Simulating the formation of Pt nanostructures utilizing molecular dynamic calculations

By

Leon Adolf Leopold Wessels

B.Sc. Hons.

Submitted in fulfilment of the requirement in respect of the master's degree qualification

MAGISTER SCIENTIAE

in the
Department of Physics
in the
Faculty of Natural and Agricultural Sciences
at the
University of the Free State
Republic of South Africa

Supervisor: Prof. J.J. Terblans
Co-supervisor: Prof H.C. Swart
Date: 2014

For my family for always being there

Acknowledgements

I would like to thank the National Research Foundation (NRF), the Cluster programme of the University of the Free State and the Inkaba yeAfrica programme for their financial assistance.

I would like to thank the following people:

- Prof Terblans for his patience and encouragement.
- Prof Swart for his encouragement.
- Abraham van der Linde for writing a library for the visualization of the simulations.
- Cornelia van der Walt for many interesting discussions.

I also want to thank the Heavenly Father for His unconditional love.

Keywords

Platinum

Nanoparticles

Simulation

Molecular dynamics

Sutton-Chen potential

Energy barrier

Physical vapour deposition

Island growth

Abstract

Platinum (Pt) is an important catalyst for applications such as catalytic converters. In this thesis the formation of platinum nanoparticles was investigated by means of simulations. For the first part of the thesis a molecular dynamics simulation using the Sutton-Chen potential was implemented. This program was used for the simulations. Low energy structures were found. It was found that the number of nearest neighbours are maximised in the low energy structures. The energy barriers that have to be overcome as atoms move around the structures were also calculated. A model is proposed for the prediction of energy barriers. The model is useful for understanding the factors that influence the energy barriers and thus the mobility of atoms. The model will also be useful for Monte Carlo simulations. Simulations were done modelling physical vapour deposition onto the Pt(111) surface and a graphite surface represented by the Steele potential. It was found that higher temperatures and lower evaporation rates lead to lower energy structures. The smaller interaction between the graphite surface and the Pt leads to structures that have more layers. The parameters of the Steele potential that determine nearest neighbour distance and interaction strength between Pt and the substrate were adjusted to simulate other materials. It was found that a mismatch between the nearest neighbour distance of the substrate and Pt causes an increase in the mobility of the Pt atoms on the surface. The results of the simulations will enable the choice of suitable substrate and experimental parameters for the growth of Pt nanoparticles of desired shapes.

Abstrak

Platinum (Pt) is ‘n belangrike katalis vir toepassings soos katalietiese omsitters. In hierdie tesis word die vorming van platinum nanodeeltjies ondersoek deur middel van simulasies. In die eerste deel van die tesis word ‘n molekulêre dinamiese simulasie wat van die Sutton-Chen potensiaal gebruik maak geïmplementeer. Die program is gebruik vir simulasies. Lae energie strukture was gevind. Dit is gevind dat die hoeveelheid naaste bure in lae energie strukture gemaksimeer is. Die energieversperrings wat oorkom moet word vir atome om langs die strukture te beweeg is ook bereken. ‘n Model waarmee hierdie energieversperrings voorspel kan word, word voorgestel. Die model lig die faktore uit wat die energieversperrings en dus die atoom beweeglikheid beïnvloed. Die model sal ook bruikbaar wees Monte Carlo simulasies. Simulasies was gemaak vir die opdamping van Pt op die Pt(111) oppervlak en op ‘n grafiet oppervlak wat met die Steele potensiaal gesimuleer is. Daar is gevind dat hoër temperature en laer opdampings tempo lei tot laer energie strukture. Die kleiner interaksie tussen die grafiet oppervlak en die Pt lei tot strukture wat meer lae bevat. Die veranderlikes van die Steele potensiaal wat die naastebuurafstand en die sterkte van interaksie tussen Pt en die substraat bepaal was verstel om ander materiale te simuleer. Dit was gevind dat ‘n wanaanpassing tussen die naastebuurafstand van die substraat en Pt ‘n verhoogde beweeglikheid van die Pt atome op die oppervlak veroorsaak. Die resultate van die simulasies gee aanduiding tot die kies van geskikte substraat en eksperimentele opstelling vir die groei van Pt nanodeeltjies van ‘n verlangde vorm.

Table of Contents

List of figures

Chapter 1: Introduction

Platinum (Pt) can be used as a catalyst in catalytic converters in automobiles. The catalytic converter changes harmful compounds, such as carbon monoxide (CO), unburnt hydrocarbons (HC), and nitrous oxides (NO_x) in the exhaust into less harmful compounds by oxidizing them [**1**]. Better catalysts are required as the requirements for the allowable emissions become more stringent. In addition it is desirable to decrease the price of these catalysts. By using Pt nanoparticles both these requirements can be met. The price will decrease because less Pt is required since a nanoparticle contains in the order of a few hundred atoms. The efficiency when using nanoparticles will increase because more surface area is exposed to the harmful gasses [**2**].

In this thesis the formation of Pt nanoparticles will be investigated. The investigation was performed by using molecular dynamic simulations. One advantage of using simulations is that more systems can be investigated. More systems can be investigated with simulations because more computer time can be obtained with relative ease. Each computer can then do a different simulation. Performing a large number of experiments with a substance as expensive as Pt would be impractical. Additionally simulations can be done for systems that would be very difficult to do experimentally. Another advantage with simulations is that more information can be gained about the movement of the atoms. More information is gained because all the information from each step is known precisely.

It is desirable to simulate a system as accurately as possible. Unfortunately there is a trade-off between the computational cost and accuracy in a simulation. This trade-off comes from factors like:

- System size (number of atoms in the system): using a larger system will resemble the real world more but computations will be slower,
- Assumptions: using fewer assumptions will be slower than a simulation with more assumptions, but closer to reality and
- Variables: the size of some variables, like the time step, influences accuracy and speed

to name a few. It is important that the trade-offs are chosen in a way such that the results will be meaningful and that the simulation is feasible to do.

On the atomic scale atoms are described by Schrodinger's wave equation (SWE). Simulations with the SWE will yield the best results. The problem with this is that such simulations take a very long time and only small systems and timescales can be simulated. Some simplifying assumptions (such as assuming that the problem of how the electrons and the nuclei move can be solved independently of each other) can be made to enable the simulation of larger systems over longer timescales [**3**] [**4**]. The accuracy of these simulations is still acceptable because molecular dynamics does not focus on the predictions concerning the exact behaviour of particles but rather properties of a whole system.

In molecular dynamics, atoms are treated as classical particles moving in a potential field. The potential field determines the forces on the atoms and thus how the atoms will move. Typically in a molecular dynamics simulation the movements and energies of tens to thousands of atoms can be simulated for a few nanoseconds.

Another simulation method is Monte Carlo simulations. With Monte Carlo simulations there are more assumptions but the simulations can be done for longer times and larger systems [**5**] [**6**]. Monte Carlo simulations work by having atoms arranged in a structure. An atom then performs a random jump towards a new position in the structure. The probability of a specific jump to occur is determined by the energy barrier that must be overcome in that jump. Thus it is more probable for a jump to occur if the energy barrier towards the new location is smaller. The barriers for the Monte Carlo simulations must be found from another source like molecular dynamics or experiments.

1.1 Purpose of study

The purpose of this study is to investigate the factors that determine the size and shape of Pt nanoparticles. The investigation focused on what happens during the physical vapour deposition of Pt clusters.

1.2 Layout of thesis

Chapter two describes molecular dynamics. It discusses the details of molecular dynamics and the basic algorithmic concepts.

Chapter three investigates small structures of less than ten atoms. The lowest energy structures were found. The energy barriers that atoms have to overcome as the atoms move around the structures were also calculated. A model is proposed in order to be able to predict the energy of the barriers.

Chapter four investigates the deposition of Pt onto a Pt surface. This is to show the effects of temperature and deposition rate on the structures that form.

Chapter five investigates the deposition of Pt on a graphite surface. The deposition rate, attractive force and size mismatch between the substrate and Pt is investigated.

Chapter six gives a brief discussion of the results.

Chapter 2: Molecular dynamics simulation details

In a molecular dynamics simulation atoms are treated as classical particles in a potential field. In order to motivate this from quantum mechanics, the following assumptions are made:

- It is assumed that the electrons move much faster than the nuclei (the Born-Oppenheimer assumption). As a result of this assumption the movement of the nuclei and electrons can be regarded as a separate problem [**7**]. This assumption is justified because the electrons are so much lighter than the nucleus.
- It is assumed that all the interactions of the electrons can be captured in a potential. This potential includes all the interactions between the electrons and the nuclei.
- It is assumed that the atoms can be treated as classical particles that move in the potential field.

With these assumptions the simulation reduces to a classical n-body problem in a force field. It is not possible to solve this n-body problem analytically for a system with more than three particles. For this reason the trajectories of all the particles have to be calculated iteratively.

2.1 Molecular dynamics calculation

The main steps performed during the molecular dynamics simulation are shown in *Figure 2.1*. The system state is the positions and velocities of all the atoms. For the initial system state of a simulation, the atoms are placed in the positions that are described by the starting state of the experiment which is to be performed. The initial velocity of each atom is calculated from the

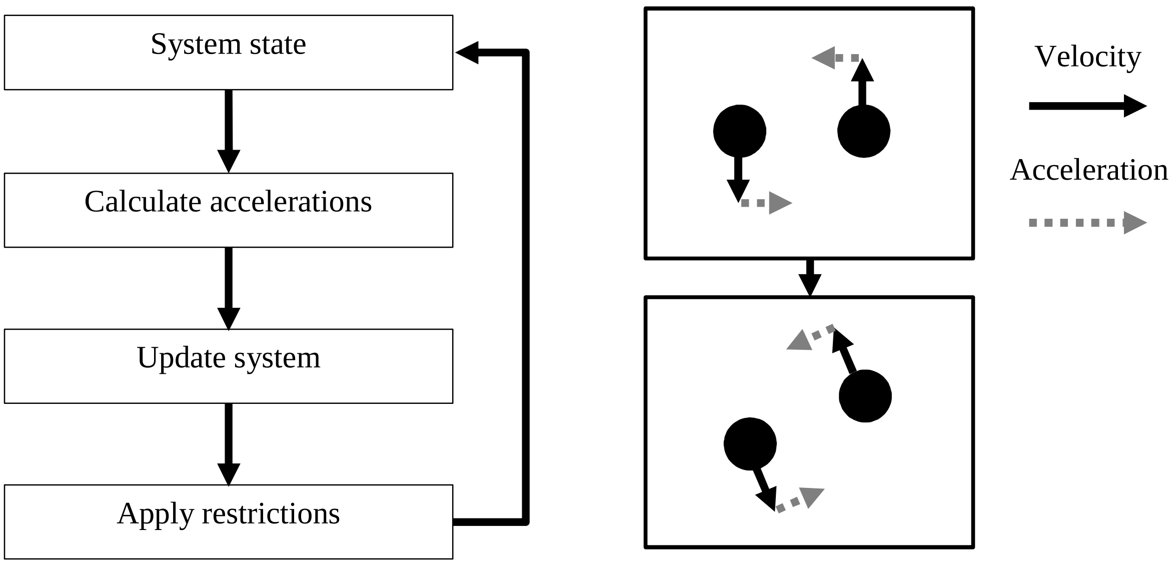

Figure 2.1: *Steps in molecular dynamics simulations are shown on the left. An example of how the positions of atoms can be updated is shown on the right.*

Maxwell-Boltzmann distribution in order that the system has the desired temperature. The acceleration, $\bar{a}$, for each atom is calculated from the force by using Newton’s second law,

$$\bar{a} = \bar{F}/m \tag{2.1}$$

where $\bar{F}$ is the force on the atom and the mass of the atom is given by *m*. The force can be calculated from the potential [**8**] as follows:

$$\bar{F} = -\nabla V \tag{2.2}$$

where *V* is the potential. The system is updated by performing a step with the integrator. There are restrictions placed on the positions by periodic boundary conditions. The velocities are controlled by the thermostat in order to keep the temperature at the desired value.

2.2 Potentials

The force as given in (2.2) is a vector field while the potential is a scalar field. It is simpler to communicate the potential. The potential can then be used to calculate the forces for the simulation.

All the interactions of the electrons are contained in the potential chosen. For this reason it is important to choose an appropriate potential. Two different types of potential will now be discussed.

2.2.1 Lennard-Jones potential

The Lennard-Jones potential is a pair potential that is useful to simulate noble gasses and as a first approximation for other systems. In a pair potential the influence between a pair of atoms is independent of the environment. An advantage of using a pair potential is that the calculations are fast because of the simplified model. A disadvantage of using a pair potential is that a pair potential will not be accurate for systems where the environment influences interactions.

The Lennard-Jones potential between atoms *i* and *j*, $V_{LJ}(ij)$, is given by [**9**]

$$V_{LJ}(ij) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{2.3}$$

where σ is the finite distance at which the inter-atomic potential is zero, r_{ij} is the distance between atoms *i* and *j,* and ε determines the strength of the binding. The potential and the contribution from the $\left(\sigma/r_{ij}\right)^{n}$ terms can be seen in *Figure 2.2*. The first term represents the repulsion between the atoms and dominates when the atoms are too close together. The second term represents the attraction between the atoms.

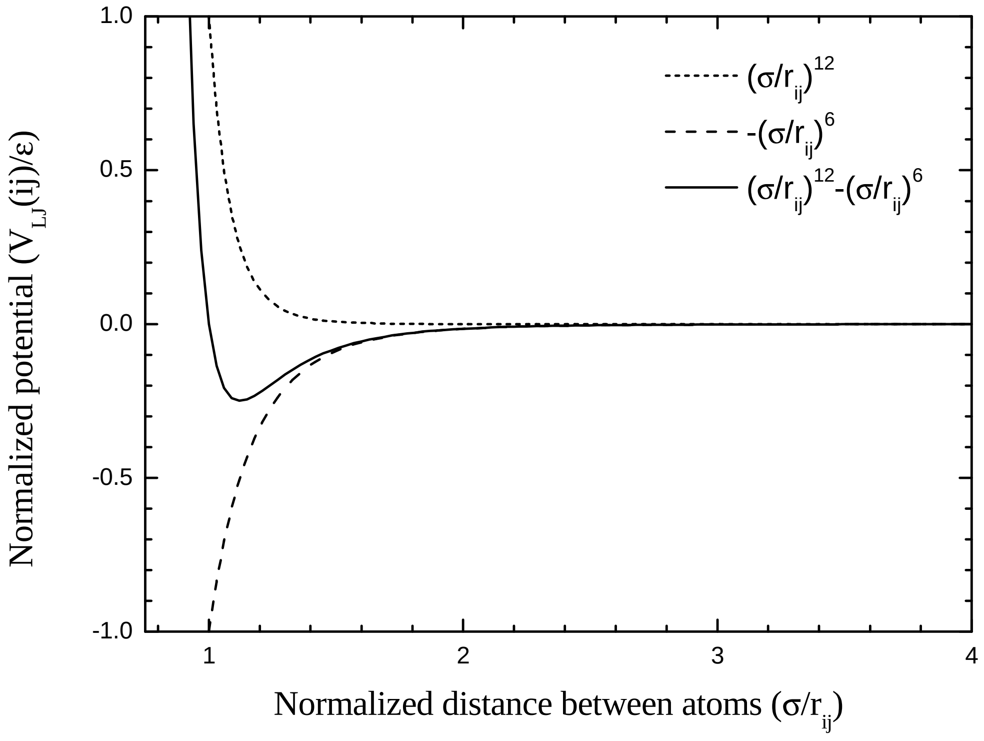

Figure 2.2: *Interaction between the repulsive and attractive terms of the Lennard-Jones potential.*

The force from the Lennard-Jones potential can be calculated by using equations (2.1) and (2.2). The derivation is done in Appendix A. The force between atoms *i* and *j*, $\overline{F_{LJ}(ij)}$, is given by

$$\overline{F_{LJ}(ij)} = \frac{48\varepsilon}{\sigma^2}\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\bar{r} \tag{2.4}$$

where $\bar{r}$ is the displacement between atoms *i* and *j*.

Table 2.1: Sutton-Chen parameters for Pt

parameter	value
m	8
n	10
c	34.408
ε	1.9833x10^{-2} eV
a	3.92 Å

2.2.2 The Sutton-Chen potential

The Sutton-Chen potential is more suitable for use with metallic atoms like Pt. A potential can be calculated by using the embedded atom model (EAM). In the embedded atom model the environment is also taken into account when calculating the interaction between two atoms. This allows for a much more accurate simulation of metallic systems. For this investigation the Sutton-Chen potential was used because it is well suited to face centred cubic (fcc) metals like Pt [**10**] [**11**] [**12**] [**13**] while the Lennard-Jones is better suited to noble gasses. The Sutton-Chen potential for atom *i* is given by

$$V_{SC}(i) = \varepsilon\left[\frac{1}{2}\sum_{j\neq i}\left(\frac{a}{r_{ij}}\right)^{n} - c\sqrt{\rho_i}\right] \tag{2.5}$$

where

$$\rho_i = \sum_{j\neq i}\left(\frac{a}{r_{ij}}\right)^{m}. \tag{2.6}$$

where r_{ij} is the distance between atoms *i* and *j*, *a* is the length of the unit cell of the fcc metal, and the parameters *m, n*, *c* and ε are determined by fitting so that the material in the simulation will have the same properties as found experimentally. The values of these parameters are given in [**10**] and reproduced in Table 2.1.

In equation (2.5) the first term, $\sum\left(a/r_{ij}\right)^{n}$, takes the repulsion between the like charges into account and the second term, $\sqrt{\rho_i}$, takes the attraction into account. The repulsion is mostly between the nuclei and is similar in form to that of the Lennard-Jones potential. For the attraction the electron cloud is taken as the density of the nuclei.

Combining equations (2.2) and (2.5) it is found that the force on atom *i* is [**7**]

$$\overline{F_i} = -\varepsilon \sum_{j \neq i} \left[n \left(\frac{a}{r_{ij}} \right)^n - \frac{cm}{2} \left(\frac{1}{\sqrt{\rho_i}} + \frac{1}{\sqrt{\rho_j}} \right) \left(\frac{a}{r_{ij}} \right)^m \right] \frac{\overline{r}_{ij}}{r_{ij}^{\ 2}} . \tag{2.7}$$

2.3 The integrator: updating the system

During the integration step the positions and velocities of the atoms are updated by a small amount. The small amount by which the simulation is advanced is called the time step. This updating is repeated until a desired condition is met. The size of the time step will determine how fast the computer simulation will run, but it also determines how large the error is in the simulation [**14**]. A larger time step will let the simulation run faster but also introduce a larger error. When the time step is larger than a critical value, which is smaller than the fastest oscillation period in the simulation, the error will dominate the results. It is therefore important to choose a time step that balances the error in the simulation and the speed at which the simulation runs. A value of 5 *fs* for the time step is a safe choice.

2.3.1 Störmer-Verlet integrator

The Störmer-Verlet integrator is one of the methods that can be used in order to update the positions of the atoms in a simulation. The Störmer-Verlet integrator is given by [**7**]

$$\overline{x}_{i,t+1} = 2\overline{x}_{i,t} - \overline{x}_{i,t-1} + \overline{a}_{i,t} dt^2 \tag{2.8}$$

where $\overline{x}_{i,t}$ is the position and $\overline{a}_{i,t}$ is the acceleration of atom *i* at time step *t*, and *dt* is the duration of one time step. This method is susceptible to rounding errors and does not calculate the velocities of the atoms.

2.3.2 Leapfrog integrator

The Leapfrog integrator advances the simulation to the next time step by updating the positions and velocities of all the atoms. The Leapfrog integrator can be written in the form

$$\bar{x}_{i,t+1} = \bar{x}_{i,t} + \bar{v}_{i,t+1/2} dt \tag{2.9}$$

$$\bar{v}_{i,t+1/2} = \bar{v}_{i,t-1/2} + \bar{a}_{i,t} dt \tag{2.10}$$

where $\bar{x}_{i,t}$ is the position, $\bar{v}_{i,t}$ is the velocity and $\bar{a}_{i,t}$ is the acceleration of atom *i* at time step *t*, and *dt* is the duration of one time step. Compared to the Störmer-Verlet integrator, the Leapfrog integrator are less susceptible to rounding errors. The velocities are calculated at times between those for which the positions are calculated, thus the Leapfrog name. Additional calculations would be required for algorithms that require the velocities and positions at the same time.

2.3.3 Velocity Verlet integrator

The Verlet integrator, also known as the Velocity Verlet or Velocity-Störmer-Verlet integrator, is even less susceptible to rounding errors than the Leapfrog integrator. Additionally, the velocities are calculated for the same times as the positions. The velocity Verlet integrator is given by

$$\bar{x}_{i,t+1} = \bar{x}_{i,t} + \bar{v}_{i,t} dt + \bar{a}_{i,t} dt^2 / 2 \tag{2.11}$$

$$\bar{v}_{i,t+1} = \bar{v}_{i,t} + \bar{a}_{i,t}\, dt/2 + \bar{a}_{i,t+1}\, dt/2\,. \tag{2.12}$$

The Velocity Verlet integrator is recommended for use since it has the best error characteristics with very little computational overhead.

2.4 Thermostat

In a system the atoms move around with velocities according to the Boltzmann distribution. The temperature, *T*, of a system is related to the average kinetic energy, *<KE>*, of the atoms in the system by Boltzmann's constant, *k*, according to the following equation [**15**]:

$$< KE >= \frac{3}{2} kT \, . \tag{2.13}$$

The relationship between the kinetic energy and the velocity, *v*, of an atom with a mass of *m* is given by [**16**]

$$KE = \frac{1}{2} mv^2 \, . \tag{2.14}$$

From equations (2.13) and (2.14) it can be deduced that the temperature of the system can be controlled by scaling the velocities by a factor λ so that

$$\overline{v}_{i,new} = \lambda \overline{v}_{i,old} \tag{2.15}$$

where $\overline{v}_{i,old}$ is atom *i*'s old velocity and $\overline{v}_{i,new}$ is atom *i*'s new scaled velocity. One method to control the temperature would be to calculate λ so that the temperature will be exactly what it should be after each rescaling. The problem with this procedure is that there are natural fluctuations in temperature when considering the short timescales of the simulations. For this reason the Berendsen thermostat was used. The Berendsen thermostat allows fluctuations in the temperature while still keeping the temperature close to the desired temperature. The Berendsen thermostat calculates the scaling factor as [**7**]

$$\lambda = \sqrt{1 + \frac{dt}{\tau}\left(\frac{T_0}{T} - 1\right)} \, . \tag{2.16}$$

where T_0 is the desired temperature, *T* is the current temperature, *dt* is the time step, and τ is the damping parameter. The damping parameter determines how long it takes to reach the desired temperature.

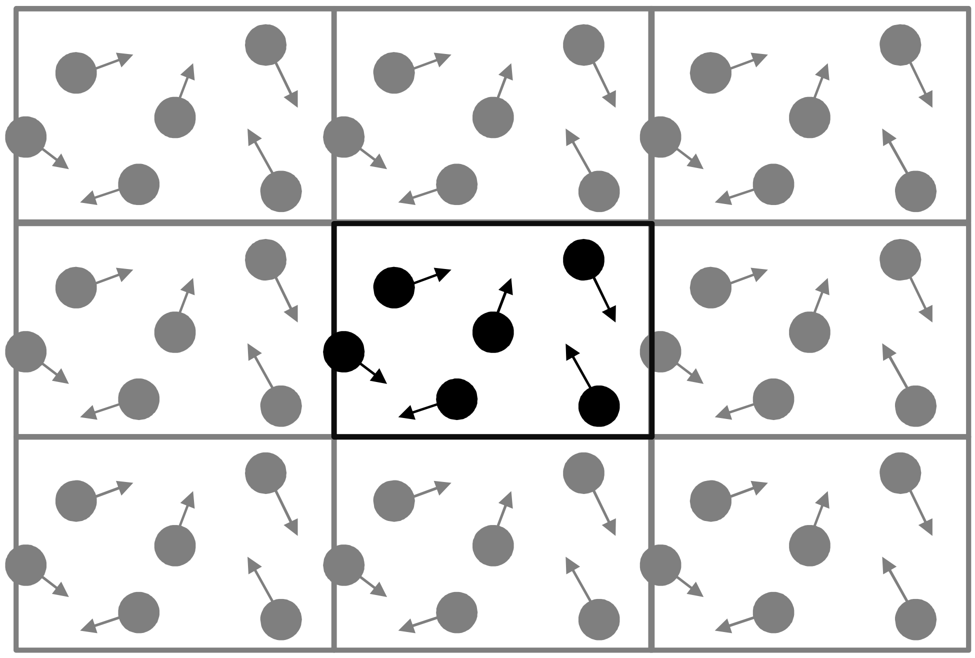

Figure 2.3: *Illustration of wraparound boundary condition. The darker central atoms are in the simulation area. The boundary conditions tile the simulation area in order to give the appearance of a larger environment to the atoms in the simulation.*

2.5 Boundary conditions

In simulations of a surface it is desirable to minimize influences like the boundary of the crystal. To remove the boundaries of the crystal wraparound boundary conditions as illustrated in *Figure 2.3* was used. This has the effect that an atom that moves out of the simulation area towards the right will enter again from the left. The force that the atoms have on each other is also affected by the wraparound conditions.

