Chapter 3

Methodology

Relevant Information

In the earlier section, we described various tools and theories needed to perform atomistic simulations in Ni-Al systems. In this section, we shall give an account of the aforementioned concepts have been implemented on a computer. All programs have been written in GNU-C and all programs and libraries used support GNU license. Complete documentation of the code is available and present as an appendix.

Constants and simulation parameters

As mentioned earlier the microstate of the system is constrained by the extensive variables. If we say that the system is isothermal then that disallows certain paths in the phase space of the system, which lead to a change in temperature. There are many constants required to restrict the system to the intelligible part of the phase space. Following is the description parameters structure.

- `struct parameter` structure description This structure contains all relevant simulation information. Following is the current list of parameters it stores.
    - `parameter::Nx` stores the size in the x direction of a primitive lattice. This can be any type of primitive lattice, cubic, tetragonal, triclinic etc. Other system descriptors would need to be changed accordingly.
    - `parameter::Ny`
    - `parameter::Nz`
    - `parameter::lattice_parameter` stores the lattice parameter of the unit cell of a cubic type. For a non-cubic lattice, more than one lattice parameter need to be declared and corresponding changes need to be made in other relevant function. **This along with Nx, Ny, Nz completely specifies the**

    volume of the simulation cell.
    - `parameter::N_MCS` this determines the number of monte Carlo simulations that need to be performed in the simulation. This is an adjustable parameter. This relates to the statistical significance of the results. A Larger number of simulations means we sample a larger part of the ensemble, and hence the result produced will have a greater accuracy because of the *law of large numbers*.
    - `parameter::temperature` this variable stores the temperature of the simulation.
    - `parameter::atoms_per_site` this stores the number of atoms in the motif of the material that we trying to simulate. So in our particular case, this is always equal to 4.
    - `parameter::no_of_atoms` the number of atoms involved in the simulation are stored here. This value is equal to `Nx * Ny * Nz * no_of_atoms`. At the time of parsing and reading from a file, this values is calculated by the functions.

Many other variables such as pressure, chemical potential haven't been added to the codebase yet, however, will be in the future. This our aim to make this as flexible as possible in Appendix we have discussed how to modify the code base to represent other crystal systems, and thermodynamic conditions.

EAM potential

The EAM potential for Al-Ni systems is taken from here. These potential have been tested and benchmarked and the results are available here. We have written our own code to represent EAM potentials. Follows the description of binary EAM data structure

- `struct binaryEAMpotential`(ref in documentation) stores all the EAM information. EAM information is in the form of structures. The data is in form tables of pairwise potentials vs distance, electron density vs distance, and embedding energy vs electron density. For a binary potential, there are 7 tables. However, size of data was minimized by avoiding storing redundant information.
- `struct energyTableConcise` was designed to store values of energies that computed repeatedly. For example, a fixed lattice simulation won't see a change in relative positions of the atom. Hence it is efficient to compute all possible

values of energy for all possible configurations once and store them. In our study, we have stored values of radius dependent quantities, like pairwise potentials, and charge density. Since we are in FCC we compute these values for up to 7th nearest neighbours and don't do these computations again.
- `double energyTableInstantLookup[2][13][7][25]` is one of the constructs that stores the energies corresponding to all possible configurations of neighbourhoods for any atom. Consider the case of first nearest neighbours in FCC. There 12 such sites. All these sites must be filled, which is our assumption that we don't assume any vacancies and gaps in the structure. Hence the sites must be either occupied by Aluminium or Nickel. Since the relative orientation doesn't change energies dependent on distance, only the number of atoms of one type determines the energy, and there are 13 possible states, i.e no aluminium atom to all aluminium atoms. Similarly, there can be 7 states for second nearest etc. Detailed description in documentation.

All the above structures are currently in use in a codebase. The efficiency of these methods hasn't been completely scrutinized.

Representation of the solid

We use a fixed lattice with ideal regular structure. There are no vacancies present, no interstitials, all atoms occur at fixed positions. The system is modelled as an Ising-type solid where 1 and 0 represent atoms Ni and Al respectively. This model is a very convenient representation of a solid in a computer simulation although it is by no means completely accurate.

The type of atom is represented using an integer. An array of integers that has indices equal to the total number of atoms in the crystal is used to store the crystal configuration. `struct point3D` is an intermediate class that is used to manipulate these arrays. It stores a point in 3D real space. It is used for intermediate calculations. Since the information is stored in an array, we needed a function that can convert these array indices to points in 3D real space. This and many other utility functions on this structure can be found in `include\point3D.h`.

