H2a: Binary Alloy

Tomas Lundberg and Jonas H. Fritz

December 8, 2021

Task №	Points	Avail. points
$\sum$		

Introduction

In this project we investigate different properties, namely the order parameter P, internal energy U and heat capacity C of a 3 dimensional two species CuZn Ising model lattice. First we consider the mean field solution, for which there are analytical quantities for these expressions, except for the order parameter P, which has to be found numerically from an implicit equation. In the second part of the project, we simulate a full 10x10x10 cube of bcc unit cells, using the metropolis algorithm. Here we also consider the short range order parameter parameter r

Problem 1

The bcc lattice on which the two species are arranged can be thought of as two intertwined sc sublattices, which we call a and b sublattice. These are filled with $N = 1000$ number of atoms of species A (Cu) and B (Zn) respectively, in a perfectly ordered state. The long range order parameter P quantifies how ordered the system is. It is defined through

$$\text{Number of atoms on the } a \text{ sublattice} = 1/2(1+P)N \tag{1}$$

We begin by considering the mean field solution, in which the mean field energy E_{MF}, which corresponds to internal energy U, is given by [1]

$$U = E_{\mathrm{MF}} = E_0 - 2NP^2\Delta E\,, \tag{2}$$

with

$$\begin{aligned} E_0 &= 2N(E_{\mathrm{AA}} + E_{\mathrm{BB}} + 2E_{\mathrm{AB}}) \\ \Delta E &= E_{\mathrm{AA}} + E_{\mathrm{BB}} - 2E_{\mathrm{AB}}. \end{aligned} \tag{3}$$

Here we take the values to be $E_{\mathrm{AA}} = E_{\mathrm{CuCu}} = -436$ meV, $E_{\mathrm{BB}} = E_{\mathrm{ZnZn}} = -113$ meV and $E_{\mathrm{AB}} = E_{\mathrm{CuZn}} = -294$ meV.

The entropy follows from the number of different configurations as

$$S = 2Nk_B \ln 2 - Nk_B \left[(1+P)\ln(1+P) + (1-P)\ln(1-P)\right]\,. \tag{4}$$

We can then derive the free energy as $F = U - TS$ and differentiate that with respect to the order parameter P to obtain the equilibrium relation between T and P

$$4P\Delta E - Nk_BT\ln\left(\frac{1+P}{1-P}\right) = 0\,. \tag{5}$$

We then proceed to solve this equation numerically, by looking for the value of P that minimizes the above expression for a fixed value of T. With this we obtain the relation $P(T)$ shown in fig. 1. For negative values of P the curve is flipped along the x-axis. We find a critical temperature of 905.15 K. In the mean field theory this can be also calculated analytically as

$$T_c = 2\Delta E/k_B\,, \tag{6}$$

which, as expected, yields the same result. However, when compared to the real critical temperature of $T_c^{\mathrm{true}} = 741$ K, the mean field theory overestimates the critical temperature. We can plug $P(T)$ into eq. (2) to obtain the temperature dependence of $U(T)$. The heat capacity is obtained from numeric differentiation of $U(T)$ using Newton's difference quotient. These are also shown in fig. 1. We see that $P(T) = 1$ for $T = 0$, i.e. the system is in a perfectly ordered state. As temperature increases, the order decreases, until finally the system reaches a completely disordered state at $T = T_c$. We find the corresponding behavior for U, which increases until it reaches a plateau at the disordered state, which is the state with maximum energy. At this point the system cannot increase its energy any further. The heat capacity is the derivative of U with respect to temperature, and drops to 0 above T_c accordingly. Taking into account the limitations of the model, the results the mean field theory predicts are reasonable and correspond to our expectations. An increasing heat capacity is also found in real solids. The drop of heat capacity to 0 above the critical temperature may not be physical, however, this is due to the model not including kinetic energy and not a result of the mean field approximation.

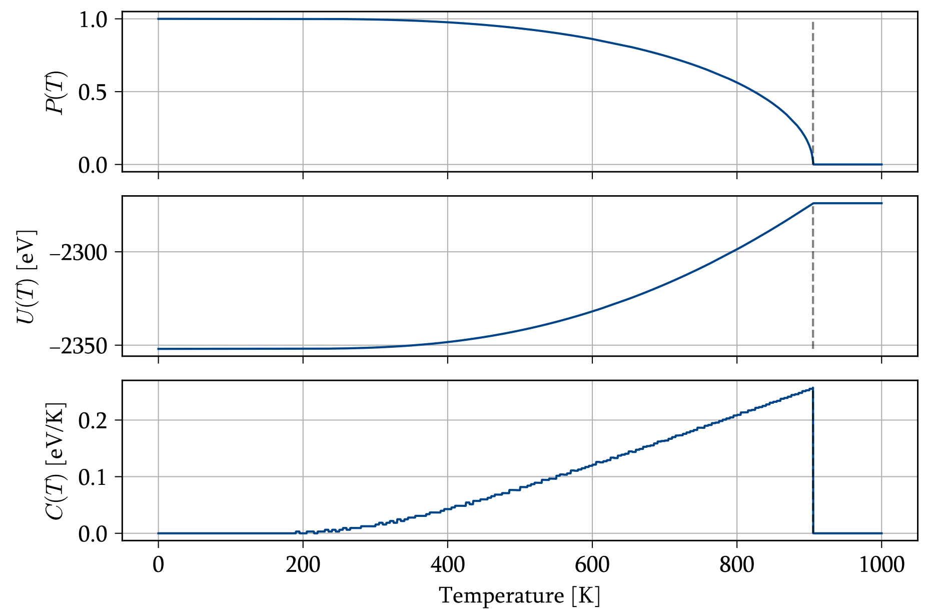

Figure 1: Order parameter P (top), energy U (middle) and heat capacity C (bottom) in dependence of temperature T. At $T = 0$ the order parameter is 1, corresponding to perfect order. As temperature increases, P decreases, until we reach a completely disordered state at $T > T_c$ with $P = 0$. This is reflected in the increase of U, which reaches a plateau, since the system is its maximum energy state. Heat capacity drops to 0, as it is the temperature derivative of internal energy.

