Why alloys prefer to be random at higher temperatures

Abstract

The aim of the project was to write a program which simulated the behaviour of metal alloys using the metropolis monte-Carlo swapping algorithm. The behaviour of the alloys was simulated taking into account a range of different temperatures, compositions and interaction energies. From this, the project set out to explain why alloys prefer to be random at high temperatures and more ordered at lower temperatures. The results of the simulation would then quantitively describe the behaviour of alloys allowing for a cohesive, convincing explanation to be formulated explaining the behaviour of a range of alloys at a range of temperatures. Finally, the project set out to demonstrate the benefits of using a computer program to simulate natural phenomenon.

From the simulations it was concluded that as temperature rose, alloys tended to take up a more disordered configuration resulting in a phase change. In addition to this it was concluded the associated transition temperature rose with composition and that the transition temperatures for the case where the interaction parameter (Eam) was – 0.1 eV resulted in the highest transition temperatures. Finally, it was concluded that when Eam = 0.0 eV the simulation was no longer modelling the behaviour of an alloy but rather that of a homogenous metal system.

Introduction

The project revolved around the key concept of alloys always looking to minimise their Gibbs free energy. The Gibbs free energy, $\Delta G = H - T\Delta S$ [1], has an entropic and enthalpic term with temperature playing a crucial role in minimising the Gibbs free energy. A key characteristic of alloy metal systems that follows as a result of this is that at high temperatures one would predict the alloying element to be distributed randomly in an effort to maximise the entropy and hence minimise the Gibbs free energy. Overall the Gibbs free energy is lowered as the increase in the entropy(at high temperatures) from more atoms mixing is greater than the Gibbs free energy increase because of the size difference of the atoms. At lower temperatures the size mismatch is not sufficiently countered by the increase in entropy and so we form more ordered alloys as this configuration results in the minimisation of the Gibbs free energy.

A computer simulation is a clean, controllable environment where the underlying physical principles governing natural phenomenon can be tested, analysed and as a result better understood. When using a computer program, it is easy to test a range of parameters in a short amount of time. Carrying out such an investigation physically in a lab would be a much more dangerous, time consuming and complex affair, where trying to have full control of all potential variables is vastly more challenging.

When considering a lattice of size 10x10 in two dimensions there up to 1.01×10^{29} possible combinations of the atoms in the lattice (at 50% alloying composition). In order to fully understand the nature of the behaviour of alloys at different temperatures one may imagine that one has to consider all these combinations. This would be a task which is utterly unfeasible to carry out at any reasonable time scale and so the metropolis monte Carlo algorithm is turned to. A random walk is carried out through a sample space and the remarkably useful characteristic of this is that if one repeats this often enough, abiding by the correct conditions of the simulation one evaluates every point in the sample space with a frequency proportional to its probability. As a result, an efficient simulation is produced which has a high fidelity to the original problem.[1]

For the simulations carried out, the principle subject of interest was the disorder of the alloy systems in relation to their temperatures, interaction energies and compositions. From the data produced by the simulations, one hoped to both quantitively and qualitatively describe the nature of an alloy at equilibrium at a range of temperatures and compositions. Along with this, the effect of changing the temperature at different equilibrium states was hoped to be analysed so as to observe any sudden changes in the order of a system and by extension phase present. All of this would help to learn about the nature of disorder in different alloys at a range of temperatures, compositions and interaction energies.

Method

In this implementation of the metropolis Monte Carlo algorithm an atom was picked at random from a lattice with periodic boundary conditions applied. A direction was chosen at random and once the neighbouring atom had been selected the program would calculate the change in energy of the system, were the swap to actually occur. If the swap reduced the energy of the system, then the swap would be carried out. This step was consistent with the laws of thermodynamics where systems looks to minimise their free energy as much as possible. If the swap did not reduce the energy of the system then it would still occur if $e^{\frac{\Delta E}{TK_b}} > R$, R being a randomly generated number between zero and one. This step accounted for the impact of temperature on minimising the Gibbs free energy such that changes which may not be energetically favourable still lower the overall free energy. The comparison to a random number is the probabilistic nature of the problem which by the use of Monte Carlo simulations results in a frequency of changes proportional to the probability of the event happening in an actual system in the natural world.[2]

The process of designing the code began by formulating the information contained within the lab script in the form of high-level pseudocode. This allowed for the identification of how many functions would need to be generated, along with the input and output of each function. It also demonstrated that using a program based on classes would not be necessary. A flow chart was produced to give an overview of the project. When it came to writing the code, care was taken to fully comment crucial lines and sections. During the writing stage, before formal testing, print statements were used to track the output of a function at different stages and the output of functions was continually compared to calculations done by hand. A system of unit tests was then used to formally check each function. Boundary test cases were chosen for all functions and the unit test class would confirm whether a function had passed or failed when compared to a known, true output.

For the program to run and carry out the simulation of the behaviour of the alloy the following information had to be inputted. The composition of the system, the temperatures at which to run the simulation at, the interaction parameter and the dimensions of the box shaped lattice.

From the simulations run, one would obtain a range of information regarding the behaviour of an alloy as temperature and interaction energy were varied, such as the equilibrium configurations, the phase transition temperatures and the plots of energy against progress in the simulation. Data however beyond the scope of the simulation were entities such as the effect of pressure, impurities, vacancies, defects as well as any potential effects of extending the model into three dimensions as well as any consideration of any potential changes of state.

Results

The program

The final program consisted of six individual functions along with a main function, where all the functions were called in the correct order. Throughout the writing process the outputs were checked against calculations done by hand and then subsequently unit testing (found in appendix) was used to rigorously prove the program worked as intended. The functions are documented as follows:

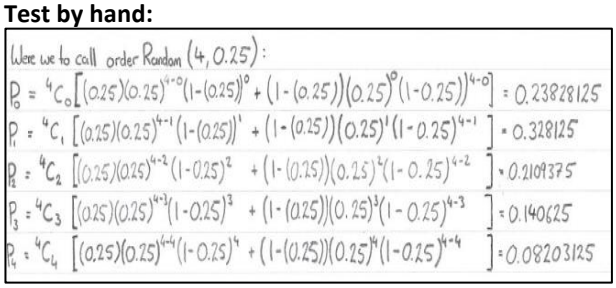
orderRandom

Inputs: The number of neighbouring sites per atom, The fraction of alloying atoms
Outputs: Probability distribution of order parameter

This function had the purpose of generating a probability distribution describing the probability that a certain lattice site would have a certain number of unlike neighbours (zero to four unlike neighbours). The crucial premise of the function was that it calculated this probability distribution for a completely random alloy. The binomial distribution was used to generate this distribution

function. The function used was $P_n = {}^{Z}C_n[ff^{(Z-n)}(1-f)^n + (1-f)f^n(1-f)^{(Z-n)}]$, with Z = number of neighbouring sites per atom and n = number of unlike neighbours.