Coordinates of neighbour atoms

To calculate energies we need to look at the neighbourhood of an atom. We aren't storing coordinates for every atom. Instead, we have designed it such that we can take any array index from crystal lattice array and use the function `point3D_indexToPoint3D_fcc(..)`(see documentation) in the context of consistent simulation parameters and data, to compute the coordinates. We have computed coordinates of 7 nearest neighbours of an FCC atom.

We store these values in `struct neighbours_fcc`. These coordinates are relative to the atom that we are observing. So we can add these coordinates to any atom in the lattice and obtain coordinates of the neighbours. However, since the information about atom types are stored in an array we need to find the index corresponding to the coordinate. This is done for the fcc case by `point3D_point3DtoIndexFCC(..)` (s.d), this function takes a point3D object and returns the index that it corresponds to based on certain simulation parameters.

It is trivial to see that for a finite simulation block there will be a lack of neighbours on the edges. However, our arrangement mention above will generate a corresponding index even if that point happens to fall outside the simulation box. This was dealt with by using a periodic boundary condition. The periodic boundary condition is applied in the `point3D_point3DtoIndexFCC(..)`. This condition makes sure that we always stay in the simulation box.

Other functionalities of datatypes

Various reading and writing functions are written for these datatypes. The code was developed in C so it is prone to memory errors. The program and the environment both don't have a flag for bad data, inconsistent use cases, and incorrect input. The system is in such a way that we can add new things relatively easily. The FCC model that is used in this case can be extended to all bravais lattices, and in this, the only things that need change would be simulation parameters, energy functions, and the transformation function from index to coordinate and vice versa.

These datatypes have all the values public as there is no encapsulation in C. The values can be adjusted on the fly. However, care has to be taken when changing these values. Input functions to these datatypes take a few precautions but not all. These functions are passed around using pointers, and presently NULL guard isn't present in all the functions which can lead to segmentation faults. There is much scope of optimization in the code, however since the current code has been producing benchmarked results, it isn't a priority.

Energy Calculation

Energy calculation depends on all the aforementioned data structures. Following is the pseudo code for energy calculation.

```
inputs : atomic array, atomic index, fcc neighbours, EAM potential, pa
rameters
indexPoint = point3D_indexToPoint3D_fcc(atomic index);
pairwise <- 0
charge density <- 0
for : iterate over fcc neighbours
    ngbrPoint = indexPoint + fccNgbr;
    ngbrIndex = point3D_point3DtoIndexFCC(ngbrPoint);
    distance  = point3D_distAtoB(...);
    read energies at this distance from the table
    if('atom at ngbrIndex' same 'as atom at atomicIndex')
    {
        if(both Aluminum)
            pairwise += 0.5 * pAl-Al;
            charge density += chargeDensityAl;
        else
            pairwise += 0.5 * pNi-Ni;
            charge density += chargeDensityNi;

    }
    else {
        pairwise += 0.5 * pNi-Al;
        if(atom at atomic index is Al)
            chargeDensity += chargeDensityNi;
        else
            chargeDensity += chargeDensityAl;
    }
endfor
finding embedding energy of calculated charge density;
total energy = pairwise + embedding energy;
return total energy
```

Shown above is the pseudo code of the function that calculates energy at a particular energy Index. The exact implementation can be looked up in the documentation. The pseudo code mentioned above gives the general algorithm used in calculating energy in various functions. The following function follows the aforementioned scheme to compute energy.

```
energyAtIndexFCC(...)
```

Many other functions that calculate the energy in the entire lattice, create lookup tables for energy all use the same function to calculate the energy, i.e for the FCC system.

```
energyInMatrix(...);
energyToSwap(...);
deltaEnergyMatrix(...);
createLookUpTable(...);
buildInstantEnergyLookup(...);
energyAtIndexFCC_fast(...);
```

The first three functions use the `energyAtIndexFCC(...)` as it is. For example `energyInMatrix(...)` computes energy for every atom in the matrix. `energyToSwap(...)` computes the energy required to switch the current state of a particular lattice site, this is the energy required to change an atom at a site from Aluminium to Nickel and vice versa. `deltaEnergyMatrix(...)` computes swaps for all atoms in the array.

The following three functions are of a different approch. `createLookUpTable(...)` creates a `struct lookUpTable` type object that stores the radius dependent data that doesn't change for a fixed lattice simulation. `buildInstantEnergyLookup(...)` can be used after creating a `lookUpTable` to initialize the `double energyTableInstantLookup[2][13][7][25]`. This table only holds up to 3rd nearest neighbours reasoning for which is discussed in this report. `energyAtIndexFCC_fast(...)` can't be used before calling `buildInstantEnergyLookup(...)`. This function uses saved values computed by building function in order to compute energies. It computes energy by iterating over fcc neighbours and counting no of atoms of each type.