Problem 2

We now model our two-species lattice as a $10 \times 10 \times 10$ bcc with periodic boundary conditions. Furthermore, we split up the bcc lattice into two sc lattices of $N = 1000$ atoms each (just as before). One of them are initialized with only A-type atoms (Cu) and the other with only B-type atoms (Zn) (this is equivalent to a cold start). To keep track of the different types of bonds we create two 1000×8 matrices where each position $0, 1, 2, \ldots 999$ corresponds to a position i, j, k with $i, j, k \in 0, 1, 2, \ldots, 9$ in one of the sc lattices. The index going from $0, 1, 2, \ldots, 7$ holds the type of atom at that position in the other sc lattice, i.e what types of atoms it has as nearest neighbours. We pick a random atom in one of the lattices and a random direction. We proceed with swapping these atoms if they are of different types (otherwise we pick two new random numbers) and calculate the energy difference

$$\Delta E = \Delta N_{AA} E_{AA} + \Delta N_{BB} E_{BB} + \Delta N_{AB} E_{AB} \,, \tag{7}$$

where ΔN is the difference in number of bonds of a specific type between the two configurations. We then generate a random number ξ from 0 to 1 and accept the trial change if

$$\xi < \exp\left(-\Delta E / k_B T\right). \tag{8}$$

If not, we count the previous configuration once more. In order to let the system reach equilibrium we run for $N_{eq} = 10^6$ iterations for all temperatures and then let it run for 10^7 iterations, from which we calculate average properties.

In this problem we also consider the short range order parameter r, which is defined as

$$r = \frac{1}{4N}(q - 4N) \,, \tag{9}$$

where q is the number of AB nearest neighbour bonds.

In fig. 2 we show the time evolution of energy U at 6 different temperatures between 500 K and 1000 K. The dashed line indicates the end of the equilibration run. We see that simulations at lower temperatures reach equilibrium later, while all the systems reach equilibrium after 10^6 iterations.

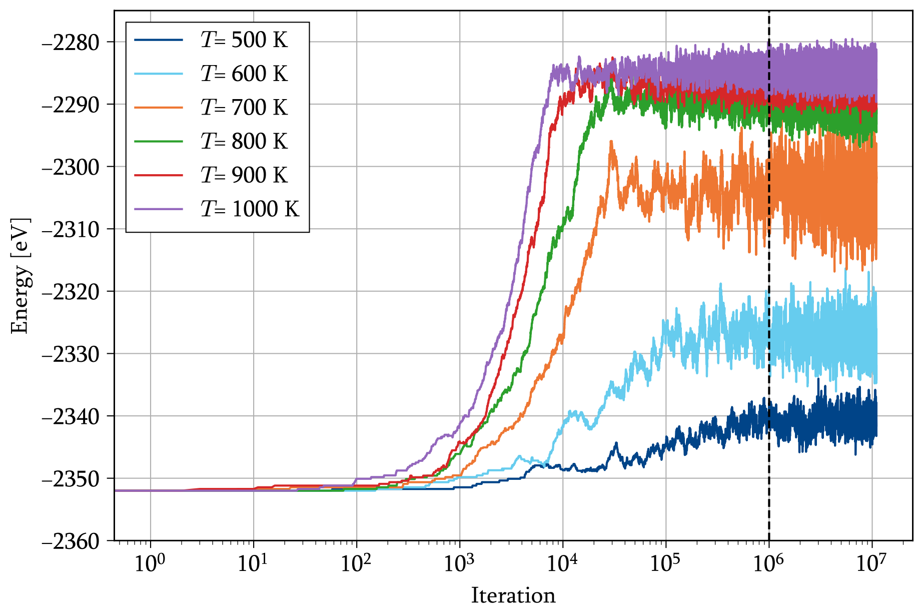

Figure 2: The energy U for six temperatures between 500 K and 1000 K as a function of iterations in the Metropolis algorithm. The vertical black dashed line represents the end of the equilibration phase. We see that the lowest temperature of 500 K equilibrates the slowest but has still reached equilibrium at 10^6 iterations.

Doing runs like this for every 10 K between 500 and 1000 K and taking the average of U, r and P yields the results shown in fig. 3. We also calculate the heat capacity $C(T)$ according to

$$C(T) = \frac{1}{k_B T^2}(\langle U^2 \rangle - \langle U \rangle^2). \tag{10}$$

The same quantities calculated with the mean field solution in Problem 1, as shown in fig. 1 are displayed in dashed black in fig. 3. The energy U, the short range parameter r and the long range parameter P are displayed with $10\sigma^2$ error bars calculated from the statistical inefficiency parameter s through

$$\mathrm{Var}[\ \cdot\ ] = \frac{s}{N}\mathrm{Var}[\text{Metropolis chain of } \cdot\ ], \tag{11}$$

where $N = 10^7$ in our case. The calculation of s is explained in the next paragraph. Looking at the energy evolution, we can note that it is rather similar in the beginning and the end of the temperature interval where both the shape of the curve and the values are similar. In between about 600 and 900 however it differs. This difference in the same interval is also present for $C(T)$ and $P(T)$. In the problem description it is says that at 741 K there is a phase shift between the ordered and the disordered bcc structure. We can see signs of this phase shift most prominently for the order parameter $P(T)$, at about 740 K it becomes 0, i.e it describes a completely disordered system. This phase transition does not show up in the mean field approximation until $T_c = 905$ K. We can also note that the heat capacity increases faster than the one obtained through the mean field approximation and drops sharply just before the phase transitions upon which it slowly goes down to 0. We can also note that $C(T)$ is not entirely smooth but instead varies quite a lot while reaching the maximum. This could be a sign that we have not used a long enough production.