2.6 Big O notation

Big O notation is a method to communicate how well an algorithm scales with an increase in the amount of data the algorithm operates on. How well an algorithm scales gives an indication of what amount of data can be processed in a feasible duration of time.

Big O notation only gives the magnitude of the number of operations that must be performed as a function of the data set size, n. An algorithm that performs a single calculation for each element will scale linearly and have a complexity of $O(n)$. An algorithm that has to consider every element for every other element will scale quadratically and have a complexity of $O(n^2)$.

It is better to use an $O(n)$ algorithm than an $O(n^2)$ algorithm because the $O(n^2)$ will take n times longer to calculate than an $O(n)$ algorithm.

2.7 Cut-off radius

From equation (2.5) it can be seen that the effect that an atom has on another decreases with $1/r$ to the power n. Thus the effect that an atom has on another quickly becomes negligible. Atoms further than some cut-off radius, r_c, from atom i can be ignored safely when calculating the potential for atom i. A cut-off radius of 2.5 times the nearest neighbour distance was used [**7**]. The influence of the atoms beyond the cut-off radius is taken as zero. The advantage of using a cut-off radius is that fewer atoms have to be considered and the simulation will be faster. The nearest neighbour distance of platinum is 2.77 Å therefore the cut-off radius was 7 Å.

Care should still be taken with very small (less than $4r_c$) simulations in order not to introduce artefacts. When the simulation is very small an atom can influence another atom twice because of wraparound. *Figure 2.4* helps to illustrate how this can happen. When calculating the force on atom *C*, all the atoms closer than r_c have to be considered. For each of these atoms their nearest

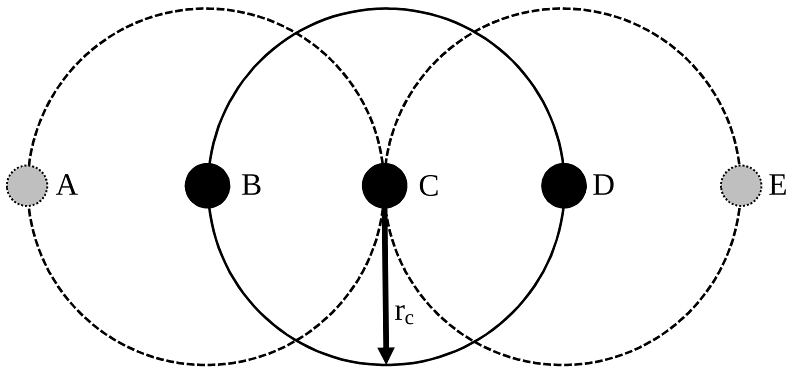

Figure 2.4: *Illustration of atoms that have an influence on atom C when a cut-off radius of r_c is used.*

neighbours also have to be considered because of the ρ_j term in equation (2.7). Atom *A* is part of the ρ_j term for *B* which influences *C*. In a similar way atom *E* also influences *C*. If the size of the simulation is $4r_c$, atom *A* will effectively be the same as atom *E*. Atom *C* will then be influenced twice by atom *A*. In order to prevent this, the simulation has to have a size larger than $4r_c$.

2.8 Cell structure for optimised calculation

Without the cut-off radius, performing a step has a time complexity of $O(n^2)$ because for each atom every other atom has to be considered. When using the cut-off radius many atoms can be skipped in the calculations because those atoms are far away. All the atoms still have to be checked for their distance which means that the algorithm is still $O(n^2)$.

There is a maximum number of atoms that can fit inside the cut-off radius. Because of this, the algorithm can be modified to have a complexity of $O(n)$. The modified algorithm will iterate through all the atoms. For each of those atoms all the atoms closer than r_c are considered. Since the number of atoms closer than r_c is independent of n, the algorithm has a complexity of $O(n)$. The key ideas to change the algorithm from $O(n^2)$ to $O(n)$ are:

- Using a cut-off radius so that each atom is only influenced by a small number of atoms that are independent of *n* and less than some maximum value.

Table 2.2: Comparison of the performance of lists and cells

	nearest neighbour lists	cells
memory usage	*n* x list size	*n* x integer size
complexity to update data structure	O(n^2)	O(n)

- Using a data structure that gives a small subset of all the atoms in the simulation. This subset must contain all the atoms that are closer than r_c.

Two data structures that can be used for this optimisation are nearest neighbour lists and cells [**9**] [**17**] [**18**].

With the neighbour list, each atom has a list of atoms that are closer than a chosen distance to that atom. The distance would be larger than r_c by an amount of Δr. All the atoms that must be considered will be in the list associated with each atom. As the atoms move around, the atoms will move to different lists. The lists will then have to be updated. Updating the lists is computationally expensive so it should only be done when absolutely necessary. The value of Δr will determine how frequently the lists need to be updated. A small value for Δr means that the lists will have to be updated frequently. Using a large value for Δr means that more atoms will be in the list. Larger lists will lead to slower calculation. The value for Δr has to be chosen carefully in order to give the best results.

When using the cell structure, the simulation area is divided into cells as shown in *Figure 2.5a*. All the atoms are put into cells. Each atom is put in the cell determined by the atom's position. In order to find all the interacting pairs for an atom in the centre cell, only the atoms in the neighbouring cells have to be considered. The choice of Δr is subject to the same considerations that were discussed for lists.

A comparison of the performance of lists and cells is given in Table 2.2. From the comparison it is clear that the cell structure will scale better as the simulation size increases. The cell structure was chosen because it scales better with larger simulations. The cell structure is only an optimisation used in order to find all the interacting pairs of atoms. The calculation of the potential from the pairs is discussed in the next section.

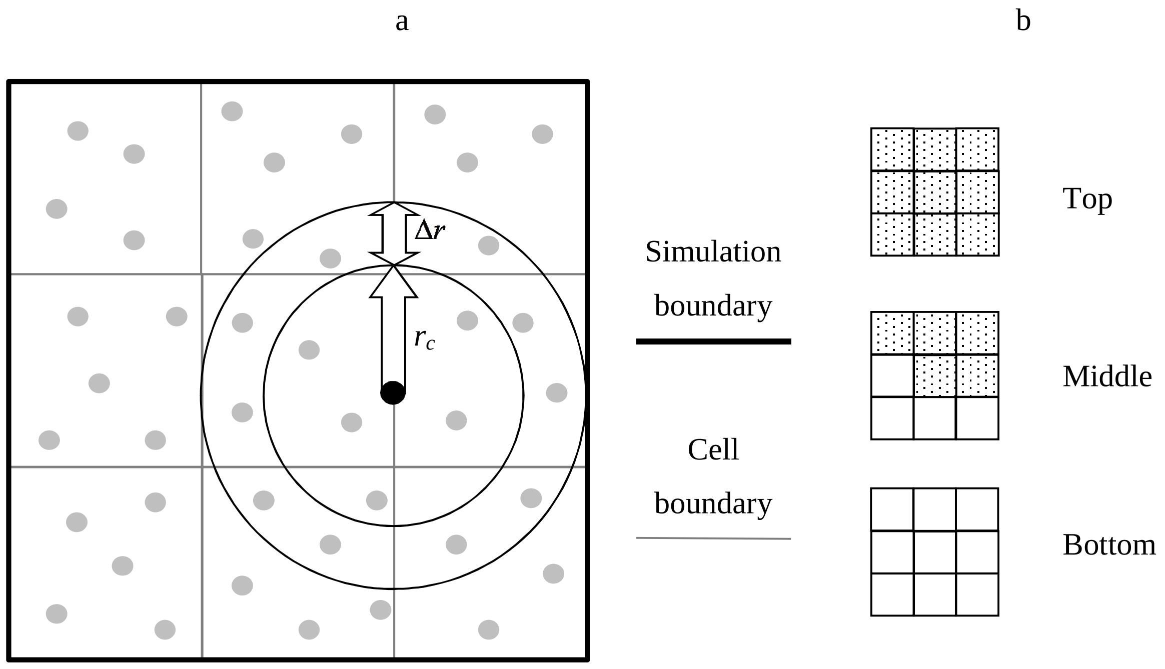

Figure 2.5: *Illustration of the cell structure in a plane in 2.5a on the left. Illustration of the cells to be considered when looking at the centre cell of a cube can be seen in 2.5b on the right.*

A further optimization can be made when finding all the pairs. In order to find all the interacting atoms when working with the atoms in one cell, all 26 neighbouring cells have to be considered. When a pair is found, the results can be saved for both atoms. This means that only half the pairs have to be found for each atom. The neighbouring cells that has to be searched is reduces from 26 to 13. The cells that are considered when looking at the middle cell in a cube of 3 x 3 cells are shaded in *Figure 2.5b*.

Setting up the cell structure is computationally expensive so it is desirable not to do it at every step in the simulation. *Figure 2.6* helps to clarify when the structure should be refreshed. Consider the configuration that would require the fastest updating of the structure. This configuration is where the atoms are just outside of the border of a cell between them and are moving towards each other with a velocity v_{max}. This velocity is found by determining the maximum velocity in the system. In a time step of length dt, the distance between any pair of atoms decreases by no more than $2v_{max}\,dt$. The cells has to be updated when the sum of all these movements are greater than Δr. The cells should thus be refreshed when

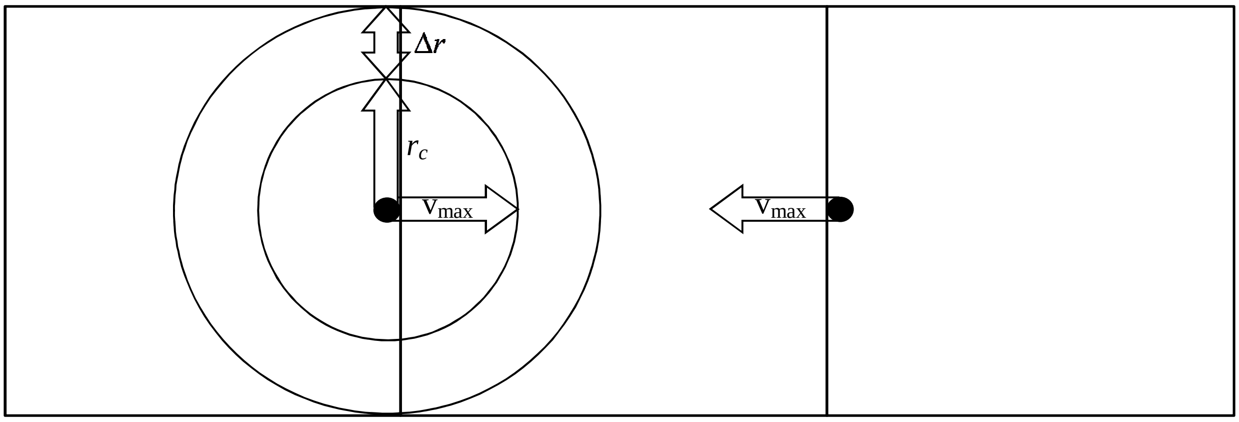

Figure 2.6: *Setup to determine when the cell structure should be refreshed.*

$$\sum_{steps} (\max_i |v_i|) \geq \frac{\Delta r}{2dt} . \tag{2.17}$$

The maximum velocity found during the current step is given by max |*v*|. When the cells are refreshed, the sum is reset to zero.

2.9 Potential calculation

For each integration time step the potential is calculated in two steps. In the first step each pair of interacting atoms are found from the cell structure and the pair's interaction parameters are saved into a temporary structure. In the second step the force and potential is calculated from the calculations in the second step.

In order that the parts of the equation that can be calculated in the first step becomes more clear, equation (2.7) can be rearranged as

$$F_i = -\varepsilon \left\{ n \sum_{j \neq i} \left[\left(\frac{a}{r_{ij}} \right)^n \frac{\bar{r}_{ij}}{r_{ij}^2} \right] - \frac{cm}{2} \sum_{j \neq i} \left(\frac{1}{\sqrt{[\rho_i]}} + \frac{1}{\sqrt{[\rho_j]}} \right) \left[\left(\frac{a}{r_{ij}} \right)^m \frac{\bar{r}_{ij}}{r_{ij}^2} \right] \right\} . \tag{2.18}$$

Each of the parts that are surrounded by square brackets in equation (2.18) can be updated when looking at a single pair, but all the pairs have to be traversed twice in order to calculate the whole equation. The terms in the square brackets together with the first term in equation (2.5) are saved during the first step. The indexes *i* and *j* are saved together with the last square bracket in equation (2.18) for the next step.

The potential and accelerations can now be calculated from the parts that were saved in the previous step. Care has to be taken with the units when doing this step. One option is to use dimensionless units, that is choose the distance between atoms, time between steps and the unit of energy to be one [**17**]. The problem with that approach is that the data has to be converted before interpretation. The approach taken here was to choose the unit of length as angstrom (Å), time as picosecond (ps), mass as atomic mass unit (amu) and energy as electron volt (eV). With SI units the numbers would have been very small. Numbers that are very small can cause rounding off errors when used with a computer's limited precision. Using the chosen units minimizes this problem.

When combining equations (2.1) and (2.7) it is found that the acceleration is given in eV/(Å amu). The integrator requires the acceleration in units of Å/ps^2. The derivation of the conversion factor can be done as

$$\frac{\text{eV}}{\text{Å amu}}\frac{1000N_a\ \text{amu}}{\text{kg}}\frac{e\ \text{kg m}^2}{\text{eV s}^2}\left(\frac{10^{10}\text{Å}}{\text{m}}\right)^2\left(\frac{\text{s}}{10^{12}\text{ps}}\right)^2=\frac{eN_a}{10}\frac{\text{eV}}{\text{Å amu}}=\frac{\text{Å}}{\text{ps}^2} \tag{2.19}$$

where Avogadro's number is N_a and an electron's charge is e. The acceleration can now be converted as

$$a_{\text{int}}\left(\frac{\text{Å}}{\text{ps}^2}\right)=\frac{eN_a}{10}a_{pot}\left(\frac{\text{eV}}{\text{Å amu}}\right)\approx 9600a_{pot}\left(\frac{\text{eV}}{\text{Å amu}}\right) \tag{2.20}$$

Where a_{int} is the acceleration as required by the integrator and a_{pot} is the acceleration as provided by the potential calculation.

Chapter 3: Small structures

In this chapter systems of less than ten atoms on a clean surface will be reported. The understanding gained from investigating the small structures will help explain the results from larger simulations.

The structures that have the lowest energies will be investigated first. The binding energy per atom for these structures will be investigated next. The energy barriers as atoms move around these structures will be calculated. A model will be proposed to predict the energy barriers.

3.1 Minimum energy structures

When adatoms move on a surface, the adatoms will tend to cluster together into structures with the lowest possible energy. The low energy structures with less than seven atoms are shown in *Figure 3.1*. *Figure 3.2* shows the low energy structures with seven, eight and nine atoms. All the structures are on the (111) plane of a fcc Pt crystal. The surface is shown in teal in *Figure 3.1* and *Figure 3.2*. The atoms that form part of the structures are shown in red. The positions where atoms can be added adjacent to the structures are indicated with smaller white spheres with letters on them. The letters are chosen in a way such that:

- Positions that are similar because of symmetry has the same letter
- The letters that denote transitions between adjacent positions can only be the same if the transitions are similar. This requirement will be discussed more in section 3.3 on energy barriers.

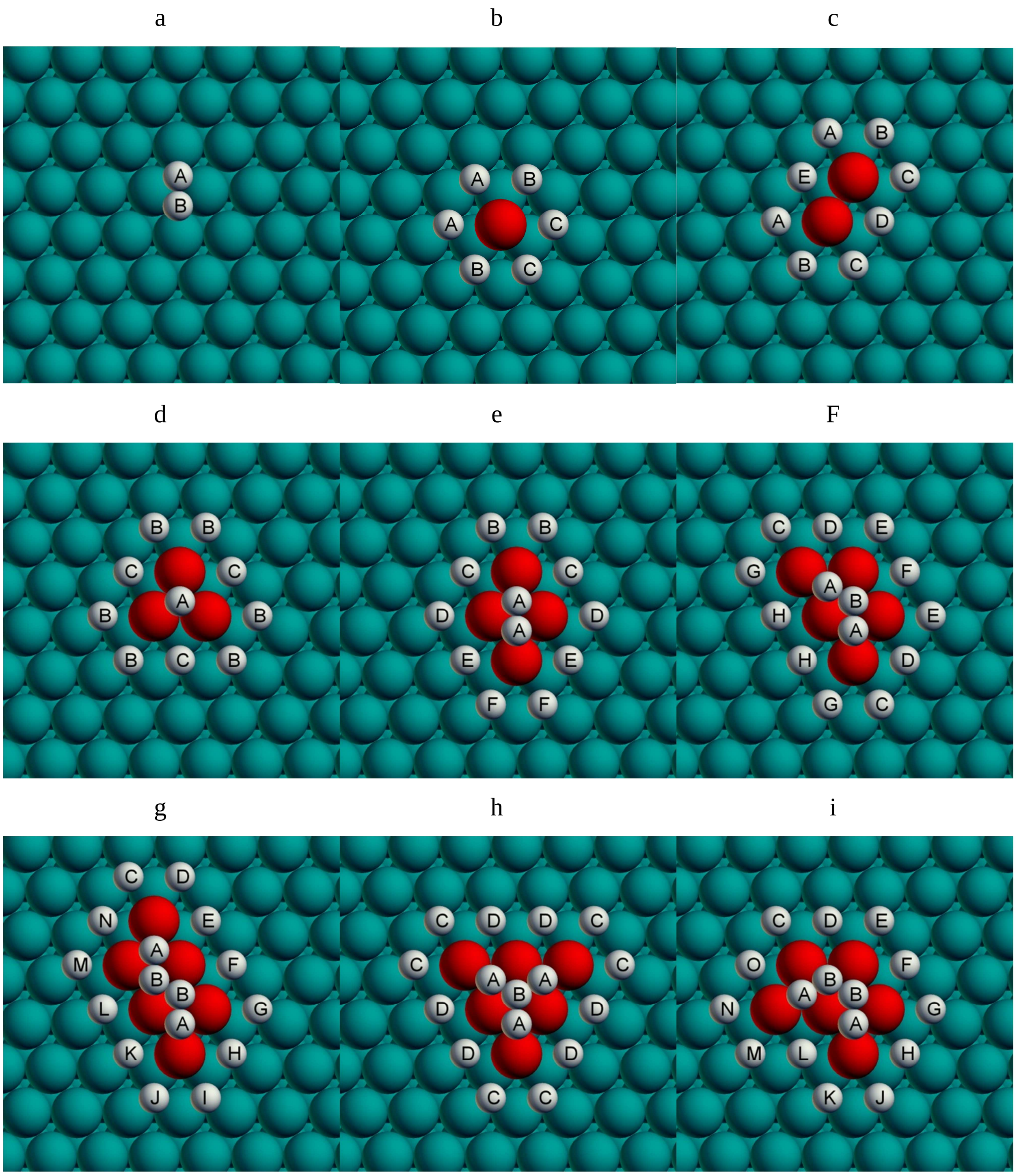

Figure 3.1: *Low energy structures with less than seven atoms. The positions that are adjacent to the structures are indicated with lettered spheres.*

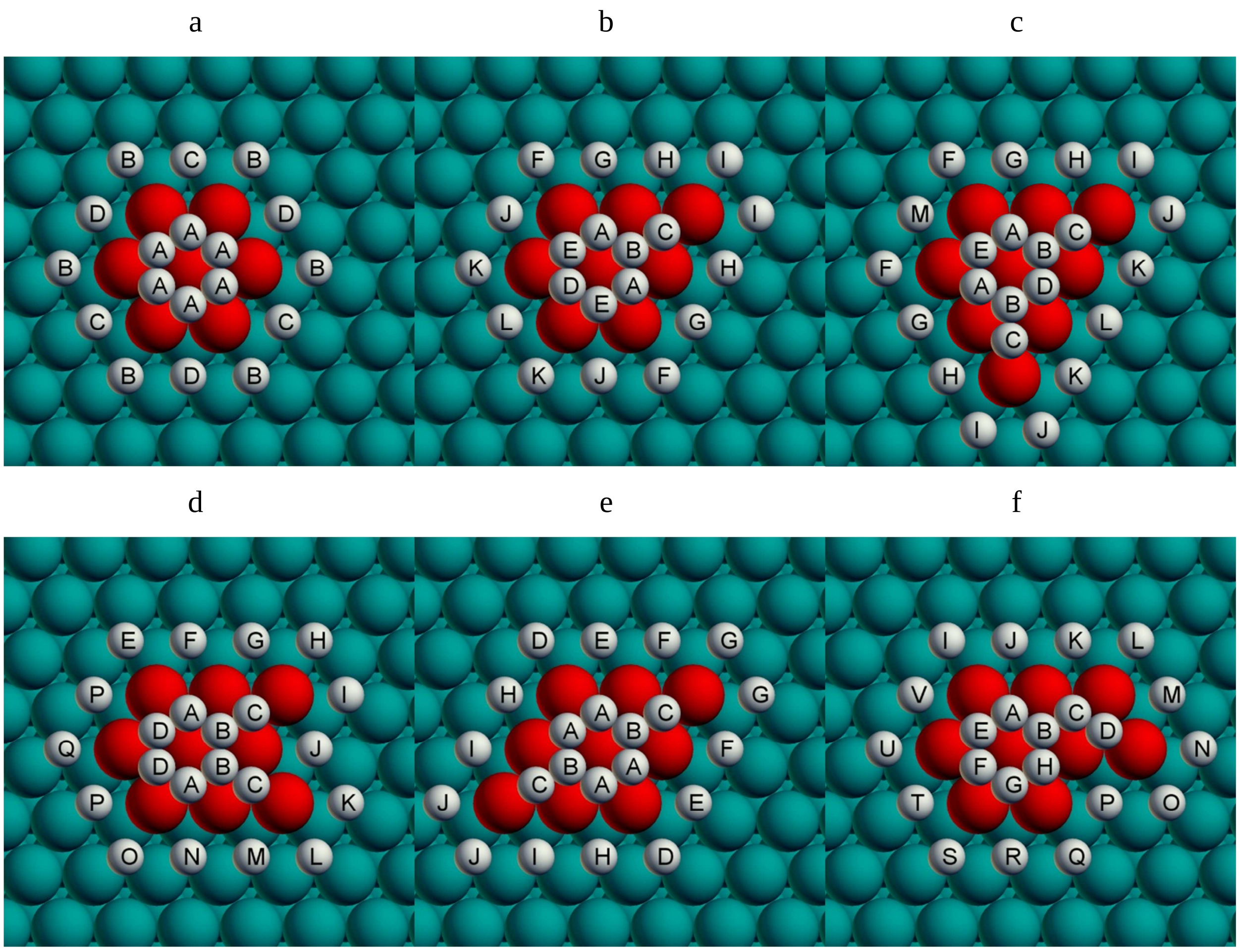

Figure 3.2: *Low energy structures with seven to nine atoms. The positions that are adjacent to the structures are indicated with lettered spheres.*

The low energy structures are found with an iterative process. At the start there is just a clean surface as shown in *Figure 3.1a*. The process to find the structures is:

- The binding energies are measured at each of the positions next to the structures as indicated in *Figure 3.1* and *Figure 3.2*.
- An atom is placed at the position with the highest binding energy. There are two special cases when finding the highest binding energy:
  - o In cases where there are positions where the binding energies are essentially the same, and those positions are similar because of symmetry, an atom is placed in only one of those positions. This can be seen *Figure 3.1a* and *Figure 3.1b.*

Table 3.1: Parameters used for simulation.

variable	value
Timestep (dt)	0.005 ps
Tau for thermostat	0.1
Desired temperature for thermostat	0 K
Mass of Pt	195.084 amu
Cut off radius	7 Å
Cell size	8 Å
Simulation size in x direction	30.47 Å
Simulation size in y direction	28.79 Å
Number of atom layers	3
Number of atoms in base	396

  - In cases where the highest energies are almost the same but different structures will be produced, all the structures are investigated. Multiple structures form for six atoms as can be seen in *Figure 3.1g* to *Figure 3.1i*. In the next iteration the binding energies for all the structures are considered when finding the highest binding energy. For seven atoms the one structure that had the highest binding energy is shown in *Figure 3.2a*.
- Repeat until the structures have the desired size.

The values of the variables used in the simulation of these structures are given in Table 3.1. In order to measure the binding energies at the testing positions, atoms are placed in those positions. The energy for only one position is measured at a time. After placing the test atom, the simulation is run for 1200 steps (6 ps). This allows the system to relax to the lowest energy. At the end of the relaxation the potential energy of the systems changes by less than 10^{-6} eV per step.

