Computational Physics - Home assignment 2a

Monte Carlo simulations - Properties of the CuZn binary alloy

Nico Guth

16 December 2022

Contents

Introduction

The goal of this project is to study the properties of a binary alloy. A binary alloy is a material consisting of two atoms, where (at least) one of the elements is a metal. As an example, this study uses the binary alloy consisting of N copper (Cu) and N zinc (Zn) atoms, which are arranged in a body-centered-cubic lattice (bcc).

The energy of this system depends on the arrangement of the two atom types. In the following, the atom type Cu is represented by A and the atom type Zn is represented by B. Let N_{ij} be the number of bonds between atom type i and j and E_{ij} the corresponding bond energy contribution, then

$$E = N_{AA}E_{AA} + N_{BB}E_{BB} + N_{AB}E_{AB}\,. \tag{1}$$

According to the assignment description, reasonable bond energies for the CuZn alloy, that are used in this project, are

$$\begin{aligned} E_{AA} &= E_{\mathrm{CuCu}} = -436\,\mathrm{meV}\,, \\ E_{BB} &= E_{\mathrm{ZnZn}} = -113\,\mathrm{meV}\,, \\ E_{AB} &= E_{\mathrm{CuZn}} = -294\,\mathrm{meV}\,[1]. \end{aligned} \tag{2}$$

Another important property of any material is the heat capacity

$$C = \frac{\mathrm{d}E}{\mathrm{d}T}\,. \tag{3}$$

To quantify the amount of order in the binary alloy, a long-range order parameter P and a short-range order parameter r are introduced.

P quantifies the separation of the atom types into the two cubic sublattices (named a and b) of the bcc lattice, where the atoms in one sublattice are in the center of the unit cells of the other sublattice and vice versa. It is defined as

$$P = 2\frac{N_{Aa}}{N} - 1 \tag{4}$$

where N_{Aa} is the number of atoms A on sublattice a and N is the total number of A atoms in the alloy. Therefore, $P = \pm 1$ if the alloy is in perfect order, so that both sublattices contain only one type of atom. And $P = 0$ if the alloy is not ordered, but each sublattice contains equal amounts of each atom type.

r quantifies the separation of the atom types into regions and is defined as

$$r = \frac{N_{AB} - 4N}{4N}\,. \tag{5}$$

The total number of bonds is $8N$, so for $r = 1$ all bonds are AB-bonds, and it is basically the same as $P = \pm 1$. However, for $r \approx 0$, only half of the bonds are AB-bonds and the system is in complete disorder. Theoretically, a negative value of r is possible, but this means that the atoms clump together in large regions of one atom type and is unlikely.

The main goal of this project is to investigate the temperature dependence of these four properties using both a numerical approximation and a Monte Carlo simulation.[1]

Task 1 4/4

Objective and procedure

Instead of simulating the binary alloy, the temperature dependence of several properties are calculated numerically using the so-called *mean field approximation* (MFA). In this approximation, the thermodynamic free energy is

$$F(N,P) = E_0 - 2NP^2\Delta E + Nk_BT[-2\ln(2) + (1+P)\ln(1+P) + (1-P)\ln(1-P)] \quad (6)$$

with

$$E_0 = 2N(E_{AA} + E_{BB} + 2E_{AB})\,, \quad (7)$$
$$\Delta E = E_{AA} + E_{BB} - 2E_{AB}\,, \quad (8)$$

the number of atoms per sublattice N, the temperature T, the Boltzmann constant k_B and the long-range order parameter P. [2]

To find the thermodynamic equilibrium state at temperature T, the free energy is minimized numerically with respect to P. In a binary alloy with equal number of A and B atoms, the long-range order parameter can take values between -1 and +1. However, because the free energy is symmetric around $P = 0$, it is sufficient to only consider positive P.

Once $P(T)$ is found (in equilibrium), the total energy of the system is calculated using Eq. (1), where in the mean field approximation

$$N_{AA} = 2(1-P^2)N, \quad (9)$$
$$N_{BB} = 2(1-P^2)N, \quad (10)$$
$$N_{AB} = 4(1+P^2)N. \quad (11)$$

And finally the heat capacity can be calculated using the numerical finite differences method

$$\frac{\mathrm{d}E}{\mathrm{d}T} \approx \frac{E(T+\Delta T) - E(T)}{\Delta T} \quad (12)$$

for a sufficiently small ΔT.

The phase transition from the β'-phase (ordered bcc-structure) to the β-phase (disordered bcc structure) occurs at the critical temperature

$$T_{c,\mathrm{MFA}} = \frac{2\Delta E}{k_B} \quad (13)$$

in the mean field approximation of a binary alloy.[2]

Results and discussion

The numerically calculated temperature dependence of the long-range order parameter P, the energy E and the heat capacity C in the mean field approximation of the CuZn binary alloy are shown in Fig. 1. A temperature range $T \in [0\,\mathrm{K}, 1200\,\mathrm{K}]$ in steps of $\Delta T = 1\,\mathrm{K}$ is chosen. To make this illustration independent of the number of atoms, each equation mentioned in the last section, which is dependent on N, is divided by N on both sides. Using Eq. (13) and the bond energies mentioned in Eq. (2) the theoretical critical temperature is

$$T_{c,\mathrm{MFA}} = 905.15\,\mathrm{K}\,. \tag{14}$$

This value corresponds very well to the plots shown in Fig. 1, where at this temperature the behavior instantly changes. However, when comparing this value to the experimentally found phase transition temperature of the CuZn binary alloy

$$T_{c,\mathrm{exp}} = 468\,^\circ\mathrm{C} = 741.15\,\mathrm{K}\,[1], \tag{15}$$

it differs by over 150 K. This indicates that the mean field approximation is not very accurate in determining the temperature of the phase transition.