To estimate the statistical inefficiency parameter s for the different runs we use both the auto correlation method and the block averaging method. The former method relies on the assumption that the auto correlation function decays as an exponential. s is then taken as the point where the correlation function has decayed to $\exp(-2)$. The correlation functions for different temperatures and for both U, r and P are shown in fig. 4. We see that for $T = 800$ K and $T = 1000$ K the correlation function decays nicely, whereas at $T = 700$ K the correlation function of the order parameter P takes much longer to decay and does not reach $\exp(-2)$ even for 500,000 steps. For $T = 500$ K the

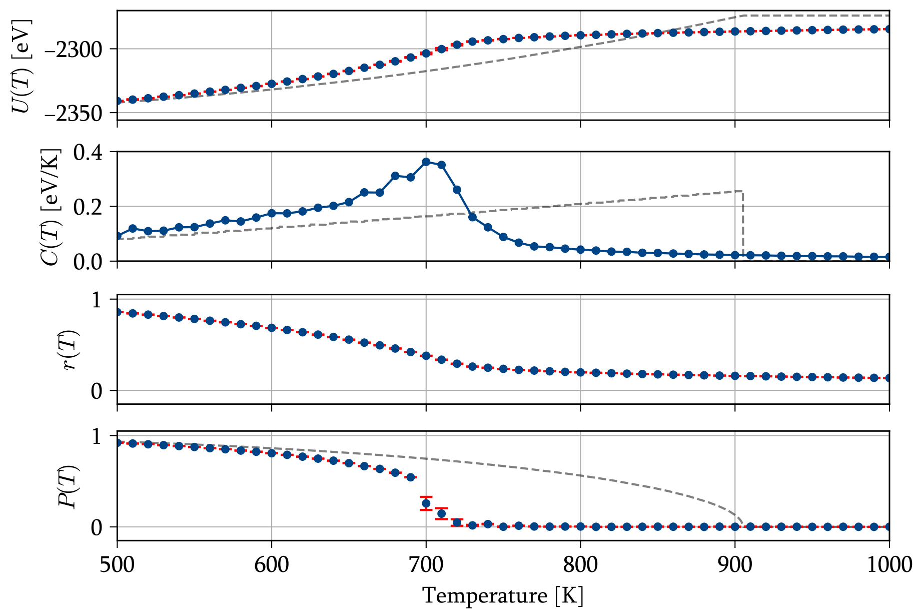

Figure 3: (blue markers with red error bars) The energy $U(T)$ (top), the heat capacity $C(T)$ (second, no error bars), the short range parameter r (third) as well as the long range order parameter P (bottom) for 51 runs at equidistant temperatures between 500 K and 1000 K. The error bars represent $10\sigma^2$ calculated for U, r and P through eq. (11). The dashed lines in black shows the corresponding quantities given from the mean field approximation. We see that the errors are very small except perhaps for P just before the phase transition, see right axis in fig. 6.

correlation functions does only roughly follow an exponential decay. We suspect that the former is a characteristic of the phase transition, while the latter is due to insufficient production steps. We did not let it run for longer due to the large computational cost and s is in these cases taken to be 500,000.

We also calculate the correlation length using the block method. For this, we divide the metropolis chain of size M into M_B blocks of size B and average an observable f along the metropolis chain in each block according too

$$F_j = \frac{1}{B}\sum_{i=1}^{B} f_{i+(j-1)B} \quad \text{for} \quad j = 1\ldots M_B \tag{12}$$

The statistical inefficiency is then given by

$$s = \frac{B\mathrm{var}\left[F\right]}{\mathrm{var}\left[f\right]}, \tag{13}$$

for large B. The statistical inefficiency (correlation length) is shown in fig. 5 as function of block size at different temperatures. For $T = 700$ K the correlation function of P does not converge. As in fig. 4 we suspect that this a property of the phase transition. The other functions seem to fluctuate around a constant value. the statistical inefficiency for U, r and P derived using both methods are shown for all 51 temperatures between 500 K and 1000 K in fig. 6 (left axis). The corresponding variance of the quantities through eq. (11) are also displayed (right axis). In calculating the variance we used the average of s obtained from both methods. We can note that the variance is very small for all parameters which is why we choose to display the rather unconventional $10\sigma^2$ in fig. 3. This is however reasonable since we use a production of N^7 steps, which is much larger than the order of s and that the variance of the metropolis chains are not too large, as can be seen for U in fig. 2. With this said it is nevertheless true that both s and the variance peak close to the phase transition at 740 K. We believe that these quantities diverge at the phase transition as can be hinted in fig. 4 and fig. 5, but lack the computational power to sample more temperatures with longer runtimes at criticality. We can also see that s and the variance are larger for lower temperatures which is probably because

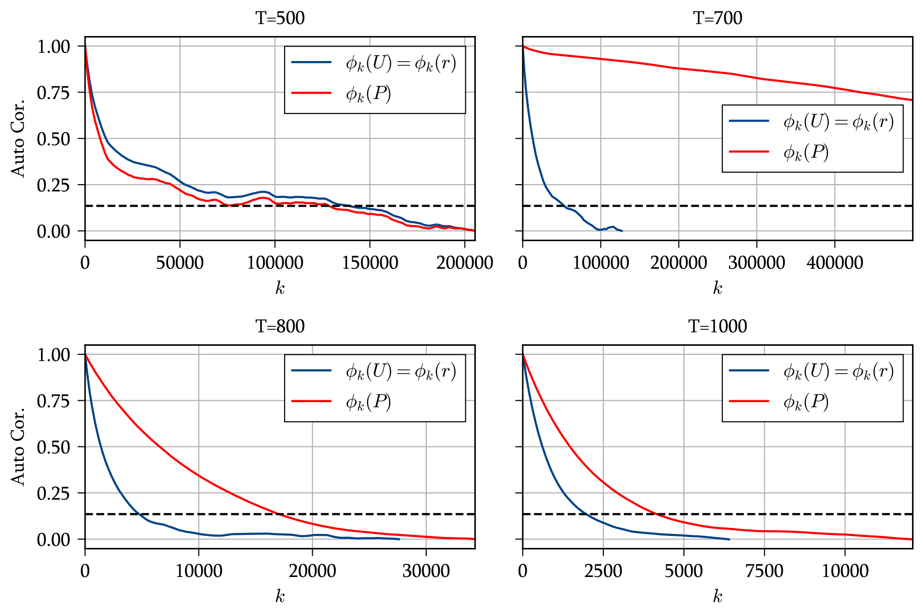

Figure 4: The auto correlation function for the energy U, short range parameter r and long range parameter P at four arbitrary temperatures between 500 K and 1000 K. The auto correlation for U and r are the same and are both represented by the blue line. The dashed black line is $\exp(-2)$ at which the statistical inefficiency parameter s equals k. Note how the auto correlation function for P at 700 K does not decay to $\exp(-2)$ at $k = 500,000$. This is probably because it is close to the phase transition at 741 K.