Test by hand:



Input

```
print("The probability distribution is ", orderRandom(4, 0.25))
```

Output

```
The probability distribution is  [0.23828125 0.328125   0.2109375  0.140625   0.08203125]
```

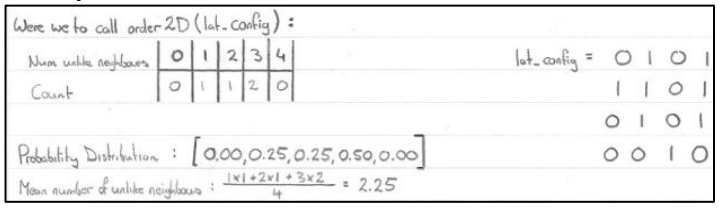
order2D

Inputs: Configuration of a lattice

Outputs: Probability distribution of order parameter, Mean number of unlike neighbours

This function was to generate a probability distribution, for a given configuration of lattice, based upon the order parameter. The order parameter was defined as the number of unlike neighbours around a lattice site. The function would go on to then calculate the mean number of unlike neighbours in a given configuration by multiplying the probability from the distribution with the associated number of unlike neighbours.

Test by hand:



Input

```
lat_config  = np.array([[0,1,0,1], [1,1,0,1], [0,1,0,1], [0,0,1,0]])
a = order2D(lat_config)
print("The probability distribution is ", a[0],"\nThe mean number of unlike neighbours is ", a[1])
```

Output

```
The probability distribution is  [0.   0.25 0.25 0.5  0.  ]
The mean number of unlike neighbours is  2.25
```

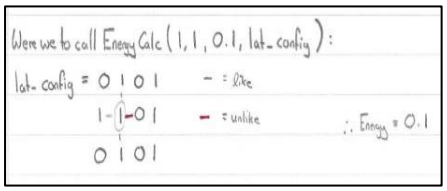
EnergyCalc

Inputs: Row index of lattice position, Column index of lattice position, Eam, lattice

Outputs: Sum of the energy of the bonds around the chosen lattice position

This function calculates the energy of the four surrounding bonds of a given lattice point. The function works by comparing the four adjacent lattice positions. If the atoms occupying a lattice position are identical to the chosen atom then the bond energy is zero, otherwise it is the energy of the interaction energy.

Test by hand:



Input

```
lat_config  = np.array([[0,1,0,1], [1,1,0,1], [0,1,0,1]])
print("The energy at lattice point 2,2 is ", EnergyCalc(1,1,0.1, lat_config))
```

Output

```
The energy at lattice point 2,2 is  0.1
```

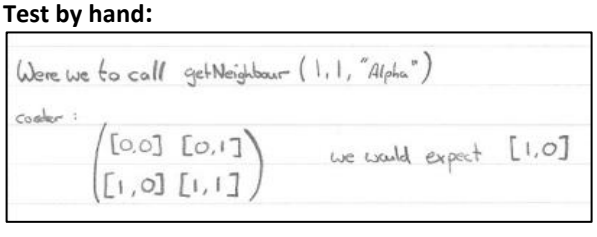
getNeighbour

Inputs: Row index of lattice position, Column index of lattice position, A direction

Outputs: Row index of neighbouring lattice position, Column index of neighbouring lattice position

The getNeighbour function returns the lattice indices adjacent to a lattice point in a given inputted direction. Depending on the direction inputted the function adjusts the rows and columns appropriately to generate the neighbouring lattice point. It must be noted that “Alpha” = Left neighbour, “Beta” = Above neighbour, “Delta” = Right neighbour, “Gamma” = Below neighbour.

Test by hand:



Input

```
print("The neighbouring atom to the right is ", getNeighbour(1,1,"Alpha"))
```

Output

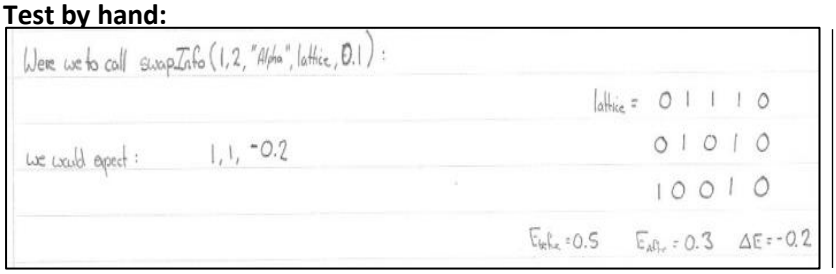
```
The neighbouring atom to the right is  (1, 0)
```

swapInfo

Inputs: Row index of lattice position, Column index of lattice position, A direction, lattice, Eam
Outputs: Row index of neighbouring lattice site, Column index of neighbouring lattice site, Energy change

The swapInfo function calculates the difference in energy between the swapped and unswapped state of a randomly chosen lattice position and its neighbour. It works by using the previously described EnergyCalc function to calculate the energy of both the swapped and unswapped states, taking care not to overcount the energy of the bond shared between the two atoms. The energy difference is calculated on a copy of the generated lattice such as to not change the lattice of the simulation which would be altered appropriately in the alloy2D function.

Test by hand:



Input

```
print(swapInfo(1,2,"Alpha", lattice, 0.1))
```

Output

```
(1, 1, -0.2000000000000007)
```

Alloy2D

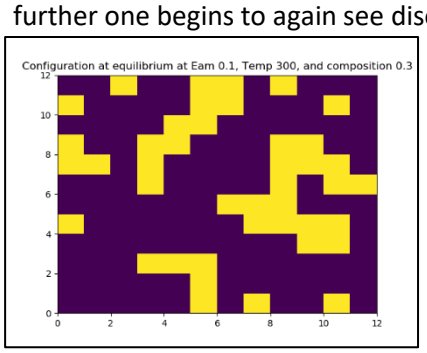
Inputs: lattice size, alloying fraction, num of total sweeps, number of sweeps to equilibrate, temperature, Eam
Outputs: Mean number of unlike neighbours, Mean energy of lattice, Heat capacity

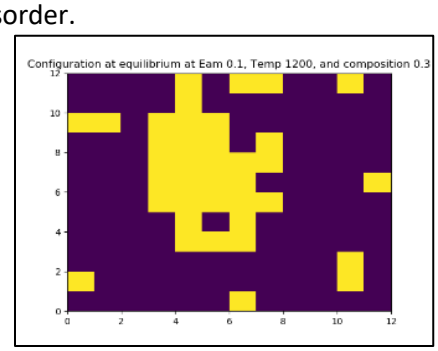
The alloy2D function begins by generating a two-dimensional lattice where the number of alloying atoms is calculated and then distributed randomly throughout the lattice. Next periodic boundary conditions are applied by copying each edge layer and then appending it to the opposite side of the lattice. The function then calculates the total initial energy of the function and then continues by carrying out the Monte Carlo simulation using the appropriate conditional statements and previously created functions. For each sweep a random direction is generated and a random lattice point is chosen, ensuring the point chosen lies within the inner lattice so that boundary conditions are observed. Once the system has reached equilibrium the Monte Carlo simulation is continued with order2D and the energy of the lattice structure used to calculate the mean number of unlike neighbours, the mean energy and mean energy squared. From this, heat capacity is calculated. Since this function comprised of the previously tested functions along with a rather long iterative portion it was only unit tested, the result of which can be found in the appendix.

Variation in final configuration

Temperature

When considering the variation of the lattice equilibrium configuration with respect to temperature one immediately concludes the following. At low temperatures the configuration is slightly random with portions of order observable, however as the temperature rises there is a point where one begins to see an ordered formation of both alloying and host atoms, on raising the temperature further one begins to again see disorder.