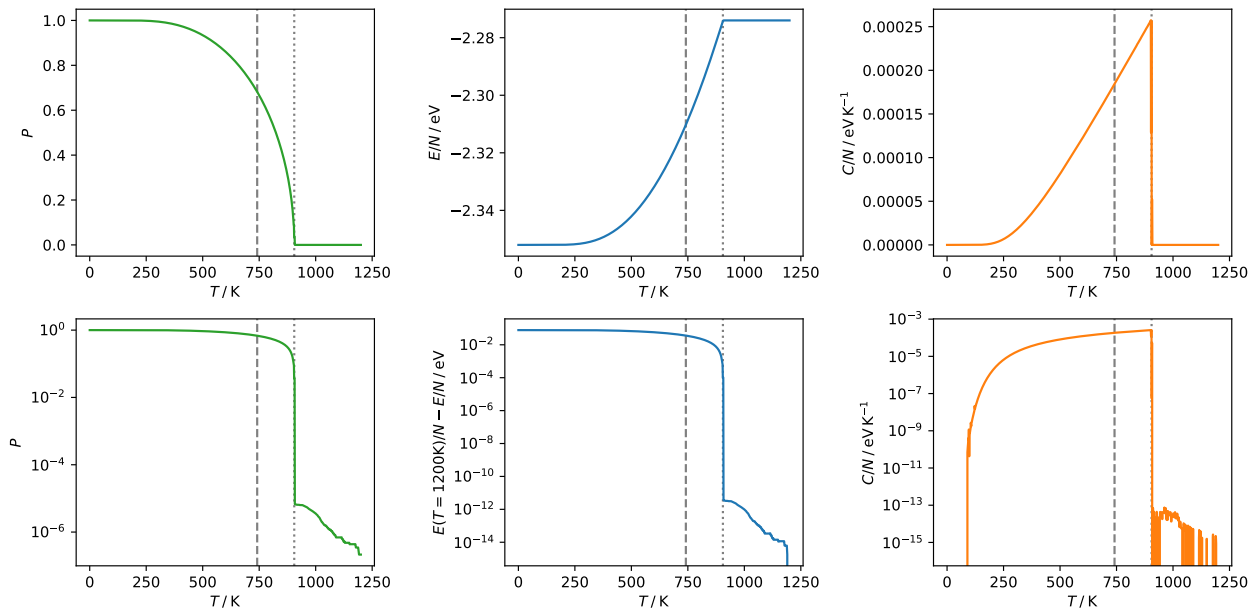

Figure 1: Temperature dependence of the long-range order parameter P, the energy E and the heat capacity C in the mean field approximation of the CuZn binary alloy. Shown are these properties per number of atoms N both on a normal and on a logarithmic scale (the difference in the energy to the last value is plotted). Also marked are the calculated critical temperature $T_{c,\mathrm{MFA}} = 905.15\,\mathrm{K}$ (dotted line) and the experimentally measured critical temperature $T_{c,\mathrm{exp}} = 741.15\,\mathrm{K}$ (dashed line).

For low temperatures, the alloy is modelled to be in almost perfect order ($P = 1$). Then P decreases with increasing rapidity until at $T_{c,\mathrm{MFA}}$ it instantly becomes 0 and stays at complete disorder for higher temperatures. The decrease in the quantities at $T > T_c$ is a result of

numerical errors and changes, when the tolerance of the minimization algorithm is decreased. $F(P)$ in Eq. (6) has the only minimum at exactly $P = 0$ for $T > T_c$. To illustrate the symmetry of $F(P)$ and the positions of its minima, it is potted for five different temperatures in Fig. 2. The general trend is physically reasonable, however the sudden change to $P = 0$ suggests an idealized behavior in the model. The same holds for the inner energy E, where it increases with the temperature but above $T_{c,\mathrm{MFA}}$ it suddenly stays constant. In a physical system, if the temperature increases and no other properties are changed, the inner energy must also increase. However, this model only considers the atom type arrangement and therefore the energy increase could be seen in the dynamics of the system. The heat capacity, which is the numerical derivative of the energy, shows an increase below the phase transition and also suddenly drops to 0. This might also be physically reasonable, since the dynamics of the system are not considered here.

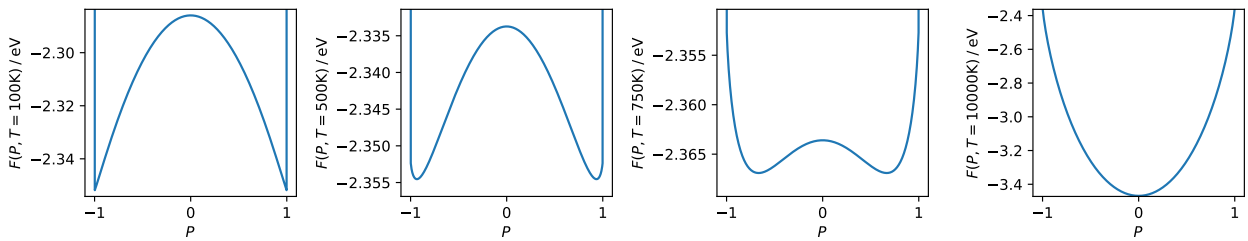

Figure 2: Dependence of the free energy F on the long-range order parameter P for five different temperatures $T = 100\,\mathrm{K}$, $T = 500\,\mathrm{K}$, $T = 750\,\mathrm{K}$ and $T = 1000\,\mathrm{K}$ respectively from left to right.

Task 2

Objective and procedure

In this task, a *Markov Chain Monte Carlo* (MCMC) simulation of a binary alloy is performed to determine the temperature dependence of the properties introduced in the first section (E,C,P,r). More specifically, the *Metropolis algorithm* is used to sample the microstates of the binary alloy at a given temperature. The system of interest is a bcc lattice with two atom types of equal proportions, which is visualized in Fig. 3. The movement of the lattice vertices is neglected in this study, and only the arrangement of the atom types (i.e. the nearest neighbor bonds) is considered.

Therefore, the state of the system is represented by a list of $2N$ atomic positions and the type of atom (A or B) that sits at this position. The positions $(x, y, z)^T$ are taken to be at integer multiples of the lattice constant a_0 (width of one unit cell) together with a binary variable $w = a, b$ that specifies whether the position is in sublattice a (i.e. $\vec{r} = (x, y, z)^T \cdot a_0$) or in sublattice b (i.e. $\vec{r} = (x + 1/2, y + 1/2, z + 1/2) \cdot a_0$). The number of simulated bcc unit cells in each direction is called n. This means that $x, y, z \in \{0, 1, ..., n - 1\}$ and $N = 2n^3$. To be able to count the types of nearest neighbor bonds (N_{AA},N_{BB},N_{AB}), a list of indices of the 8 nearest neighbors is also generated for each position. Since only a finite and potentially small number of atoms can be simulated compared to the number of atoms in a real crystal,

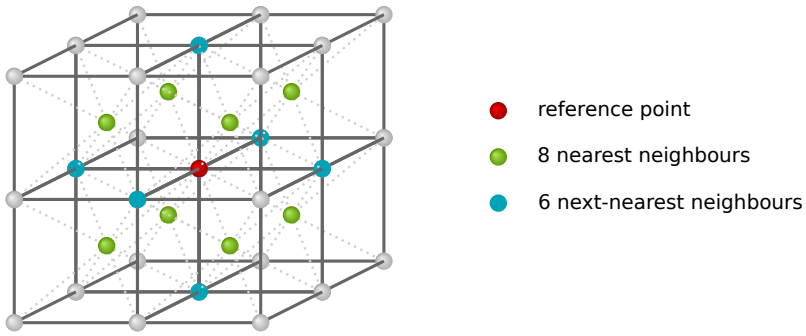

Figure 3: Illustration of a bcc lattice with the green atoms in one sublattice and all other atoms in a different sublattice. Shown are only $2 \times 2 \times 2$ unit cells. [3]

periodic boundary conditions are applied to the list of nearest neighbors. This means that, for example, an atom in sublattice b with $x = n - 1$ has nearest neighbors in sublattice a with both $x = n - 1$ and $x = 1$.

Given a specific temperature T and an initial state of the system, the microstates of the system are sampled using the Metropolis algorithm. At each simulation step, a random change in the system's state is proposed, and then the probability of the current state and the proposed state is compared to accept or decline the proposal. The new current state is used to calculate and save various instantaneous properties, which can later be used to calculate canonical ensemble averages.