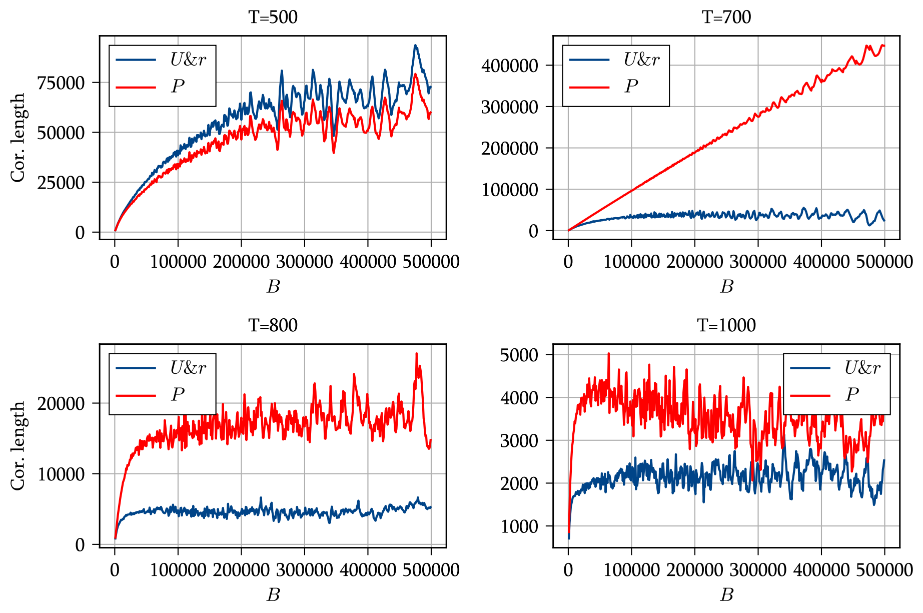

Figure 5: The correlation length for different block sizes B for the energy U, short range parameter r and long range parameter P at four temperatures between 500 K and 1000 K. The auto correlation for U and r are the same and are both represented by the blue line. The correlation length at which the lines converge to are taken as the statistical inefficiency parameter s. Note how the correlation length for P at 700 K does not converge even for $B = 500,000$. This is probably because it is close to the phase transition at 741 K.

lower temperatures correspond to lower acceptance ratios according to eq. (8) and thus the chains require more production steps in order to fully represent the distribution. This can easily be understood when considering the auto correlation functions. If the

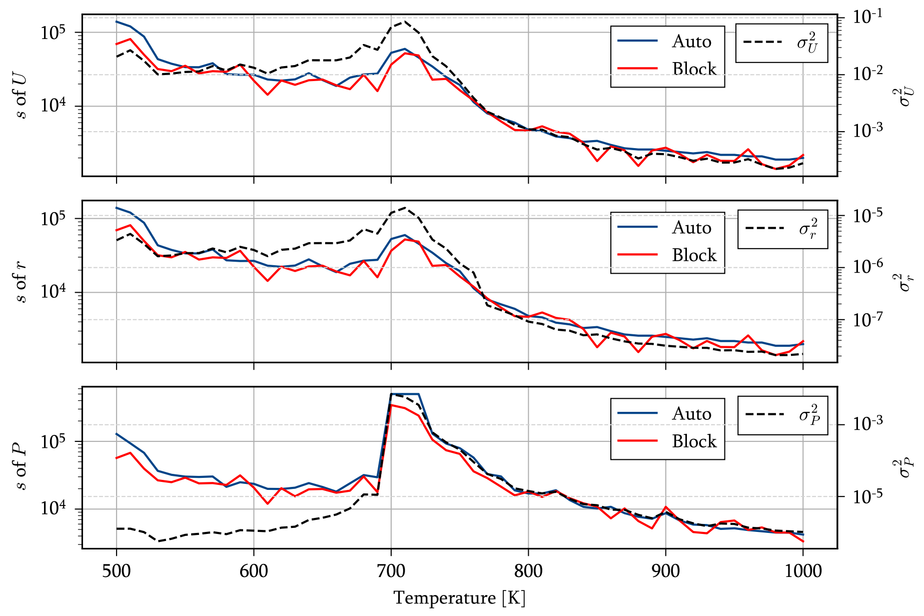

Figure 6: (left axis) The statistical inefficiency parameter s for the energy U (top), short range parameter r (middle) and long range parameter P (bottom) as a function of temperature derived from the auto correlation (Blue) and block (red) methods depicted in fig. 4 and fig. 5 respectively. (right axis) The black dashed line is the variance of U, r and P computed through eq. (11).

acceptance ratio is very small the correlation in the chain will of course go up since we to a higher degree save the same configuration as for the previous step.

In order to improve the results in this problem we would use longer production runs for the lower temperatures, the need for which is most prominent when looking at the auto correlation functions for low temperatures. Furthermore, we would allocate more computational resources in computing the statistical inefficiency for temperatures close to the phase transition. Finally, we would use more samples close to the critical temperature in order to more fully capture the behaviour of the phase transition.