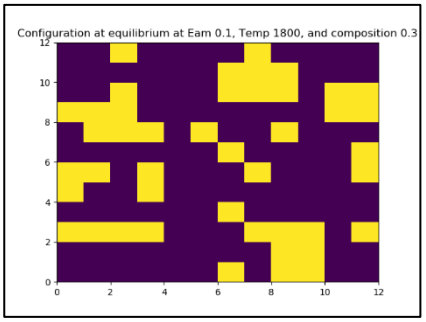
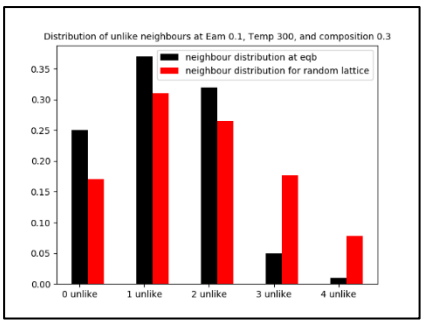
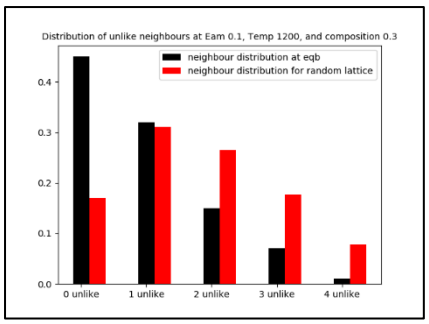


Fig1,2,3(Left to right). Fig1: Equilibrium configuration at 300K (portions of order observable). Fig 2: Equilibrium configuration at 1200K (clear formation of ordered region at the centre). Fig 3: Equilibrium configuration at 1800K (system returns to a disordered state) [all systems having constant Eam of 0.1 and composition of 30%]





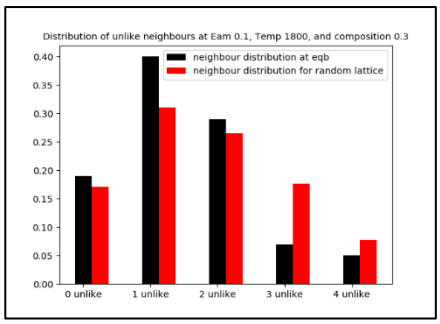
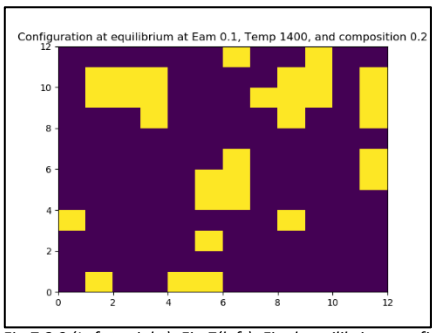
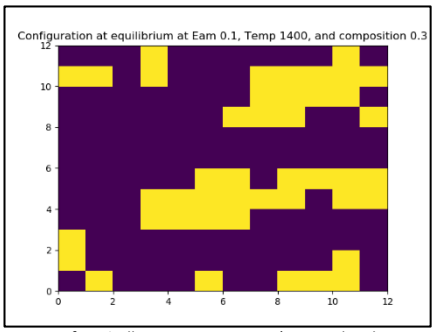


Fig 4,5,6 (Left to right). Fig 4: Distribution of unlike neighbours compared to random alloy at 300K (distribution in favour of low values of order parameter). Fig 5: Distribution of unlike neighbours compared to random alloy at 1200K (configuration heavily skewed to 0 unlike neighbours indication formation of ordered regions of like atoms). Fig 6: Distribution of unlike neighbours compared to random alloy at 1800K (once more similar to a random alloy).

Concentration of alloying element

While maintaining a constant interaction energy and temperature one observes that the final equilibrium configuration varies in how ordered it is, depending on composition. At a chosen temperature, certain lower compositions have changed from an ordered state and begin to be disordered , while systems with higher compositions are still in an ordered state. Naturally also the region of order formed is larger for higher composition systems since there are more alloying atoms present to form it.





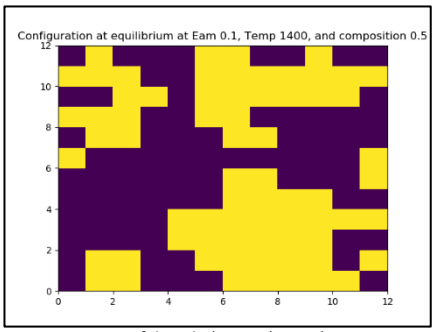
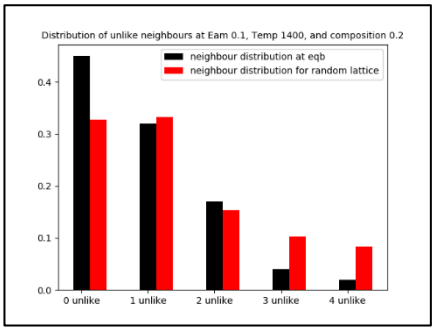
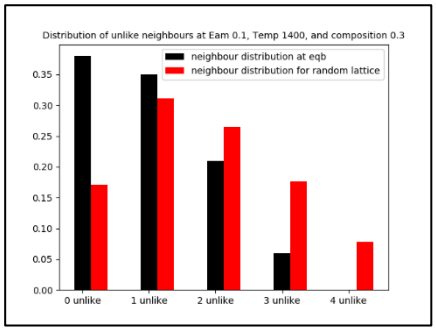


Fig 7,8,9 (Left to right). Fig 7(left): Final equilibrium configuration of 20% alloying composition (System has begun to turn into a state of disorder). Fig 8 (centre) : Final equilibrium configuration of 30% alloying composition (System is beginning to return to a state of disorder however there are still large regions of order). Fig 9 (right): Final equilibrium configuration of 50% alloying composition (System is still very much in a state of order with an increase in entropy at 1400K unfavourable as it is outweighed by the energetic cost of unlike neighbours interacting). [all systems having constant Eam of 0.1 and temperature of 1400K]





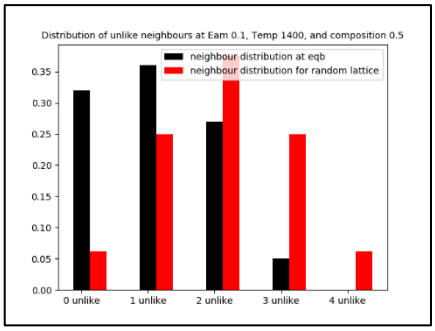
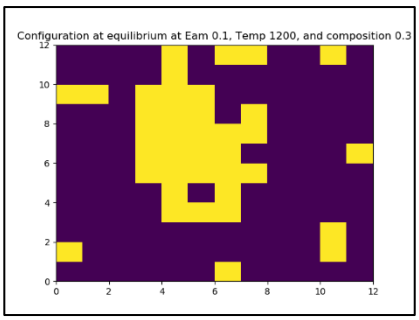
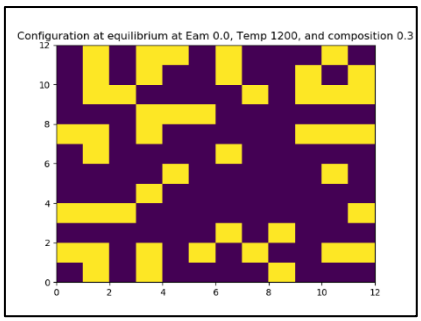


Fig 10,11,12 (Left to right) Distribution of unlike neighbours compared to a random alloy. Fig 10(left): Distribution has close resemblance with the distribution of a random alloy indicating the system moved to a state of disorder. Fig 11 (centre) : Distribution is skewed in favour of 0 unlike neighbours, indicating the system is still in an ordered state. Fig 12(right): Distribution is very skewed in favour of fewer unlike neighbours indicating the ordered nature of the system despite the high temperature.