The proposed change used to simulate the binary alloy is the swap of two randomly picked atoms. To increase the efficiency of this simulation without loss of generality, only swaps are proposed where both atoms are of different type. Otherwise, the proposed state would be exactly the same as the current state, because the only difference of the atoms are the atom type. Then, the number of bonds for each bond type (N_{AA},N_{BB},N_{AB}) are calculated for the proposed state to calculate the difference in energy

$$\Delta E = E_{\text{proposed}} - E_{\text{current}} \tag{16}$$

using Eq. (1). If $\Delta E \leq 0$, the proposed state is more likely and therefore accepted. Otherwise, the proposed state is accepted with the probability of

$$P(\text{accept proposal}) = \exp\left(-\frac{\Delta E}{k_B T}\right) \text{ [2].} \tag{17}$$

To decrease the computation time, instead of counting the bonds in each simulation step, the bond counts are updated by only evaluating how the bonds between the swapped atoms and their nearest neighbors change. It is important to also consider the special case, when the swapped atoms are nearest neighbors themselves.

Since the first steps in the Metropolis algorithm are needed to find the region of interest (thermodynamic equilibrium), each simulation must consist of a number of burn-in/equilibration steps followed by the actual production steps where physical quantities are calculated. The quantities of interest in this study are calculated in each simulation step using Eq. (1), Eq. (4)

and Eq. (5). Afterwards, the averages of these quantities are calculated per simulation (one temperature), and the heat capacity is calculated as

$$C = \frac{1}{k_B T^2} \left(\langle E^2 \rangle - \langle E \rangle^2 \right) \text{ [1].} \tag{18}$$

The error estimation of the calculated average quantities needs a simulation property called *statistical inefficiency* s, which quantifies how many subsequent simulated measurements are highly correlated. The correct uncertainty (standard deviation) of the average of measured quantities $\{f_i\}$ is therefore

$$\sigma_{\langle f \rangle} = \sqrt{\frac{s}{M} \sigma_f} \tag{19}$$

where M is the number of measurements and σ_f is the standard deviation of $\{f_i\}$.

The statistical inefficiency can be estimated in several ways. The method chosen here is to set $s = k_0$, where k_0 is the lowest integer for which the autocorrelation function

$$\Phi(k) = \frac{\langle f_i \cdot f_{i+k} \rangle - \langle f \rangle^2}{\langle f^2 \rangle - \langle f \rangle^2} \tag{20}$$

is less than $\Phi_0 = \exp(-2) \approx 0.135$. Since calculating $\Phi(k)$ for all values $k \in \{1, 2, ..., M\}$ after each simulation would take a lot of computation time, the search for k_0 is implemented in two steps. First, the value of k is steadily increased by a factor of 2 until $\Phi(k_{\text{start}} \cdot 2^i) < \Phi_0$. Then, the exact value k_0 is searched using a binary search that starts with the boundaries $k_{\text{start}} \cdot 2^{i-1}$ and $k_{\text{start}} \cdot 2^i$. This way, k_0 is found using only a few tens of $\Phi(k)$ computations instead of the up to M computations.

A different method of calculating the statistical inefficiency is block averaging. Here, the variance of the measurable f is compared to the variance over the block averages

$$F_j = \frac{1}{M} \sum_i^B f_{i+(j-1)B} \tag{21}$$

where B is a specific block size and $j \in \{1, 2, ..., M/B\}$. The statistical inefficiency is then estimated as

$$s = B \frac{\text{Var}[F]}{\text{Var}[f]} \tag{22}$$

and converges to the correct value for large B. [4]

All the procedures explained above are repeated for multiple temperatures, so that the temperature dependence of the mentioned quantities can be investigated. The number of needed equilibration steps is heavily reduced by simulating the system with increasing temperatures and using the last system state of the previous temperature as the initial state for the next temperature. The first initial state is chosen to be the perfectly ordered state, where all atoms A are in sublattice a and all atoms B are in sublattice b.

Results and discussion

The procedures described in the previous section are used to simulate the CuZn binary alloy using the bond energies given in Eq. (2). The simulation cell consists of $10 \times 10 \times 10$ unit cells ($\Rightarrow N = 1000$) with the previously mentioned periodic boundary conditions.

The resulting ensemble averages for several temperatures are shown in Fig. 6. These results are however discussed only later in this section. Included are the uncertainties calculated through Eq. (19). However, the uncertainties are too small to be visible and are therefore plotted separately in Fig. 7. Chosen is the temperature range between 300 K and 1000 K with a $\Delta T = 1\,\mathrm{K}$ in the range between 600 K and 800 K (region of the phase transition) and a $\Delta T = 20\,\mathrm{K}$ outside of this range. Each simulation is performed with $M_{eq} = 0.5 \times 10^6$ equilibration steps and $M = 4.5 \times 10^6$ production steps.

To show that M_{eq} is indeed enough to find the thermodynamic equilibrium, Fig. 4 shows the evolution of the quantities during several simulations. All quantities show a more or less constant behavior for all shown temperatures, with some degree of noise. The number of equilibration steps appears to be more than sufficient, since equilibrium is reached with a lot less steps for all temperatures. For temperatures in the region of the phase transition (700 K and 750 K), the variation/oscillation happens over much longer step intervals. Especially for $T = 750\,\mathrm{K}$ (red) a large oscillation around $P = 0$ is visible. This explains the large uncertainties in the phase transition region in Fig. 7 and why $P(T)$ in Fig. 6 is oscillating around $P = 0$.

To illustrate how the statistical inefficiency is calculated, Fig. 5a shows the autocorrelation function $\Phi_E(k)$ of the energy for several temperatures. Even though the procedure used to find the first k_0 for $\Phi(k_0) < \exp(-2)$, explained at the end of the last section, does not need to calculate all $\Phi(k)$, the dashed lines show that the k_0 found is indeed the correct k_0 (at least for these five temperatures). It also shows why a simple binary search would not work, because for higher k the autocorrelation function is very noisy and becomes greater than $\exp(-2)$ again.

The autocorrelation method of calculating the statistical inefficiency should yield about the same results as the block averaging method. To verify this, the block averaging method is performed for several temperatures. This is shown in Fig. 5b together with the statistical inefficiencies calculated during the production. In theory, the statistical inefficiency should converge to the correct value for very large block sizes. Fig. 5b shows that it converges to similar values as calculated through the autocorrelation method. However, for too large block sizes, there are too few blocks to take the variance over. It could not be determined with certainty, why the block averaging for the temperatures $T = 650\,\mathrm{K}$ and $T = 750\,\mathrm{K}$ only converges for a relatively small B interval and drops so quickly for larger B. However, it might be because for temperatures near the phase transition, P jumps between positive and negative values and stays there for a certain number of simulation steps. In the cases, where the block averaging result differs from value during production, the production value is higher. Therefore, the statistical inefficiencies in this study are not underestimated.