Concluding discussion

In this report we have studied a two species CuZn bcc lattice. We have in particular looked at the temperature dependence of the energy, heat capacity and order parameters. We have done this using two different approaches; the mean field approximation and Monte Carlo simulations with the Metropolis algorithm for a $10 \times 10 \times 10$ bcc lattice. We find that the mean field solution can not accurately represent the phase transition, as it yields a too high critical temperature. This is to be expected however, since the mean-field assumption does not hold true near criticality. It is however much simpler and faster and for studies at temperatures that are not too close to the phase transition it gives accurate results. The Monte Carlo simulated system closely resembles how a physical system works in reality where atoms can switch positions if it is energetically likely. This lets the model capture the phase transitions much more accurately.

References

[1] Charles Kittel and Donald F. Holcomb. "Introduction to Solid State Physics". In: *American Journal of Physics* 35.6 (June 1967), pp. 547–548. DOI: 10.1119/1.1974177. URL: https://doi.org/10.1119/1.1974177.

A Source Code

In this appendix we display the source code used in obtaining the results presented in this report.

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <gsl/gsl_rng.h>

#define PI 3.1415926535
#define KB (8.61733326*pow(10,-5))
#define DELTA_E -0.039
#define E_AA -0.436
#define E_BB -0.113
#define E_AB -0.294
#define N_atoms 1000

// Functions for Task 1
double Temp(double P){
    /*Returns temperature given long range order P*/
    return -4*P*DELTA_E/(KB*log((1+P)/(1-P)));
}

double U(double P){
    /*Returns energy given long range order P*/
    return 2 * N_atoms * (E_AA+E_BB+2*E_AB) + 2*N_atoms*pow(P,2)*DELTA_E;
}

double C(double U_last, double U_now, double delta_T){
    /*Returns heat capacity as the numerical derivative of the energy U*/
    return (U_now - U_last)/delta_T;
}

// Functions for Task 2
void i_j_k_from_state(int state, int coord[3]){
    /* sets i,j,k coordinates from state 0,1,2,...,999*/
    coord[0]=state/100; //i
    coord[1]=(state-coord[0]*100)/10; //j
    coord[2]=(state-coord[0]*100-coord[1]*10); //k
}

int state_from_i_j_k(int coord[3]){
    /* Returns state from i,j,k coordinates. Takes care of periodic boundary ↩
        conditions*/
    for (int i=0;i<3;i++){
        if (coord[i] > 9)
        {
            coord[i]=0;
        }
        if (coord[i] < 0)
        {
            coord[i]=9;
        }
    }
    return coord[0]*100 + coord[1]*10 + coord[2];
}

double short_range_order(double q){
    /* Calculates short range order r given parameter q (N_ab bonds)*/
    return 1.0/(4*N_atoms)*(q-4*N_atoms);
}

double long_range_order(double N_a){
    /* Returns long range order P given number of A atoms on A lattice */
    return N_a*2.0/N_atoms - 1.0;
}

double heat_capacity(double E[], int len, double T){
    /* returns the heat capacity given energy vector of length len and ↩
        temperature T */
    double mean = 0;
    double mean_2 =0;
    for (int i=0; i<len;i++){
        mean += 1.0/len*E[i];
        mean_2 += 1.0/len*E[i]*E[i];
    }
    return 1.0/(KB*T*T)*(mean_2-mean*mean);
}

double get_mean(double v[], int len){
    /* returns the mean of vector v of length len*/
    double mean = 0;
    for (int i=0; i<len;i++){
        mean += 1.0/len*v[i];
    }
```

```c
    return mean;
}

double standard_deviation(double v[], int len){
    /* returns the standard deviation of vector v of length len*/
    double mu = get_mean(v, len);
    double var =0;
    for (int i=0; i<len;i++){
        var += pow(v[i]-mu,2)*1.0/len;
    }
    return sqrt(var);
}

void auto_cor(char *filename, int N_metropolis, double vec[]){
    /* calculates the autocorrelation function of vector vec of lenth ↩
        N_metropolis.
    Used for calculating the statistical inneficiency*/
    FILE *fpc = fopen(filename, "w");
    fprintf(fpc, "k,phi,var_f\n");
    double *f = malloc(N_metropolis*sizeof(double));
    double *f_2 = malloc(N_metropolis*sizeof(double));
    double *f_temp = malloc(N_metropolis*sizeof(double));
    double f_mean; double f2_mean;
    int to_k = 500000; //limits
    double phi;

    f_mean = get_mean(vec,N_metropolis);
    for (int i =0; i < N_metropolis; i++){
        f[i] = vec[i] -f_mean; //subtract mean to make nuerically stable
    }
    f_mean = get_mean(f, N_metropolis);
    double f_var = pow(standard_deviation(f, N_metropolis),2);

    for (int i=0; i<N_metropolis; i++){
        f_2[i] = f[i]*f[i];
    }
    f2_mean = get_mean(f_2, N_metropolis);

    // caluclate auto correlation function phi(k)
    for (int k=0; k<to_k; k+=100){
        for (int i=0; i<N_metropolis-k; i++){
            f_temp[i]=f[i+k]*f[i];
        }

        phi = get_mean(f_temp, (N_metropolis-k)) / f2_mean;
        fprintf(fpc, "%d,%f,%f\n", k, phi, f_var);
        if (phi<0){ // if phi has passed 0, break.
            break;
        }

    }
    fclose(fpc);
    free(f);free(f_2);free(f_temp);f=NULL;f_2=NULL;f_temp=NULL;
}

void block(char *filename, int N_metropolis, double vec[]){
    /* Block method for calculating the statistical inneficiency. same as E3*/
    double f_mean; double f_var; double F_var; double s;
    double *f = malloc(N_metropolis*sizeof(double));
    double *F = malloc(N_metropolis*sizeof(double));
    int to; int to_block=500000; //limits