Interaction parameter

Upon analysing the variation in final equilibrium configurations with respect to the interaction parameter one observes that for Eam = 0.0 the alloy system behaves as a homogenous metal with a distribution almost identical to that of a random lattice. For Eam = 0.1 one observes clusters of alloying elements coming together since although there is a reduction in entropy in this arrangement, there is less of a large energy penalty between unlike atoms (at low enough temperatures). For Eam = -0.1 there is no energy penalty at all and spreading out atoms is the most energetically favourable state, as a result we see a very strong skew in the distribution of unlike neighbours towards the maximum (four) unlike neighbours.





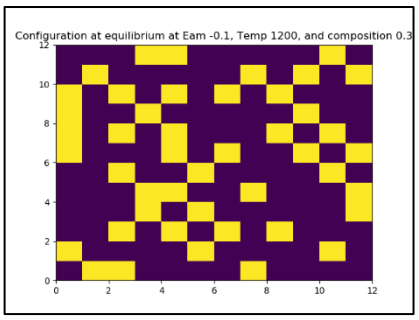
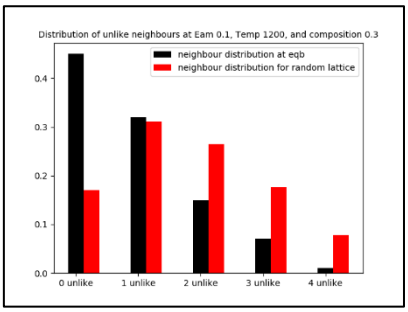
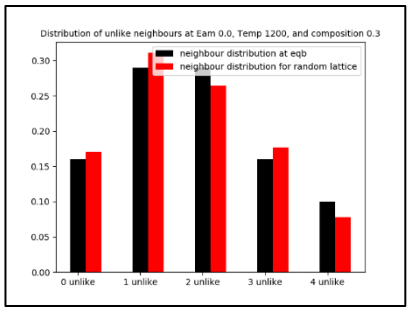


Fig 11,12,13 (Left to right) Configuration of system at equilibrium. Fig 11lLeft): Some small regions of order. Fig 12(centre): Random arrangement. Fig 13(right): Alternating formation of alloying and non-alloying atom since a maximum number of neighbours gives the lowest possible energy configuration





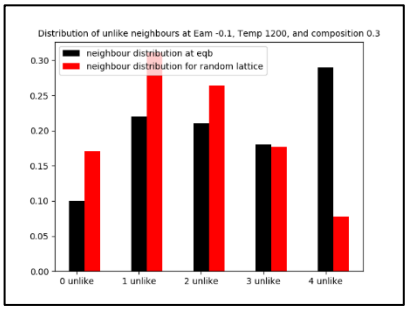


Fig 14,15,16 (Left to right) Distribution of unlike neighbours compared to a random alloy. Fig 14 (left): Slight skew to 0 unlike neighbours due to energy penalty of the unlike bond energy. Fig 15(centre): analogous to homogenous metal, distribution very similar to random lattice. Fig 16(right): Due to reduction in energy in formation of unlike bonds a strong skew towards four unlike neighbours is observed.

The Transition Temperature

After each simulation reached equilibrium, the simulation was further continued the equilibrium configuration for a further 55,000 steps. For each temperature and composition, the mean of the order parameter and the associated heat capacity was recorded and calculated. These were plotted against temperature and the temperature at which a sharp variation in either value occurred was indicative of a transition temperature. For the system with Eam = 0.0 eV, since it was representative of a homogenous metal there was no fluctuation in the order parameter and the associated heat capacity was always zero. As a result, a table for Eam = 0.0 eV was not included. The best estimate of the transition temperature was obtained by calculating the mean between the values obtained for the order parameter and the heat capacity.

Interaction Energy = 0.1 eV

Temp of significant change in order parameter (K)	Temp of significant change in heat capacity (K)	Best Estimate of transition Temperature (K)	Composition (%)
550	600	575	10
700	800	750	20
1250	1300	1275	30
1700	1800	1750	40
1900	2000	1950	50

Table 1 : Temperature of significant variation in order parameter and head capacity for all alloy compositions with an Eam = 0.1eV

Interaction Energy = - 0.1 eV

Temp of significant change in order parameter (K)	Temp of significant change in heat capacity (K)	Best Estimate of transition Temperature (K)	Composition (%)
1100	1100	1100	10
1500	1600	1550	20
1800	1750	1775	30
2000	1900	1950	40
2400	2300	2350	50

Table 2 : Temperature of significant variation in order parameter and heat capacity for all alloy compositions with an Eam = - 0.1Ev

Example of Variation of order parameter and heat capacity, C with temperature

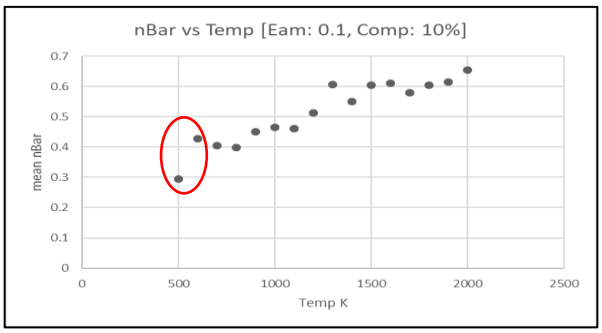


Fig 17: Variation in order parameter with temperature [Eam = 0.1, Comp = 10%] Circled in red is the sharp change in the value of mean order parameter

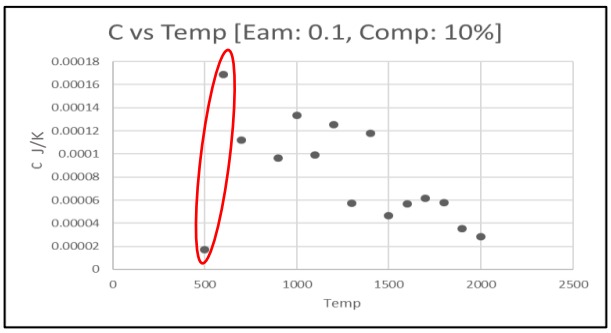


Fig 18: Variation in heat capacity with temperature [Eam = 0.1, Comp 10%] Circled in red is the sharp change in the heat capacity

Discussion

General rules emerging from the results

General rules emerging for temperature dependence

From analysing the data produced, the clearest and most consistent conclusion drawn with regards to temperature dependence was as follows. At lower temperatures an ordered formation of a phase field was favoured however as the temperature rose the alloy systems tended towards more disordered configurations. The underlying explanation for this phenomenon is found by considering the different components which contribute to the overall Gibbs free energy $\Delta G = H - T\Delta S$. At lower temperatures, increasing the disorder of the system would not have reduced the free energy as much as the arrangement of phase field where the enthalpy term was lowest, and the entropy term was also very low (the states had a high degree of order). In the case of Eam = 0.1eV this meant a cluster of alloying particles and in the case of Eam = -0.1eV this meant the formation of an alternating, checkerboard like structure. For each of these cases (with the system at a low temperature), were the system to become more disordered one can observe that the lowering in Gibbs energy due to increased entropy would be offset by the much larger increase in Gibbs energy due to the bond interaction energy and so as a result the Gibbs energy would not be at a minimum and such a process would not occur. At higher temperatures however this increase in entropy outweighs the bond energy penalty incurred and as a result the alloy becomes more disordered. The temperature at which the alloy starts becoming disordered is signified as the transition temperature.