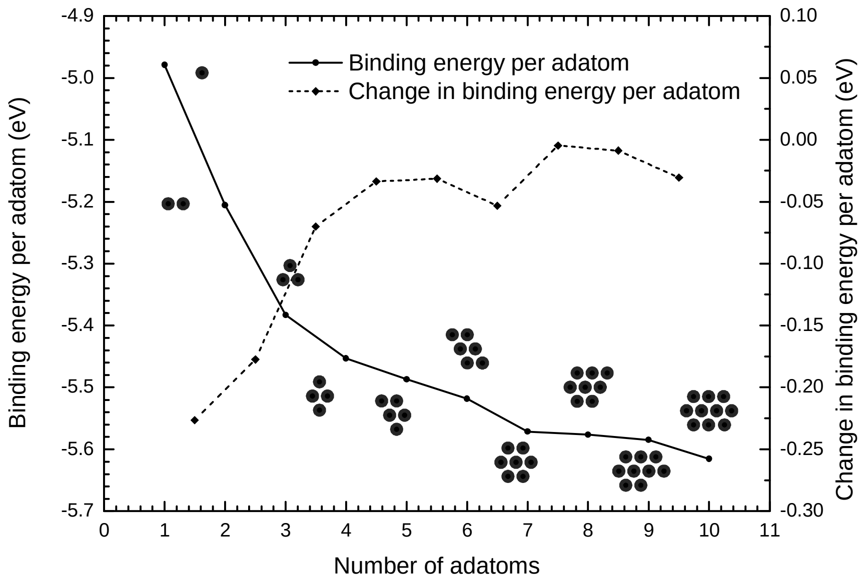

Figure 3.3: *Binding energy per adatom as a function or the number of adatoms. The structures that form are shown close to the corresponding energies.*

From the structures that are shown in *Figure 3.1* and *Figure 3.2* it can be seen that the low energy structures maximise the number of bonds to neighbouring atoms. The number of bonds are maximised in a rounded, closed structure where no atoms stick out from the structure. When considering the binding energy per atom it will be shown that the binding energy is higher for the structures that are more rounded.

3.2 Binding energy per atom

In *Figure 3.3* the binding energy per adatom as a function of number of adatoms can be seen. The binding energy per adatom is calculated as the total potential energy the adatoms added to the system divided by the total number of adatoms. The kinetic energy is zero since the target temperature for the thermostat was 0 K. A more negative binding energy means that the structure is more stable.

Table 3.2: Energies needed to remove a single atom from various positions.

Position atom is removed from	Energy (eV)
On surface	-4.984
In surface	-6.492
In bulk	-6.866

From *Figure 3.3* it can be seen that adding an atom increases the average strength by which all the adatoms are bound together. Each additional atom increases the number of bonds to nearest neighbours and next nearest neighbours. It can be seen that the change in binding energy tends to decrease as the number of adatoms increase. There are exceptions to the trend in the change in binding energy when the structures with seven and ten atoms form. For the structures with seven and ten atoms, all the atoms have at least three nearest neighbours on the surface. All the other structures contain at least one atom with only two nearest neighbours.

For comparison, the energies to remove atoms from various positions are given in Table 3.2. The energies were measured with a simulation of a cubical crystal that contained 7140 atoms. The binding energy per atom for this larger crystal is -5.811 eV/atom. This is more than the value shown in *Figure 3.3* because surface has a smaller influence in the larger simulation. The binding energy per atom is the energy that must be added to each atom in order to completely break up the crystal. The binding energy per atom is the same as the sublimation energy. The sublimation energy for Pt is -5.856 eV/atom [**19**]. This compares very favourably to the value of -5.811 eV/atom found in the simulation.

The vacancy formation energy is the energy required to remove an atom from the bulk and place it on the surface. From the values in Table 3.2 the energy required to form the first vacancy can be calculated as 6.866 eV – 4.984 eV in order to give 1.88 eV. This compares well to the theoretical values of 1.68 eV [**20**], 1.60 eV [**21**], 1.28 eV [**22**] and 1.45 eV [**23**] and the experimental value of 1.6 eV [**20**].

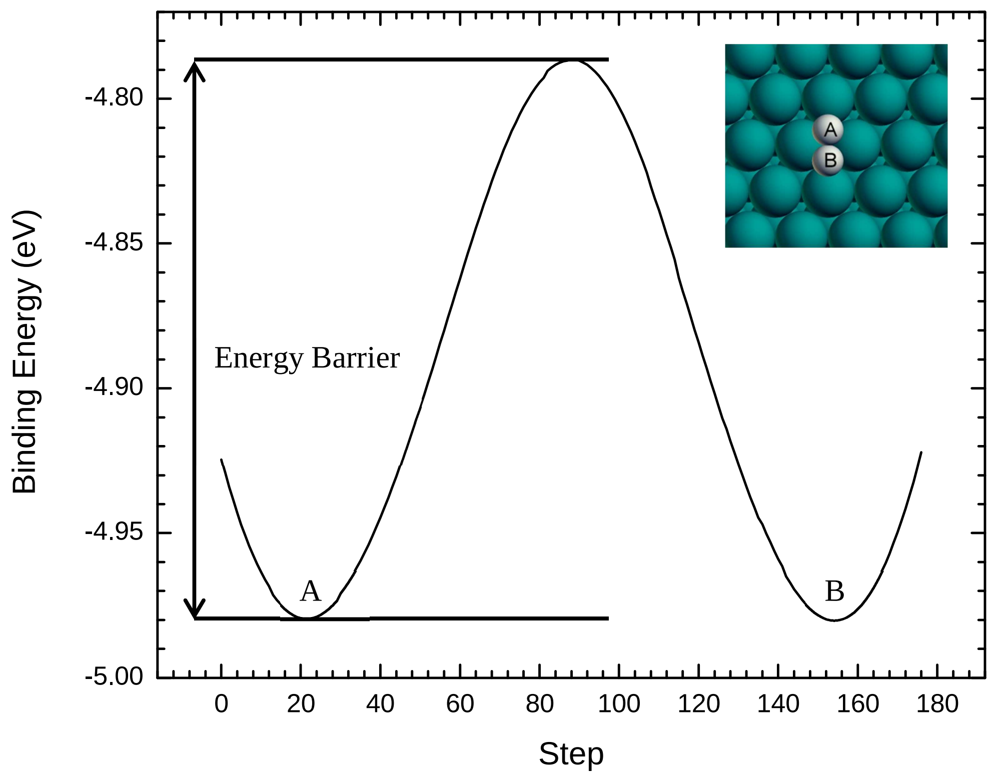

Figure 3.4: *The energy along the path taken when an atom hops from A to an adjacent site, B, on a clean Pt(111) surface. The energy barrier is indicated as the difference between the minimum at the starting position and the maximum. Figure 3.5 shows the hop from the side.*

3.3 Energy barriers

In order for the adatoms to move, an energy barrier must be overcome. Heat provides the energy required for the adatoms to overcome this energy barrier. The energy barriers for atoms to move around the structures will now be calculated. The barriers will help to understand the structures that form during deposition.

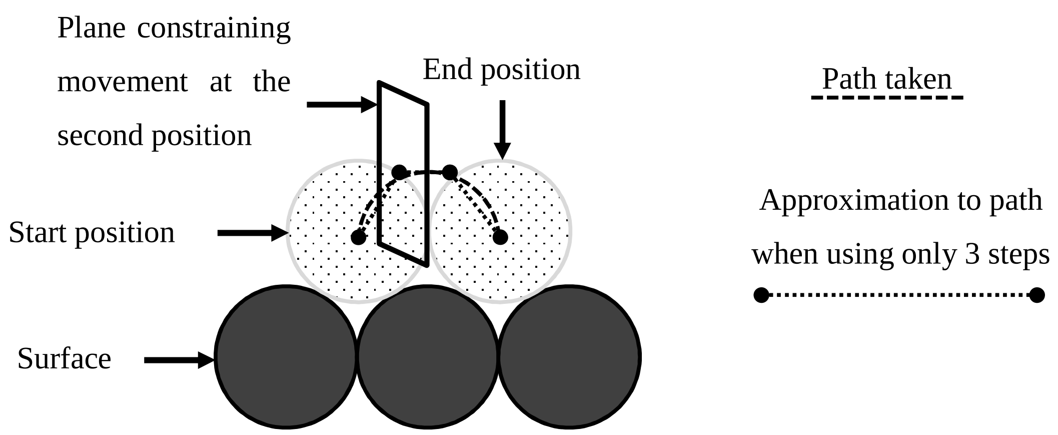
Plane constraining movement at the second position
End position
Path taken
Start position
Approximation to path when using only 3 steps
Surface

Figure 3.5: *Finding the path of minimum energy.*

An example of an energy barrier can be seen in *Figure 3.4*. *Figure 3.4* shows the potential energy for a single jump as on a clean surface from position *A* to *B* as shown in the inset. The energy barrier is the difference between the first minimum, which is where the atom would start, and the maximum.

In order to calculate the energy barrier for an atom to move to another position it has to be moved along the lowest energy path. Therefore the first step is to determine the path along which the atom will move. The change in potential energy of the system will then be used to determine the size of the energy barriers. Since the potential energy fluctuates when the system is not at 0 K all the simulations were done at 0 K.

Figure 3.5 illustrates how the path of minimum energy was found. The test atom is moved in the direction from the start to the end positions in small increments. After each movement of the test atom, the structure is allowed to relax while the test atom is constrained to a plane perpendicular to the direction of movement. Since there was only a small change from the previous state the system was relaxed for only 400 steps. After the 400 steps the change in energy was less than 10^{-6} eV. Only the atoms closer than 16 Å to the test atom were allowed to relax, in order to prevent the whole structure from deforming. With small increments the test atom will move along the path of lowest energy. A hundred steps were used to move the atom from the start to end positions.

The activation energy that is required to move an atom from one position to the next is the difference between the binding energy at the beginning and the binding energy at the highest point in the barrier. In Table 3.3 it can be seen that the activation energy for self-diffusion on a clean Pt(111) surface is 0.193 eV. This value compares well with previously published values of 0.194 eV [**24**], 0.260 eV [**25**] [**26**], 0.250 eV [**27**], 0.160 eV [**28**] and 0.176 eV [**29**].

The energy barriers between all adjacent test atoms in *Figure 3.1* and *Figure 3.2* were calculated. Table 3.5 gives the energies for *Figure 3.1c*. The first two columns give the letters that are used to indicate the start and the end position of the atom when it was moved. The binding energy in the start and end positions are given in the next two columns. The fifth column gives the barrier energy as calculated in the simulation. The last column gives the barrier energy as predicted by the model proposed in the next section.

When there are jumps that are similar because of symmetry only one of them are given. Care has to be taken to ensure that the jumps are actually similar. Consider *Figure 3.1*c. The A-B and B-C jumps appear symmetric when only considering the adatoms. Those jumps are not symmetric when the surface structure is considered.

The barrier energies for the first four structures in *Figure 3.1* are given in Table 3.3, Table 3.4, Table 3.5, and Table 3.6. The energies for the rest of the structures in *Figure 3.1* and *Figure 3.2* are given in Appendix B.

Table 3.3: Energies from *Figure 3.1a*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.979	4.980	0.193	0.010

Table 3.4: Energies from *Figure 3.1b*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	5.428	5.432	0.244	0.258
B	C	5.428	5.432	0.244	0.258
C	A	5.428	5.432	0.244	0.269

Table 3.5: Energies from *Figure 3.1c*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	E	5.377	5.737	0.120	0.209
A	B	5.377	5.348	0.323	0.265
B	C	5.345	5.363	0.188	0.257
C	D	5.376	5.738	0.201	0.215

Table 3.6: Energies from *Figure 3.1d*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	C	4.992	5.654	0.424	0.429
B	B	5.328	5.332	0.298	0.288
B	C	5.328	5.663	0.216	0.219

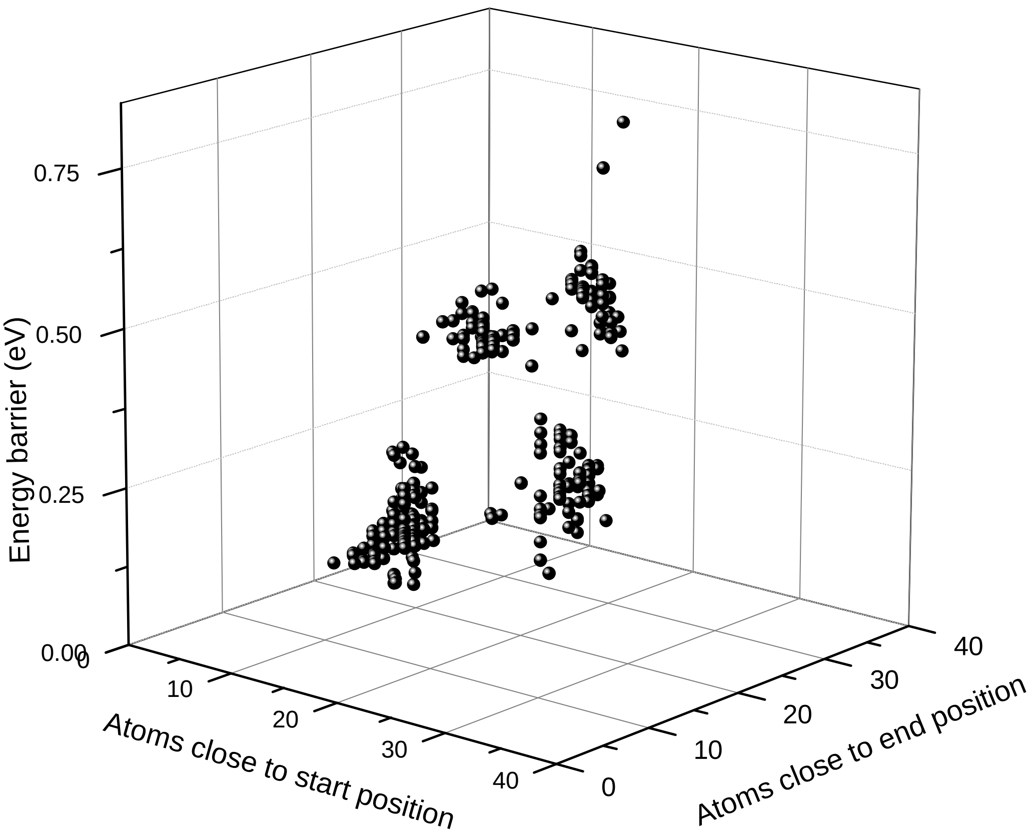

Figure 3.6: *The energy barriers clusters into four groups depending on how many atoms are closer than twice the nearest neighbour distance to the start and end positions. The four clusters are characterised in* Table 3.7.

The influence of the number of atoms close to the start and end positions on the energy barrier is shown in *Figure 3.6*. The four clusters seen in *Figure 3.6* are characterised in Table 3.7. It is important to notice that there is quite a large energy barrier for atoms that jump down from an island. The energy barrier to move to a position that is more favourable or slightly less favourable is in the range of 0.1 to 0.3 eV. The energy barrier is much larger for a jump to a position that has a binding energy that is much lower. The energy barrier to move around on top of an island is slightly lower than moving around on a clean surface. The energy barrier increases as the size of the island increase.

Table 3.7: Description of the four clusters from *Figure 3.6*.

Description of jump	Number of atoms that are close to start.	Number of atoms that are close to end.	Energy barrier (eV)
Movement on top of island	10-16	12-16	0.1-0.3
Jump down from island	10-16	23-25	0.4-0.48
Jump to more favourable position	21-23	21-25	0.1-0.3
Jump to less favourable position	23-25	22-24	0.45-0.6

3.4 The uncorrelated contribution model for predicting energy barriers

A model to predict the energy barriers for the jumps of atoms on a surface is proposed in this section. This model would be useful in Monte Carlo simulations. For the model it is assumed that:

1. Each atom has a fixed influence on the barrier irrespective of that atom's environment.
2. The influence of an atom will decrease proportional to 1/r.
3. The influence of an atom is in the direction of the jump.
4. The barrier is affected by the environment at the start and the end of the jump.
5. The contributions from the start and end positions have a similar form.
6. All the atoms are in lattice positions.

Figure 3.7 shows how the nearest neighbour surface atoms influence the energy barrier for the jump by hindering or helping. *Figure 3.9* shows how the position of an atom influences the atom's contribution the energy barrier. The nearest neighbours to the starting position have a much greater influence on the barrier than the next nearest neighbours. Atoms further than the next nearest neighbours have almost no contribution to the barrier energy. There can be no atoms closer than the nearest neighbour distance because of assumption 6.

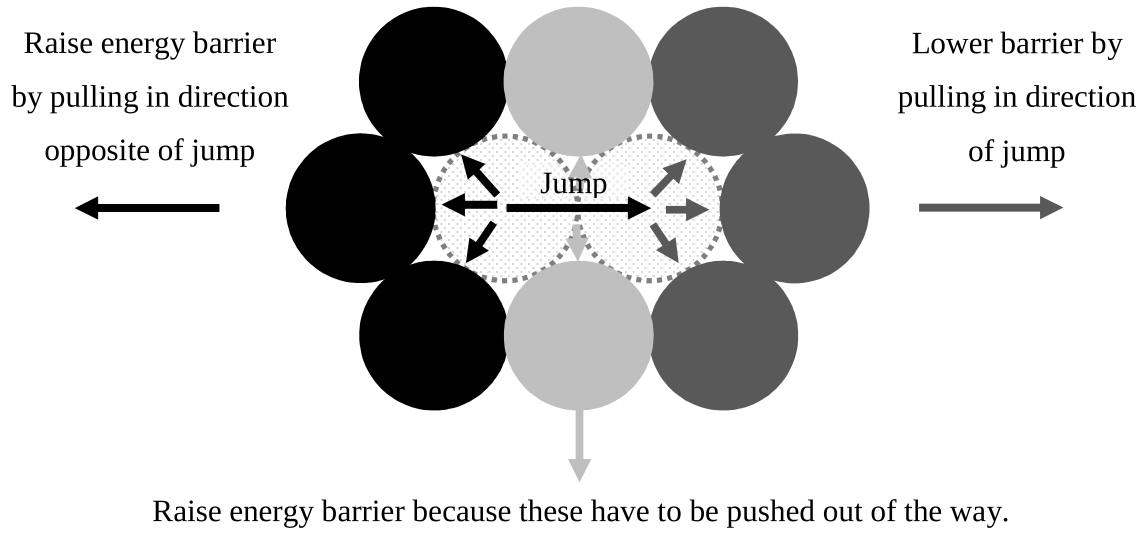
Raise energy barrier by pulling in direction opposite of jump
Jump
Lower barrier by pulling in direction of jump
Raise energy barrier because these have to be pushed out of the way.

Figure 3.7: *Influences of surface atoms in jump.*

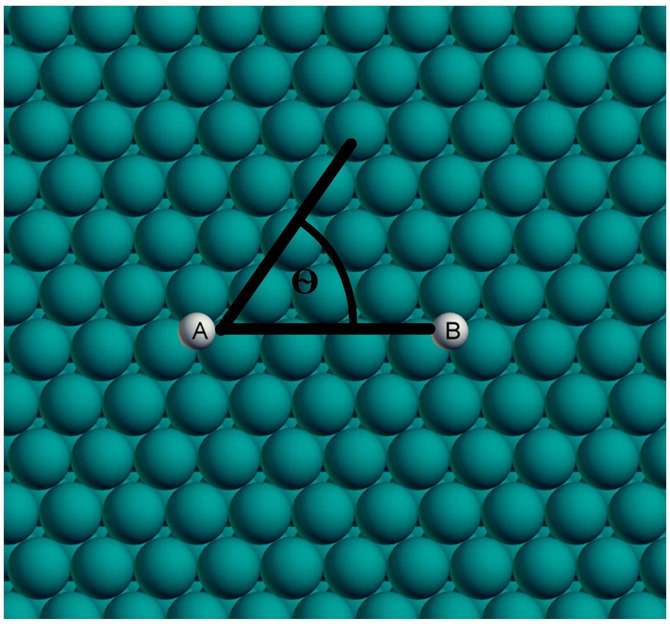

Figure 3.8: *When an atom moves, the angle is taken as the deviation from the line that connects the start, A, and end, B, positions.*

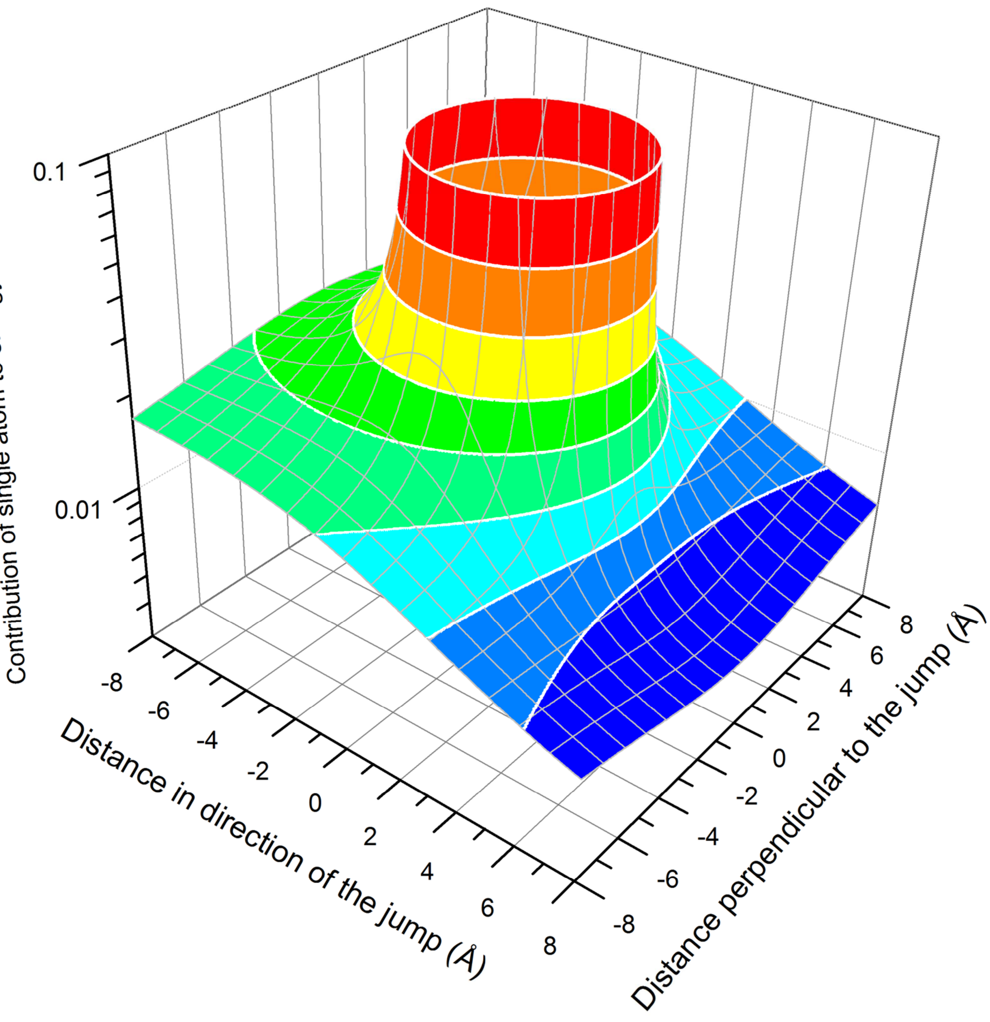

Figure 3.9: *The effect of relative position on the size of the contribution to the energy barrier can be seen here. The start position is at (0, 0) and the jump is towards the right. The contour lines indicate what positions have the same energy contribution towards the barrier. The positions closer to the start position have a greater energy contribution.*

From assumptions 4 the energy barrier, E_B, is

$$E_B = p_0 + E_{start} + E_{end} \tag{3.1}$$

where p_0 is a fitting parameter, E_{start} is the contribution from the start position and E_{end} is the contribution from the end position.

From assumption 1 the contributions for the atoms can be added together in order to obtain the energy barrier, therefore the energy barrier for position *pos*, E_{pos}, is given by

$$E_{pos} = p_1 \sum_{i}^{r_{pos,i} < r_{rc}} C_{pos,i} \tag{3.2}$$

where p_1 is a fitting parameter, r_{rc} is the cut-off radius beyond which the contributions of the atoms become negligible and are discarded, *pos* is either *start* or *end* in order to indicate the position and $C_{pos,i}$ is the contribution of atom *i* to position *pos*.

From assumptions 2 and 3 the contribution, $C_{pos,i}$, of atom *i* to position *pos* is

$$C_{pos,i} = \left(\frac{1}{r_{pos,i} + p_1} + p_2 \right) \left(\cos\theta_{pos,i} + p_3 \right) \tag{3.3}$$

where $r_{pos,i}$ is the distance from atom *i* to position *pos*, $\theta_{pos,i}$ is the angle between atom *i* and the line that connects the start and end positions as shown in *Figure 3.8*, and the parameters p_1, p_2, and p_3 are used to fit the model to the results from the simulations.

From assumption 5 the contributions from the start and end positions form the same equation but with different fitting parameters. Combining equations (3.1), (3.3), and (3.2) and providing different fitting parameters to the start and end positions, the energy barrier is given by

$$\begin{aligned} E_B = p_0 + p_1 \sum_{i}^{r_{start,i} < r_{rc}} \left(\frac{1}{r_{start,i} + p_2} + p_3 \right) \left(\cos\theta_{start,i} + p_4 \right) \\ + p_5 \sum_{i}^{r_{end,i} < r_{rc}} \left(\frac{1}{r_{end,i} + p_6} + p_7 \right) \left(\cos\theta_{end,i} + p_8 \right) \end{aligned} \tag{3.4}$$

where p_0 to p_8 are fitting parameters.