The resulting statistical inefficiencies s for each temperature are shown in Fig. 5c. It turns out that the $s(E)$ and $s(r)$ are the same, which is based on the definitions Eq. (1) and Eq. (5) with the approximation that $N_{AA} \approx N_{BB}$. The statistical inefficiencies show a general decrease with temperature, except that in the phase transition region it increases a lot. This corresponds to

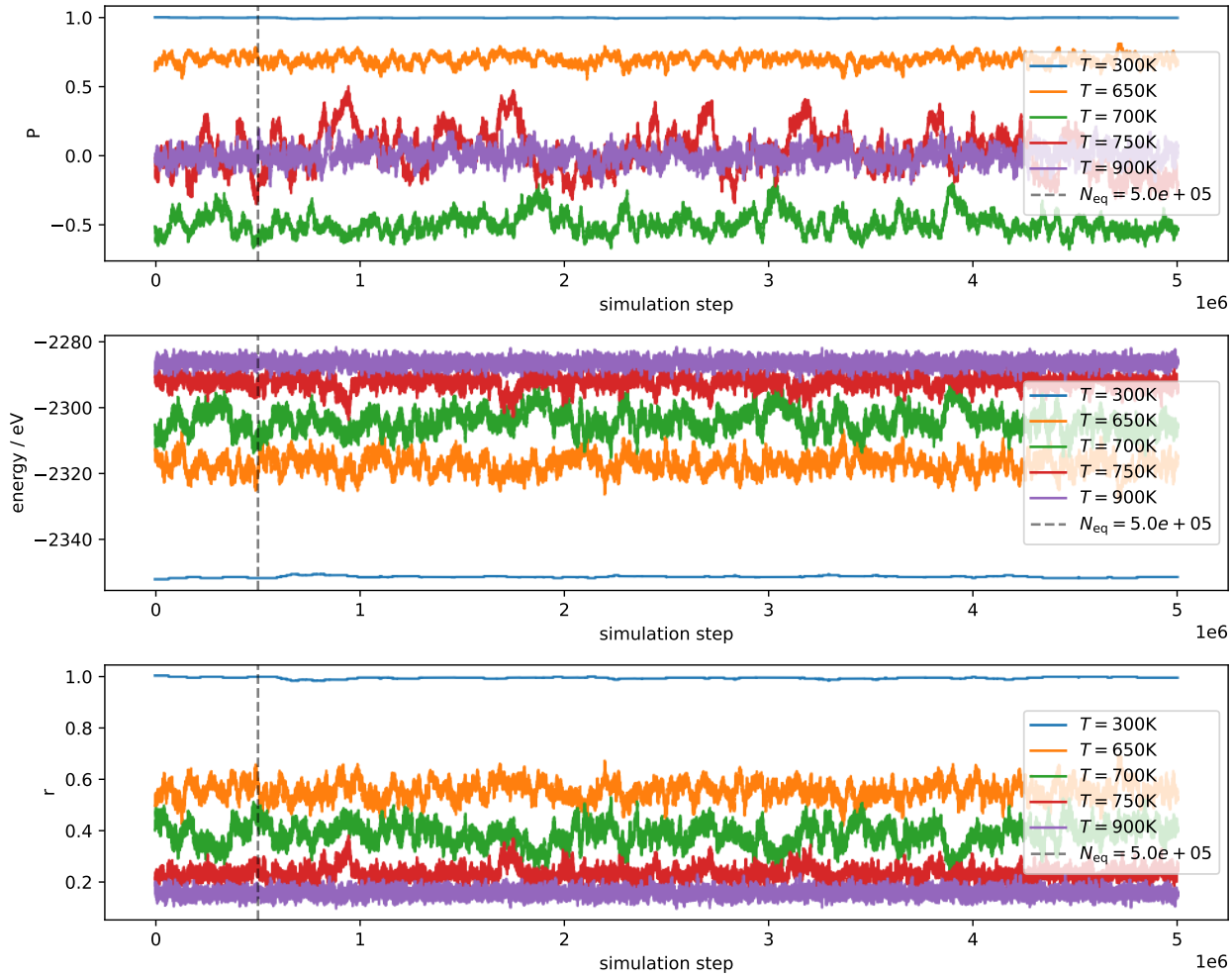

Figure 4: Evolution of the energy E, the long-range order parameter P and the short-range order parameter r in simulations of the temperatures 300 K, 650 K, 700 K, 750 K and 900 K. Shown is each tenth simulation step. The first step of the production is marked with a vertical dashed line at $M_{eq} = 0.5 \times 10^6$.

the oscillations seen in Fig. 4 and shows an instable behavior of the CuZn binary alloy during phase transition. For temperatures below the phase transition, additionally $s(P) \approx s(E)$.