Monte Carlo Sampling

We have mentioned that Monte Carlo is a sampling method. In the context of thermodynamics, there will be too many states to sample them all, in any reasonable

amount of time. Most of these states will be too high energy and won't contribute much to the partition function or the equilibrium property.
Looking at the expression for the probability of a state it is quite clear to see that large positive value for the hamiltonian will have very low probability of occurring.

There are different methods of sampling ensembles. Let's consider the simplest case. In Buffon's Needle problem we stated that randomly dropping pin many times will be a legitimate experiment for establishing the probability of the event. However, we didn't mention the properties that need to be satisfied in order to confirm that a particular trial experiment trying to model the system is valid.

Following are the conditions that need to be satisfied for any experiment to used in an MC simulation.

1. Equal apriori probability.
2. Detailed balance.

Equal apriori probability means that if we fix the macrostate of the system, i.e properties such as volume, temperature, energy, pressure in such a way that an ensemble is defined, then all microstates that have these aforementioned fixed properties will be equally probable. This is a fundamental postulate of statistical mechanics and is supported by Ergodic hypothesis and Jaynes idea of maximum information entropy[ref]. This condition is implicit in our simulation.

The second assumption, however, isn't trivial. Mathematically the second expression can be written as follows.

$$\pi_i P_{ij} = \pi_j P_{ji}$$

This innocuous looking expression has been called upon multiple times in Physics by Boltzmann, Maxwell, and Einstein and many others to explain natural phenomenon. Markov chain Monte Carlo need to satisfy this detailed balance. In the given expression π_i, π_j are probabilities of 'i' and 'j' states respectively and, P_{ij} is the probability of transition from i to j state. The expression now reads "the probability of going to a state j given that we are in state i must be equal to the probability of going to i if we are in j". This is the criteria for a reversible Markov chain. It now becomes easy to see why this condition is needed. In simple terms, it can be understood as, if a system makes a particular transition then the expectation of that event is same as that of a back transition.

If repeat the experiment similar to Buffon's needle for thermodynamics then we won't get correct results. This style of sampling is called simple sampling.Thermodynamic systems are very complex but, the majority of viable states are around the equilibrium value. So if we try to sample all states equally we will generate erroneous results. Say we are trying to sample energy, then we know that equilibrium is generally a low energy state. However, in an ensemble average through simple sampling, the high energy results will dominate the result. Simple sampling won't distinguish low or high energies so they will be uniformly distributed, leading to the said result.

Metropolis and hastings came up with a way to solve this problem in their paper[ref]. This idea is called important sampling, and we have used this to define our MC trail experiment. Here we add a bias in sampling. We must compensate for this bias in some way.

In metropolis algorithm we make a trial change $i \to j$ in the system, let's say this change constitutes a change in energy of $\Delta E_{i\to j}$ then

$$\Delta E_{i\to j} > 0 \quad \text{Accept with probability, } P = \exp\left(\frac{-\Delta E_{i\to j}}{k_B T}\right)$$

$$\Delta E_{i\to j} < 0 \quad \text{Accept with probability, } P = 1$$

It can be shown that this satisfies detailed balance, and hence is valid sampling method for the ensemble. Another method of sampling is Gibbs Sampling[ref]. In next section, we have described the trial moves we did for various ensembles.

Random Numbers

It is necessary to discuss random numbers in context with Monte Carlo. When John von Neumann was programming the ENIAC to run simulations he used a particularly bad random number generator. His reasoning was that it was fast and it misbehaved in predictable ways.

There are two types of random numbers. True random numbers and pseudorandom numbers and, generators, i.e computer programs that generate them, are called TRNGs and PRNGs respectively. We haven't programmed the pseudo random number generator ourselves. A case has been made for the use of GSL random number generator that is under GNU license.

The generator we are using is standard gsl generator which is known as r1279.
This is a lagged fibonacci generator[Katzgraber] and, are mathematically given by,

$$x_i = (x_{i-j} * x_{i-k}) \mod m$$

Here $M = 64$ for our system, $j = 1279$ and $k = 418$ and $*$ is a binary operator(not sure which one).
For our system, the period of this generator will be $2^{k-1}2^{M-3}$, or 10^{403}. With this type of period, we don't need to worry about randomness.

This alone, however, doesn't make this a good choice, there are other factors like covariance, autocorrelation, equidistribution etc. Investigations of the validity of r1279 haven't been done in our case. Also, the tail analysis is extremely important, i.e the behaviour or probability distribution of random number near edges of its range.