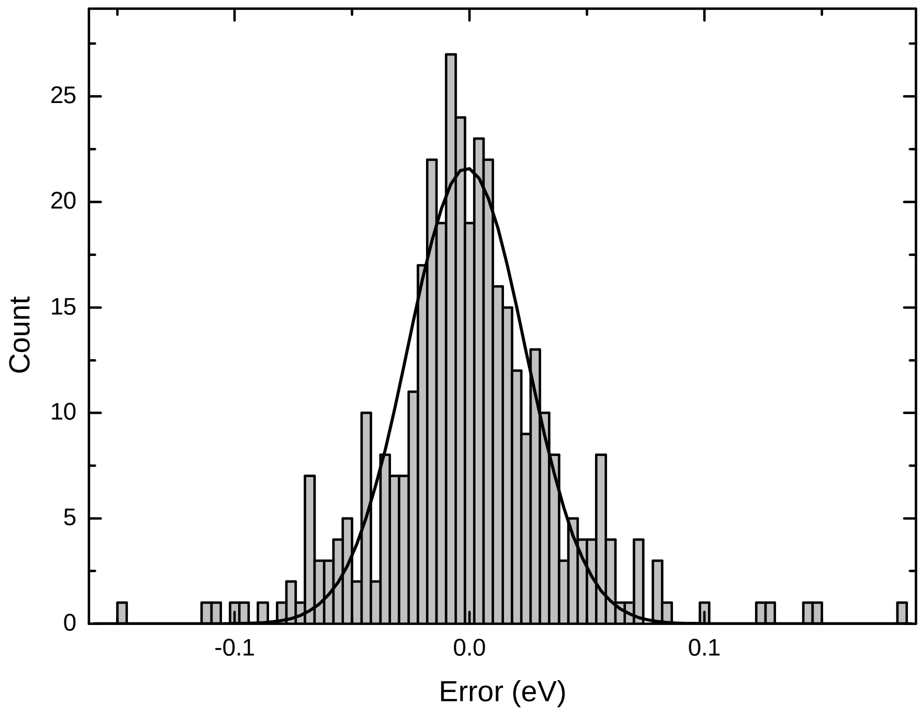

Figure 3.10: *Histogram with Gaussian fit for the errors in the prediction of the model. The Gaussian has a mean of 1.3 x 10^{-3} eV and a standard deviation of 2.5 x 10^{-2} eV.*

The model was fitted to the barriers from all the jumps in *Figure 3.1* and *Figure 3.2* and a histogram of the errors, which is the difference between the barrier energies from the calculations and the model, was created. A Gaussian curve was fitted to the histogram in order to get the standard deviation of the model. This can be seen in *Figure 3.10*. Using the standard deviation of the Gaussian fit, the quality of the model could be measured. It was found that using the nearest and next-nearest neighbours, which translate to a cut-off radius of 5.7 Å, provided good results. The standard deviation of the Gaussian in *Figure 3.10* is 0.025 eV, which means that the difference between the barrier energy predicted by the model and the barrier energy as calculated using molecular dynamics will differ by less than 0.025 eV in approximately 68% of the cases and differ by less than 0.05 eV in approximately 95% of the cases. The fitting parameters that were found are given in Table 3.8. For further comparison the predicted and calculated barriers are given in Table 3.3, Table 3.4, Table 3.5, and Table 3.6 and the tables in Appendix B.

Table 3.8: Values of the fitting parameters for the model for Pt.

Fitting parameter	Value
p_0	-7.920926 x 10^{-1}
p_1	-3.727818 x 10^{-5}
p_2	-2.767271 x 10^{0}
p_3	-1.256022 x 10^{1}
p_4	-1.727271 x 10^{1}
p_5	9.408896 x 10^{-3}
p_6	-3.310704 x 10^{0}
p_7	6.439961 x 10^{-1}
p_8	1.616139 x 10^{0}

3.5 Discussion

The lowest energy shapes was found for surface structures with less than 11 atoms. It was found that rounded shapes have lower energy. Atoms that are bound to only one or two atoms will easily move to positions of lower energy. Atoms that are in a position of low energy will tend to stay there because of the high energy barrier for leaving a low energy site.

The energy barriers for atom movement next to the low energy structures were calculated. It was found that the energy barriers could be roughly classified into four groups based on the number of atoms near the start and end positions of the jump. A simplified model was created in order to predict the energy barrier based on the environment. The proposed model for predicting energy barriers has an error of 0.025 eV. Depending on the simulation parameters the model can be more than six orders of magnitude faster than using a simulation to find the energy barrier.

Chapter 4: Island growth on Pt(111)

In this chapter the growth of islands with hundreds of atoms will be discussed. The process of physical vapour deposition (PVD) will be used as the basis for the investigation. The influence of temperature and evaporation rate will be investigated.

4.1 PVD and model used to represent PVD

PVD is used to create thin layers of a material on a substrate. A schematic illustrating how PVD functions is shown in *Figure 4.1a*. The material to be deposited is evaporated in a vacuum in order to prevent contamination and allow the evaporated material to reach the substrate. The substrate is in the plume of evaporated material. The material sticks to the substrate and layers form.

The model used in the simulations is shown in *Figure 4.1b*. In the model a surface of four layers of atoms were used. The atoms in the bottom layer were held stationary. Atoms were projected towards the surface from random locations. The interval between the addition of the atoms determines the evaporation rate. All the projected atoms had energies of 0.13 eV, corresponding to a temperature of 1000 K. The temperature of all the atoms was controlled by using the Berendsen thermostat.

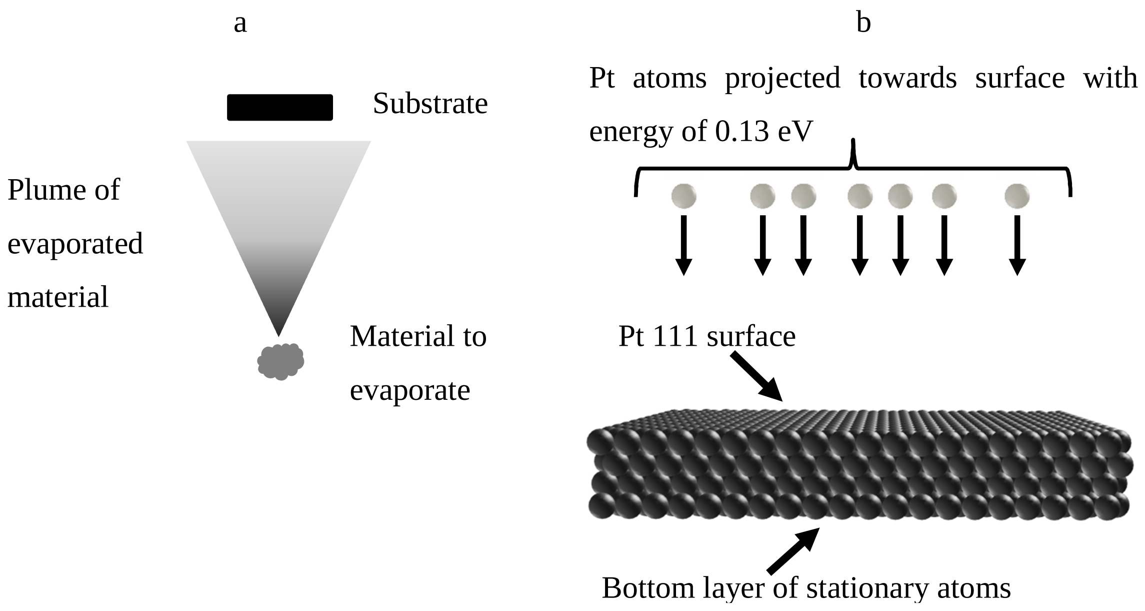

Figure 4.1: *A simplified illustration of PVD is given on the left. The model used to represent PVD is given on the right.*

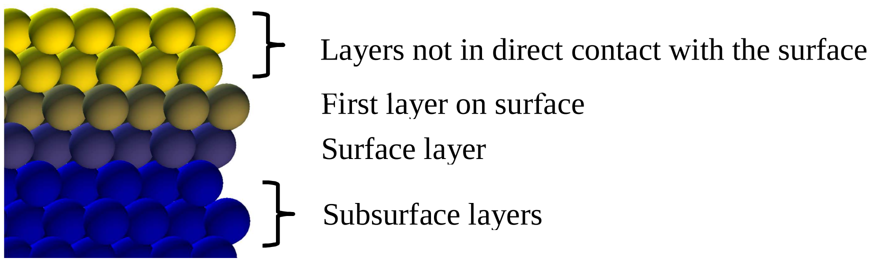

Figure 4.2: *Colour codes used in the visualization of island growth.*

4.2 Effect of temperature on island growth

The first simulations were to determine the effect of substrate temperature on island growth. Simulations using the model for PVD were done at various temperatures. Results from the simulation at 300 K, 700 K, and 1100 K are shown in *Figure 4.3*. The parameters used in the simulation are shown in Table 4.1. The colour of the atoms in *Figure 4.3* is determined by height. *Figure 4.2* shows the colours used for the different layers.

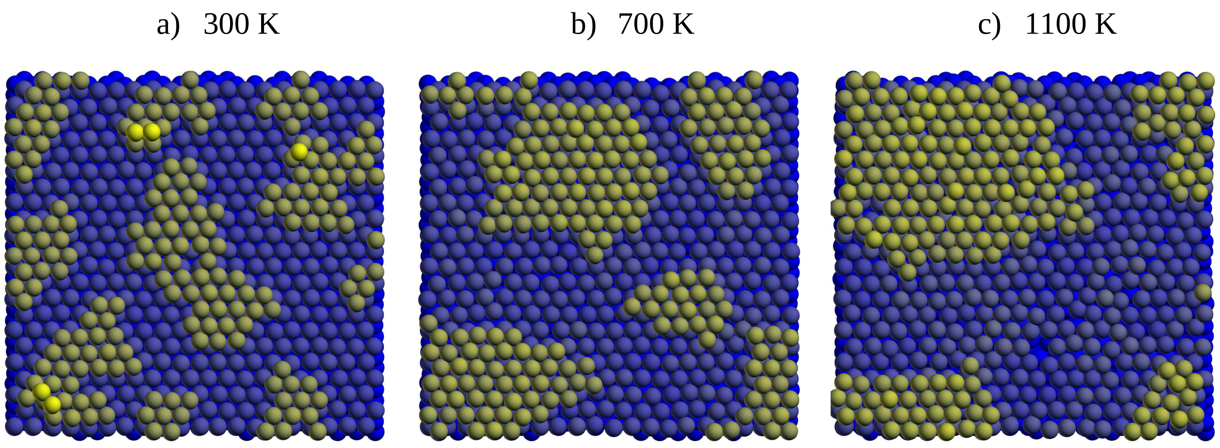

Figure 4.3: *Simulation state at 80 % surface coverage for three selected temperatures.*

Table 4.1: Parameters used for simulation.

variable	value
Timestep (dt)	0.01 ps
Length of simulation	1,000,000 steps (10 ns)
Time between addition of atoms	5000 steps (50 ps)
Tau for thermostat	0.1
Cut off radius	7 Å
Cell size	8 Å
Simulation size in x direction	55.4 Å
Simulation size in y direction	52.78 Å
Number of atom layers	4
Number of atoms in base	1760

With an increase in temperature, the energy available to atoms increases. With more energy, the atoms have a higher probability of jumping across an energy barrier. Thus a higher temperature leads to increased mobility of atoms. As atoms cluster and the size of the island increases, the islands become less mobile. Some of the reasons for the decrease in mobility of larger clusters are that the atoms are bound more tightly in larger islands and that more atoms have to move in order for the larger island to move. The effect of the decreased mobility of clusters is that an atom moves around on the surface until the atom can cluster together with other atoms.

The number of islands that form are related to the number of jumps that an atom can perform between the addition of subsequent atoms. The likelihood that an atom reaches an existing structure will increase as the number of jumps that the atom performs increases. If a new atom is added to the surface before the previous atom found a structure, the two atoms that have not found a structure can cluster in order to form a new structure. The likelihood that new structures form is related to the number of jumps that occur between the addition of atoms. As the temperature increase the frequency of jumps increase and larger structures form.

There are a few islands in the simulation at 300 K as seen in *Figure 4.3a*. These islands are not in the lowest energy state because the atoms didn't have enough mobility to find the low energy positions. The islands from the simulation at 700 K as seen in *Figure 4.3b* are closer to the lowest energy state because of the increased mobility of the atoms. During the simulation at 700 K some of the low energy structures from the previous chapter are seen to have formed. In the simulation at 1100 K, as seen in *Figure 4.3c*, only one island forms. The single island in *Figure 4.3c* appears as multiple islands because of the periodic boundary conditions. Only one island forms because the atoms are mobile enough to form a cluster before more atoms are added onto the surface.

For each of the temperatures from 100 K to 1200 K in steps of 100 K, a simulation was done. These simulations are shown in *Figure 4.4*. In order to show the evolution of the islands, snapshots of the simulations are given. At low temperatures the atoms are relatively stationary because the atoms are effectively frozen in place. When the atoms are deposited the atoms have some energy, but the energy from the heat quickly dissipates into the neighbouring atoms and are removed. This causes the atoms to stay very close to the point of impact and as a result small structures that are faceted is formed. As the temperatures get higher, the atoms become more mobile and find a more energetically favourable position. The structures become larger and smoother. When the temperature gets very high, the edges get less smooth. At these temperatures the atoms have enough energy and move away from the structure edge. Almost all the atoms are

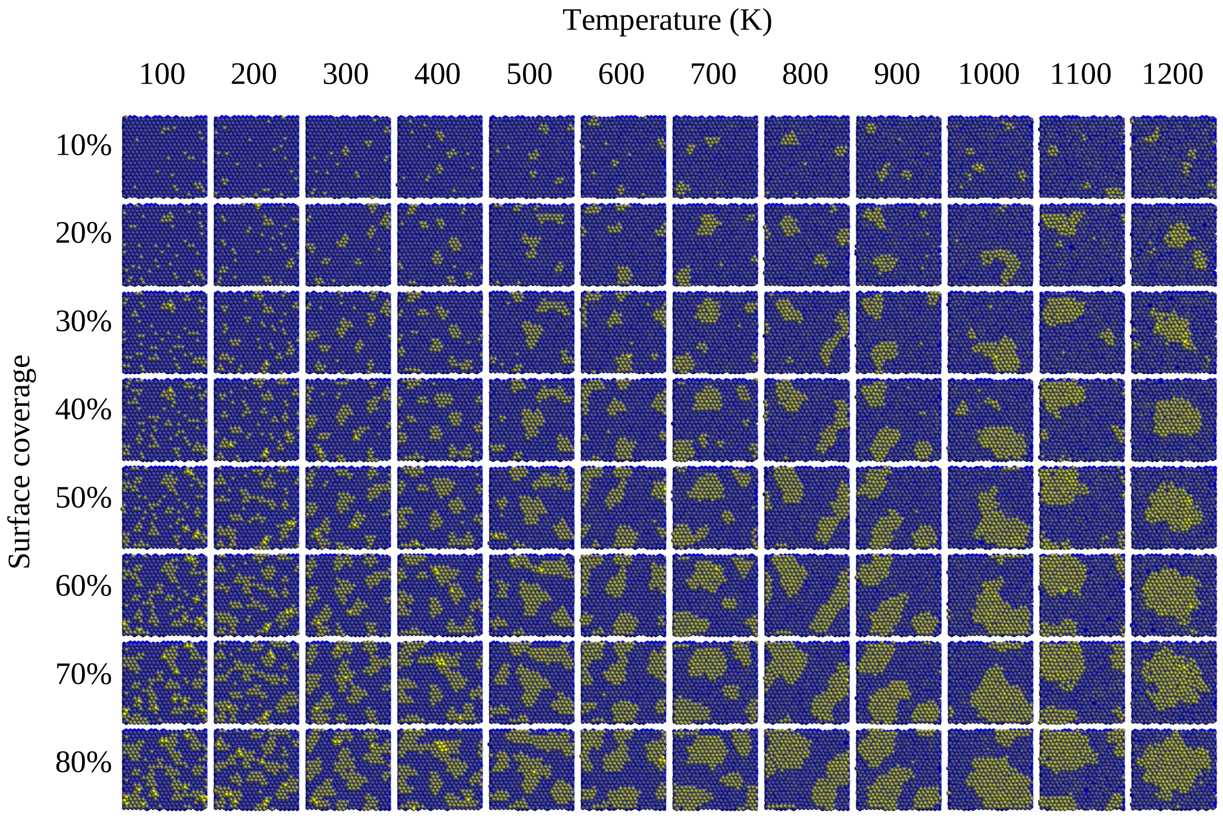

Figure 4.4: *Growth of islands as a function of temperature. Each column shows how the structures develop at that temperature. The rows show how the structures are different when the temperatures are different. The colours of the atoms are chosen according to their height. The layer below the surface layer is dark blue. The surface layer is light blue. The atoms directly on the surface are yellow. The atoms in the second layer above the surface are bright yellow.*

directly on the surface. At the low temperatures the islands are so small that not many atoms land on the islands. At higher temperatures the atoms have enough energy to jump down from the island.

4.3 Effect of evaporation rate on island growth

The evaporation rate can be varied by changing the number of steps between the addition of atoms. By increasing the time between the addition of atoms the evaporation rate is lowered. Halving the evaporation rate requires running the simulation for twice the amount of time in order to retain the same percentage surface coverage. The parameters used for the simulation at different evaporation rates are shown in Table 4.2.

Figure 4.5 shows the structures that form at three different evaporation rates. In *Figure 4.5* it can be seen that larger islands form with a lower evaporation rate. With lower evaporation rates an atom can perform more jumps before the next atom is deposited. The result of more jumps before the addition of extra atoms is that less structures form, and that those structures are larger. The increase in structure size due to lower evaporation rates is similar to the increase in structure size due to higher temperatures.

For *Figure 4.6* both the evaporation rate and the temperature were varied. This was done in order to show how different values for evaporation rate and temperature can produce similar results. In order to get results over the greatest range, the evaporation rate was halved for each simulation. In all the simulations the same number of atoms was deposited. The number of atoms deposited is enough to cover 50% of the surface. The evaporation rate in the first column corresponds to an evaporation rate of almost 10^8 layers/s.

In *Figure 4.6* it can be seen the structures that form at 0 K are very similar because the atoms are effectively frozen in place once they are on the surface. The structures are not exactly the same because the atoms that are deposited on the surface are deposited on random locations.

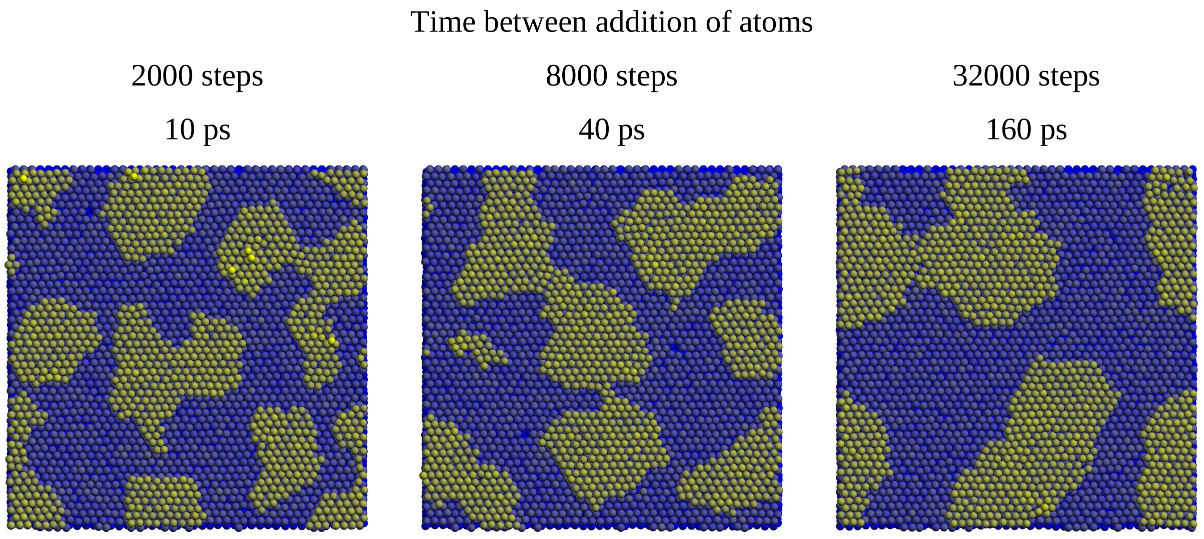

Figure 4.5: *Structures that form at different evaporation rates at 900 K.*

Table 4.2: Parameters used for simulation.

variable	value
Timestep (dt)	0.005 ps
Tau for thermostat	0.1
Mass of Pt	195.084 amu
Cut off radius	7 Å
Cell size	8 Å
Simulation size in x direction	119.11 Å
Simulation size in y direction	119.94 Å
Number of atom layers	4
Number of atoms in base	8600

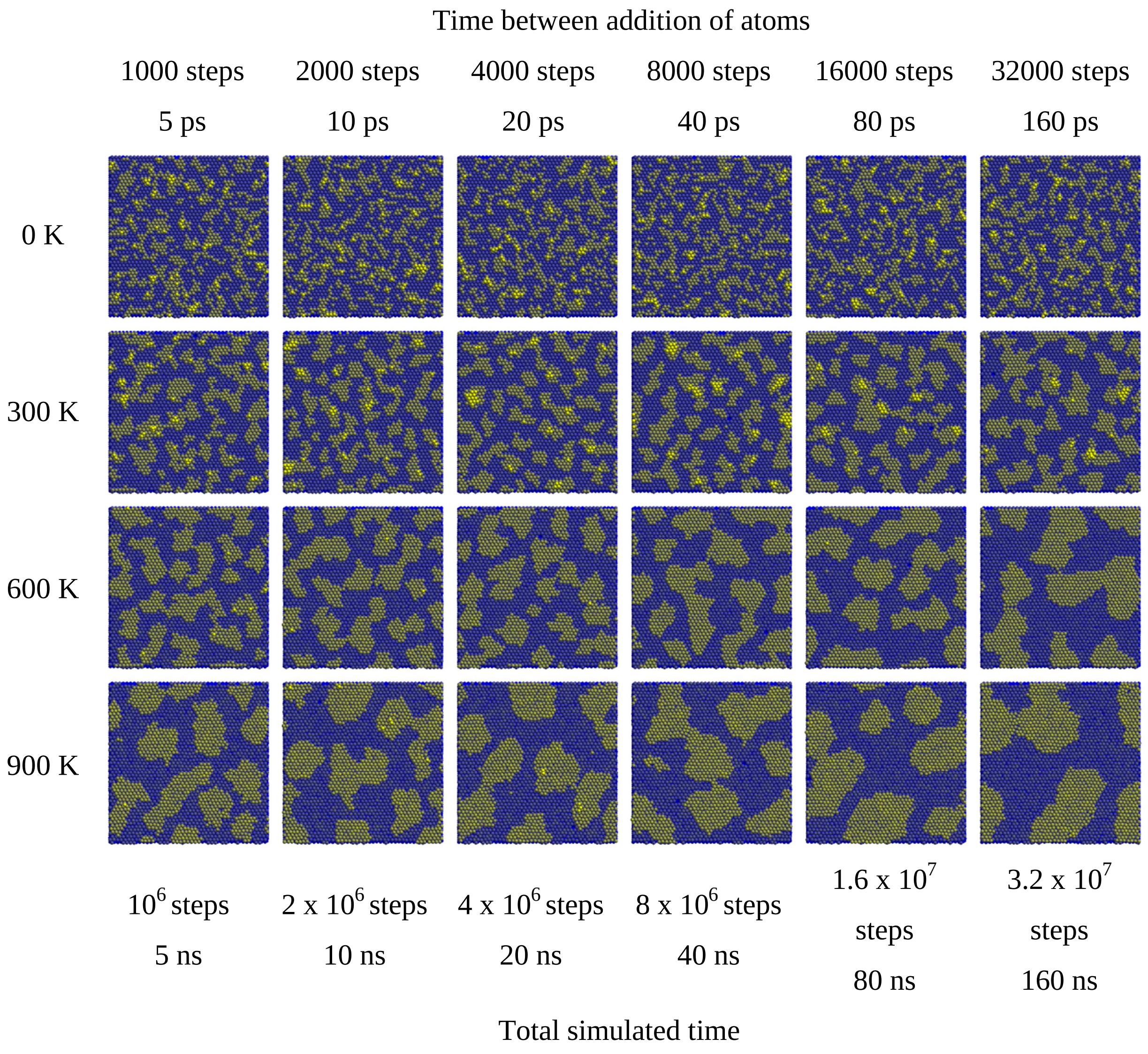

Figure 4.6: *The effect of evaporation rate and temperature on the formation of structures. Each row was done at the same temperature. Each column was done with the same number of steps between the addition of subsequent atoms, and thus the same evaporation rate.*

At the other temperatures a slower evaporation rate leads to larger structures. Larger structures can form because the combination of higher temperature and lower evaporation rate enables more jumps to occur. From *Figure 4.6* it can be seen that an increase in temperature combined with a decrease in evaporation rate is much more effective in the creation of large structures than a change in only one of them.

