Binary Alloy Thermodynamic Simulation: Method and details of program operation

Tom Walton

March 3, 2017

1 Introduction

Monte Carlo methods employ probability distributions to obtain numerical results to problems that are either impossible, or very difficult, to solve analytically.[1] More specifically, accurately simulating how atoms in a binary alloy (in the gaseous state) arrange themselves would be very challenging. However, by making several simplifying assumptions one can obtain a reasonable approximation of a real-world system using the Monte Carlo Method. These assumptions include like atoms have no interaction energy, atoms can only swap with their nearest neighbours and considering solely the thermodynamics of the system, rather than the kinetics. Therefore, for this simulation, we are sampling from the canonical ensemble in order to produce equilibrium atomic arrangements, with the driving force being minimisation of Gibbs free energy.[2]

This simulation focuses on how varying the system composition, temperature and bond energy between unlike atoms (E_{am}) affects the arrangement of atoms in a binary alloy after reaching its state of lowest energy. One key result is how like atoms arrange themselves at the three different E_{am} values tested (-0.1, 0.0, 0.1 eV) in order to reach thermal equilibrium. It is expected that the non-zero E_{am} cases will either prefer like or dislike neighbours, while the zero case shouldn't exhibit a preference. Furthermore, the effect of increasing temperature on these arrangements is also an important result. It is expected that atoms will become more randomly distributed at higher temperatures due to greater thermal energy and so the approximate temperature at which this random distribution dominates (compared to a more ordered distribution at lower temperatures) is of key interest.

2 Method

The Metropolis Monte Carlo method is an algorithm for generating N configurations of a system (c_1, c_2,c_3,...,c_N) according to a probability distribution, such as the Boltzmann distribution. This is described by equation 1, where N_c is the number of c configurations, N is the total number of configurations and P(c) is the probability of configuration c according to a given probability distribution.

$$\lim_{N\to\infty} \frac{N_c}{N} = P(c) \tag{1}$$

In terms of actually implementing this method, step 1 would involve choosing a (random) configuration of the system (c_1). Step 2 is to choose a trial configuration c_t, typically one than is similar to c_n, and calculate the ratio of the probability of this trial configuration compared to the probability of the current configuration (c_n), as per equation 2. Then pick a random number between 0 and 1. If R is greater than or equal to this random number, set c_{n+1} equal to c_t; otherwise set c_{n+1} equal to c_n (i.e. no change in system configuration, as unfavourable according to given probability distribution). Step 3 is to repeat step 2, replacing c_n with c_{n+1}. Step 2 should be repeated in this manner N times, where N is a sufficiently large number for the system to converge to a certain configuration, if possible.[3] In this way the system goes towards the most likely configuration as the simulation progresses.

$$R = \frac{P(c_t)}{P(c_n)} \tag{2}$$

The specific implementation of the Monte Carlo method used involved setting up a random configuration of atoms for step 1, with fractions of host and alloy atoms as specified. For step 2, a random pair of nearest neighbour atoms were chosen and the energy change (ΔE) of the system when these atoms were swapped was calculated. For this simulation, nearest neighbour atoms were defined to be atoms that share a face (i.e. not nearest neighbour if only share a corner or edge). If this change in energy is negative then the atoms are swapped, as this move lowers the overall energy of the system and so is more

energetically favourable. If the change in energy is greater than or equal to 0 then the effect of thermal energy is considered. This meant that the atoms are only swapped if $exp(\frac{-\Delta E}{k_B T}) > R$, where k_B is the Boltzmann constant, T is the system temperature and R is a random number between 0 and 1. Step 2 is repeated N times, where N is specified at the program start. N is chosen so that it is sufficiently large for the order value, which quantifies the order of the system, to converge. In order to account for edge effects, the boundaries were assumed to be periodic, meaning that the grid repeats itself on all sides.

The input parameters varies were the size of the grid (gridLength), the fraction of alloying atoms present in the system (alloyFraction), N, T, and the bond energy between an alloying atom and a host matrix atom (E_{am}). The bond energy between like matrix atoms (E_{mm}) and the between like alloying atoms (E_{aa}) are both kept constant at 0 eV, while the allowed values for E_{am} are -0.1, 0.0 and 0.1 eV. Therefore the simulation can provide information on how changing alloyFraction and T affects the configuration of the system, after N iterations. Furthermore, the effect of changing E_{am} on these relationships can be investigated, where alloy-matrix bonds are considered either favourable, neither favourable nor unfavourable or unfavourable (-0.1, 0.0 and 0.1 eV respectively).

3 Program Details

3.1 2D Program

Firstly, the initializeGrid() function creates a square matrix of size as specified by the gridLength parameter. The square matrix contains 0's and 1's to represent matrix and alloying atoms respectively, which are randomly distributed and the concentration of alloying atoms is as specified by the alloyFraction parameter. Next, step 2 is carried out N times by executing the runSim() function. Within this function, firstly a random atom is chosen and then one of its nearest neighbours is randomly chosen, via the randomAtomPairChooser() function. It should be noted that whenever the coordinate of an atom being considered could potentially be outside of the grid (e.g. when choosing a nearest neighbour atom), the cValidate() function is run on that coordinate, which adjusts the coordinate value to be inside of the grid, according to the periodic boundaries stipulation.

With a random pair of atoms now chosen, the total energy of the bonds between the selected atoms and their nearest neighbours is calculated using the localEnergyCount() function, before running this function again, but with the positions of the pair of atoms swapped. The localEnergyCount() function works by summing up the number of unlike bonds in this local area, and then multiplying this value by the bond energy (E_{am}). The difference between the two values returned by localEnergyCount() is the energy change of the system (deltaE). The deltaE value is then passed into the energyAct() function, which carries out the swapping of position of the pair of nearest neighbour atoms using the performSwap() function if deltaE is negative or if $exp(\frac{-\delta E}{k_B T}) > R$, where R is a random number between 0 and 1. Now 1 iteration of step 2 has been carried out and so the above sequence is repeated N times, until a final configuration is reached.