General rules emerging for concentration dependence

From the simulations carried out it was concluded that as the amount of alloying fraction increased the transition temperature for the associated system subsequently was also higher. In explaining this it is important to consider the energetic consequences of forming unlike bonds compared to the consequences of forming a disordered system. At low compositions the fact that there are fewer alloying atoms means that there is less of an energy penalty when they begin to spread themselves in a random, disordered fashion throughout the host atoms. As a result, for the increase in entropy to make the transition feasible the temperature need not be as high as for higher composition alloys. When higher composition alloys spread themselves out in a more disordered state there is a much greater energetic cost due to the formation of unlike bonds and only at higher temperatures is this offset by the entropic increase. (It must be noted that when higher composition alloys spread out there is a higher associated entropy, however this does not offset the argument, it merely reduces the difference in how much bigger one might expect the transition temperatures to be).

General rules emerging for the interaction parameter dependence

For the system where the Eam = 0.0eV it was swiftly concluded that there was no transition temperature and the system behaved almost identically to a homogenous metal. Since the interaction energy is 0.0 eV there is nothing differentiating the host atoms from the alloy atoms and as a result the idea of an order parameter and the associated heat capacity lose their relevance. The only possible phase transition would have been a change of state however this was beyond the model developed.

When analysing the system where Eam = 0.1 eV it was observed that generally the system began with alloying atoms clumping together to form separate phase fields and then as the temperature rose the systems became more disordered. This is explained by the fact that at higher temperatures the entropy term outweighed any penalty from unlike bonds forming and so the system increased in disorder.

When analysing the system where Eam = - 0.1 eV, initially the checkerboard pattern was observed and then at high temperatures (higher than for 0.1 eV) the system became more disordered. The highly ordered checkerboard formation initially formed since a high number of unlike neighbours resulted in the lowest possible energy configuration. Since the interaction energy was already negative the entropy term only began to dominate at very high temperatures, becoming more negative than the already negative Eam value. Hence the transition temperatures for Eam = -0.1eV were observed at higher temperatures than for Eam = 0.1 eV and were characterised by a drop in the mean value of the order parameter unlike a sharp rise as in the case of Eam = 0.1eV.

Variation in transition temperature with alloy concentration and bond energy

Trends in transition temperature

For all non-zero Eam systems it was observed that the transition temperature increased as the composition of the system increased. This was explained by the fact that a larger enthalpy term meant that only at higher temperature would the entropy term dominate and result in a reduction of the Gibbs free energy. When comparing between systems it was clearly observed that the Eam = -0.1 eV system had higher transition temperatures at all compositions in comparison to the 0.1 eV system. One can account for this by considering that the -0.1 eV value means that the enthalpy term in the Gibbs free energy expression is already negative. As a result, the temperature has to be very high for the entropy to dominate and result in a disordered state, breaking up the ordered checkerboard like formation.

Difference Between specific heat and order parameter results

In most cases the heat capacity and the order parameter were similar in value however upon analysing the nature of the systems investigated it was concluded that the specific heat would be a more accurate measure of the transition temperature. Each transition analysed could be classified as a second order or continuous phase transition.[3] Our model dealt only with solid to solid phase transitions and the model did not account for latent heat. The order parameter is best at identifying first order transitions, where it will characteristically appear to indicate a change of phase. Since there was no latent heat however and we were dealing with second order, continuous phase transitions the heat capacity would be the more accurate indicator of transition temperature as it accounted for each bond over the second order phase change. A fluctuation in heat capacity is due to a change of position of atoms in the lattice and is directly related to the entropy. In our systems phases changes occur when the entropic term dominates and as a result the heat capacity is a better indicator a phase transition.[3] The order parameter is less suited since it does not show such sharp fluctuations as we are dealing with a second order continuous phase transition.

Size of system

The premise of the simulations carried out was that the metropolis Monte Carlo method of sampling the nature of a system was accurate of a true real-world system. Since we were dealing with probabilities it follows that an increased system size would result in more accurate results, provided that the simulation would be run with enough steps in a reasonable amount of time.[2] In addition to this one needs to consider the effects of finite size. Although periodic boundary conditions were applied for a 10 by 10 box it is likely that certain thermodynamic quantities such as the heat capacity get smoothed out and broadened since we are dealing with a repeating lattice. [5]This being due to the fact that the system may not be as truly random as hoped for and not indicative of a system in the real world as well as the fact that any potential long-range ordering effects cannot be accounted for. Hence the values for the transition temperature could be more accurately obtained if a larger box size was used.

Appendix

References:

[1] *: Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A. and Teller, E. (1953). Equation of State Calculations by Fast Computing Machines. The Journal of Chemical Physics, 21(6), pp.1087-1092.*

[2]: Monte Carlo Methods Stéphane Paltani ISDC Data Center for Astrophysics Astronomical Observatory of the University of Geneva Statistics Course for Astrophysicists, 2010–2011

[3] : Web.mit.edu. (2020). *Criticality*. [online] Available at: https://web.mit.edu/8.334/www/grades/projects/projects10/AlexanderPapageorge/Page5.html [Accessed 19 Feb. 2020].

[4]: University of California, Irvine : [online] https://ps.uci.edu/~cyu/p238C/class.html [Accessed 26 Feb. 2020].

[5]: Binder, K. and Young, A. (1986). Spin glasses: Experimental facts, theoretical concepts, and open questions. *Reviews of Modern Physics*, 58(4), pp.801-976.