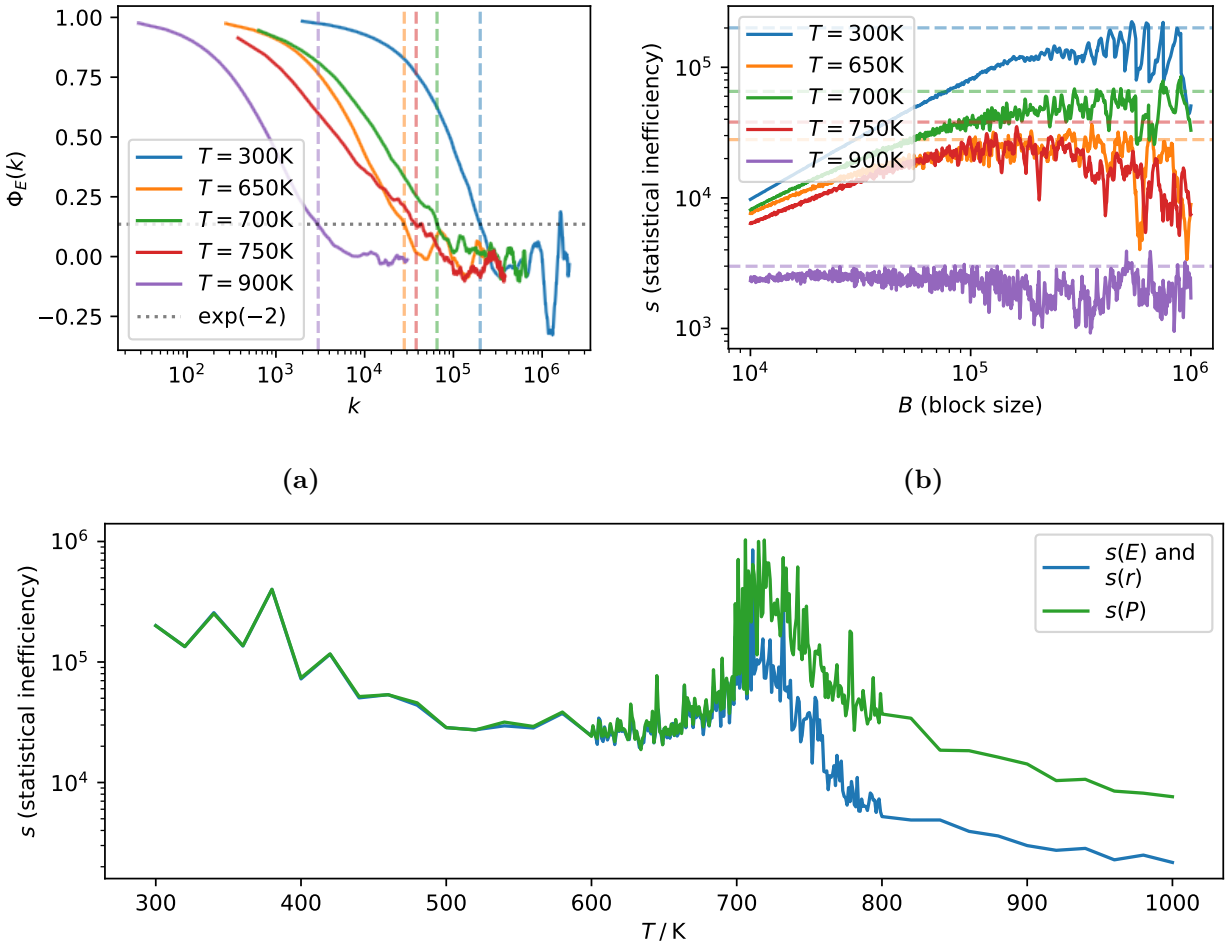

Figure 5: (a) shows a section of the autocorrelation functions for the energy of several different temperatures. The vertical dashed lines show the statistical inefficiency that is calculated during production. (b) shows the calculated statistical inefficiencies using block averaging for several block sizes and several temperatures. The horizontal dashed line show the statistical inefficiency that is calculated during production. (c) shows the statistical inefficiencies s for the quantities E, r and P that are calculated during production. Since $s(E)$ and $s(r)$ are the same, they are plotted together.

The final results of this study are shown in Fig. 6 with the uncertainties in Fig. 7. Rough comparisons to the mean field approximation in Fig. 1 show the same general trends. The energy E is increasing and flattens out after the phase transition. The heat capacity C is increasing until around 700 K and is then decreasing rapidly to almost $C = 0\,\mathrm{eV/K}$. The long-range order parameter $|P|$ is decreasing more and more until it rapidly goes to the unordered state $P \approx 0$. The short-range order parameter r shows a similar trend as P but converges much slower towards $r = 0$. In general, the values of the quantities also roughly correspond to the values in Fig. 1, when multiplying by $N = 1000$.

A major difference to the mean field approximation is that the phase transition does not happen instantly, but continuously over a temperature range of around 50 K starting at around $T = 700\,\mathrm{K}$. Additionally, the phase transition in the simulation is much closer to the experimentally found critical temperature (dashed line) than the mean field approximation critical temperature (dotted line). Since the transition happens gradually, no concrete critical temperature could be determined. That said, the critical temperature in the simulation from which point onwards $P \approx 0$ (disordered β-phase) coincides very well with the experimentally found critical temperature. It is also worth to note, that the same physical constants (Eq. (2)) are used in both models and the simulation still outperforms the mean field approximation.

The energy is monotonously increasing with temperature, which is physically reasonable. The slower flattening of the energy at high temperatures leads to higher values of C than found in the mean field approximation. The heat capacity looks very noisy in the region of the phase transition, however this is a result of the more precise sampling in this region. Although, around the turning point at $T \approx 700\,\mathrm{K}$, the variation is very large.

The same turning point can be observed in P, where the value of P suddenly oscillates between positive and negative values. As mentioned before, this oscillation also happens during one simulation, as seen in Fig. 4 for $T = 750\,\mathrm{K}$. However, the fact that P is purely positive before the phase transition is not a physical property, but a result of choosing the initial state as the $P = +1$ state. The absolute value $|P|$ shows a steady decrease (with some noise) towards $P \approx 0$ with increasing temperature. The short-range order parameter r is also steadily decreasing, but presumably reaches $r \approx 0$ only for much higher temperatures. The fact, that both order parameters start in almost complete order for low temperatures and ends in almost complete disorder for high temperatures, is also physically reasonable and shows indeed that there are the two different phases β' and β in the CuZn binary alloy.

When considering the uncertainties in Fig. 7 and the statistical inefficiencies in Fig. 5c, it becomes apparent that a lot more simulation steps would be needed to achieve precise results in the region of the phase transition. Additionally, a larger amount of simulated atoms would potentially show a more sharp phase transition and also increase the precision. Both of which could not be done in this project, due to time limitations and limitations in the available computing power.