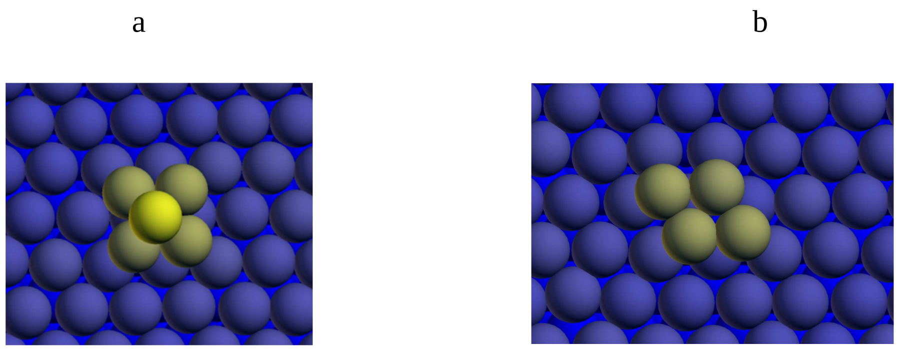

Figure 4.7: *Square pyramid found in the simulation. A four atom cluster that matches the substrate is show on the right.*

An interesting feature that was observed was square pyramids as seen in *Figure 4.7a*. It was very interesting because the square doesn’t match the triangular surface. The pyramid lasted a bit longer than 0.8 ns at 300 K. During that time the pyramid performed more jumps than the structures that matched the surface. The relationship between surface mismatch and mobility will be discussed in greater detail in the next chapter.

4.4 Conclusion

It was found that an increase in the temperature and a decrease in the evaporation rate have similar effects because both lead to an increase in the number of jumps that an atom can perform before another atom was added. As the atoms perform more jumps, the structures that form become larger. Low energy structures discussed in the previous chapter formed during the course of the simulations.

Chapter 5: Deposition on graphite

The change mobility of the square pyramid found in the previous chapter suggested that a mismatch with the substrate can influence the mobility of the deposited atoms. Different surfaces will be required in order to test this effect. The Steele potential provides a model for a surface that will be well suited for such a test.

5.1 Steele potential

In the simulations in chapter 4 the surfaces were represented by four atom layers. Less than one atom layer was deposited onto the surfaces. The simulations will be much faster if an analytical method to represent the surface was used instead. The Steele potential is an analytical expression that provides the potential between an atom and a regular surface. An additional advantage of the Steele potential is that the surface parameters can be varied easily.

The Steele potential in reduced units, V^*_{Steele}, is given by [**30**]

$$V^*_{\text{Steele}} = E_0(z^*) + \sum_{n>0} E_n(z^*) f_n(s_1, s_2) \tag{5.1}$$

where

$$E_0(z^*) = \frac{2\pi q A^6}{a_s^*} \sum_{p=0}^{\infty} \left(\frac{2A^6}{5(z^* + p\Delta z^*)^{10}} - \frac{1}{(z^* + p\Delta z^*)^4} \right) \tag{5.2}$$

and

Table 5.1: First five values for g_s^* and f_n as given in [**30**].

n	$g_n^*/2\pi$	$f_n(s_1,s_2)/2$
1	$2/\sqrt{3}$	$-\left[\cos(2\pi s_1)+\cos(2\pi s_2)+\cos(2\pi(s_1+s_2))\right]$
2	2	$2\left[\cos(2\pi(s_1+2s_2))+\cos(2\pi(2s_1+s_2))+\cos(2\pi(s_1-s_2))\right]$
3	$4/\sqrt{3}$	$-\left[\cos(4\pi s_1)+\cos(4\pi s_2)+\cos(4\pi(s_1+s_2))\right]$
4	$2\sqrt{7/3}$	$-\left[\begin{matrix}\cos(2\pi(3s_1+s_2))+\cos(2\pi(s_1+3s_2))+\cos(2\pi(3s_1+2s_2))+\\ \cos(2\pi(2s_1+3s_2))+\cos(2\pi(s_1-2s_2))+\cos(2\pi(2s_1-s_2))\end{matrix}\right]$
5	$6/\sqrt{3}$	$2\left[\cos(6\pi s_1)+\cos(6\pi s_2)+\cos(6\pi(s_1+s_2))\right]$

Table 5.2: Parameters for graphite using the Steele potential.

Variable	Formula and/or value	Description
q	2	Number of atoms in unit cell
a_1	2.46 Å	Length of one side of the unit cell
a_s*	$a_s / a_{12} = 3^{1/2}/2$	Reduced surface area of the unit cell
$z*$	z / a_1	Reduced distance of atom from the surface
$\Delta z*$	1.38 Å	Reduced distance between graphite layers
ε_{gs}	Depends on material	Parameter giving energy scale of gas-solid interaction
σ_{gs}	Depends on material	Parameter giving distance scale of gas-solid interaction
$V*_{\text{Steele}}$	$V_{\text{Steele}} / \varepsilon_{gs}$	Steele potential in reduced units
A	σ_{gs} / a_1	Ratio to scale potential to correct distance

$$E_n(z^*)=\frac{2\pi A^6}{a_s^*}\left[\frac{A^6}{30}\left(\frac{g_n^*}{2z^*}\right)^5 K_5(g_n^* z_*)-2\left(\frac{g_n^*}{2z^*}\right)^2 K_2(g_n^* z_*)\right]. \tag{5.3}$$

The values of g_s* and f_n for graphite with n lest than six are given in Table 5.1. A description and the values of the parameters in equations (5.1), (5.2) and (5.3) are given in Table 5.2. The values for the interaction parameters between graphite and platinum are given in Table 5.3.

Table 5.3: Interaction parameters for graphite and platinum [**12**] [**31**] [**32**].

Parameter	Value
$\varepsilon_{\mathrm{pc}}$	256 K = 256 x 8.617 x 10^{-5} eV/K
σ_{pc}	2.905 Å

The values for s_1 and s_2 in equation (5.1) are found from the projection of the (x, y) coordinate onto the graphite crystal structure so that the position of the atom on the surface, $\boldsymbol{\tau}$, is given by the formula $\boldsymbol{\tau} = s_1\boldsymbol{a}_1 + s_2\boldsymbol{a}_1$.

The E_0 term in equation (5.1) incorporates the uniform contribution from all the layers. Since the effect of each subsequent layer gets less, only the first few layers have to be used. Only the first 25 layers were used as in [**31**]. The E_n term in equation (5.1) gives the amplitude and the f_n term gives the shape of the *n*th term. Only the first 5 terms are used because the effect of the others terms are negligible.

To further speed up the process, simple functions were used in the place of E_0 and E_n. It was found that a sufficiently accurate fit was produce by fitting functions with the form $az + b(z + c)^d$ to E_n while a function with the form $az + b(z + c)^d + e(z + f)^g$ was required for E_0.

Using the Steele potential with simplified fits for the E terms results in much faster simulation times. For similarly sized systems and 8 million steps using the Sutton-Chen potential for the base takes 47 days, while using the Steele potential takes only 16 hours, which is a speedup of about 70 times. The speedup is similar for other simulations.

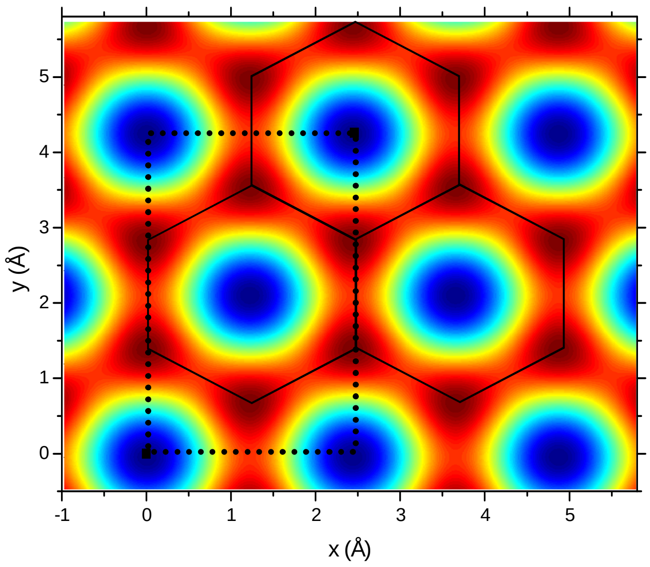

Figure 5.1: *The Steele potential as calculated for graphite is shown here. The colour indicates the potential. An atom would be bound more tightly in the blue locations. The solid lines highlight the hexagonal structure of graphite. The dotted rectangle shows the minimum cell that must be repeated for smooth boundary conditions.*

5.2 Simulation Setup

The Steele potential was used to simulate a graphite surface and the Sutton-Chen potential was used to simulate the platinum atoms. The Steele potential of the graphite surface is shown in *Figure 5.1*. Atoms are projected along the z axis towards the surface. The surface is along the x and y axis. Periodic boundary conditions was used in the x and y directions. The sizes were chosen such that the cells from the graphite surface fits perfectly into the simulated area. The *x* direction had to be a multiple of 2.46 Å while the *y* direction had to be a multiple of 4.26 Å. An elastic wall was used in the *z* direction so that atoms that bounce of the surface can have another chance to stick to the surface. The Berendsen thermostat was used to control the temperature of

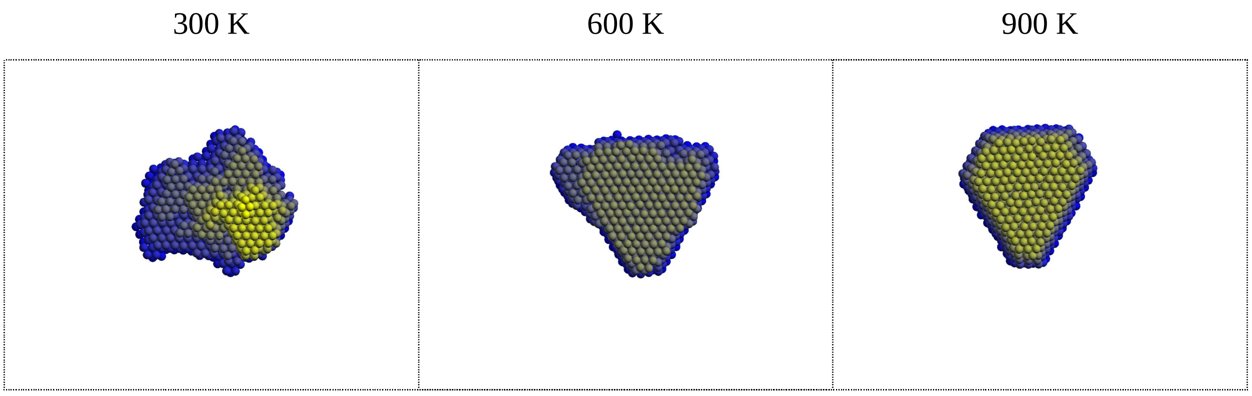

Figure 5.2: *Structures that form at different temperatures when there are 64000 steps (320 ps) between the addition of atoms.*

all the atoms in the system. The atoms were projected towards the surface from uniformly distributed random locations with a velocity of 3.6 Å/ps, which would correspond to a temperature of about 1000 K.

5.3 Investigating evaporation rate

The evaporation is done using the same model as described in chapter 4, with the modification that the surface is represented by the Steele potential instead of layers of atoms. Atoms are deposited onto the surface with certain intervals between addition of subsequent atoms. By varying the time interval, the evaporation rate can be changed. The simulation was done for various temperatures and evaporation rates.

Figure 5.2 shows the structures that form at three different temperatures when there are 64000 steps between the addition of atoms. There is enough time for atoms and small islands to come together before becoming immobile, thus only a single structure formed at this evaporation rate. When an atom reaches a bigger island, the atom will most likely be in position that is not the lowest possible energy. At lower temperatures the atoms do not have enough energy to easily move out of this local minimum. As the temperature gets higher it becomes more likely that an atom will find a position that minimizes the energy of the structure. As the temperature gets

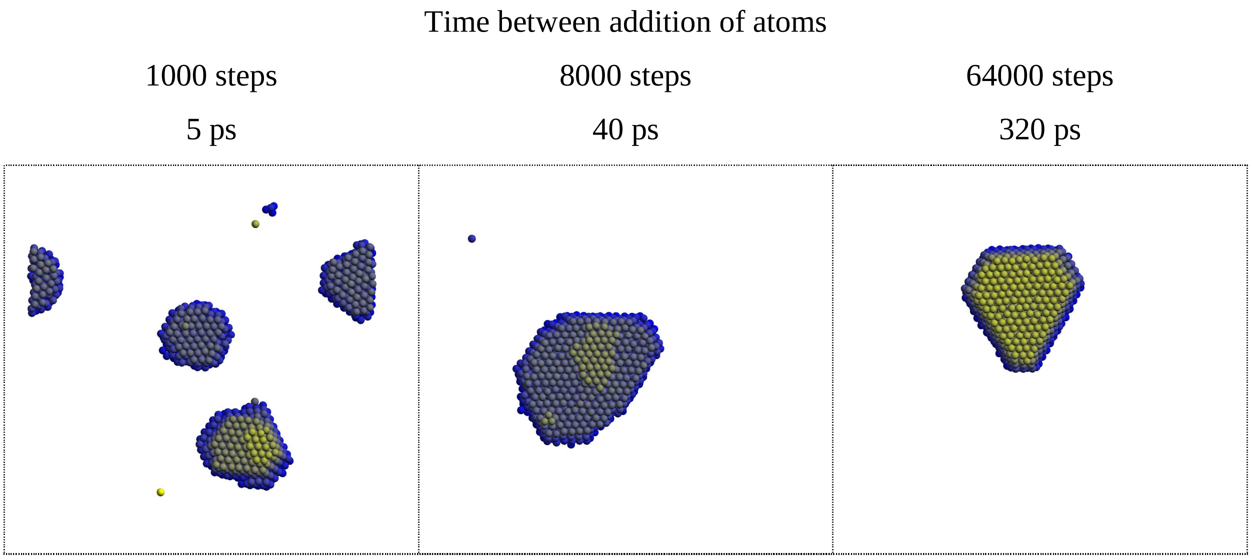

Figure 5.3: *Structures that form at different evaporation rates at 900 K.*

higher, the sides of the structures become more flat. The number of nearest neighbour bonds is larger in the structures that formed at higher temperatures. The truncated triangle seen in *Figure 5.2* is close to the structure of minimum energy for platinum on platinum(111) surface [**33**].

Figure 5.3 shows how the structures change when the evaporation rate is varied and the temperature is kept constant. At the fastest evaporation rate the atoms do not even have enough time to reach a single structure so more than one structure form. With 8000 steps between the addition of atoms, only a single structure forms, but that structure is not yet in the lowest energy state. With 64000 steps between the addition of atoms, the structure is much closer to the lowest energy state.

The results from more variations of temperature and evaporation rate are shown in *Figure 5.4*. At the fastest evaporation rates there were a few small structures. As the evaporation rate slowed the number of structures decreased and the structures became larger until there was only one particle. The combination of structures requires a slower evaporation rate at lower temperatures.

A single atom on the graphite surface is very mobile. When the atom bonds to another atom, the cluster that forms becomes less mobile. As more atoms are joined to the cluster, the cluster tends to become more stationary. When atoms are added to the cluster sooner, the atoms have less time to move to equilibrium positions. This explains why at very high evaporation rates there are smaller clusters while at low evaporation rates there is one large particle. When the temperature is higher the atoms move faster so the atoms can form larger structures at higher evaporation rates.

When the platinum atoms are in a large cluster, the individual atoms are bound in place so the cluster cannot change much to find the equilibrium structure. When the temperature is increased the atoms can move around more so it is more likely that the equilibrium structure will be found. In *Figure 5.4* it can be seen that the structures tend towards the equilibrium structure when the evaporation rate is slower and the temperature is higher. The equilibrium structure that forms is a truncated triangle.

The attraction between the platinum is about an order of magnitude stronger than the attraction between the platinum and the carbon. The effect of this is that it will be energetically more favourable for the platinum to bond together than to just spread out on the graphite. This can be seen in that the platinum does not form a single layer on the graphite but rather a layered structure. The structure tends to grow layer by layer. The layers tend to cover the layers below them completely, except at low temperatures. At low temperatures the energy of the atoms are too low for the atoms to easily jump down to the layer below.

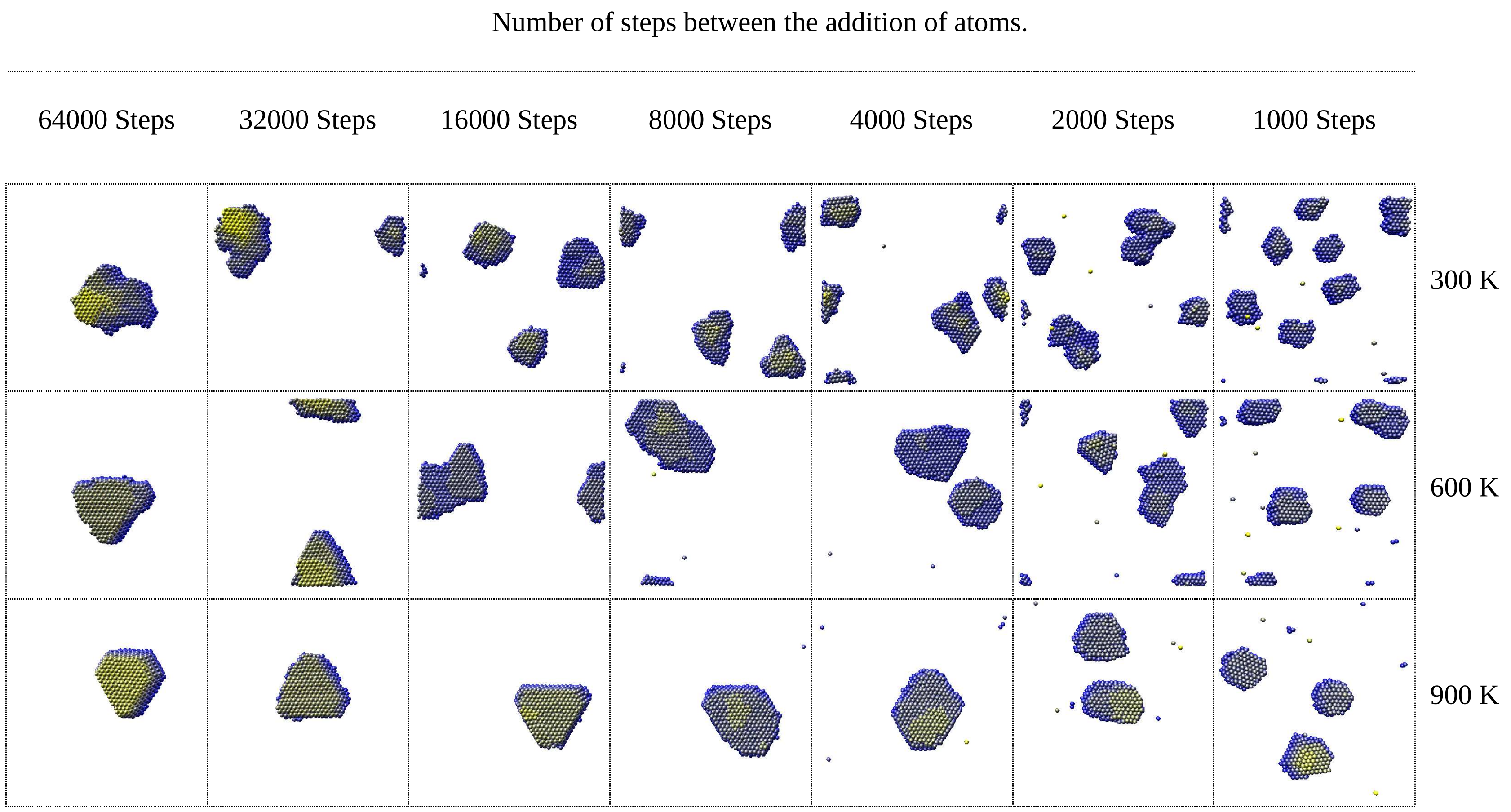

***Figure 5.4**: The structures that form at different temperatures and evaporation rates are shown. Each column contains the results from simulations at the given temperature. Each row contains the results from simulations at the given evaporation rate. Note that the steps between atom projections in subsequent rows doubles, which means that the evaporation rate halves for each row down. The first row corresponds to about 3 x 10^8 Å/s or 1.5 x 10^8 monolayers/s. In each of the simulations the same number of atoms was used.*

5.4 Effect of substrate parameters on structure growth

In this section the influence of the mismatch between the substrate and the material deposited on the surface and the strength of the interaction between the substrate and the material deposited is investigated. In order to investigate the effect of mismatch, the lattice parameter, *a*, of the Steele potential was varied. In order to vary the interaction between Pt and the substrate, ε_{gs} is taken as $\varepsilon_{\mathrm{pc}}$ multiplied by a scaling factor. The scaling factor scales the whole potential. Since the whole potential is scaled, a scaling factor greater than one will increase both the binding energy for the surface and the barrier for an atom to move across the surface. A scaling factor greater than one will thus decrease the mobility of an atom on the surface.

Figure 5.5 shows the results for simulations for various values for the substrate lattice parameter and scaling factor for the substrate potential. The simulations were done at a temperature of 900 K with an interval of 64000 steps between the addition of atoms. This corresponds to the bottom right (or left, if the page is in the 'landscape' orientation) simulation in *Figure 5.4*.

The force that the substrate exerts on the Pt atoms increases towards the bottom in *Figure 5.5*. As the attraction and barriers get larger it is expected that the atoms become less mobile. This can be seen by the shape of the structures that form. An unexpected effect is that the mobility of the atoms is affected by the amount of mismatch with the substrate. This effect becomes evident when the structures become larger than a critical size. This is because the neighbouring atoms help to overcome the barriers.

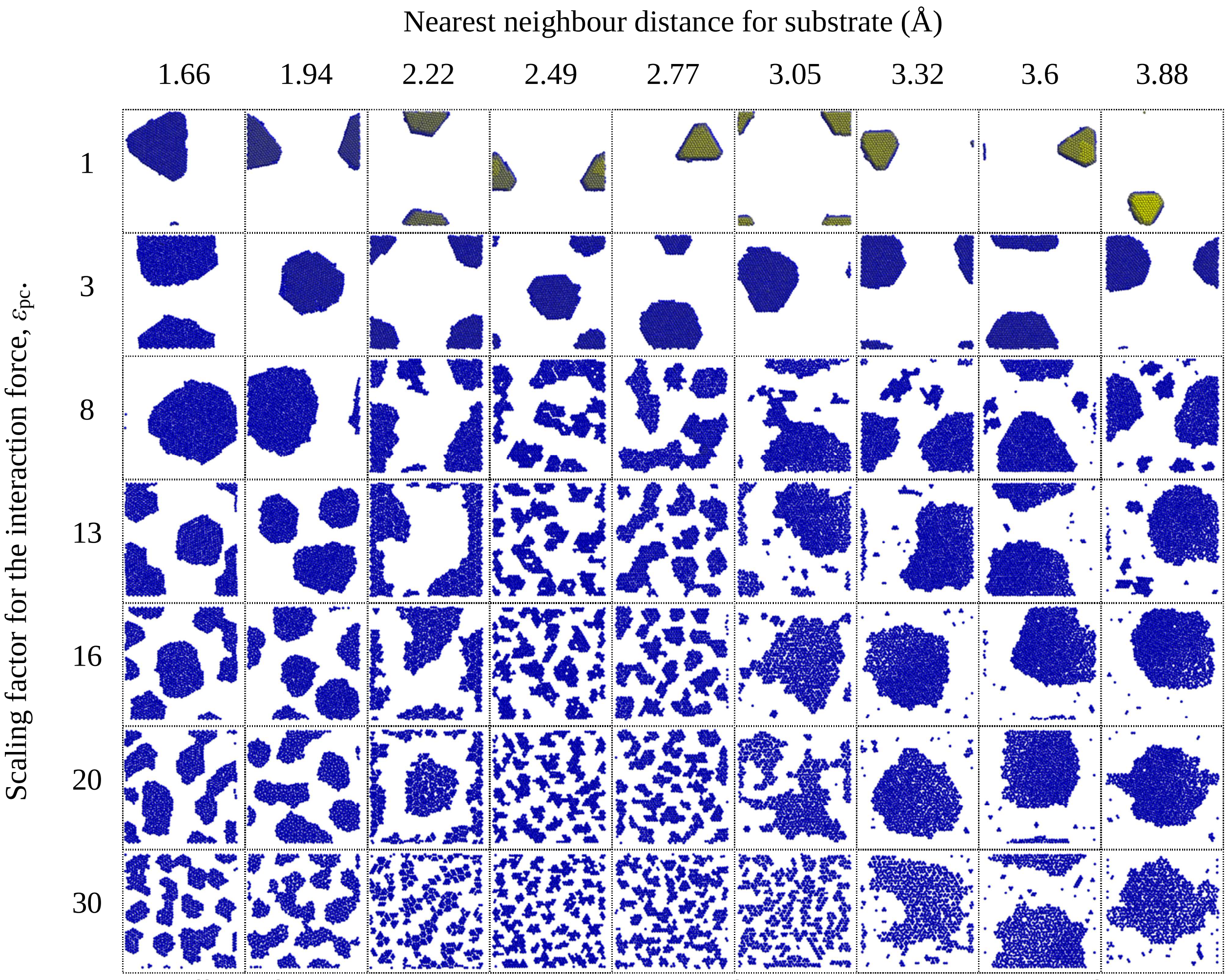

Figure 5.5: *Effect of substrate mismatch and attractive force on the structures that form. The column heading gives the value of lattice parameter, a, that was used. The first row gives the factor by which the potential was scaled.*

To see how neighbouring atoms can increase mobility, consider the illustrations in *Figure 5.6*. In *Figure 5.6a* the nearest neighbour distance on the substrate is smaller than that of Pt. When a Pt atom is in a minimum position, another Pt atom cannot enter the neighbouring minimum position. The atoms will push on each other and get lifted out of the minimum position. The atom will be bonded to the surface with less energy and energy barrier for a jump will be lower. In *Figure 5.6b* the nearest neighbour distance on the substrate is larger than that of Pt. Neighbouring atoms decrease the barrier in much the same ways as in *Figure 5.6a*. The only difference is that the neighbouring atoms attract each other instead of repelling.

a) Nearest neighbour distance of substrate smaller than that of Pt

b) Nearest neighbour distance of substrate larger than that of Pt

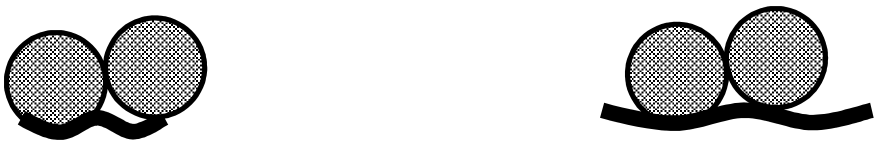

Figure 5.6: *Illustration of how neighbouring atoms can help overcome barriers when there is a size mismatch with the substrate. The heavy line indicates the surface. Atoms want to be on the lowest point on the surface.*

In the centre column with the heading of 2.77 in *Figure 5.5*, the substrate has exactly the same nearest neighbour distance as that of Pt. As the scaling parameter, and thus the barriers, increases, the Pt atoms become less mobile. This has a similar effect to that of a lower temperature. There is a clear resemblance between the low temperature structures in *Figure 4.6* and the structures in *Figure 5.5* where the barriers are high and the substrate mismatch is small.