Copy of code

```
## imports ##
import numpy as np
import matplotlib.pyplot as plt
import numpy.ma as ma
import math
from math import exp
from random import randrange
import copy
import tqdm as tqdm


########################################
    # G L O B A L  V A L U E S #
########################################
cellA = 0 # Matrix atom
cellB = 1 # Alloy atom


def orderRandom(Z,f):
    """
    OrderRandom gives the distribution function of unlike neighbours for a completely random allo
y.
    The order parameter here is the NUMBER OF AB BONDS around a lattice site.
    The binomial distribution is used to calculate the distribution fucntion.

    Input Arguments
        Z => The number of neighbouring sites per site
        f => The fraction of alloying atoms

    Output Arguments
        N_Rand => List of possible number of neighbours
        P_Rand => The probabilty distribution of the order parameter

    """
```

```
    # ================================================================================================
================================================

    Z = 4
    prob1 = 0
    P_Rand = np.linspace(0,0,Z+1)
    N_Rand = np.linspace(0,Z,Z+1)

    # Generate distribution based on binomial expression
    for n in range(5):
        prob1 = (math.factorial(Z)/((math.factorial(n))*( math.factorial(Z-
n) ) ))*     ((f)*(f**(Z-n))*((1-f)**n) + (1-f) *(f**n)* ((1-f)**(Z-n)) )
        P_Rand[n] = prob1
        N_Rand[n] = n

    return(P_Rand, N_Rand)

#-------------------------------------------------------------------------------------------------
---------------------------------

def order2D(eqb_lattice):
    """
    ORDER2D produces a distribtion function of the orderparameter.
    The order parameter is just the number of AB bondsaround a site.

    Input arguments
        config => configuration of the system at eqb

    Output arguments
        P_Alloy => List of possible number of neighbours
        N_Alloy => The probabilty disitribution of the order parameter
        mean_unlike_neigh => The mean number of unlike neighbours for a given configuration

    """
    total_unlike = 0
    zero_unlike = 0
    one_unlike = 0
    two_unlike = 0
    three_unlike = 0
    four_unlike = 0



    P_Alloy = np.linspace(0,0,5)
    N_Alloy = np.linspace(0,4,5)

    # total number of atoms under consideration
    dimensions = eqb_lattice.shape
    total_atoms = (dimensions[0]-2) * (dimensions[1] - 2)


    #loop through the lattice
    for i in range (1, dimensions[0]-1):
```

```
        for j in range(1,dimensions[1]-1):

            if eqb_lattice[i,j] != eqb_lattice[i, j-1]:
                total_unlike = total_unlike + 1
            if eqb_lattice[i,j] != eqb_lattice[i-1,j]:
                total_unlike = total_unlike + 1
            if eqb_lattice[i,j] != eqb_lattice[i, j+1]:
                total_unlike = total_unlike + 1
            if eqb_lattice[i,j] != eqb_lattice[i+1, j]:
                total_unlike = total_unlike + 1

            if total_unlike == 4:
                four_unlike = four_unlike + 1
            elif total_unlike == 3:
                three_unlike = three_unlike + 1
            elif total_unlike == 2:
                two_unlike = two_unlike + 1
            elif total_unlike == 1:
                one_unlike = one_unlike + 1
            elif total_unlike == 0:
                zero_unlike = zero_unlike + 1

            total_unlike = 0



    # list with the number of atoms which have N_unlike neighbours
    unlike = [zero_unlike, one_unlike, two_unlike, three_unlike, four_unlike]

    #generating the distribution
    #Generating the distribution
    distr = []
    for i in unlike:
        prob = i/total_atoms
        distr.append(prob)

    #assigning the distribution
    P_Alloy = np.array(distr)

    # Mean unlike number of neighbours
    sum_unlike = 0
    for i in range(len(unlike)):
        sum_unlike = sum_unlike +  i*unlike[i]

    mean_unlike_neigh = sum_unlike/total_atoms


    return(P_Alloy, N_Alloy, mean_unlike_neigh)


# ------------------------------------------------------------------------------------------------
------------------------
# This function is extra to the script
def EnergyCalc (Atom_iIndex, Atom_jIndex, Energy_AB, lattice):
```

```
    """
    This function calculates the energy of the four surrounding bonds of a given atom

    Inputs
        Atom_iIndex => The i index of the chosen element in the numpy array
        Atom_jIndex => The j index of the chosen element in the numpy array
        Energy_AB => Energy of a bond between an alloying and host atom
        lattice => numpy array representing the atoms

    Outputs
        Energy => Energy of the four bonds surrounding the inputted atom

    """
    # Energy before

    # left bond
    if lattice[Atom_iIndex, Atom_jIndex] == lattice[Atom_iIndex, Atom_jIndex - 1]:
        Bond_One = 0
    else:
        Bond_One = Energy_AB
    # top bond
    if lattice[Atom_iIndex, Atom_jIndex] == lattice[Atom_iIndex - 1, Atom_jIndex]:
        Bond_Two = 0
    else:
        Bond_Two = Energy_AB
    # right bond
    if lattice[Atom_iIndex, Atom_jIndex] == lattice[Atom_iIndex, Atom_jIndex + 1]:
        Bond_Three = 0
    else:
        Bond_Three = Energy_AB
    # bottom bond
    if lattice[Atom_iIndex, Atom_jIndex] == lattice[Atom_iIndex + 1, Atom_jIndex]:
        Bond_Four = 0
    else:
        Bond_Four = Energy_AB
    Energy = Bond_One + Bond_Two + Bond_Three + Bond_Four
    return(Energy)

# ------------------------------------------------------------------------------------------------
--------------------------------

def swapInfo(FirstAtom_iIndex, FirstAtom_jIndex, dab, lattice, Energy_AB):
    """
    This function calculates the difference in energy between a the swapped and unswapped state a
nd returns the energy change and the coordinates of the
    neighbouring atom

    Inputs
        FirstAtom_iIndex => This is the i index of the chosen atom (will be chosen randomly by th
e simulation)
        FirstAtom_jIndex => This is the j index of the chosen atom
        dab => Direction of swap (selected at random)
        lattice => numpy array representing all the atoms
```

```
        Energy_AB => Energy of bond between two unlike atoms

    Returns
        SecondAtom_iIndex => The i coordinate of the atom chosen in the respective direction
        SecondAtom_jIndex => The j coordinate of the atom chosen in the respective direction
        energy_change => The energy change as a result of a swap


    """
    temp_lattice = copy.deepcopy(lattice)
    # print("This is lat from function \n", lattice)


    energy_before  = 0
    energy_after = 0
    a = getNeighbour(FirstAtom_iIndex, FirstAtom_jIndex, dab)
    SecondAtom_iIndex = a[0]
    SecondAtom_jIndex = a[1]

    # calculating the energy before
    energy_site_one = EnergyCalc(FirstAtom_iIndex, FirstAtom_jIndex, Energy_AB, temp_lattice)
     # energy of chosen atom
    energy_site_two = EnergyCalc(SecondAtom_iIndex, SecondAtom_jIndex, Energy_AB, temp_lattice)
   # energy of neighbouring atom


    # swapping the atoms to see
    temp_lattice[FirstAtom_iIndex,FirstAtom_jIndex], temp_lattice[SecondAtom_iIndex, SecondAtom_j
Index] = temp_lattice[SecondAtom_iIndex, SecondAtom_jIndex], temp_lattice[FirstAtom_iIndex, First
Atom_jIndex]

    energy_site_oneSWAP =  EnergyCalc(FirstAtom_iIndex, FirstAtom_jIndex, Energy_AB, temp_lattice
)
    energy_site_twoSWAP =  EnergyCalc(SecondAtom_iIndex, SecondAtom_jIndex, Energy_AB, temp_latti
ce)


    energy_before = (energy_site_one + energy_site_two - Energy_AB)
    energy_after = (energy_site_oneSWAP + energy_site_twoSWAP - Energy_AB)
    # Calculate energy change
    energy_change = energy_after - energy_before

    return (SecondAtom_iIndex, SecondAtom_jIndex, energy_change)

#-------------------------------------------------------------------------------------------------
------------------------

def getNeighbour(Atom_iIndex, Atom_jIndex, d12):
    """
    Returns to the user the coordinates of the neighbouring atom to the atom refereced in the fun
ction call.

    Inputs
```

```
        Atom_iIndex => The i component of the atom chosen
        Atom_jIndex => The j componenet of the atom chosen

    Outputs
        NeighbourAtom_iIndex => The i compoenent of the atom chosen
        NeighbourAtom_jIndex => The j compoenent of the atom chosen

    """
    # inital empty array which will go on to contain the indicies of the chosen neighbour
    index = np.array([0,0])

    # Calculating the indicies of each possible neighbour for the atom inputed into the function
    a = np.array([Atom_iIndex, Atom_jIndex -1])
    b = np.array([Atom_iIndex - 1, Atom_jIndex])
    c = np.array([Atom_iIndex, Atom_jIndex + 1])
    d = np.array([Atom_iIndex + 1, Atom_jIndex])

    if d12 == "Alpha":
        index = a
    elif d12 == "Beta":
        index = b
    elif d12 == "Delta":
        index = c
    elif d12 == "Gamma":
        index = d

    NeighbourAtom_iIndex = index[0]
    NeighbourAtom_jIndex = index[1]

    return(NeighbourAtom_iIndex, NeighbourAtom_jIndex)
#-------------------------------------------------------------------------------------------------
-----------


def alloy2D(nBox, fAlloy, nSweeps, nEquil, Temp, Energy_AB, job):

    """
        This function generates the lattice, runs the monte carlo simulation and then subsequentl
y analyses the data

        Inputs
                nBox => Dimensions of the (square) lattice
                fAlloy => Fraction of alloying atoms present in the lattice
                nSweeps => The number of steps for the monte carlo simulation
                Temp => The temperature chosen to run the simulation at
                Energy_AB => The bond energy of unlike bonds
                Job => Job


        Outputs
                nBar => The average number of unlike neighbours
                Ebar => The average energy
                C => The heat capacity
```

```
"""
#################################
### GENERATE LATTICE PROPERLY ###
#################################