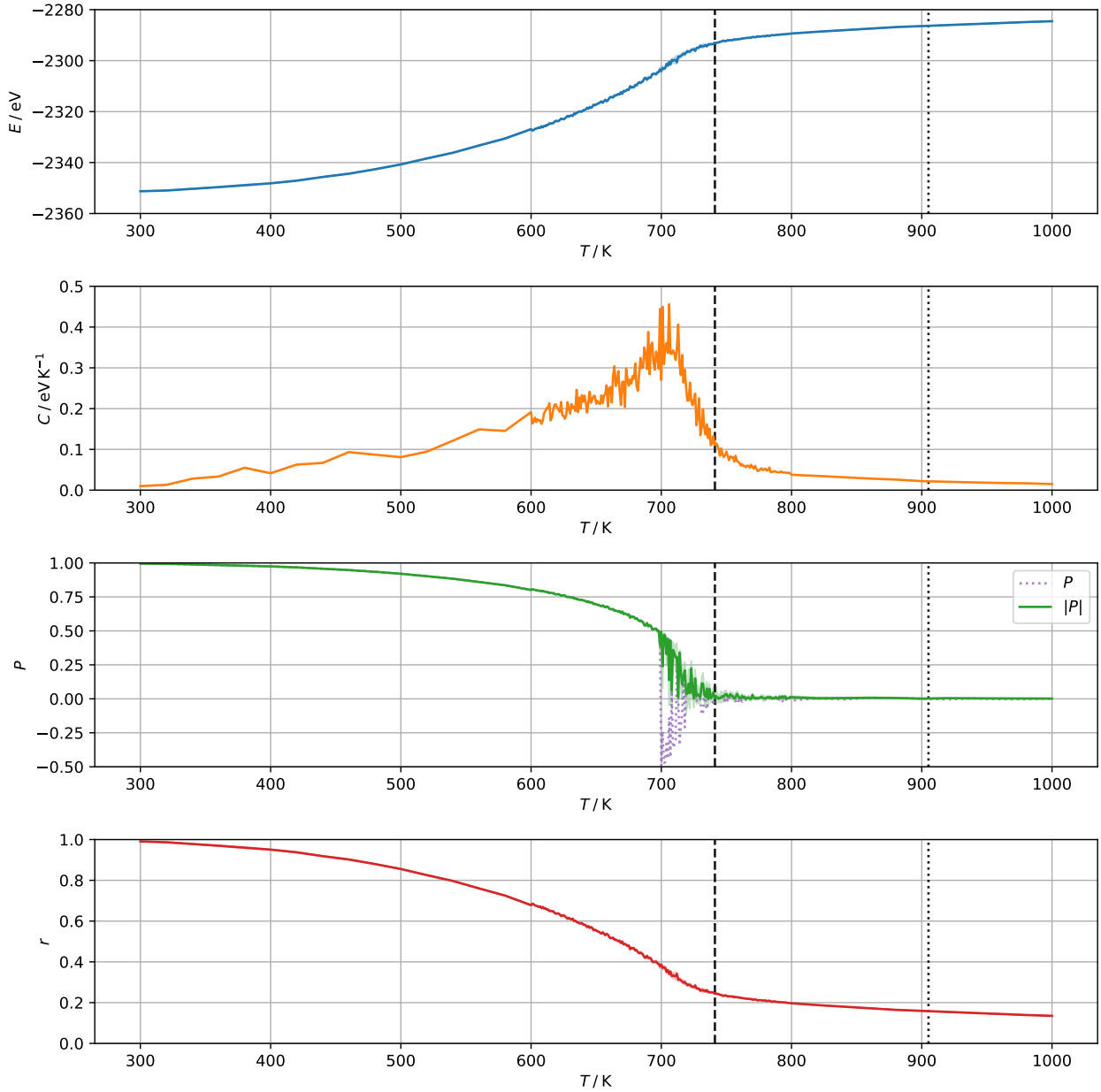

Figure 6: Temperature dependence of the energy E, the heat capacity C, the long-range order parameter P and the short-range order parameter r as the resulting canonical ensemble averages of MCMC simulations of a CuZn binary alloy. Each value (except for C) is plotted with the corresponding uncertainty. However, most uncertainties are too small to be visible and are therefore shown in Fig. 7. Both P and $|P|$ are plotted, however the error region is only plotted for $|P|$. Also marked are the mean field approximation critical temperature $T_{c,\mathrm{MFA}} = 905.15\,\mathrm{K}$ (dotted line) and the experimentally measured critical temperature $T_{c,\mathrm{exp}} = 741.15\,\mathrm{K}$ (dashed line).

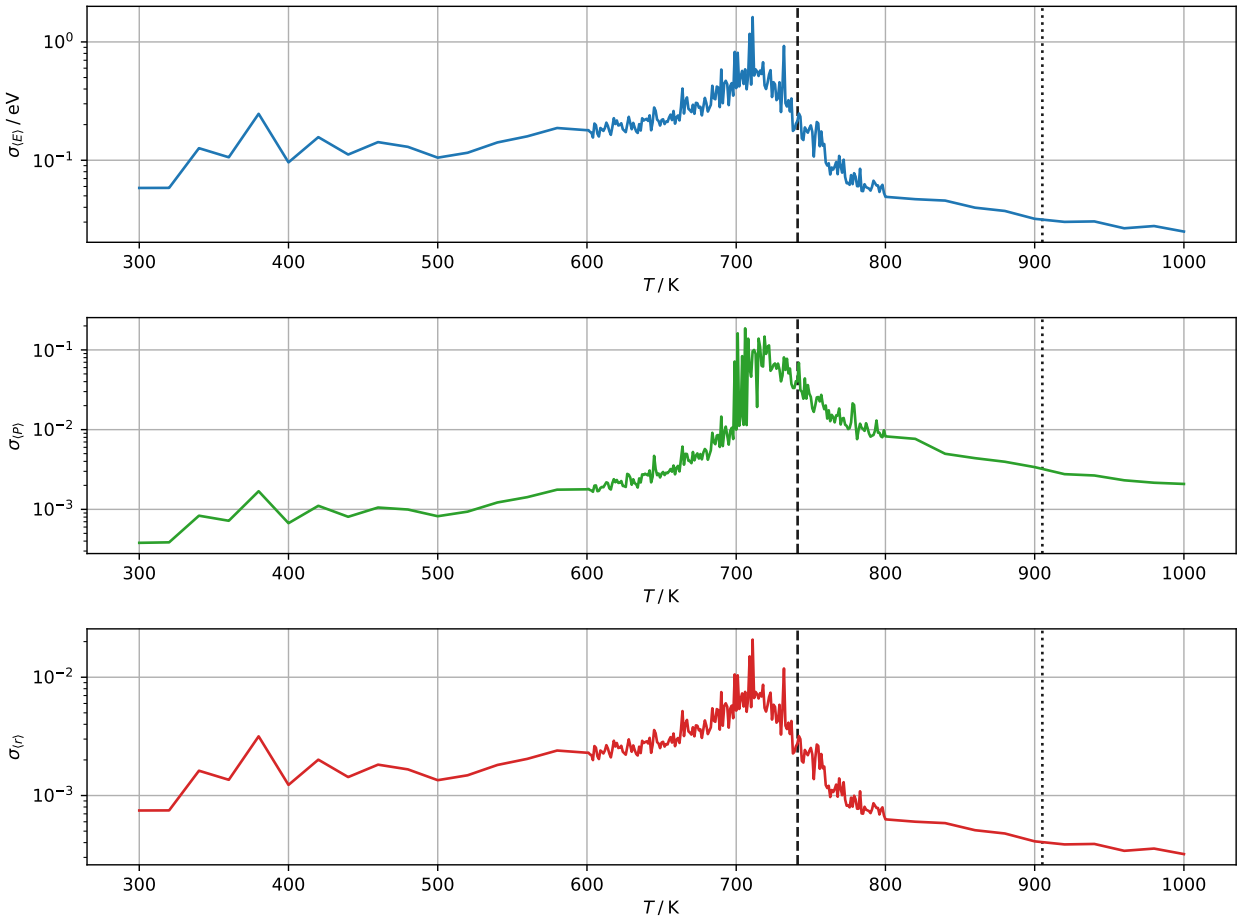

Figure 7: Uncertainties of the quantities in Fig. 6 calculated through Eq. (19).

Concluding discussion

In this study, it is shown that both the mean field approximation and a MCMC-simulation yield physically reasonable behaviors of several properties of the CuZn binary alloy. The critical temperature of the phase transition (between the ordered β'-phase and the disordered β-phase) in the mean field approximation differs by over 100 K from the experimentally found temperature. However, the phase transition found through the simulation is much closer to the experimentally found value. In the mean field approximation, the phase transition is very sharp, but the transition found through simulation is more gradual and therefore probably closer to physical reality. Overall, this study illustrates that Monte Carlo simulations of physical systems can be used to investigate their thermodynamic behavior.

References

[1] Göran Wahnström, Department of Physics, Chalmers, Göteborg. *Computational Physics: H2a Binary Alloy*. 2022.

[2] Göran Wahnström, Department of Physics, Chalmers, Göteborg. *Computational Physics lecture 7*. 2022.

[3] Physics in a nutshell, Tobias Wegener. *Solid State Physics - Common Crystal Structures - Body Centered Cubic (bcc)*. URL: `https://www.physics-in-a-nutshell.com/article/12/body-centered-cubic-bcc` (visited on 14/12/2022).

[4] Göran Wahnström, Department of Physics, Chalmers, Göteborg. *Monte Carlo lecture notes*. 2019.

Source Code

Included in this appendix is all the relevant code that I wrote myself. The entire project is attached as a zip file, where also the code is included that was already provided.

