatoms on the Graphene-Boron Nitride Lattice

The other model that was constructed with this code was the diffusion of Fe-atoms on the graphene-boron nitride (BN) lattice described in section 2.6.

To simulate multiple adatoms interacting we require a model interpotential, as described in section 3.4. The potential from a single atom $A$ needs to have its minimum at a distance that is greater than the distance $r_{min}$ from $A$ to the surrounding saddle points if the simulation is not to break. Otherwise there is a high probability for an adjacent atom to transition to the hexagonal site occupied by $A$, after this transition any $r^{-n}$ dependence
in the potential would result in division by zero. This also implies that the potential must approach infinity rapidly as $r \to 0$. For a potential on the form

$$V(r) = c(r^{-n} - br^{-m})$$

it was determined through trial-and-error that $n = 4$ would suffice for this to be true. For the other term $m = 1$ was chosen since any larger $m$ either diminishes long-range attraction or requires $c$ to be huge. By solving $V'(r) = 0$ for the minimum we get the condition

$$b < \frac{4}{r_{\text{min}}^3},$$

in line with the reasoning above. For $r$ measured in the lattice unit vectors this means $r_{\text{min}} = \frac{1}{2}$ and so $4 \leq b < 32$, if we also require bonding to occur when the atoms are adjacent ($r = 1$). Within this range $b \approx 7$ was chosen ($r_{\text{min}} = 0.84$), although slightly lower values in the range would deliver similar results. $c$ can then be used to set the overall strength of the attractive interaction, here $c = 3$ was used.

The diffusion barriers close to the junction, where they do not seem to follow any overall pattern, was obtained from Sanyal et al. [3]. Here "close to the junction" is a zone extending 3 lattice sites in both directions. Outside of this zone fixed diffusion barriers of 0.3 eV and 0.2 eV was used for the graphene and BN halves, respectively.

Using this setup 2-4 Fe adatoms was placed at different starting positions and the movements was then directly observed through a minimal graphical interface shown in figure [10]. By looking at many runs the general characteristics of the motion of the atoms could be understood. Of course due to the model potential it is no surprise that the atoms come together and cluster whenever they are placed sufficiently close in the starting position. When we have two atoms they bond and then get stuck in the first column to the right of the junction, performing rotational motions into the junction and vertical translations along the junction. This is similar to the findings of Sanyal et al. [3], but there they had a periodic pattern of graphene and BN. In their molecular dynamics simulation the atoms cluster but get stuck at a different site, namely at the junction of graphene and boron, a site not included in my unperiodic model. They also had 4 adatoms on the surface that formed a tetrahedral structure, which this simulation is unable to recreate since the adatoms are stuck on the plane. This is discussed further in the following section.
4.3 Limitations

4.3.1 Adatom Clustering

As two atoms come close, the interatomic potential becomes larger and larger (negatively) which skews the transition rates substantially. Assuming a properly chosen potential, the only non-zero rates (the rates rarely become identically zero, but for a bonded pair the transitions for splitting the bond is negligible) are the transitions to the unoccupied hexagonal sites that are adjacent to both atoms. The combined effect of this is that the two-atom system performs fast rotational and shuffling motions. This is fine as long as there is only two adatoms, but when there is three or more adatoms they often get stuck in various patterns with none or a few of the atoms oscillating between a few nearby sites. In reality the atoms can, as previously mentioned, bond in 3 dimensions and form stable 3 dimensional structures. This code might be with some extention be able to model such behavior as well, the details are discussed in section 4.4.2.

4.4 Improvements

4.4.1 Dynamical Rate Catalogues

In the current code, all the rates are precalculated and stored in the `LKMNode` objects of the lattice. This works very well for the models constructed here, but in a more general case the number of total rate constants could be huge (if not infinite) and the memory footprint and initialization time would be proportionally huge. To solve this one could instead calculate the rate constants dynamically "on-the-go", and store only these rate constants, since there in many cases would be probable that they would be reused. So only the rate constants that are actually used are calculated. This could be fairly easily implemented by using for example a hashtable as a cache/lookup table, the rates that can’t be found are then evaluated and added to the hashtable. So instead of storing the rates directly in the lattice nodes, some key is stored that gives back the correct rate constant when it is looked up in the hashtable.

4.4.2 Modelling Multi-Atom Clustering

One method that might be able to model the advanced clustering behavior seen with more than 2 or 3 adatoms is to allow adjacent adatoms to form new objects on the lattice, which their own rate constants. Though it is uncertain how accurate one could make such a model. Again from the findings of Sanyal et al. \[^3\], where their MD simulation produced a tetrahedral Fe cluster, it
seems as if it could be possible, since the Fe atom at the top of the tetrahedron diffuses in a similar way as a single adatom. What is meant by this is that this top Fe atom spends most of the time centered over the middle of the hexagonal cells and then "diffuses" to adjacent hexagonal cells due to the motion of the atoms in the base. Thus if the transition rates for this process could be established one could take the tetrahedron to be a single object on the lattice and then use the already existing code. For general atom clusters the center of mass could be taken to be represented by the lattice object.
Figure 6: Random walks of a single Fe-atom on a surface of graphene, both plots show 5000 steps. The path starts at the origin (green dot) and ends at the red dot. The axes are in the units of the transformed index coordinates (see sec 3.1).
Figure 7: Trajectory of the single Fe-atom for $T=300\text{K}$ and $T=350\text{K}$.

Figure 8: Mean square distance of 1000 simulations of a Fe-atom on a graphene surface for different temperatures.
Figure 9: Plot of the estimated diffusion coefficients with the corresponding regression line. See the text for details.
Figure 10: Snapshot from the graphical interface, shows two clustered Fe atoms (purple) that are stuck at the junction of graphene and boron-nitride (the black, pink and green dots are carbon, boron and nitrogen, respectively). The two atom cluster can only move vertically by doing rotations into the junction according to the red arrows.
5 References