lattice = np.zeros((nBox, nBox))
count = 0
a = lattice.shape
num_atoms = a[0]* a[1]
while count < math.ceil(fAlloy*num_atoms):
    rand_row = randrange(0, a[0])
    rand_coloumn = randrange(0, a[1])
    if lattice[ rand_row, rand_coloumn] == 0:
        lattice[rand_row, rand_coloumn] = 1
        count = count + 1

#####################################
### APPLYING BOUNDARY CONDITIONS ###
#####################################
shp = np.array(lattice.shape)
top = lattice[0,:]   # coloumn to be reshaped
left = lattice[:,0]
bottom = lattice[shp[0] - 1,:]
right = lattice[:, shp[1] -1]   # coloumn to be reshaped

lattice = np.vstack((bottom, lattice))
lattice = np.vstack((lattice, top))

left = np.append(left, 0)
left = np.insert(left, 0, 0)
left = left.reshape(-1,1)

right = np.append(right, 0)
right = np.insert(right, 0, 0)
right = right.reshape(-1,1)

lattice = np.hstack((lattice,left))
lattice = np.hstack((right, lattice))

#################################
### ENERGY OF INITAL  LATTICE ###
#################################

initial_lat_energy = 0
b = lattice.shape

for i in range(b[0]):
    for j in range(b[1] - 1):
        if lattice[i,j] == lattice[i, j + 1]:
            initial_lat_energy = initial_lat_energy + 0
        else:
            initial_lat_energy = initial_lat_energy + Energy_AB

for k in range(b[0]-1):
```

```
        for l in range(b[1]):
            if lattice[k,l] == lattice[k + 1, l]:
                initial_lat_energy = initial_lat_energy + 0
            else:
                initial_lat_energy = initial_lat_energy + Energy_AB


    print("Initial energy of lattice is", initial_lat_energy)




    ##########################################
    ###  Carrying out the simulation to eqb ###
    ##########################################
    values = ["Alpha", "Beta", "Delta", "Gamma"]
    probability = [0.25, 0.25, 0.25, 0.25]
    energy = initial_lat_energy
    energy_list = []
    R = np.random.random_sample()

    for i in range(nEquil):


        shape_array = lattice.shape
        rand_row = randrange(2, shape_array[0]-
2)                                # Generate lattice point
        rand_column = randrange(2, shape_array[1]-
2)                           # Generate lattice point
        direction = np.random.choice(values, p = probability)                    # Generate random
 direction

        a = swapInfo(rand_row, rand_column, direction, lattice, Energy_AB)
        if a[2] <= 0:
            lattice[rand_row,rand_column], lattice[a[0], a[1]] = lattice[a[0], a[1]], lattice[ran
d_row, rand_column]
            energy = energy + a[2]
            energy_list.append(energy)
        elif a[2] > 0:
            R = np.random.random_sample()
            if(exp(-(a[2])/ ((0.00008617332)*(Temp)))) > R:
                lattice[rand_row,rand_column], lattice[a[0], a[1]] = lattice[a[0], a[1]], lattice
[rand_row, rand_column]
                energy = energy + a[2]
                energy_list.append(energy)


    eqb_lattice = lattice


    ########################################
    ####### Data analysis at eqb ###########
    ########################################
```

```
    distr_analysis = order2D(eqb_lattice)
    rand_dist = orderRandom(4, fAlloy)


    # Plot the configuration at eqb
    # CONFIG AT EQB
    config_plot = np.zeros((nBox+2, nBox+2))
    config_plot[0:nBox + 2, 0:nBox + 2] = lattice
    plt.figure(0)
    plt.pcolor(config_plot)
    plt.title('Configuration at equilibrium at Eam {}, Temp {}, and composition {}'.format(Energy
_AB, Temp, fAlloy))
    plt.savefig(str(job)+'Configuration at equilibrium at Eam {}, Temp {}, and composition {}.png
'.format(Energy_AB, Temp, fAlloy) )
    # plt.show()
    plt.clf()

    # ENERGY PLOT (TO EQB)
    iteration_list = list(range(nEquil))
    x  = np.linspace(0,len(energy_list),len(energy_list))
    y = energy_list
    plt.figure(1)
    plt.plot(x,y, color = "r")
    plt.title('Plot of energy at Eam {}, Temp {}, and composition {}'.format(Energy_AB, Temp, fAl
loy))
    plt.ylabel('Energy')
    plt.xlabel('Steps')
    plt.savefig(str(job)+ 'Plot of energy at Eam {}, Temp {}, and composition {}.png'.format(Ener
gy_AB, Temp, fAlloy))
    # plt.show()
    plt.clf()

    # DISTRIBUTION OF UNLIKE NEIGHBOURS
    num_unlike = ('0 unlike', '1 unlike', '2 unlike', '3 unlike', '4 unlike')
    y_pos = np.arange(len(num_unlike))
    plt.bar(y_pos + 0.00, distr_analysis[0], color= "k", width = 0.25, label = "neighbour distrib
ution at eqb" )
    plt.bar(y_pos + 0.25, rand_dist[0], color = "r", width = 0.25, label = "neighbour distributio
n for random lattice")
    plt.title("Distribution of unlike neighbours at Eam {}, Temp {}, and composition {}".format(E
nergy_AB, Temp, fAlloy), fontsize = 10)
    plt.xticks(y_pos, num_unlike)
    plt.legend(fontsize = 10)
    plt.savefig(str(job)+ "Distribution of unlike neighbours at Eam {}, Temp {}, and composition
{}.png".format(Energy_AB, Temp, fAlloy))
    # plt.show()
    plt.clf()

    ##############################
    ##### CONTINUING AT EQB ######
    ##############################

    Energy_eqb_lat = 0
```

```
    n_bar = 0
    Total_energy = 0
    Total_energy_squared = 0


    # WE LOOK TO OBTAIN:
                        # 1.) mean nBar
                        # 2.) Ebar
                        # 3.) C

    for i in range(nSweeps - nEquil):
        shape_array = eqb_lattice.shape
        rand_row = randrange(2, shape_array[0]-
2)                              # Generate lattice point
        rand_column = randrange(2, shape_array[1]-
2)                            # Generate lattice point
        direction = np.random.choice(values, p = probability)                    # Generate random
 direction

        a = swapInfo(rand_row, rand_column, direction, eqb_lattice, Energy_AB)
        if a[2] <= 0:
            eqb_lattice[rand_row,rand_column], eqb_lattice[a[0], a[1]] = eqb_lattice[a[0], a[1]],
 eqb_lattice[rand_row, rand_column]
            energy = energy + a[2]
            energy_list.append(energy)

        elif a[2] > 0:
            R = np.random.random_sample()

            if(exp(-(a[2])/ ((0.00008617332)*(Temp)))) > R:
                eqb_lattice[rand_row,rand_column], eqb_lattice[a[0], a[1]] = eqb_lattice[a[0], a[
1]], eqb_lattice[rand_row, rand_column]
                energy = energy + a[2]
                energy_list.append(energy)


        #Calculating the average value for nBar
        new_n_bar = order2D(eqb_lattice)
        n_bar = n_bar + new_n_bar[2]

        #Calculating Ebar (the average energy) [loop through each lattice site to calculate energ
y of system]
        for i in range(b[0]):
            for j in range(b[1] - 1):
                if eqb_lattice[i,j] == eqb_lattice[i, j + 1]:
                    Energy_eqb_lat = Energy_eqb_lat + 0
                else:
                    Energy_eqb_lat = Energy_eqb_lat + Energy_AB

        for k in range(b[0]-1):
            for l in range(b[1]):
                if eqb_lattice[k,l] == eqb_lattice[k + 1, l]:
                    Energy_eqb_lat = Energy_eqb_lat + 0
```

```
                else:
                    Energy_eqb_lat = Energy_eqb_lat + Energy_AB


        Total_energy = Total_energy + Energy_eqb_lat
        Total_energy_squared = Total_energy_squared + Energy_eqb_lat**2
        Energy_eqb_lat = 0

    mean_n_bar = n_bar/(nSweeps - nEquil)
    Ebar = Total_energy/(nSweeps - nEquil)      # mean of energy
    E2bar = Total_energy_squared/(nSweeps - nEquil)
    print("No. of eqb sweeps", nSweeps- nEquil)
    print("the total energy is", Total_energy)
    print("the total energy squared is", Total_energy_squared)
    print("the mean energy is", Ebar)
    print("the mean total energy squared is", E2bar)
    C = (E2bar - Ebar*Ebar)/((Temp**2)*(0.0008617332))
    print("the heat capacity is", C)

    # return(eqb_lattice, distr_analysis[2], mean_n_bar,Temp)
    return(mean_n_bar, Ebar, C)

#-------------------------------------------------------------------------------------------------
-------------------------------------------------------

######################
### MAIN FUNCTION ###
######################


def main():

    # SIMULATION PARAMETERS
    nBox = 10
    nEquil = 20000
    nSweeps = 75000
    fAlloy_list = [0.1, 0.2, 0.3, 0.4, 0.5]
    T_list = [300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700,
1800 , 1900, 2000,2100,2200,2300,2400,2500,2600,2700,2800,2900,3000, 3100, 3200, 3300, 3400, 3500
, 3600, 3700, 3800, 3900, 4000, 4300, 4600, 4900, 5100 ]
    Eam_list = [0.1, 0.0, -0.1]

    # Open file to save data
    file = open("data.csv", "w")
    file.write('Job number, Alloy fraction, Temperature(K), Unlike bond energy (eV), Average numb
er of unlike neighbours, Average energy (eV), Heat capacity (kB) \n')

    #Looping over values
    count = 0
    for fAlloy in fAlloy_list:
        for Temp in T_list:
            for Energy_AB in Eam_list:
                count = count + 1
```

```
                job = '{:04d}'.format(count)

                # Echos parameters back to the user
                print("")
                print("Simulation", job)
                print("----------------------")
                print("Cell size                =", nBox)
                print("Alloy fraction           =", fAlloy)
                print("Total number of moves    =", nSweeps)
                print("Number of equilibration moves  =", nEquil)
                print("Temperature                    =", Temp, "K")
                print("Bond energy                    =", Energy_AB, "eV")

                # Run the simulation
                mean_n_bar, Ebar, C = alloy2D(nBox, fAlloy, nSweeps, nEquil, Temp, Energy_AB, job
)
                # Write out the statistics
                file.write('{0:4d}, {1:6.4f}, {2:8.2f}, {3:5.2f}, {4:6.4f}, {5:14.7g}, {6:14.7g}
\n'.format(count, fAlloy, Temp, Energy_AB, mean_n_bar, Ebar, C))

    # close the file
    file.close()

    # sign off
    print('')
    print('Simulations completed')

# Ensure main is invoked
if __name__ == "__main__":
    main()
```