Main code for all calculations/simulations: `run.c`

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

#include "tools.h"
#include "lattice_tools.h"
#include "statistical_ineff.h"

// physical constants
const double k_B = 8.617333262e-5; // eV/K;

// parameters of the simulation
const int n_cells = 10;
const int N_atoms = 2*n_cells*n_cells*n_cells;

void update_bond_counts_after_swap(int N_atoms, int* atype,
                int** nn_idxs, int** pos,
                int swap_idx_a, int swap_idx_b,
                int* N_AA, int* N_BB, int* N_AB, int* N_Aa) {
    int atype_a = atype[swap_idx_a]; // old atype at position a
    int atype_b = atype[swap_idx_b]; // old atype at position b
    if (atype_a == atype_b) {
        // the swap introduces no change
        return;
    }

    // update the Number of A atoms in sublattice a
    // if position a is on a different sublattice than position b, N_Aa can change
    if (pos[swap_idx_a][3] != pos[swap_idx_b][3]) {
        if (pos[swap_idx_a][3] == 0) {
            // position a is on sublattice a
            if (atype_a == 0) {
                (*N_Aa)--;
            } else {
                (*N_Aa)++;
            }
        } else {
            // position b is on sublattice a
            if (atype_b == 0) {
                (*N_Aa)--;
            } else {
                (*N_Aa)++;
            }
        }
    }

    // update bonds at position a
    for (int j=0; j<8; j++) {
        int atype_nn = atype[nn_idxs[swap_idx_a][j]];

        // if the nn of a is b then the change is already acounted for
        if (nn_idxs[swap_idx_a][j] == swap_idx_b) {
            continue;
        }

```

```
        // substract old bond and add new bond
        if (atype_a == 0 && atype_nn == 0) {
            (*N_AA)--;
            (*N_AB)++;
        } else if (atype_a == 1 && atype_nn == 1) {
            (*N_BB)--;
            (*N_AB)++;
        } else if (atype_a == 0 && atype_nn == 1) {
            (*N_AB)--;
            (*N_BB)++;
        } else if (atype_a == 1 && atype_nn == 0) {
            (*N_AB)--;
            (*N_AA)++;
        }
    }

    // update bonds at position b
    for (int j=0; j<8; j++) {
        int atype_nn = atype[nn_idxs[swap_idx_b][j]];

        // if the nn of b is a then the change is already acounted for
        if (nn_idxs[swap_idx_b][j] == swap_idx_a) {
            continue;
        }

        // substract old bond and add new bond
        if (atype_b == 0 && atype_nn == 0) {
            (*N_AA)--;
            (*N_AB)++;
        } else if (atype_b == 1 && atype_nn == 1) {
            (*N_BB)--;
            (*N_AB)++;
        } else if (atype_b == 0 && atype_nn == 1) {
            (*N_AB)--;
            (*N_BB)++;
        } else if (atype_b == 1 && atype_nn == 0) {
            (*N_AB)--;
            (*N_AA)++;
        }
    }
}

void metropolis_algorithm(int N_atoms, int n_steps, double T,
                          int* atype, int** pos, int** nn_idxs, gsl_rng* rng,
                          double* E_out, double* P_out, double* r_out) {
    int N_AA, N_BB, N_AB, N_Aa=0;
    count_bonds(N_atoms,atype,nn_idxs,&N_AA, &N_BB, &N_AB);
    // count A atoms on a sublattice
    for (int i=0; i<N_atoms; i++) {
        if (atype[i] == 0 && pos[i][3] == 0) {
            N_Aa++;
        }
    }

    double E_curr = calc_energy(N_AA, N_BB, N_AB);
    print_progress(0,0,n_steps-1,true);
    // calc the energy of the current configuration
    for (int i_step=0; i_step<n_steps; i_step++) {
        // make trial change by swapping two atoms
        // choose which atoms to swap
        int swap_idx_a = gsl_rng_uniform_int(rng, N_atoms);
        int swap_idx_b = gsl_rng_uniform_int(rng, N_atoms);
        while (swap_idx_a == swap_idx_b || atype[swap_idx_a] == atype[swap_idx_b]) {
            swap_idx_b = gsl_rng_uniform_int(rng, N_atoms);
        }

        // save the current bond counts if the trial change is not accepted
        int N_AA_trial = N_AA;
        int N_BB_trial = N_BB;
        int N_AB_trial = N_AB;
        int N_Aa_trial = N_Aa;
```

```
        // check if the swap should be accepted as the new configuration
        update_bond_counts_after_swap(N_atoms,atype,nn_idxs, pos,
                            swap_idx_a,swap_idx_b,
                            &N_AA_trial, &N_BB_trial, &N_AB_trial, &N_Aa_trial);
        double E_trial = calc_energy(N_AA_trial, N_BB_trial, N_AB_trial);
        double DeltaE = E_trial - E_curr;
        //DeltaE = E_curr - E_trial;
        bool accept = (DeltaE <= 0);
        if (!accept) {
            accept = gsl_rng_uniform(rng) <= exp(-DeltaE/(k_B*T));
        }

        if (accept) {
            E_curr = E_trial;
            // perform the swap
            int temp_atype = atype[swap_idx_a];
            atype[swap_idx_a] = atype[swap_idx_b];
            atype[swap_idx_b] = temp_atype;
            // update the counts
            N_AA = N_AA_trial;
            N_BB = N_BB_trial;
            N_AB = N_AB_trial;
            N_Aa = N_Aa_trial;
        }

        //if (T>600 && i_step>8e6) {
        //    printf("N_AA = %i, N_BB = %i, N_AB = %i, N_Aa = %i, E = %f\n",N_AA,N_BB,N_AB,N_Aa,E_curr);
        //}

        if (4*N_atoms != N_AA+N_BB+N_AB || N_AA<0 || N_BB<0 || N_AB<0) {
            printf("\nERROR: bonds calculated wrong: N_AA = %i, N_BB = %i, N_AB = %i\n", N_AA, N_BB, N_AB);
            exit(1);
        }

        // calculate various instantaneous quantities
        // save the quantities somehow
        E_out[i_step] = E_curr;
        P_out[i_step] = calc_P(N_atoms, N_Aa);
        r_out[i_step] = calc_r(N_atoms, N_AB);

        print_progress(i_step,0,n_steps-1,false);
    }
}

void perform_simulation(double T, int n_eq_steps, int n_steps,
                    int* atype, int** pos, int** nn_idxs, int idx_by_pos[n_cells][n_cells][n_cells][2],
                    bool save_steps, char* save_steps_file_path, gsl_rng* rng, int n_skip_saves,
                    double* E_avg, double* P_avg, double* r_avg, double* C,
                    double* E_std, double* P_std, double* r_std,
                    int* s_E, int* s_P, int* s_r) {

    // Monte Carlo Simulation, Metropolis Algorithm
    int n_tot_steps = n_eq_steps+n_steps;
    double* E = (double*)malloc(n_tot_steps*sizeof(double));
    double* P = (double*)malloc(n_tot_steps*sizeof(double));
    double* r = (double*)malloc(n_tot_steps*sizeof(double));

    metropolis_algorithm(N_atoms, n_tot_steps, T, atype, pos, nn_idxs, rng, E, P, r);

    // calculate the average quantities (without the equilibration steps)
    *E_avg = average(E+n_eq_steps, n_steps);
    *P_avg = average(P+n_eq_steps, n_steps);
    *r_avg = average(r+n_eq_steps, n_steps);
    *E_std = standard_deviation(E+n_eq_steps, n_steps);
    *P_std = standard_deviation(P+n_eq_steps, n_steps);
    *r_std = standard_deviation(r+n_eq_steps, n_steps);

```

```c
    // calculate the heat capacity: C = 1/k_BT * (<E^2>-<E>^2)
    double* E_squared = (double*)malloc(n_steps*sizeof(double));
    elementwise_multiplication(E_squared, E+n_eq_steps, E+n_eq_steps, n_steps);
    double E_squared_avg = average(E_squared, n_steps);
    *C = (E_squared_avg - (*E_avg)*(*E_avg))/(k_B*T*T);