    FILE *fpb = fopen(filename, "w");
    fprintf(fpb, "B,s,var_I\n");

    f_mean = get_mean(vec,N_metropolis);
    for (int i =0; i < N_metropolis; i++){
        f[i] = vec[i] - f_mean; //subtract mean to make numerically stable
    }
    f_mean = get_mean(f, N_metropolis);
    f_var = pow(standard_deviation(f,N_metropolis),2);

    // Iterate over different block sizes B
    for (int B=1000; B < to_block; B+=1000){
        to = N_metropolis/B;
        for (int j = 0; j<to;j++){
            F[j]=0;
            for (int i=0; i<B; i ++){
                F[j]+=1.0/B*f[i+1+(j)*B];
            }
        }
        F_var=pow(standard_deviation(F, to),2);

        s=B*F_var/f_var;
        fprintf(fpb, "%d,%f,%f\n", B, s, s*f_var/N_metropolis);
    }
    fclose(fpb);
    free(f);free(F);f=NULL;F=NULL;
}
```

```c
int main(){

    // ***** PROBLEM 1 *****
    bool run_task_1 = false;
    int N_temp_steps = 10000;
    double T_min = 0; //run from T_min to T_max
    double T_c = -2*DELTA_E/KB; //critical temperature
    double T_max=1000;
    double T[N_temp_steps];
    double P[N_temp_steps];
    double delta_T = (T_max-T_min)/(N_temp_steps-1);
    double delta_P = 1.0/N_temp_steps;
    // Initialize lin spaced temperatures.
    for (int i=0; i<N_temp_steps; i++){
        T[i]=T_min+i*delta_T;
        P[i]=i*delta_P;
    }

    int min_p = 0; //index for the minimum of P
    double min; //the minimum value P
    double U_last_min=0; //used for calculating derivative of U, i.e C.
    int save_u_every_iter = 50; // save data every n iterations.
    double C_temp=0;
    if (run_task_1){
        printf("Running task 1 \n");

        FILE *fp = fopen("data/task1.csv", "w");
        fprintf(fp, "T,P,U,C\n");
        for (int t=0; t<N_temp_steps; t++){
            // Invert the function T(P) into T(P) by finding minimum of T-T(P).
            min=10000;
            for (int p=0; p<N_temp_steps; p++){
                if (abs(T[t]-Temp(P[p])) < min )
                {
                    min = abs(T[t]-Temp(P[p]));
                    min_p = p;
                }
            }
            if (t==0){ // save data for first iteration when no derivative ↩
                 exists
                fprintf(fp, "%f,%f,%f,%f\n",T[t], P[min_p], U(P[min_p]), 0.0);
                U_last_min = U(P[min_p]);
            }
            if (t%save_u_every_iter == 0 && t>0){ //calculate C every n ↩
                 iterations
                C_temp = C(U_last_min, U(P[min_p]), delta_T*save_u_every_iter);
                if (T[t]>T_c){
                    C_temp=0;
                }
                fprintf(fp, "%f,%f,%f,%f\n",T[t], P[min_p], U(P[min_p]), C_temp)↩
                     ;
                U_last_min = U(P[min_p]);
            }
            else{ // save data and save the previously calculated C again.
                if (T[t]>T_c){
                    C_temp=0;
                }
                fprintf(fp, "%f,%f,%f,%f\n",T[t], P[min_p], U(P[min_p]), C_temp)↩
                     ;
            }
        }
        fclose(fp);
    }


    // ***** PROBLEM 2 *****
    int N=1000; // Number of A/B atoms.
    int A[N][8]; // matrix with nearest neighbors of atoms in A lattice.
    int B[N][8]; // matrix with nearest neighbors of atoms in B lattice.
    // Used for proposing trial steps in Metropolis algorithm
    int A_trial[N][8];
    int B_trial[N][8];

    // Initialize to perfect order P=1. all NN in A are b-atoms (0) and all NN ↩
         in
    //B are a-atoms (1)
    for (int i=0;i<N;i++){
        for (int j=0;j<8;j++){
            A[i][j]=0;
            B[i][j]=1;
            A_trial[i][j]=0;
            B_trial[i][j]=1;
        }
    }

    // state in A and B matrix corresponds to (i,j,k) through state=N*N*i + N*j ↩
         + k
    // A matrix lies "before" B matrix, i.e we "see" A matrix at (0,0,0) before ↩
         B
    // for a given atom its nearest neigbours are given by: starting above at ↩
         corner with lowest index,
    // going clockwise (0,1,2,3) and simliarily below (4,5,6,7).

    // This transformation matrix describes the above convention:
    int transformation[8]; // index A to nearest neighbour B
    transformation[0] = 6;transformation[1] = 7;transformation[2] = 4;↩
        transformation[3] = 5;
    transformation[4] = 2;transformation[5] = 3;transformation[6] = 0;↩
        transformation[7] = 1;

    int A_atom; int B_atom; // The atoms in the A lattice and B lattice ↩
        respectively
    int temp_coord[3]; int temp_state; // temporary coordinates and states used

    int coord_A[3]; int coord_B[3]; // Coordinates in Nearest neigbour matrixes.
    int state_B; // corresponds to coordinates in nearest neighbour matrix B.

    double delta_E=0; // energy difference from proposed atom swaps.
    // Number of bonds, before and after proposed swap and the difference delta.
    int N_aa=0; int N_aa_i; int N_aa_f; int delta_aa=0;
    int N_bb=0; int N_bb_i; int N_bb_f; int delta_bb=0;
    int N_ab=N*8; int N_ab_i; int N_ab_f; int delta_ab=0;
    double E=N_ab*E_AB; // Energy
    double q=0; // q parameter in metropolis algorithm

    // Random number generator.
    const gsl_rng_type * T_r;
    gsl_rng * r;
    gsl_rng_env_setup();
    T_r = gsl_rng_default;
    r = gsl_rng_alloc(T_r);
    double random;