Copy of unit testing code

```
import unittest
import long_lab
import numpy as np


class TestLongLab(unittest.TestCase):
    # def test_orderRandom(self):
    #     a = [0.23828125, 0.328125, 0.2109375, 0.140625, 0.08203125]
    #     b = long_lab.orderRandom(4,0.25)
    #     self.assertEqual(b[0], 0.23828125 )
    #     self.assertEqual(b[1], 0.328125 )
    #     self.assertEqual(b[2], 0.2109375)
    #     self.assertEqual(b[3], 0.140625)
    #     self.assertEqual(b[4], 0.08203125)
```

```
    def test_order2D(self):
        try_lattice = np.array([[0,1,0,1], [1,1,0,1], [0,1,0,1], [0,0,1,0]])
        b = long_lab.order2D(try_lattice)
        self.assertEqual(b[2], 2.25)

    def test_EnergyCalc(self):
        try_lattice = np.array([[0,1,0], [0,0,1], [0,1,0]])
        b = long_lab.EnergyCalc(1,1, 0.1,try_lattice)
        self.assertAlmostEqual(b, 0.3)

    def test_getNeighbour(self):
        try_lattice = np.array([[0,1,0], [5,0,1], [0,1,0]])
        b = long_lab.getNeighbour(1,1,"Alpha")
        c = try_lattice[b[0], b[1]]
        self.assertEqual(c , 5)

    def test_swapInfo(self):
        try_lattice = np.array([[0,1,1,1,0], [0,1,0,1,0], [1,0,0,1,0]])
        Energy_AB = 0.1
        b = long_lab.swapInfo(1,2,"Alpha", try_lattice, Energy_AB)
        self.assertAlmostEqual(b[2], -0.2)

    def test_alloy2D(self)
        resultArray = np.array([[1,0,1,0],[0,1,0,1],[1,0,1,0],[0,1,0,1]])
        Energy_AB = 0.1
        b = long_lab.alloy2D(10,0.1,2,2,300,1)
        self.assertAlmostEqual(b[2], 5.7)


if __name__ == '__main__':
    unittest.main()
```

output of unit testing

```
PS C:\Users\xamir\Documents\Imperial\(year 2)\Labs\LongLab\longlabformatted> & C:/Users/xamir/AppData/Local/Programs/Python/Python37-32/python.exe "c:/Users/xamir/Documents/Imperial/(year 2)/Labs/LongLab/longlabformatted/test_long_lab.py"
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```