Since the presence of neighbours increases the mobility, the atoms on the edges of the structures are very mobile. The high mobility of the edges gives the appearance that the structures are molten.

Another very interesting effect was seen in the first row in *Figure 5.5*. In the first row, most of the particles that form have more than one layer. The particles would be stationary and then suddenly start moving across the surface. The particle would move quickly and freely. The particle would then suddenly stop again. The movement of the particle is especially interesting since the Velocity Verlet integrator conserves momentum [**34**], but momentum is not conserved in the particle’s movement. The answer to this conundrum is that the thermostat does not conserve momentum. The thermostat only ensures that the average temperature of the system is as close as possible to the desired temperature. On closer inspection of the particles as they start and stop moving, some interesting observations are made. When the particle is stationary, the atoms that make up the particle vibrate quite a bit because of the high temperature. When the particle moves, the atoms are almost stationary relative to each other. These effects suggest that the influence of the thermostat on the simulations can be the subject of future investigations.

5.5 Conclusion

The Steele potential provided an analytical representation of a surface for physical vapour deposition. Using an analytical expression for the surface provided a significant speedup for the simulations and allowed more parameters to be investigated. When the evaporation was done onto graphite, truncated triangles formed. This corresponds well to known low energy structures of Pt. In order to form the truncated triangle, the evaporation rate had to be low and the temperature had to be high. It was found that a mismatch between the nearest neighbour distance and that of Pt resulted in increased mobility of atoms on the surface. Weak interaction between the surface and Pt caused layered structures to form. Strong interaction resulted in only a single layer forming. Anomalous movement of particles suggests that it might be useful to investigate the effect of the thermostat in the simulations.

Chapter 6: Discussion

The mobility of an atom is determined by the strength of interaction with the substrate, the amount of energy the atom has from the temperature of the system, and the structure in the vicinity. The mismatch affects how the structure influences movement. The evaporation rate will determine how long an atom has to move before it interacts with another atom. Once atoms join together in an island their mobility decreases drastically. With fast evaporation rates the atoms do not have enough time to move to equilibrium positions so the structures that form are small. With slower evaporation rates the structures become larger and closer to an equilibrium structure.

The structure with the highest surface to volume ratio is found in the top right of *Figure 5.5*. The evaporation rate is low, the temperature is high, the nearest neighbour distance in the substrate is larger than that of Pt and interaction strength between Pt and the substrate is the same as between Pt and graphite. When doing PVD the deposition rate will be much lower, thus the temperature would have to be lowered to get a similar structure.

6.1 Future work

The model proposed in chapter 3 will be adapted for the other metals for which Sutton-Chen parameters exist. In addition it will be adapted for other surface orientations and mixtures of those metals.

A Monte Carlo simulation will be implemented using the model proposed in chapter 3. Using Monte Carlo simulations will make it possible to investigate much larger systems than was possible with molecular dynamics.

The Steele potential for other surfaces will be implemented. This will allow the investigation of the effect of surface orientation on the growth of structures.

Different thermostats will be investigated in order to determine whether the Berendsen thermostat had a significant effect on the structures that formed.

Appendix A: Derivation of the force from the Lennard-Jones potential

The Lennard-Jones potential is given by

$$V_{LJ} = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \qquad (7.1)$$

From a given potential, *V*, the force, *F*, can be calculated with the formula

$$\overline{F} = -\nabla V \qquad (7.2)$$

For the derivation of the force it is helpful to first derive the gradient of *r* and 1/*r*. Let the displacement vector be given by $\overline{r} = \hat{x}r_x + \hat{y}r_y + \hat{z}r_z$, where $\hat{x}$ is the unit vector in the *x* direction and r_x is the component of $\overline{r}$ in the *x* direction. The length of the displacement vector is $r = \left\|\overline{r}\right\| = \sqrt{r_x^2 + r_y^2 + r_z^2}$. From the definitions of gradient , it follows that

$$\nabla r = \left(\hat{x}\frac{\partial}{\partial x} + \hat{y}\frac{\partial}{\partial y} + \hat{z}\frac{\partial}{\partial z}\right)r \qquad (7.3)$$

This can also be written as

$$\nabla r = \sum \hat{x}_i \frac{\partial}{\partial x_i} r \qquad (7.4)$$

Where $\hat{x}_i$ is the unit vector in the *i*th direction and $\frac{\partial}{\partial x_i}$ is the derivative in the *i*th direction. From the definition of *r*

$$\nabla r = \sum \hat{x}_i \frac{\partial}{\partial x_i}\left(r_x^2 + r_y^2 + r_z^2\right)^{1/2} \tag{7.5}$$

By using the chain rule

$$\nabla r = \sum \hat{x}_i \frac{1}{2}\left(r_x^2 + r_y^2 + r_z^2\right)^{-1/2} \frac{\partial}{\partial x_i}\left(r_x^2 + r_y^2 + r_z^2\right) \tag{7.6}$$

Now using the definition of *r* and taking the derivative

$$\nabla r = \sum \hat{x}_i \frac{1}{2} r^{-1} 2 r_i \frac{\partial r_i}{\partial x_i} \tag{7.7}$$

Noting that $\frac{\partial r_i}{\partial x_i}$ =1 and simplifying

$$\nabla r = r^{-1} \sum \hat{x}_i r_i \tag{7.8}$$

Finally using the definition of $\overline{r}$

$$\nabla r = r^{-1}\overline{r} \tag{7.9}$$

Equation (7.9) can now be used to simplify calculations such as

$$\nabla r^{-1} = -r^{-2}\nabla r = -r^{-3}\overline{r} \tag{7.10}$$

The gradient of the potential can now be calculated.

$$\nabla V_{LJ} = \nabla 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{7.11}$$

Applying the chain rule

$$\nabla V_{LJ} = 4\varepsilon\left[12\left(\frac{\sigma}{r_{ij}}\right)^{11} \nabla\left(\frac{\sigma}{r_{ij}}\right) - 6\left(\frac{\sigma}{r_{ij}}\right)^{5} \nabla\left(\frac{\sigma}{r_{ij}}\right)\right] \tag{7.12}$$

Now using equation (7.10) and doing some simplification

$$\nabla V_{LJ} = 48\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{11} \sigma\left(\frac{-\overline{r}}{r_{ij}^{\,3}}\right) - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{5} \sigma\left(\frac{-\overline{r}}{r_{ij}^{\,3}}\right)\right] \tag{7.13}$$

Rearranging gives

$$\nabla V_{LJ} = 48\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{11}\left(\frac{\sigma}{r_{ij}}\right)^{3}(-\overline{r}) - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{5}\left(\frac{\sigma}{r_{ij}}\right)^{3}(-\overline{r})\right]\left(\frac{1}{\sigma}\right)^{2} \tag{7.14}$$

Simplifying gives

$$\nabla V_{LJ} = -48\frac{\varepsilon}{\sigma^2}\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\overline{r} \tag{7.15}$$

By combining equations (7.2) and (7.15) it is found that

$$\overline{F} = -\nabla V_{LJ} = 48\frac{\varepsilon}{\sigma^2}\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \frac{1}{2}\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\overline{r} \tag{7.16}$$

Appendix B: Data for barriers

Table 0.1: Energies from *Figure 3.1e*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	A	4.994	4.994	0.118	0.122
A	C	4.994	5.622	0.447	0.433
A	E	4.994	5.652	0.453	0.432
B	B	5.337	5.340	0.312	0.288
B	C	5.337	5.637	0.191	0.243
C	D	5.640	5.320	0.607	0.558
D	E	5.322	5.655	0.247	0.246
E	F	5.653	5.335	0.553	0.537
F	F	5.333	5.334	0.292	0.259

Table 0.2: Energies from *Figure 3.1f*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.968	4.976	0.126	0.142
A	D	4.968	5.643	0.428	0.425
A	H	4.968	5.632	0.466	0.438
C	D	5.315	5.647	0.235	0.221
D	E	5.645	5.305	0.600	0.528
E	F	5.303	5.589	0.257	0.250
F	E	5.592	5.300	0.547	0.550
C	G	5.315	5.318	0.265	0.268
G	H	5.317	5.636	0.227	0.233
H	H	5.634	5.633	0.477	0.491

Table 0.3: Energies from *Figure 3.1g*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.977	4.965	0.133	0.192
A	E	4.977	5.605	0.460	0.454
A	H	4.980	5.648	0.433	0.423
A	K	4.980	5.641	0.464	0.435
A	N	4.977	5.630	0.424	0.443
B	B	4.971	4.964	0.125	0.146
B	F	4.966	5.590	0.495	0.450
B	L	4.970	5.609	0.502	0.443
C	N	5.323	5.629	0.167	0.199
C	D	5.322	5.327	0.319	0.253
D	E	5.323	5.621	0.175	0.211
E	F	5.624	5.585	0.552	0.498
F	G	5.590	5.309	0.533	0.541
G	H	5.309	5.650	0.257	0.246
H	I	5.649	5.321	0.568	0.537
I	J	5.318	5.322	0.257	0.259
J	K	5.321	5.644	0.227	0.233
K	L	5.642	5.611	0.483	0.517
L	M	5.612	5.317	0.546	0.554
M	N	5.312	5.629	0.245	0.254

Table 0.4: Energies from *Figure 3.1h*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.910	4.935	0.129	0.181
A	D	4.910	5.595	0.437	0.430
C	C	5.274	5.275	0.239	0.259
C	D	5.274	5.598	0.229	0.233
D	D	5.596	5.594	0.494	0.491

Table 0.5: Energies from *Figure 3.1i*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.940	4.955	0.145	0.180
A	H	4.940	5.639	0.402	0.415
A	L	4.940	5.887	0.422	0.410
A	O	4.940	5.605	0.400	0.424
B	B	4.955	4.952	0.138	0.145
B	D	4.955	5.608	0.461	0.431
B	F	4.950	5.584	0.450	0.428
C	O	5.297	5.623	0.241	0.235
C	D	5.297	5.611	0.258	0.220
D	E	5.610	5.299	0.576	0.520
E	F	5.295	5.586	0.236	0.250
F	G	5.589	5.293	0.546	0.550
G	H	5.296	5.641	0.257	0.246
H	J	5.640	5.314	0.560	0.537
J	K	5.313	5.307	0.270	0.285
K	L	5.307	5.889	0.213	0.204
L	M	5.890	5.323	0.744	0.781
M	N	5.326	5.325	0.311	0.283
N	O	5.321	5.623	0.204	0.221

.Table 0.6: Energies from *Figure 3.2a*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	A	4.940	4.936	0.187	0.160
A	C	4.940	5.630	0.413	0.423
A	D	4.937	5.606	0.404	0.420
B	D	5.313	5.607	0.249	0.253
B	C	5.314	5.634	0.259	0.272

Table 0.7: Energies from *Figure 3.2b*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.911	4.835	0.182	0.196
A	E	4.911	4.929	0.223	0.166
A	G	4.911	5.603	0.429	0.437
B	C	4.933	4.963	0.093	0.120
C	H	4.962	5.626	0.449	0.436
E	D	4.933	4.935	0.173	0.190
E	J	4.933	5.589	0.414	0.436
D	L	4.934	5.611	0.426	0.429
F	J	5.296	5.589	0.227	0.262
F	G	5.296	5.606	0.255	0.251
G	H	5.604	5.628	0.476	0.484
H	I	5.628	5.304	0.556	0.534
I	I	5.301	5.301	0.232	0.277
J	K	5.575	5.277	0.525	0.545
K	L	5.292	5.615	0.264	0.228

Table 0.8: Energies from *Figure 3.2c*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.909	4.930	0.170	0.196
A	E	4.909	4.933	0.153	0.152
A	G	4.909	5.587	0.443	0.437
B	C	4.930	4.953	0.107	0.120
B	D	4.930	4.886	0.207	0.226
C	H	4.952	5.613	0.450	0.432
C	K	4.952	5.617	0.445	0.436
D	L	4.884	5.581	0.445	0.452
E	M	4.932	5.576	0.423	0.432
F	M	5.279	5.574	0.207	0.262
F	G	5.279	5.592	0.260	0.251
G	H	5.590	5.615	0.482	0.484
H	I	5.615	5.290	0.560	0.534
I	J	5.286	5.288	0.220	0.277
J	K	5.288	5.620	0.227	0.236
K	L	5.618	5.579	0.495	0.543

Table 0.9: Energies from *Figure 3.2d*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.912	4.907	0.183	0.228
A	D	4.912	4.932	0.160	0.152
A	F	4.912	5.596	0.436	0.437
B	B	4.907	4.799	0.198	0.236
B	C	4.907	4.930	0.199	0.152
C	G	4.931	5.623	0.419	0.424
C	J	4.931	5.890	0.406	0.403
D	D	4.935	4.933	0.169	0.160
D	P	4.935	5.604	0.434	0.429
E	P	5.287	5.579	0.217	0.262
E	F	5.287	5.600	0.255	0.251
F	G	5.598	5.625	0.477	0.484
G	H	5.624	5.302	0.553	0.534
H	I	5.299	5.292	0.234	0.303
I	J	5.292	5.890	0.215	0.208
J	K	5.892	5.307	0.744	0.788
K	L	5.310	5.308	0.322	0.285
L	M	5.304	5.606	0.173	0.216
M	N	5.608	5.568	0.529	0.493
N	O	5.575	5.285	0.512	0.537
O	P	5.286	5.607	0.265	0.228
P	Q	5.605	5.287	0.588	0.531

Table 0.10: Energies from *Figure 3.2e*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	A	4.922	4.923	0.175	0.197
A	B	4.922	4.941	0.176	0.196
A	C	4.922	4.963	0.177	0.140
A	E	4.922	5.605	0.438	0.437
A	H	4.918	5.583	0.432	0.452
B	C	4.941	4.964	0.105	0.120
C	F	4.962	5.629	0.447	0.432
C	I	4.959	5.610	0.441	0.444
D	H	5.299	5.582	0.216	0.283
D	E	5.299	5.609	0.255	0.251
E	F	5.606	5.631	0.476	0.484
F	G	5.630	5.306	0.557	0.534
G	G	5.303	5.304	0.226	0.277
H	I	5.584	5.605	0.499	0.510
I	J	5.612	5.307	0.460	0.541
J	J	5.305	5.310	0.327	0.256

Table 0.11: Energies from *Figure 3.2f*.

		Binding energy (eV)		Barrier (eV)	
Start	End	Start	End	Calculated	Predicted
A	B	4.935	4.837	0.185	0.227
A	E	4.935	4.951	0.168	0.152
A	J	4.935	5.622	0.428	0.437
B	C	4.933	4.975	0.090	0.163
B	H	4.881	4.911	0.171	0.208
C	D	4.974	4.969	0.155	0.120
C	K	4.974	5.615	0.483	0.438
D	M	4.969	5.632	0.411	0.426
D	P	4.969	5.888	0.418	0.425
E	F	4.952	4.954	0.174	0.191
E	V	4.952	5.604	0.416	0.432
F	G	4.953	4.954	0.175	0.158
F	T	4.953	5.627	0.429	0.429
G	H	4.954	4.910	0.182	0.244
G	R	4.954	5.608	0.416	0.436
H	P	4.911	5.887	0.386	0.405
I	V	5.311	5.605	0.216	0.262
I	J	5.311	5.625	0.256	0.251
J	K	5.623	5.618	0.480	0.511
K	L	5.617	5.312	0.562	0.559
L	M	5.307	5.632	0.211	0.244
M	N	5.635	5.329	0.487	0.537
N	O	5.327	5.337	0.313	0.262
O	P	5.333	5.887	0.167	0.179
P	Q	5.890	5.316	0.814	0.791
Q	R	5.311	5.608	0.228	0.245
R	S	5.610	5.305	0.534	0.545
S	T	5.308	5.631	0.264	0.228
T	U	5.629	5.312	0.586	0.531
U	V	5.309	5.604	0.229	0.219

Appendix C: Code listings

Organization of code

The code is organised in classes. Each class performs one function. When there is multiple ways for a function to be performed a base class is made for that function. A subclass then implements the specifics of how the function is performed. This allows for easy moving to a new function implementation. Examples include using a different integrator or potential.

The settings class is passed to the constructor of most classes. This allows all the variables to be loaded when the classes are constructed. All the variables that are to be loaded are put together in a region for easy maintenance.

Code for the viewer, generator and analysis components is not included here.

Conversion Functions

The conversion functions are included in the file ‘Simulator.cs’.

```
        //All the conversion functions work in eV and A/ps
        private static double avogadro = 6.02214179e23;
        private static double electronCharge = 1.602176565e-19;
        private static double boltzmann = 1.3806e-23;// in joule

        static public double Energy2Velocity(double kineticEnergyIneV, double mass) {
//mass = 195.084 for Pt
            double massInKg = mass / (avogadro * 1000);
            double energyInJ = kineticEnergyIneV * electronCharge;
            double vInMeterPerSecond = Math.Sqrt(2 * energyInJ / massInKg);
            double vInAPerPs = vInMeterPerSecond / 100;
            return vInAPerPs;
        }

        static public double Velocity2Energy(double velocity, double mass) {
            return VelocitySquared2Energy(velocity * velocity, mass);
        }

        static public double VelocitySquared2Energy(double velocitySquared, double ma
ss) {
            double energy = 0.5 * mass * velocitySquared * 10 / (avogadro * electronC
harge);// ~ 0.0101 * velocitySquared//*10e8
            return energy;
        }

        static public double Energy2Temperature(double energy, double mass) {
            return energy * 2 * electronCharge / (3 * boltzmann);
        }

        static public double Temperature2Energy(double temperature, double mass) {
            return boltzmann * temperature * 3 / (2 * electronCharge);
        }

        static public double Temperature2Velocity(double temperature, double mass) {
            return Energy2Velocity(Temperature2Energy(temperature, mass), mass);
        }

        static public double Velocity2Temperature(double velocity, double mass) {
            return Energy2Temperature(Velocity2Energy(velocity, mass), mass);
        }

        static public double VelocitySquared2Temperature(double velocitySquared, doub
le mass) {
            return Velocity2Temperature(Math.Sqrt(velocitySquared), mass);
        }
```

Integrator

The integrator as implemented in ‘integrator.cs’ is:

```
using System;
using FileUtils;
using MyStuff;

namespace MD_Simulation {

    public abstract class Integrators {
        #region Variables that need saving & loading; Code that does that.-----------
----------------------

        protected double dt;

        public void Save(SettingHandler loader) {
            loader.SaveMember(() => dt);
        }

        private void Load(SettingHandler loader) {
            dt = loader.LoadMember(() => dt);
        }

        #endregion Variables that need saving & loading; Code that does that.--------
-------------------------

        protected Potential.UpdateDelegate UpdateAccelerations;
        protected VectorArray x, v, a;//These are just references to the actual data.

        protected Integrators(Simulator sim, SettingHandler loader) {
            x = sim.x;
            v = sim.v;
            a = sim.a;
            UpdateAccelerations = sim.UpdateAccelerations;

            Load(loader);
        }

        abstract public void Step();
    }

    public class Leapfrog : Integrators {

        public Leapfrog(Simulator sim, SettingHandler loader)
            : base(sim, loader) {
        }

        public override void Step() {
            x.PlusEqual(v, dt);
            UpdateAccelerations();
            v.PlusEqual(a, dt);
        }
    }

    internal class VelocityVerlet : Integrators {

        public VelocityVerlet(Simulator sim, SettingHandler loader)
            : base(sim, loader) {
        }

        public override void Step() {
            x.PlusEqual(v, dt, a, dt * dt / 2, false);
            v.PlusEqual(a, dt / 2);
            UpdateAccelerations();
            v.AddToCurrent(a, dt / 2);
        }
    }
}
```

Vector array

In order to make the handling of the arrays of vectors easier in the integrator, a class to assist with the array was implemented in ‘VectorArray.cs’.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MyStuff {

    public class VectorArray {

        // This class is only to provide a more convenient view of the simulation dat
a for the integrators.

        private MyVector[] simdata;

        public VectorArray(int numberOfSteps, int numberOfAtoms) {
            simdata = new MyVector[numberOfAtoms];
            for(int atom = 0; atom < NumberOfAtoms; atom++) {
                simdata[atom] = MyVector.Zero;
            }
        }

        public VectorArray(int numberOfSteps, VectorArray source, int numberOfExtraAt
oms = 0) {
            int numberOfAtoms = source.NumberOfAtoms + numberOfExtraAtoms;
            simdata = new MyVector[numberOfAtoms];
            for(int atom = 0; atom < numberOfAtoms; atom++) {
                simdata[atom] = MyVector.Zero;//step,
            }
            for(int atom = 0; atom < source.NumberOfAtoms; atom++) {
                this[atom] = source[atom];
            }
        }

        public override string ToString() {
            return ToString(0);
        }

        public string ToString(int stepInc) {
            string s = "";
            for(int atom = 0; atom < NumberOfAtoms; atom++) {
                s += this[atom].ToString();
                s += " ; ";
            }
            return s;
        }
        public int NumberOfAtoms {
            get {
                return simdata.Length;
            }
            set { }
        }

        public MyVector this[int index] {
            get { return simdata[index]; }
            set { simdata[index] = value; }
        }

        public MyVector this[int index, double scale] {
            get { return this[index] * (float)scale; }
            set { }
        }

        public void PlusEqual(VectorArray vec1, double vec1Scale) {

            //This is more efficient than using the operators because there are less
temporary objects.
            for(int i = 0; i < NumberOfAtoms; i++) {
                simdata[i] = this[i] + (vec1[i, vec1Scale]);
            }
        }

        public void AddToCurrent(VectorArray vec1, double vec1Scale) {

            //This is more efficient than using the operators because there are less
temporary objects.
            for(int i = 0; i < NumberOfAtoms; i++) {
                simdata[i] += (vec1[i, vec1Scale]);
            }
        }

        public void PlusEqual(VectorArray vec1, double vec1Scale, VectorArray vec2, d
ouble vec2Scale, bool lookAtAllAtoms = true) {

            //This is more efficient than using the operators because there are less
temporary objects.
            int numberOfAtomsToLookAt = MD_Simulation.Simulator.NUMBER_OF_ATOMS_CURRE
NTLY_AFFECTING_SIMULATION;
            if(lookAtAllAtoms) {
                numberOfAtomsToLookAt = NumberOfAtoms;
            }

            for(int i = 0; i < numberOfAtomsToLookAt; i++) {
                simdata[i] = this[i] + (vec1[i, vec1Scale]) + (vec2[i, vec2Scale]);
            }
        }
    }
}
```

Potentials

The potentials as implemented in ‘potential.cs’ is:

```
using System;
using System.Collections.Generic;
using FileUtils;
using MyStuff;

namespace MD_Simulation {

    public class Potential {

        //The purpose of this class is to calculate the potential at a point.
        //This class can't be used directly.
        //It is the base class for the class that implements a specific potential. Se
e below for example.

        protected VectorArray x, v, a;//These are just references to the actual data.
        public double PE, KE;
        public double maxV;//Used to check whether the cell structure should be
updated.
        private double[] vSquared;
        public double potentialForAtomAdded = 0;
        public int atomAdded = 0;

        #region Variables that need saving & loading; Code that does that.-----------
----------------------

        protected bool useSteele = false;
        protected double steele_size = 2.46;
        protected double steele_scale_force = 1;

        private double dt;
        private double targetDistanceTolerance;
        private string pullAtomTowardsTarget;

        public void Save(SettingHandler loader) {
        }

        private void Load(SettingHandler loader) {
            dt = loader.LoadMember(() => dt);
            targetDistanceTolerance = loader.LoadMember(() => targetDistanceTolerance
);
            pullAtomTowardsTarget = loader.LoadMember(() => pullAtomTowardsTarget);
            useSteele = loader.SettingExists("useSteele");
            if(useSteele) {
                steele_size = loader.LoadMember(() => steele_size);
                steele_scale_force = loader.LoadMember(() => steele_scale_force);
            }
            double desiredTemperature = 0;
            desiredTemperature = loader.LoadMember(() => desiredTemperature);
            desiredVelocity = Simulator.Temperature2Velocity(desiredTemperature, 195.
084);
        }

        #endregion Variables that need saving & loading; Code that does that.--------
-------------------------

        public Cells cells;

        public Potential(Simulator sim, SettingHandler loader) {
            x = sim.x;
            v = sim.v;
            a = sim.a;

            cells = sim.cells;
            vSquared = sim.vSquared;
            Load(loader);
            atomAdded = x.NumberOfAtoms - 2;
        }

        public delegate void UpdateDelegate();

        public delegate void PairHandler(int a1, int a2, MyVector r, double r2);

        public void Update() {
            for(int atom = 0; atom < a.NumberOfAtoms; atom++) {
                a[atom].MakeZero();
            }
            pot = 0;

            PreCalculate();
            cells.ForAllAtomPairs(HandlePair);
            PostCalculate();

            //Calculate the potential energy
            PE = pot;
            PE *= peScale;

            //Calculate the kinetic energy and the maximum velocity squared
            double maxV2 = 0;
            KE = 0;
            for(int i = 0; i < x.NumberOfAtoms; i++) {
                double l2 = v[i].LengthSquared();
                vSquared[i] = l2;
                KE += l2;
                if(l2 > maxV2) {
                    maxV = Math.Sqrt(l2);//This can possible be moved out of the loop
                    maxV2 = l2;
                }
            }
            KE = Simulator.VelocitySquared2Energy(KE, 195.084);
        }
        #region helper functions
        private double desiredVelocity;

        #endregion helper functions

        #region Variables for the calculation by child class

        protected double pot;//Temporary variable to hold the potential during one st
ep.

        protected double peScale;//scaling factor that needs to be applied to the pot
ential energy

        #endregion Variables for the calculation by child class

        #region extern members that the Potential classes HAVE to impliment.

        protected virtual void PreCalculate() {
        }

        protected virtual void PostCalculate() {
        }

        protected virtual void HandlePair(int a1, int a2, MyVector r, double r2) {
        }

        #endregion extern members that the Potential classes HAVE to impliment.
    }

    /*Convention:
     * rN = r^N
     * r_N = r^-N
     * R_N = r^-N*R where R is a vector
     */

    internal class LennardJones : Potential {

        public LennardJones(Simulator sim, SettingHandler loader)
            : base(sim, loader) {
            peScale = 4;
        }

        protected override void HandlePair(int a1, int a2, MyVector r, double r2) {
            double r_6 = 1 / (r2 * r2 * r2);
            double r_12 = r_6 * r_6;
            double potT = r_12 - (r_6);
            MyVector force = r * (float)((r_12 - r_6 / 2) / r2);

            pot += potT;

            a[a1] += force;
            a[a2] -= force;
        }

        protected override void PreCalculate() {
        }

        protected override void PostCalculate() {
            for(int i = 0; i < x.NumberOfAtoms; i++) {
                a[i] *= 48;
            }
        }
    }

    internal class ShiftedLennardJones : Potential {

        public ShiftedLennardJones(Simulator sim, SettingHandler loader)
            : base(sim, loader) {
            peScale = 4;
            double cutoffRaduis = 0;
            cutoffRaduis = loader.LoadMember(() => cutoffRaduis);
            shift = Math.Pow(cutoffRaduis, -12) - Math.Pow(cutoffRaduis, -6);
        }

        private double shift;

        protected override void HandlePair(int a1, int a2, MyVector r, double r2) {
            double r_6 = 1 / (r2 * r2 * r2);
            double r_12 = r_6 * r_6;
            double potT = r_12 - (r_6);
            MyVector force = r * (float)((r_12 - r_6 / 2) / r2);

            //lock(potentialLock)            {
            pot += potT - shift;

            a[a1] += force;
            a[a2] -= force;
        }

        protected override void PreCalculate() {
        }

        protected override void PostCalculate() {
            for(int i = 0; i < x.NumberOfAtoms; i++) {
                a[i] *= 48;
            }
        }
    }

    //----------------------------------------------------------------------------
    internal struct NearestNeighbour {

        //The purpose of this structure is to cache the data during the calculation.

        public NearestNeighbour(int nn, MyVector R2_5) {
            index = nn;
            aDivr_8multRdivr2 = R2_5;
        }

        public int index;// = the index of the neighbour atom
```

```
        public MyVector aDivr_8multRdivr2;// (a/r)^m * R/(r^2)  = R / r^(m+2) * a^m
    }

    internal class SuttonChenPtInRealUnits : Potential {

        public SuttonChenPtInRealUnits(Simulator sim, SettingHandler loader)
            : base(sim, loader) {
            aDivr_10multRdivr2 = new MyVector[x.NumberOfAtoms];
            aDivr_8 = new double[x.NumberOfAtoms];
            aDivr_10 = new double[x.NumberOfAtoms];

            neigbours = new List<NearestNeighbour>[x.NumberOfAtoms];

            for(int i = 0; i < x.NumberOfAtoms; i++) {
                aDivr_10multRdivr2[i] = MyVector.Zero;
                neigbours[i] = new List<NearestNeighbour>();
            }

            //-------------------------------
            peScale = 1;

            double mass = 195.084;
            double avogadro = 6.02214179e23;
            double electronCharge = 1.602176565e-19;
            forceConversionFactor = electronCharge * avogadro / (10 * mass);// ~ 9600
/mass

            if(useSteele) {
                steelePot = new SteelePotential(steele_size, steele_scale_force);
            }
        }

        private SteelePotential steelePot;

        private double forceConversionFactor;

        #region material specific parameters

        private const double n = 10;
        private const double m = 8;
        private const double c = 34.408;
        private const double a_lattice = 3.92;
        private const double a_lattice2 = a_lattice * a_lattice;
        private const double epsilon = 1.9833e-2;
        private const double ft1scale = -epsilon * n;
        private const double ft2scale = epsilon * c * m / 2;
        private const double cmOver2 = c * m / 2;

        #endregion material specific parameters

        private MyVector[] aDivr_10multRdivr2;//First part in the force calculation.
        private double[] aDivr_8;//This is the p_i terms.
        private double[] aDivr_10;//This is the first part for the potential calculat
ion.
```

```
        private List<NearestNeighbour>[] neigbours;
        /* All the lists should probably be changed to arrays for optimisation.
         * (http://www.codeproject.com/Articles/340797/Number-crunching-Why-you-
should-never-ever-EVER-us)
         */

        protected override void PreCalculate() {

            //Clear all the arrays.
            for(int i = 0; i < x.NumberOfAtoms; i++) {
                aDivr_10multRdivr2[i].MakeZero();
                aDivr_8[i] = 0;
                aDivr_10[i] = 0;
                neigbours[i].Clear();
            }
        }

        protected override void HandlePair(int a1, int a2, MyVector r, double r2) {
            /* Using multiplication and addition instead of power because of speed
             * {intel optimisation manual p 762-
763; http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-
manual.pdf, 30 aug 2012,
             * software developers manual http://download.intel.com/products/processo
r/manual/325462.pdf}
             * */
            double r_2 = 1 / r2;
            double ar_2 = a_lattice2 * r_2;// (a/r)^2
            double ar_4 = ar_2 * ar_2;
            double ar_8 = ar_4 * ar_4;
            double ar_10 = ar_8 * ar_2;

            //r2_n is a length so it will be the same in both directions.
            aDivr_8[a1] += ar_8;
            aDivr_8[a2] += ar_8;

            aDivr_10[a1] += ar_10;
            aDivr_10[a2] += ar_10;

            //R2_n is a vector so its direction will reverse for the second atom.
            MyVector term1 = r * (ar_10 * r_2);// (a/r)^n * R/(r^2)
            aDivr_10multRdivr2[a1] += term1;
            aDivr_10multRdivr2[a2] -= term1;

            MyVector term2 = r * (ar_8 * r_2);// (a/r)^m * R/(r^2)
            neigbours[a1].Add(new NearestNeighbour(a2, term2));
            neigbours[a2].Add(new NearestNeighbour(a1, -term2));
        }

        protected override void PostCalculate() {
            double[] pSqrtUnder1 = new double[x.NumberOfAtoms];
            for(int i = 0; i < pSqrtUnder1.Length; i++) {
                if(aDivr_8[i] == 0) {
                    pSqrtUnder1[i] = 0;//This means there was no nearest neighbours
                } else {
                    pSqrtUnder1[i] = 1 / Math.Sqrt(aDivr_8[i]);
                }
            }

            //-------------------------------

            for(int atom1 = 0; atom1 < x.NumberOfAtoms; atom1++) {
                MyVector sum2 = MyVector.Zero;// sum { (1/sqrt_p_i + 1/sqrt_p_j) (a/r
)^m R/(r^2) }
                for(int nn = 0; nn < neigbours[atom1].Count; nn++) {
                    sum2 += (pSqrtUnder1[atom1] + pSqrtUnder1[neigbours[atom1][nn].in
dex])
                        * neigbours[atom1][nn].aDivr_8multRdivr2;
                }

                a[atom1] = -epsilon * (-
n * aDivr_10multRdivr2[atom1] + cmOver2 * sum2);//now a = f in eV/A
                pot += 0.5 * aDivr_10[atom1] - c * Math.Sqrt(aDivr_8[atom1]);

                if(atom1 == atomAdded) {//Save info for special atom
                    potentialForAtomAdded = 0.5 * aDivr_10[atom1] -
 c * Math.Sqrt(aDivr_8[atom1]);
                    potentialForAtomAdded *= epsilon;
                }
            }
            pot *= epsilon;//don't have to do it for each atom now.

            if(useSteele) {
                double potTemp = 0;

                for(int atom = 0; atom < Simulator.NUMBER_OF_ATOMS_CURRENTLY_AFFECTIN
G_SIMULATION; atom++) {
                    a[atom] += steelePot.PotentialAndForceAt(x[atom], out potTemp);//
Apply Steele potential.
                    pot += potTemp;
                    if(x[atom].z > 30 && v[atom].z > 0) {
                        if(v[atom].z > 20) {
                            v[atom].z *= -0.9;
                        } else {
                            v[atom].z *= -1;
                        }
                    }
                }
            }

            for(int atom = 0; atom < a.NumberOfAtoms; atom++) {
                a[atom] *= forceConversionFactor;//a=f/m;
            }
        }
    }
    public class SteelePotential {

        //Possible improvements:
        // Add lookup for a*s1 + b*s2. Savings would probably be very small.
        // Move common sine calculations out of the derivatives. This would probably
make it about 15% faster.
```

```
        public SteelePotential(double steele_size = 2.46, double steele_scale_force =
 1) {
            oneOverA = 1 / steele_size;
            epc *= steele_scale_force;
        }

        public MyVector PotentialAndForceAt(MyVector v, out double potentialForAtom)
{
            double x = v.x * oneOverA;//scaled for the first part to
            double y = v.y * oneOverA;// the projection to s1 and s2
            double z = v.z * oneOverSigmapc;//scaled because of z*
            double s1, s2;
            xy2s(x, y, out  s1, out  s2);

            double[] EiVal = { E[0](z), E[1](z), E[2](z), E[3](z), E[4](z), E[5](z) }
;
            double[] fVal = { fscale[0] * f[0](s1, s2), fscale[1] * f[1](s1, s2), fsc
ale[2] * f[2](s1, s2), fscale[3] * f[3](s1, s2), fscale[4] * f[4](s1, s2), fscale[5]
* f[5](s1, s2) };
            double[] ds1fVal = { 0, dsfscale[1] * ds1f[1](s1, s2), dsfscale[2] * ds1f
[2](s1, s2), dsfscale[3] * ds1f[3](s1, s2), dsfscale[4] * ds1f[4](s1, s2), dsfscale[5
] * ds1f[5](s1, s2) };
            double[] ds2fVal = { 0, dsfscale[1] * ds2f[1](s1, s2), dsfscale[2] * ds2f
[2](s1, s2), dsfscale[3] * ds2f[3](s1, s2), dsfscale[4] * ds2f[4](s1, s2), dsfscale[5
] * ds2f[5](s1, s2) };

            double potential = 0;
            for(int i = 0; i < 6; i++) {
                potential += fVal[i] * EiVal[i];
            }

            double fx = 0;
            for(int i = 1; i < 6; i++) {
                fx += EiVal[i] * ds2fVal[i];
            }

            double fy = 0;
            for(int i = 1; i < 6; i++) {
                fy += EiVal[i] * (ds1fVal[i] - 0.5 * ds2fVal[i]) * yscale;
            }

            double fz = 0;
            for(int i = 0; i < 6; i++) {
                fz += dE[i](z) * fVal[i] * oneOverA;
            }

            potentialForAtom = potential * epc;
            return new MyVector(-fx * epc, -fy * epc, -fz * epc);
        }

        private double yscale = 1 / (Math.Sqrt(0.75));
        private const double oneOverPi = 1 / Math.PI;
        private const double sigmapc = 2.905;
        private const double oneOverSigmapc = 1 / sigmapc;
        private double oneOverA = 1 / 2.46;
        private double epc = 256 * 8.617e-5;

        private void xy2s(double x, double y, out double s1, out double s2) {
            s1 = y * yscale;
            s2 = x - 0.5 * s1;

            //Now scale to be able to put into the simplified f_i functions.
            s1 *= Math.PI;
            s2 *= Math.PI;
        }

        private Func<double, double>[] E = {
z=> 0.0096486935771095821978704520915926*z - 0.059521754060441273148640561885259 -
    26.471611160744238588904408970848*Math.Pow(z -
 0.12184210937936890140065315790707, -3.5322725491679580755999268149026) +
    42.541422108820711400767322629 69*Math.Pow(z -
 0.0028279720814790030095076112 10438, -9.9288944978424673593053739750758),

z=> 0.0026771860330833561689156674390233*z - 0.0080001335758203631631779728650145 +
    9085.2083334329945500940084457397*Math.Pow(z + 0.49598067010776697438600990608393
, -21.573040985022547744165422045626),

z=> 0.00052913207800193648184389205724187*z - 0.0015240825479119592469889887809131 +
    399854.43656259006820619106292725*Math.Pow(z + 0.67316046205391855128397082808078
, -30.624379896293120850714331027 12),

z=> 0.00030380309088065782186990904101265*z - 0.00086694009177576891386762802937938 +
    3532181.0755637940019369125366211*Math.Pow(z + 0.74515058420753998991159505749238
, -34.669004507003592152614146471024),

z=> 0.000086304674668712965160920214113816*z -
 0.00024179075872386040536865525751864 +
    3172851494.7974438667297363281 25*Math.Pow(z + 0.91574980579651088508086331785307,
 -45.530829200028662739896390121 43),

z=> 0.000046458000261504449335259181452784*z -
 0.00012959543941364331121751662934116 +
    247826959323.0172119140625*Math.Pow(z + 1.0001007632683678316709574573906, -
51.877749048159977007799170678481)
        };

        private Func<double, double>[] dE = {
z=> 93.504945435345021246081687042533*Math.Pow(z -
 0.12184210937936890140065315790707, -4.5322725491679580755999268149026) -
    422.38929190666385613181726974841*Math.Pow(z -
 0.0028279720814790030095076112 10438, -
3076210324253425/281474976710656) + 0.0096486935771095821978704520915926,

z=> 0.0026771860330833561689156674390233 -
 195995.57173461838813365148735 26*Math.Pow(z + 0.49598067010776697438600990608393, -
22.573040985022547744165422045626),
z=> 0.00052913207800193648184389205724187 -
 12245294.168510996303070072279607*Math.Pow(z + 0.67316046205391855128397082808078, -
31.624379896293120850714331027 12),
z=> 0.00030380309088065782186990904101265 -
 122457201.62827396995265461263811*Math.Pow(z + 0.74515058420753998991159505749238, -
35.669004507003592152614146471024),
z=> 0.000086304674668712965160920214113816 -
 144462559486.67804790978443904502*Math.Pow(z + 0.91574980579651088508086331785307, -
46.53082920002866273989639012143),
z=> 0.000046458000261504449335259181452784 -
 12856704803128.037505266833196187*Math.Pow(z + 1.0001007632683678316709574573906, -
52.877749048159977007799170678481),
        };

        private double[] fscale = { 1, -2, 4, -2, -2, 4 };
        private double[] dsfscale = { 1, 4 * Math.PI, -
8 * Math.PI, 8 * Math.PI, 4 * Math.PI, -24 * Math.PI };

        private Func<double, double, double>[] f = {
(s1,s2)=> 1,//This is to align the indexes of f and E.
(s1,s2)=> Math.Cos(2*s1) + Math.Cos(2*s2) + Math.Cos(2*(s1 + s2)),
(s1,s2)=> Math.Cos(2*s1 - 2*s2) + Math.Cos(2*s1 + 4*s2) + Math.Cos(4*s1 + 2*s2),
(s1,s2)=> Math.Cos(4*s1) + Math.Cos(4*s2) + Math.Cos(4*s1 + 4*s2),
(s1,s2)=> Math.Cos(2*s1 + 6*s2) + Math.Cos(6*s1 + 2*s2) + Math.Cos(4*s1 + 6*s2) + Mat
h.Cos(6*s1 + 4*s2) + Math.Cos(2*s1 - 4*s2) + Math.Cos(4*s1 - 2*s2),
(s1,s2)=> Math.Cos(6*s1 + 6*s2) + Math.Cos(6*s1) + Math.Cos(6*s2)
        };

        private Func<double, double, double>[] ds1f = {
(s1,s2)=> 1,
(s1,s2)=> Math.Sin(2*s1) + Math.Sin(2*s1 + 2*s2),
(s1,s2)=> Math.Sin(2*s1 - 2*s2) + Math.Sin(2*s1 + 4*s2) + 2*Math.Sin(4*s1 + 2*s2),
(s1,s2)=> Math.Sin(4*s1) + Math.Sin(4*s1 + 4*s2),
(s1,s2)=> Math.Sin(2*s1 + 6*s2) + 3*Math.Sin(6*s1 + 2*s2) + 2*Math.Sin(4*s1 + 6*s2) +
 3*Math.Sin(6*s1 + 4*s2) + Math.Sin(2*s1 - 4*s2) + 2*Math.Sin(4*s1 - 2*s2),
(s1,s2)=> Math.Sin(6*s1 + 6*s2) + Math.Sin(6*s1)
        };

        private Func<double, double, double>[] ds2f = {
(s1,s2)=> 1,
(s1,s2)=> Math.Sin(2*s2) + Math.Sin(2*s1 + 2*s2),
(s1,s2)=> Math.Sin(2*s1 - 2*s2) + 2*Math.Sin(2*s1 + 4*s2) + Math.Sin(4*s1 + 2*s2),
(s1,s2)=> Math.Sin(4*s2) + Math.Sin(4*s1 + 4*s2),
(s1,s2)=> 3*Math.Sin(2*s1 + 6*s2) + Math.Sin(6*s1 + 2*s2) + 3*Math.Sin(4*s1 + 6*s2) +
 2*Math.Sin(6*s1 + 4*s2) - 2*Math.Sin(2*s1 - 4*s2) - Math.Sin(4*s1 - 2*s2),
(s1,s2)=> Math.Sin(6*s1 + 6*s2) + Math.Sin(6*s2)
        };
    }
}
```

Cells

The cell structure as implemented in ‘Cells.cs’ is:

```
using System;
using System.Collections.Generic;
using System.Linq;
using FileUtils;
using MyStuff;

namespace MD_Simulation {

    public class Cells {
        #region variables and accessors

        #region Variables that need saving & loading; Code that does that.-----------
----------------------

        private double cellSize;
        private int xcellCount, ycellCount, zcellCount;
        private double xWrap, yWrap, zWrap;
        private double cutoffRaduis;

        public void Save(SettingHandler loader) {
            loader.SaveMember(() => cellSize);

            loader.SaveMember(() => xcellCount);
            loader.SaveMember(() => ycellCount);
            loader.SaveMember(() => zcellCount);

            loader.SaveMember(() => xWrap);
            loader.SaveMember(() => yWrap);
            loader.SaveMember(() => zWrap);

            loader.SaveMember(() => cutoffRaduis);
        }

        private void Load(SettingHandler loader) {
            cellSize = loader.LoadMember(() => cellSize);

            xcellCount = loader.LoadMember(() => xcellCount);
            ycellCount = loader.LoadMember(() => ycellCount);
            zcellCount = loader.LoadMember(() => zcellCount);

            xWrap = loader.LoadMember(() => xWrap);
            yWrap = loader.LoadMember(() => yWrap);
            zWrap = loader.LoadMember(() => zWrap);

            cutoffRaduis = loader.LoadMember(() => cutoffRaduis);
        }
        #endregion Variables that need saving & loading; Code that does that.--------
-------------------------

        #region nearest neigbours list

        private MyVector[] nearestNeighbours;//index offsets to nearest neighbour cel
ls

        private void InitializeNearestNeighbours() {
            int i = 0;
            nearestNeighbours = new MyVector[13];
            nearestNeighbours[i++] = new MyVector(0, 0, 1);
            nearestNeighbours[i++] = new MyVector(0, 1, -1);
            nearestNeighbours[i++] = new MyVector(0, 1, 0);
            nearestNeighbours[i++] = new MyVector(0, 1, 1);
            nearestNeighbours[i++] = new MyVector(1, -1, -1);
            nearestNeighbours[i++] = new MyVector(1, -1, 0);
            nearestNeighbours[i++] = new MyVector(1, -1, 1);
            nearestNeighbours[i++] = new MyVector(1, 0, -1);
            nearestNeighbours[i++] = new MyVector(1, 0, 0);
            nearestNeighbours[i++] = new MyVector(1, 0, 1);
            nearestNeighbours[i++] = new MyVector(1, 1, -1);
            nearestNeighbours[i++] = new MyVector(1, 1, 0);
            nearestNeighbours[i++] = new MyVector(1, 1, 1);
        }

        #endregion nearest neigbours list

        private VectorArray atomPositions;//the x vector array from simulation.
        private List<int>[, ,] cells;//There has to be at least 3 cells in all direct
ions to prevent serious wrap around artifacts.

        private List<int> atomsNotInCells;

        private readonly double halfXWrap, halfYWrap, halfZWrap;
        private double minWrapDist2;

        private double cutoffRaduisSquared;//Critical raduis squared
        private Potential.PairHandler HandlePair;

        private int this[int x, int y, int z] {
            get { return cells[x, y, z].Count(); }
            set { cells[x, y, z].Add(value); }
        }

        private int this[int x, int y, int z, int atomIndex] {
            get { return cells[x, y, z][atomIndex]; }
            set { cells[x, y, z][atomIndex] = value; }
        }

        #endregion variables and accessors

        #region init

        public Cells(Simulator sim, SettingHandler loader) {
            InitializeNearestNeighbours();

            atomPositions = sim.x;

            Load(loader);

            //Calculate values & init
            cutoffRaduisSquared = cutoffRaduis * cutoffRaduis;

            halfXWrap = xWrap / 2;
            halfYWrap = yWrap / 2;
            halfZWrap = zWrap / 2;

            minWrapDist2 = Math.Min(xWrap, yWrap) - cutoffRaduis;
            minWrapDist2 *= minWrapDist2;

            //Allocate space for arrays
            atomsNotInCells = new List<int>();
            cells = new List<int>[xcellCount, ycellCount, zcellCount];
            for(int xcell = 0; xcell < xcellCount; xcell++) {
                for(int ycell = 0; ycell < ycellCount; ycell++) {
                    for(int zcell = 0; zcell < zcellCount; zcell++) {
                        cells[xcell, ycell, zcell] = new List<int>();
                    }
                }
            }
        }

        #endregion init

        #region functions to go through all pairs

        private delegate void CellHandler();

        public void ForAllAtomPairs(Potential.PairHandler _HandlePair) {

            //This is the function that finds all the pairs of atoms that are interac
ting.
            HandlePair = _HandlePair;
            for(int xcell = 0; xcell < xcellCount; xcell++) {
                for(int ycell = 0; ycell < ycellCount; ycell++) {
                    ForAllZCells(xcell, ycell);
                }
            }

            //Now for atoms not in the cells.
            for(int atom = 0; atom < atomsNotInCells.Count; atom++) {
                for(int compareAtom = atom + 1; compareAtom < atomsNotInCells.Count;
compareAtom++) {
                    HandlePair_Check(atomsNotInCells[atom], atomsNotInCells[compareAt
om]);
                }
            }
        }
        public void ForAllZCells(int xcell, int ycell) {
            bool onEdge = false;
            onEdge = onEdge || xcell == 0 || xcell == xcellCount - 1;
            onEdge = onEdge || ycell == 0 || ycell == ycellCount - 1;
            for(int zcell = 0; zcell < zcellCount; zcell++) {
                if(cells[xcell, ycell, zcell].Count == 0) {
                    continue;
                }
                bool onEdge2 = onEdge || zcell == 0 || zcell == zcellCount - 1;

                HandlePairsInCell(xcell, ycell, zcell, onEdge2);
                HandleNearestNeighbours(xcell, ycell, zcell, onEdge2);
            }
        }

        private void HandlePairsInCell(int xcell, int ycell, int zcell, bool onEdge)
{

            //Find all interacting pairs in a cell.
            for(int atom = 0; atom < cells[xcell, ycell, zcell].Count; atom++) {//for
 all atoms in current cell
                for(int compareAtom = atom + 1; compareAtom < cells[xcell, ycell, zce
ll].Count; compareAtom++) {//for compareAtom in cell
                    HandlePair_Check(this[xcell, ycell, zcell, atom],
                                     this[xcell, ycell, zcell, compareAtom]);

                    //have to do wrap checking for atom in same cell becuase it can m
ove out of cell without an update
                }
            }
        }

        private void HandleNearestNeighbours(int xcell, int ycell, int zcell, bool on
Edge) {

            //Find all interacting pairs between this cell and its nearest neighbours
.