    // Initialize coordinate transformation matrix.
    // coord_trans[n][i] gives transformation between A matrix atom in direction
    // n=1,2,...,8 for coordinate i=i,j,k
    int coord_trans[8][3];
    for (int i=0;i<8;i++){
        if(i<4){
            coord_trans[i][0]=0;
        }
        else{
            coord_trans[i][0]=1;
        }
    }
    for (int i=0;i<8;i++){
        if(i<2 || i==4 || i==5 ){
            coord_trans[i][1]=0;
        }
        else{
            coord_trans[i][1]=1;
        }
    }
    for (int i=0;i<8;i++){
        if(i==0 || i==3 || i==4 || i==7){
            coord_trans[i][2]=0;
        }
        else{
            coord_trans[i][2]=1;
        }
    }

    // Pick random instance
    int rand_atom = 0; //0,1,...,999 random atom
    int rand_nn = 0; // 0,1,...,7 random nearest neighbour
    int accepted=0; //accepted steps
    int N_A_atoms=N; //A atoms in A lattice
    int N_B_atoms=0; //B atoms in A lattice
    int N_metropolis=10000000; // number of metropolis steps
    double T_eq = 500; // equilibration temperature
    int N_eq = 1000000; // Equilibration steps

    double *E_metropolis = malloc(N_metropolis*sizeof(double)); // Energy
    double *r_metropolis = malloc(N_metropolis*sizeof(double)); // short range ↩
        order
    double *P_metropolis = malloc(N_metropolis*sizeof(double)); // long range ↩
        order

    FILE *fp2b = fopen("data3/task2b.csv", "w"); // File for saving E, C, r, P ↩
        of T
    fprintf(fp2b, "T,E_mu, E_std,C,r,P\n");

    // Loop over different starting temperatures (500,..,1000)
    int N_temps_mc = 51; // number of temperatures to calculate quantitites for.
    double T_mc[N_temps_mc];
    T_min=500;T_max=1000;
    delta_T = (T_max-T_min)/(N_temps_mc-1);
    // Initialize temperatures
    for (int i=0; i<N_temps_mc; i++){
        T_mc[i]=T_min+i*delta_T;
    }
    // Beginning of for loop over temperatures
    for (int t=0; t<N_temps_mc;t++){
        T_eq = T_mc[t];
        // Initialize to perfect order P=1. all NN in A are b-atoms (0) and all ↩
             NN in
        // B are a-atoms (1)
        for (int i=0;i<N;i++){
            for (int j=0;j<8;j++){
                A[i][j]=0;
                B[i][j]=1;
                A_trial[i][j]=0;
                B_trial[i][j]=1;
            }
        }
        // Initialize number of atoms, bonds and energy.
        N_A_atoms=N; N_B_atoms=0;
        N_aa=0; N_bb=0; N_ab=8*N;
        E=N_ab*E_AB;

        // Initialize file to save E E,N_aa,N_bb,N_ab,accepted,N_A_Atoms,↩
             N_B_atoms,q
        char filename[32];
        snprintf(filename, sizeof(char) * 32, "data3/task2b_%d.csv", t);
        printf("T_eq=%.4f\n",T_eq);
        FILE *fp2 = fopen(filename, "w");
        fprintf(fp2, "E,N_aa,N_bb,N_ab,accepted,N_A_Atoms,N_B_atoms,q\n");
        fprintf(fp2, "%f,%d,%d,%d,%d,%d,%d,%f\n", E, N_aa, N_bb, N_ab, 0, ↩
             N_A_atoms, N_B_atoms,0.0);

        // Beginning of for loop for metropolis algorithm
        for (int m=0; m<N_eq+N_metropolis; m++){
            // Only consider cases where swaps are from A atoms to B atoms.
            do
            {
                rand_atom = gsl_rng_uniform_int (r, 1000); //pick random ↩
                     position in A matrix
                rand_nn= gsl_rng_uniform_int (r, 8); //pick random NN from this ↩
                     atom
                i_j_k_from_state(rand_atom, coord_A); //convert to i,j,k state

                for (int i =0; i<3; i++){ // this corresponds to atom in B ↩
                     lattice with coordinates...
                    coord_B[i] = coord_A[i] - coord_trans[transformation[rand_nn↩
                         ]][i];
                }

                state_B = state_from_i_j_k(coord_B); // whicih corresponds to ↩
                     the state state_B

                // what type of atoms are at the specific locations?
                A_atom = B[state_B][transformation[rand_nn]];
                B_atom = A[rand_atom][rand_nn];
            } while (A_atom==B_atom);

            // if atoms are the same, do nothing
            if (A_atom==B_atom){
                delta_E = 0;
                delta_aa = 0;
                delta_bb = 0;
                delta_ab = 0;
                q=0;
            }
            else{
                // COUNT BONDS
                N_aa_i=0;
                N_bb_i=0;
                N_ab_i=0;
                for (int i=0;i<10;i++){
                    temp_coord[0]=i;
                    for (int j=0;j<10;j++){
                        temp_coord[1]=j;
                        for (int k=0;k<10;k++){
                            temp_coord[2]=k;
                            // what A atom is at the position in the A matrix?
                            //(get this from the B matrix in the 0'th direction)
                            temp_state = state_from_i_j_k(temp_coord); //state ↩
                                 for A atom in A lattice
                            A_atom = B[temp_state][0]; // The type of atom in ↩
                                 the A lattice
                            for (int nn=0;nn<8;nn++){
                                if (A[temp_state][nn]==1 && A_atom==1){
                                    N_aa_i += 1;
                                }
                                else if (A[temp_state][nn]==0 && A_atom==0)
                                {
                                    N_bb_i += 1;
                                }
                                else
                                {
                                    N_ab_i += 1;
                                }
                }
            }
        }
    }
    A_atom = B[state_B][transformation[rand_nn]];

    // make the trial swap of atoms between the different positions
    for (int i=0; i<8; i++){
        // Change NN in A lattice
        temp_coord[0] = coord_A[0] - coord_trans[i][0];
        temp_coord[1] = coord_A[1] - coord_trans[i][1];
        temp_coord[2] = coord_A[2] - coord_trans[i][2];

        temp_state = state_from_i_j_k(temp_coord);
        B_trial[temp_state][i] = B_atom; //change atom type