Now that a final arrangement of atoms has been reached, the getOrder() function is run on this configuration in order to determine a value that quantifies the order of the system, with a higher value corresponding to a more ordered system. The getOrder() function works by first iterating over every atom in the grid. For each atom the number of its unlike neighbours is calculated using the unlikeNeighbourCount() function. This function works by considering all of the nearest neighbour atoms around the selected atom, and counting the number of these atoms which are different to the selected atom, then returning this value. In this way the number of atoms in the grid with 0, 1, 2, 3 & 4 unlike neighbours is determined and stored in a list called nList. Next, the number of atoms with 0, 1, 2, 3 & 4 unlike neighbours is calculated for the binomial distribution and stored in a list called EXPList. This is done by inputting n values from 0 to 4 into the binProb() function. This function calculates the probability that a given atom has n unlike neighbours using the binomial distribution equation (equation 3), where Z is the number of neighbouring sites (4 for the 2D case), f is equal to alloyFraction, n is the number of unlike neighbours (i.e. desired outcome) and ${}^{Z}C_{n}$ is the number of ways to choose n items from a collection of Z items. The function then returns the expected number of atoms with n unlike bonds by multiplying the calculated probability by the total number of atoms in the system. Lastly, the order value is calculated by summing up the magnitudes of the differences between corresponding values in nList and EXPList.

$$P_n =^{Z} C_n[f f^{Z-n}(1-f)^n + (1-f)f^n(1-f)^{Z-n}] \tag{3}$$

$${}^{Z}C_n = \frac{Z!}{n!(Z-n)!}$$

Extensive test functions were written to ensure that each individual component (i.e. function) of the system performed its role correctly, such that when combined the simulation runs as it is supposed to. The initializeGridTest() function checks that initializeGrid() produces a random arrangement of matrix and alloying atoms by displaying the calculated order value of the grid produced (which should be low) as well as displaying the matrix for visual inspection. The cValidate() function

is tested by calling the cValidateTest() function, which passes coordinates into the cValidate() function which are outside of the grid (by 1 lattice position) and checks that the coordinate values are changed to the correct value inside of the grid. The performSwapTest() functions check that performSwap() correctly swaps the atoms by considering a specific grid configuration and checking that the position of the two atoms to be swapped does indeed change. The getTotalEnergy() function calculates the number of unlike bonds in the system by taking the number of 0, 1, 2, 3 & 4 unlike neighbours, then multiplying each element in this list by 0.5 and its index, before summing up these elements. This value is then multiplied by E_{am}. To test this function, getTotalEnergyTest() takes a specific case of a 3x3 grid where the total energy has been worked out by hand, then checks that the getTotalEnergy() function computes the correct answer. The localEnergyCountTest() function tests that localEnergyCount() computes the correct answer in a similar way, using a specific case worked out by hand. However, it also checks that the change in energy value produced using localEnergyCount() is equal to the value produced via getTotalEnergy(), ensuring that both functions are consistent with each other.

The choosing of random pair of nearest neighbour atoms is tested by running the randomAtomPairChooserTest(), which runs the randomAtomPairChooser() function on a matrix of 0's and displays the selected atoms as 1's, so that it can be confirmed visually that they are indeed nearest neighbours. Lastly, the getOrderTest() function produces a random grid and an ordered grid (2 blocks, one of 1's, one of 0's). It then runs the getOrder() function on these grids and checks that the random and ordered grids produce a low and high order value respectively. This extensive unit testing ensures that all components of the system work well independently, however integration testing was also carried out throughout the development process. This done simply by running functions (that depend on other functions) and ensuring that the outcomes are as expected.

3.2 3D Program

The first alteration to the 2D program required to make it work in 3D is in setting up the initial grid. The grid length must be multiplied by 3 instead of 2, in order to add another dimension to the system. Furthermore, when populating this grid with alloying atoms, and indeed any time atoms in this grid are referenced, a z coordinate must be introduced. The cValidate() function works by accepting which type of coordinate has changed (e.g. x, y) and so now a case for when the z-coordinate changes must be added. The localEnergyCount() function must be altered to account for the nearest neighbours above and below the plane of the pair of selected atoms, thereby adding an extra 2 atoms that need to be considered every time this function is run.

Additionally, when performing a swap and when choosing a random pair of atoms, the z-coordinate must be taken into account, adding another dimension to which atoms may be selected. For the binomial distribution formula, the Z value must be changed from 4 to 6, as there are no 6 nearest neighbours per atom. When iterating over each atom in functions such as unlikeNeighbourCount() and getOrder() an extra for loop must be added in order to iterate over the z coordinate, which adds a significant amount of processing time for that function. Furthermore, the list containing frequency of each number of unlike bonds, for both the binomial distribution case and the actual simulation case, must now range from 0 to 6, rather than from 0 to 4, to account for the fact that 6 is now the maximum number of unlike neighbours that an atom may have.

4 References

[1] http://dx.doi.org/10.1002/wics.1314 - 28/02/2017; 'Why the Monte Carlo method is so important today', Kroese, Dirk P., Brereton, Tim, Taimre, Thomas, Botev, Zdravko I., John Wiley & Sons, Inc.
[2] http://micro.stanford.edu/ caiwei/me334/Chap8_Canonical_Ensemble_v04.pdf - 28/02/2017
[3] http://xbeams.chem.yale.edu/ batista/vaa/node42.html - 25/02/2017