            //if cell is on edge
            if(onEdge) {
                for(int nn = 0; nn < nearestNeighbours.Length; nn++) {//for compareAt
om in nearest neigbour cell
                    int xc = xcell + (int)nearestNeighbours[nn].x;
                    int yc = ycell + (int)nearestNeighbours[nn].y;
                    int zc = zcell + (int)nearestNeighbours[nn].z;

                    WrapCellIndex(ref xc, xcellCount);
                    WrapCellIndex(ref yc, ycellCount);
                    WrapCellIndex(ref zc, zcellCount);

                    if(cells[xc, yc, zc].Count == 0) {
                        continue;//No atoms in the neighbouring cell.
                    }
                    for(int atom = 0; atom < cells[xcell, ycell, zcell].Count; atom++
) {//for all atoms in current cell
```

```
                        for(int compareAtom = 0; compareAtom < cells[xc, yc, zc].Coun
t; compareAtom++) {//for all atoms in neighbouring cell
                            HandlePair_Check(this[xcell, ycell, zcell, atom],
                                             this[xc, yc, zc, compareAtom]);
                        }
                    }
                }

                for(int atom = 0; atom < cells[xcell, ycell, zcell].Count; atom++) {/
/for all atoms in current cell
                    for(int compareAtom = 0; compareAtom < atomsNotInCells.Count; com
pareAtom++) {//for compareAtom in list
                        HandlePair_Check(this[xcell, ycell, zcell, atom], atomsNotInC
ells[compareAtom]);
                    }
                }
            } else {
                for(int nn = 0; nn < nearestNeighbours.Length; nn++) {//for compareAt
om in nearest neigbour cell
                    int xc = xcell + (int)nearestNeighbours[nn].x;
                    int yc = ycell + (int)nearestNeighbours[nn].y;
                    int zc = zcell + (int)nearestNeighbours[nn].z;

                    if(cells[xc, yc, zc].Count == 0) {
                        continue;
                    }
                    for(int atom = 0; atom < cells[xcell, ycell, zcell].Count; atom++
) {//for all atoms in current cell
                        for(int compareAtom = 0; compareAtom < cells[xc, yc, zc].Coun
t; compareAtom++) {
                            HandlePair_Check(this[xcell, ycell, zcell, atom],
                                             this[xc, yc, zc, compareAtom]);
                        }
                    }
                }
            }
        }

        protected void HandlePair_Check(int a1, int a2, bool doWrapAroundCheck = true
) {

            //Checks whether atoms are interacting.
            MyVector r = atomPositions[a1] - atomPositions[a2];

            double r2 = r.LengthSquared();
            if(r2 > cutoffRaduisSquared) {
                if(!doWrapAroundCheck) {
                    return;
                }
                if(r2 >= minWrapDist2) {
                    ForceWrap(r);

                    r2 = r.LengthSquared();
                    if(r2 > cutoffRaduisSquared)
                        return;
                } else {
                    return;
                }
            }

            //Atoms are interacting so pass the details to the potential calculation.
            HandlePair(a1, a2, r, r2);
        }

        #endregion functions to go through all pairs

        #region wrapping functions

        private void WrapCellIndex(ref int index, int size) {
            if(index < 0) {
                index += size;
            } else if(index >= size) {
                index -= size;
            }
        }

        public void WrapAll() {
            for(int atom = 0; atom < MD_Simulation.Simulator.NUMBER_OF_ATOMS_CURRENTL
Y_AFFECTING_SIMULATION; atom++) {
                ForceWrap(atomPositions[atom]);
            }
        }

        private void ForceWrap(MyVector v) {
            WrapPosition(ref v.x, halfXWrap);
            WrapPosition(ref v.y, halfYWrap);
            WrapPosition(ref v.z, halfZWrap);
        }

        private void WrapPosition(ref double pos, double bound) {
            if(pos < -bound) {
                pos += bound + bound;
            } else if(pos > bound) {
                pos -= (bound + bound);
            }
        }

        #endregion wrapping functions

        #region other functions

        public void Update() {

            //Updates the cell structure
            //The atoms not inserted into the cell structure are ignored in the inter
actions but taken
            //into account in the kinetic energy calculations so that atoms can be in
troduced easier.
            //NOTE: The cell structure is centered around 0,0,0. The simulation shoul
d also be centered there.
```

```
            Clear();
            for(int atom = 0; atom < MD_Simulation.Simulator.NUMBER_OF_ATOMS_CURRENTL
Y_AFFECTING_SIMULATION; atom++) {
                int xt = PositionToCellIndex(atomPositions[atom].X, xcellCount);
                int yt = PositionToCellIndex(atomPositions[atom].Y, ycellCount);
                int zt = PositionToCellIndex(atomPositions[atom].Z, zcellCount);

                if((xt < 0 || yt < 0 || zt < 0 ||
                    xt >= (xcellCount) ||
                    yt >= (ycellCount) ||
                    zt >= (zcellCount))) {
                    atomsNotInCells.Add(atom);
                } else {
                    this[xt, yt, zt] = atom;
                }
            }
        }

        private int PositionToCellIndex(double pos, int count) {
            double index = (pos + count * cellSize / 2) / cellSize;
            return (int)index;
        }

        private void Clear() {
            atomsNotInCells.Clear();
            for(int xcell = 0; xcell < xcellCount; xcell++) {
                for(int ycell = 0; ycell < ycellCount; ycell++) {
                    for(int zcell = 0; zcell < zcellCount; zcell++) {
                        cells[xcell, ycell, zcell].Clear();
                    }
                }
            }
        }

        #endregion other functions
    }
}
```

Performing fit for Steel potential

The Matlab® scritp used to perform the fitting for the Steele potential is provided in this section. The equations and their derivatives were also printed in order for to be used in the c# program. This was done with the following code.

```
function SteelePotential()
clc
clear
close all

%All the Pt-C parameters are global and defined here
% so that they can easily be used everywhere.
    global a1 sigmapc dz q epc as A E0 Ei Fi X Y x y z s1 s2 fz
    a1=2.46;
    sigmapc=2.905;
    dz = 1.38;%This is dz*
    q=2;
    epc=256*8.617e-5;
    as=(sqrt(3)/2);%This is a_s*
    A = sigmapc / a1;

    E0 = getfE0();
    Ei = getfEi();
    Fi = getfFi();

    z = 0.5:.01:4;

    X=-.2:.005:1.2;
    X = X.*2;
    Y=X;
    [x,y] = meshgrid(X,Y);

    [s1,s2] = toParms(x,y);

    fEi_n = getfEi();
    fittingStructure = {%{function2FitTo function2FitWith zeroCrossing
startValue}
        {getfE0(), @ffitLinear2, 1.0071603744, [-2.647161116074423 90e+001 -
1.21842109379368900e-001 -3.53227254916795810e+000 4.25414221088207110e+001 -
2.82797208147900300e-003 -9.92889449784246740e+000 9.64869357710958220e-003 -
5.95217540604412730e-002]};
        {fEi_n{1}, @ffitLinear, 1.6900000000, [9.08520833343299460e+003
4.95980670107766970e-001 -2.15730409850225480e+001 2.67718603308335620e-003 -
8.00013357582036320e-003]};
        {fEi_n{2}, @ffitLinear, 1.3264898345, [3.99854436562590070e+005
6.73160462053918550e-001 -3.06243798962931210e+001 5.29132078001936480e-004 -
1.52408254791195920e-003]};
```

```
        {fEi_n{3}, @ffitLinear, 1.2458359721, [3.53218107556379400e+006
7.45150584207539990e-001 -3.46690045070035920e+001 3.03803090880657820e-004 -
8.66940091775768910e-004]};
        {fEi_n{4}, @ffitLinear, 1.0394434840, [3.17285149479744390e+009
9.15749805796510890e-001 -4.55308292000286630e+001 8.63046746687129650e-005 -
2.41790758723860410e-004]};
        {fEi_n{5}, @ffitLinear, 1.0000000000, [2.47826959323017210e+011
1.00010076326836780e+000 -5.18777490481599770e+001 4.64580002615044490e-005 -
1.29595439413643310e-004]}
        };

    DoFit(fittingStructure);
    fprintf('\n\nprinting all for c#\n\n')
    PrintFunctionsForCSharp(fittingStructure)%Remember: s1&s2 are to be
divided by pi and z by sigmapc. Also the derivatives of the sines should be
multiplied by pi
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Utility functions

function [s1,s2] = toParms(x,y)
    global a1
    %Projection from x,y coordinates into crystal coordinates.
    s1 = y/sqrt(0.75);
    s2 = x-0.5*s1;
    %Taking just the decimal part because s1 and s2 are between 0 and 1.
    s1 = s1-fix(s1);
    s2 = s2-fix(s2);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%The basic function parts

function E0 = getE0()
    global a1 sigmapc dz q epc as A
    syms z n

    E0_preConstant = 2*pi*q*A^6/as;
    E0_sum = 0;
    E0_partial_sum = (2/5)*A^6/((z+n*dz)^10)-(1/(z+n*dz))^4;
    for i=0:25
        E0_sum = E0_sum + subs(E0_partial_sum, n, i);
    end
    E0_sum = E0_sum * E0_preConstant;
    E0_sum = simplify(E0_sum);

    E0 = E0_sum;
end

function Ei = getEi()
    global a1 sigmapc dz q epc as A
    syms s1 s2 z gi n

    gi_vals = (2*pi)*[2/sqrt(3), 2, 4/sqrt(3), 2*sqrt(7/3), 6/sqrt(3)];%
    Ei_preConstant = 2*pi*(A^6)/as;
    Ei_part = (A^6/30)*(gi/(2*z))^5*besselk(5,gi*z)-
2*(gi/(2*z))^2*besselk(2,gi*z);

    for i = 1:5
        Ei{i} = subs(Ei_preConstant*Ei_part,gi,gi_vals(i));
    end
end

function Fi = getFi()
    syms s1 s2

    f = 2*[...
     - (cos(2*pi*s1) + cos(2*pi*s2) + cos(2*pi*(s1 + s2))) ;...
     2*(cos(2*pi*(s1 + 2*s2)) + cos(2*pi*(2*s1 + s2)) + cos(2*pi*(s1 - s2)))
;...
     - (cos(4*pi*s1 ) + cos(4*pi*s2 ) + cos(4*pi*(s1 + s2 ))) ;...
     - (cos(2*pi*(3*s1 + s2)) + cos(2*pi*(s1 + 3*s2)) + cos(2*pi*(3*s1 +
2*s2)) + cos(2*pi*(2*s1 + 3*s2) ) + cos(2*pi*(s1 - 2*s2)) + cos(2*pi*(2*s1 -
s2))) ;...
     2*(cos(6*pi*s1 ) + cos(6*pi*s2) + cos(6*pi*(s1 + s2))) ];

    for i=1:5
        Fi{i} = f(i);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Getting the function handles to the parts

function fE0 = getfE0()
    syms z
    fE0 = @(height)subs(getE0(),z,height);
end

function fEi = getfEi()
    syms z
    Ei = getEi();
    for i = 1:5
        fEi{i} = @(height)subs(Ei{i},z,height);
    end
end

function fFi = getfFi()
    syms s1 s2
    Fi = getFi();
    for i = 1:5
        fFi{i} = @(x,y)subs(Fi{i}, {s1,s2}, {x,y});
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Getting the function handles to the derivatives
```

```
function [fdzE0,dzE0] = getdzE0()
    syms z
    temp = getE0();
    dzE0 = diff(temp,z);
    fdzE0 = @(height)subs(dzE0,z,height);
end

function [fdzEi,dzEi] = getdzEi()
    syms z
    temp = getEi();
    for i = 1:5
        dzEi{i} = diff(temp{i},z);
        fdzEi{i} = @(height)subs(dzEi{i},z,height);
    end
end

function [fds1Fi,ds1Fi] = getds1Fi()
    syms s1 s2
    temp = getFi();
    for i = 1:5
        ds1Fi{i} = diff(temp{i},s1);
        fds1Fi{i} = @(x,y)subs(ds1Fi{i}, {s1,s2}, {x,y});
    end
end

function [fds2Fi,ds2Fi] = getds2Fi()
    syms s1 s2
    temp = getFi();
    for i = 1:5
        ds2Fi{i} = diff(temp{i},s2);
        fds2Fi{i} = @(x,y)subs(ds2Fi{i}, {s1,s2}, {x,y});
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Performing the fit

function DoFit(fittingStructure)
    for i = 1: length(fittingStructure)
        finalParms{i} = Fitf(fittingStructure{i}{:}, i);
    end
    fprintf('\n')
    for i = 1: length(finalParms)
        fprintf('%f [',i-1)
        fprintf(' %20.17e', finalParms{i});
        fprintf(']\n')
    end
end

function parms = Fitf(f, fittingFunction, zeroCrossing, parms, i)
    global z fz
    fz = f(z);
    errorFunc = @(parms)Error(parms, fittingFunction);
    startError = errorFunc(parms);
    parms = fminsearch(errorFunc, parms,
optimset('MaxFunEvals',500000,'MaxIter', 50000));

    figure
    ErrorComparePlot(fz, fittingFunction(parms))
    subplot(4,1,1),title(['E' int2str(i-1)])

    stopError = errorFunc(parms);

    fprintf('start error = %7.4e\n', startError)
    fprintf('stop error = %7.4e\n', stopError)

%     fprintf('\n[')
%     fprintf(' %10.6e', parms);
%     fprintf('] \n\n')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Error functions for the fit

function err = ErrorBetween(fnew)
    global z fz
    err = fnew -fz;
    err = (err)./(fz+1);%+1 so that the error doesn't explode when f is
approximately 0
    err(z>0.7) = err(z>0.7)*1.2;%make the error on the approach count more
    err(z>0.95&z<2.4) = err(z>0.95&z<2.4)*1.3;%make the error in the minimum
count more
    err = sqrt(mean(err.^2));%calculate the rms error
end

function err = Error(parms, fittingFunction)
    err = ErrorBetween(fittingFunction(parms));
%     fprintf('err: %7.1e parms:',err);
%     fprintf(' %7.1e;', parms);
%     fprintf('\n')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Functions used in the fit

function y = ffitLinear(parms)
    global z
    y = 0;
    i = 0; y = y+ parms(1+i).*(z+parms(2+i)).^parms(3+i);
    i = 3; y = y+ parms(1+i).*z+parms(2+i);
end

function y = ffitLinear2(parms)
    global z
    y = 0;
    i = 0; y = y+ parms(1+i).*(z+parms(2+i)).^parms(3+i);
    i = 3; y = y+ parms(1+i).*(z+parms(2+i)).^parms(3+i);
    i = 6; y = y+ parms(1+i).*z + parms(2+i);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function ErrorComparePlot(fz, fparm)
    global z
    subplot(4,1,1), plot(z, fz)%original
    subplot(4,1,2), plot(z, fparm)%fit
    subplot(4,1,3), plot(z, fparm - fz)%error
    subplot(4,1,4), plot(z, (fparm - fz)./(fz+1))%percentage error
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function PrintFunctionsForCSharp(fittingStructure)
    syms s1 s2
    for i = 1: length(fittingStructure)
        fprintf(['\n\nE' int2str(i-1)])
        PrintEi(fittingStructure{i}{4})
    end
    Fi = getFi();
    [fds1Fi,ds1Fi] = getds1Fi();
    [fds2Fi,ds2Fi] = getds2Fi();
    for i = 1:5
        fprintf(['\n\nf' int2str(i)])
        PrintFi(i, subs(Fi{i},{s1,s2},{s1/pi,s2/pi}),
subs(ds1Fi{i},{s1,s2},{s1/pi,s2/pi}), subs(ds2Fi{i},{s1,s2},{s1/pi,s2/pi}) )
        %PrintFi(i, Fi, ds1Fi, ds2Fi)
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Printing helpers

function PrintEi(parms)
    global sigmapc
    syms x y z
    if length(parms)>5
        eq = parms(1).*(z+parms(2)).^parms(3) +
parms(4).*(z+parms(5)).^parms(6) + parms(7).*z + parms(8);
    else
        eq = parms(1).*(z+parms(2)).^parms(3) + parms(4).*z + parms(5);
    end
    %f = simplify(subs(eq, z, z/sigmapc))
    f = vpa(simplify(subs(eq, z, z)))
    df = vpa(simplify(diff(f)))
end

function PrintFi(i, Fi, ds1Fi, ds2Fi)
    f = vpa(simplify(Fi))
    ds1f = vpa(simplify(ds1Fi)/pi)
    ds2f = vpa(simplify(ds2Fi)/pi)
end
```

References

1 Tsybukh, R. *A comparative study of platinum nanodeposits on HOPG (0001), MnO(100) and MnOx/MnO(100) surfaces by STM and AFM after heat treatment in UHV, O2 , CO and H2*. Leiden University, Amsterdam, Netherland, 2010.

2 Gregor, C., Hermanek, M., Jancik, D., Pechousek, J., Filip, J., Hrbac, J. and Zboril, R. The Effect of Surface Area and Crystal Structure on the Catalytic Efficiency of Iron(III) Oxide Nanoparticles in Hydrogen Peroxide Decomposition. *Eur. J. Inorg. Chem.*, 16 (June 2010), 2343–2351.

3 Winsberg, E. Simulated Experiments: Methodology for a Virtual World. *Philosophy of Science*, 70 (January 2003), 105–125.

4 Haile, J.M. *Molecular Dynamics Simulations: Elementary Methods*. John Wiley & Sons, New York, USA, 1992.

5 Schroeder, M., Smilauer, P., and Wolf, D.E. Bond counting in a simulation model of epitaxial growth. *Phys. Rev. B*, 55 (1997).

6 Villarba, M. and Jonsson, H. Diffusion mechanisms relevant to metal crystal growth: Pt/Pt(lll). *Surf. Sci.*, 317 (1994), 15-36.

7 Griebel, M., Knapek, S., and Zumbusch, G. *Numerical Simulations in Molecular Dynamics*. Springer-Verlag, Heidelberg, Germany, 2007.

8 Burghaus, U., Stephan, J., Vattuone, L., and Rogowska, J.M. *A Practical Guide to Kinetic Monte Carlo Simulations and Classical Molecular Dynamics Simulations*. Nova Science Publishers, New York, 2005.

9 Satoh, A. *Introduction to Practive of Molecular Simulation*. Elsevier, Burlington, USA, 2011.

10 Sutton, A.P. and Chen, J. Long-range Finnis-Sinclair potentials. *Philosophical Magazine Letters*, 61, 3 (1990), 139-146.

11 Huang, S. and Balbuena, P.B. Platinum nanoclusters on graphite substrates: a molecular

dynamics study. *Molecular Physics*, 100, 13 (2002), 2165-2174.

12 Wu, G. and Chan, K. Molecular Simulation of Platinum Clusters on Graphite. *Surface Review and Letters*, 4, 5 (1997), 855-858.

13 Rafii-Tabar, H. and Mansoori, G.A. *Interatomic Potential Models for Nanostructures*. American Scientific Publishers. 2003.

14 Burden, R.L. and Faires, J.D. *Numerical Analysis*. Brooks/Cole, Belmont, USA, 2005.

15 Huang, K. *Introduction to statistical physics*. CRC Press, Florida, USA, 2001.

16 Halliday, D., Resnick, R., and Walker, J. *Fundamentals of physcs, 7th edition*. Wiley, Hoboken, USA, 2005.

17 Rapaport, D.C. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, New York, USA, 2004.

18 Caturla, M.J. and Marti, A.G. Molecular dynamics simulations of energy deposition in solids. *Advances in Quantum Chemistry*, 45, 1 (2004), 79-98.

19 Arblaster, J.W. Thermodynamic Properties of Platinum. *Platinum Metals Rev*, 49, 3 (Jul 2005), 141-149.

20 Kraftmakher, Y. Equilibrium vacancies and thermophysical properties of metals. *Physics reports*, 299 (1998), 79-188.

21 Krause, U., Kuska, J.P., and Wedell, R. Monovacancy Formation Energies in Cubic Crystals. *Phys. Status Solidi B,* 151 (1989), 479-494.

22 Rosato, V., Guillope, M., and Legrand, B. Thermodynamical and structural properties of f.c.c. transition metals using a simple tight-binding mode. *Phil. Mag. A*, 59, 2 (1989), 321-336.

23 Korhonen, T., Puska, M.J., and Nieminen, R.M. Vacancy-formation energies for fcc and bcc transition metals. *Phys. Rev. B*, 51 (1995), 9526-9532.

24 Panagiotides, N. and Papanicolaou, N.I. Diffusion of Platinum Adatoms and Dimers on Pt(111) Surface by Molecular-Dynamics Simulation. *Int J Quantum Chem*, 110 (2009), 202–209.

25 Kyuno, K., Gölzhäuser, A., and Ehrlich, G. Growth and the diffusion of platinum atoms and dimers on Pt(111). *Surf Sci*, 397 (1998), 191.

26 Bott, M., Hohage, M., Morgenstern, M., Michely, T., and Comsa, G. New Approach for Determination of Diffusion Parameters of Adatoms. *Phys Rev Lett*, 76 (1996), 1304.

27 Feibelman, P.J., Nelson, R.S., and Kellogg, G.L. Energetics of Pt adsorption on Pt(111). *Phys Rev B*, 49 (1994), 10548.

28 Stoltze, P.J. Simulation of surface defects. *Phys Condens Matter*, 6, 45 (1994), 9495.

29 Bulou, H. and Massobrio, C. Mechanisms of exchange diffusion on fcc(111) transition metal surfaces. *Phys Rev B*, 72, 20 (Nov 2005), 205427.

30 Steele, W.A. The Physical Interaction of Gases with Crystalline Solids. *Surface Science*, 36 (1973), 317-352.

31 Liem, S.Y. and Chan, K.Y. Simulation study of platinum adsorption on graphite using the Sutton-Chen potential. *Surface Science*, 328 (1995), 119-128.

32 Chen, J. and Chan, K.Y. Size-dependent mobility of platinum cluster on a graphite surface. *Molecular Simulation*, 31, 6-7 (May 2005), 527–533.

33 Zhang, P., Xie, Y., Ning, X., and Zhuang, J. Equilibrium structures and shapes of clusters on metal fcc(111) surfaces. *Nanotechnology*, 19 (2008), 255704.

34 Hairer, E., Lubich, C., and Wanner, G. Geometric numerical integration illustrated by the Störmer/Verlet method. *Acta Numerica*, 12 (2003), 399-450.

35 Allen, M.P. Introduction to Molecular Dynamics Simulation. *NIC Series*, 23 (2004), 1-28.

36 Bavel, A.P. van, Hermse, C.G.M., Hopstaken, M.J.P., Jansen, A.P.J., Lukkien, J.J., Hilbers, P.A.J., and Niemantsverdriet, J.W. Quantifying latera ladsorbate interactions by kinetic Monte-Carlo simulations and density-functional theory: NO dissociation on Rh(100). *Physical Chemistry*, 6 (Feb 2004), 1830-1836.

37 Lee, B. and Cho, K. Extended embedded-atom method for platinum nanoparticles. *Surface Science*, 600, 10 (May 2005), 1982-1990.