        // Change NN in B lattice
        temp_coord[0] = coord_B[0] + coord_trans[i][0];
        temp_coord[1] = coord_B[1] + coord_trans[i][1];
        temp_coord[2] = coord_B[2] + coord_trans[i][2];

        temp_state = state_from_i_j_k(temp_coord);
        A_trial[temp_state][transformation[i]] = A_atom; //change ↩
             atom type
    }

    // Count atoms after trial swap
    N_aa_f=0;
    N_bb_f=0;
    N_ab_f=0;
    for (int i=0;i<10;i++){
        temp_coord[0]=i;
        for (int j=0;j<10;j++){
            temp_coord[1]=j;
            for (int k=0;k<10;k++){
                temp_coord[2]=k;
                // what A atom is at the position in the A matrix?
                //(get this from the B matrix in the 0'th direction)
                temp_state = state_from_i_j_k(temp_coord); //state ↩
                     for A atom in A lattice
                A_atom = B_trial[temp_state][0]; // The type of atom↩
                      in the A lattice
                for (int nn=0;nn<8;nn++){
                    if (A_trial[temp_state][nn]==1 && A_atom==1){
                        N_aa_f += 1;
                    }
                    else if (A_trial[temp_state][nn]==0 && A_atom↩
                         ==0)
                    {
                        N_bb_f += 1;
                    }
                    else
                    {
                        N_ab_f += 1;
                    }
                }
            }
        }
    }
    // this corresponds to a delta in bonds before and after swap.
    delta_aa = N_aa_f-N_aa_i;
    delta_bb = N_bb_f-N_bb_i;
    delta_ab = N_ab_f-N_ab_i;
    // which corresponds to a delta E
    delta_E = delta_aa*E_AA + delta_bb*E_BB + delta_ab*E_AB;
    // thst gives the probability of accepting trail step.
    q = exp(-delta_E/(KB*T_eq));
}

// random number between 0 and 1
random = gsl_rng_uniform (r);

// if this random number is larger than q, accept the trial change
if (random < q){
    // Accept swap and save new A and B lattice
    E += delta_E;
    N_aa += delta_aa;
    N_bb += delta_bb;
    N_ab += delta_ab;
    accepted = 1;

    // set A and B matrix to the proposed trial step
    for (int i=0;i<N;i++){
        for (int j=0;j<8;j++){
            A[i][j] = A_trial[i][j];
            B[i][j] = B_trial[i][j];
        }
    }

}
else{
```

```
                // energy and bonds are the same as before. (add 0)
                E +=0;
                N_aa += 0;
                N_bb += 0;
                N_ab += 0;

                // reverse back the proposed change.
                accepted = 0;
                for (int i=0;i<N;i++){
                    for (int j=0;j<8;j++){
                        A_trial[i][j] = A[i][j];
                        B_trial[i][j] = B[i][j];
                    }
                }
            }

            // Number of a atoms and b atoms on the A sublattice
            N_A_atoms=0;
            N_B_atoms=0;
            for (int i=0;i<N;i++){
                    for (int j=0;j<8;j++){
                        if (B[i][j]==1){
                            N_A_atoms+=1;
                        }
                        else{
                            N_B_atoms+=1;
                        }
                    }
                }
            N_A_atoms = N_A_atoms/8;
            N_B_atoms = N_B_atoms/8;

            // save relevant data for this metropolis step
            fprintf(fp2, "%f,%d,%d,%d,%d,%d,%d,%f\n", E, N_aa, N_bb, N_ab, ↩
                 accepted,
            N_A_atoms, N_B_atoms, q);
            // save data for E, r and P if the equilibration phase is over.
            // used for calculating mean quantities and the statistical ↩
                 inneficiency s.
            if (m>=N_eq){
                E_metropolis[m-N_eq] = E;
                r_metropolis[m-N_eq] = short_range_order(N_ab);
                P_metropolis[m-N_eq] = long_range_order(N_A_atoms);
            }
        }

        fclose(fp2);
        // Calculate statistical inneficiency s for U, r and P using two methods
        // (1) CORRELATION FUNCTION
        printf("\tCorrelation function\n");
        char filename_cor[32];
        snprintf(filename_cor, sizeof(char) * 32, "data3/task2b_cor_E_%d.csv", t↩
             );
        auto_cor(filename_cor, N_metropolis, E_metropolis);
        snprintf(filename_cor, sizeof(char) * 32, "data3/task2b_cor_r_%d.csv", t↩
             );
        auto_cor(filename_cor, N_metropolis, r_metropolis);
        snprintf(filename_cor, sizeof(char) * 32, "data3/task2b_cor_P_%d.csv", t↩
             );
        auto_cor(filename_cor, N_metropolis, P_metropolis);

        // (2) BLOCK METHOD
        printf("\tBlock method\n");
        char filename_block[32];
        snprintf(filename_block, sizeof(char) * 32, "data3/task2b_block_E_%d.csv↩
             ", t);
        block(filename_block, N_metropolis, E_metropolis);
        snprintf(filename_block, sizeof(char) * 32, "data3/task2b_block_r_%d.csv↩
             ", t);
        block(filename_block, N_metropolis, r_metropolis);
        snprintf(filename_block, sizeof(char) * 32, "data3/task2b_block_P_%d.csv↩
             ", t);
        block(filename_block, N_metropolis, P_metropolis);

        // Save mean energy, heat capacity, short range parameter r
        // and long range parameter P for this temperatur T
        fprintf(fp2b, "%f,%f,%f,%f,%f,%f\n",T_eq, get_mean(E_metropolis,↩
             N_metropolis),
        standard_deviation(E_metropolis,N_metropolis), heat_capacity(↩
             E_metropolis,N_metropolis, T_eq),
        get_mean(r_metropolis,N_metropolis), get_mean(P_metropolis, N_metropolis↩
             ));
    }
    fclose(fp2b);

    return 0;
}
```