    // calculate the statistical inefficiency

    printf("Calculate statistical inefficiencies...\n");
    print_progress(0,0,3,true);
    *s_E = calc_s_corr(E+n_eq_steps, n_steps);
    print_progress(1,0,3,false);
    *s_P = calc_s_corr(P+n_eq_steps, n_steps);
    print_progress(2,0,3,false);
    *s_r = calc_s_corr(r+n_eq_steps, n_steps);
    print_progress(3,0,3,false);


    // save a few simulations with E(t),P(t),...
    if (save_steps) {
        int n_save_steps = n_tot_steps / n_skip_saves;
        printf("Save %i steps from %i for T = %.4f K ...\n", n_save_steps, n_tot_steps, T);
        // write it to a file
        FILE* file = fopen(save_steps_file_path, "w");
        fprintf(file, "# {\"T[K]\": %.2f, \"n_eq_steps\":%i, \"n_steps\": %i, \"n_skip_saves\": %i,\n", T, n_eq_steps, n_steps, n_skip_saves);
        fprintf(file, "# \"P\": %.10f, \"E[eV]\": %.10f, \"C[eV/K]\": %.10f, \"r\": %.10f}\n", *P_avg, *E_avg, *C, *r_avg);
        fprintf(file, "# i_step, E[eV], P, r\n");
        for (int i_step=0; i_step<n_eq_steps+n_steps; i_step+=n_skip_saves) {
            fprintf(file, "%i, %.10e, %.10e, %.10e\n", i_step, E[i_step], P[i_step], r[i_step]);
        }
        fclose(file);

        // calculate the correlation function
        printf("Calculate and save the correlation function...\n");
        int k_min = (*s_E)/100;
        int k_max = (*s_E)*10;
        int k_step = (*s_E)/100;
        int n_k = (k_max-k_min)/k_step + 1;
        int* k = (int*)malloc(n_k*sizeof(int));
        double* Phi = (double*)malloc(n_k*sizeof(double));
        for (int i=0; i<n_k; i++) {
            k[i] = k_min + i*k_step;
        }
        calc_all_corr_func(E+n_eq_steps, n_steps, k, Phi, n_k);

        // save the file to data/corr_simulations/
        save_steps_file_path[5] = 'c';
        save_steps_file_path[6] = 'o';
        save_steps_file_path[7] = 'r';
        save_steps_file_path[8] = 'r';
        file = fopen(save_steps_file_path, "w");
        fprintf(file, "# {\"s_E\": %i, \"T[K]\": %.2f}\n", *s_E, T);
        fprintf(file, "# k, Phi(E,k)\n");
        for (int i=0; i<n_k; i++) {
            fprintf(file, "%i, %.10f\n",k[i], Phi[i]);
        }
        fclose(file);
        free(k);
        free(Phi);

        // calculate the block average statistical inefficiencies
        printf("Calculate and save the block averaging...\n");
        int n_B = 1000;
        int B_min = 1e4;
        int B_max = 1e6;
        double logB_min = log10(B_min);
        double logB_max = log10(B_max);
        double dlogB = (logB_max-logB_min)/(n_B-1);
```

```
        int* B = (int*)malloc(n_B*sizeof(int));
        double* s = (double*)malloc(n_B*sizeof(double));
        print_progress(0,0,n_B,true);
        for (int j=0; j<n_B; j++) {
            B[j] = (int)pow(10.,logB_min + j*dlogB);
            s[j] = calc_s_block_avg(E+n_eq_steps, (*E_std)*(*E_std), B[j], n_steps);
            print_progress(j+1,0,n_B,false);
        }

        // write the block average statistical inefficiencies to a file
        // to data/blok_simulations/
        save_steps_file_path[5] = 'b';
        save_steps_file_path[6] = 'l';
        save_steps_file_path[7] = 'o';
        save_steps_file_path[8] = 'k';
        file = fopen(save_steps_file_path, "w");
        fprintf(file, "# {\"s_E\": %i, \"T[K]\": %.2f}\n", *s_E, T);
        fprintf(file, "# B, s_E\n");
        for (int i=0; i<n_B; i++) {
            fprintf(file, "%i, %.10f\n",B[i], s[i]);
        }
        free(B);
        free(s);
        fclose(file);
    }

    // tidy up
    free(E);
    free(E_squared);
    free(P);
    free(r);
}

int
run(
    int argc,
    char *argv[]
   )
{
    // Write your code here
    // This makes it possible to test
    // 100% of you code
    gsl_rng* rng = init_rng(42);

    int n_eq_steps = 0.5e6;
    int n_steps = 4.5e6;
    int n_skip_saves = 10;

    // make 3 sections of temperature, where around the critical temperature it more dense
    int T_0 = 300;
    int T_1 = 600;
    int T_2 = 800;
    int T_3 = 1000;
    // delta T steps in the sections
    int dT01 = 20;
    int dT12 = 1;
    int dT23 = 20;
    // number of T steps in each section
    int n_T01 = (T_1-T_0)/dT01;
    int n_T12 = (T_2-T_1)/dT12;
    int n_T23 = (T_3-T_2)/dT23 + 1; // plus the last one
    int n_T = n_T01 + n_T12 + n_T23;

    int T_saves[] = {300, 650, 700, 750, 900};
    int n_T_saves = sizeof(T_saves)/sizeof(T_saves[0]);

    double* T = (double*)malloc(n_T*sizeof(double));
    double temp_T = T_0;
    for (int i=0; i<n_T; i++) {
        T[i] = temp_T;
        if (T_0 <= temp_T && temp_T < T_1) {
```

```
            temp_T += dT01;
        } else if (T_1 <= temp_T && temp_T < T_2) {
            temp_T += dT12;
        } else if (T_2 <= temp_T && temp_T < T_3) {
            temp_T += dT23;
        }
    }

    double* E = (double*)malloc(n_T*sizeof(double));
    double* P = (double*)malloc(n_T*sizeof(double));
    double* r = (double*)malloc(n_T*sizeof(double));
    double* C = (double*)malloc(n_T*sizeof(double));
    double* E_std = (double*)malloc(n_T*sizeof(double));
    double* P_std = (double*)malloc(n_T*sizeof(double));
    double* r_std = (double*)malloc(n_T*sizeof(double));
    int* s_E = (int*)malloc(n_T*sizeof(int));
    int* s_P = (int*)malloc(n_T*sizeof(int));
    int* s_r = (int*)malloc(n_T*sizeof(int));


    // init the lattice completely ordered
    int* atype = (int*)malloc(N_atoms*sizeof(int));
    int** pos = create_2D_int_array(N_atoms, 4);
    int** nn_idxs = create_2D_int_array(N_atoms, 8);
    int idx_by_pos[n_cells][n_cells][n_cells][2];
    construct_bcc_binary_alloy(n_cells, atype, pos, nn_idxs, idx_by_pos);

    // perform the simulations
    for (int i=0; i<n_T; i++) {
        bool save = false;
        char save_step_file_path[100];
        // check if this step should be saved
        for (int j=0; j<n_T_saves; j++) {
            if (T_saves[j] == T[i]) {
                save = true;
                sprintf(save_step_file_path, "data/full_simulations/H2a_simsteps_%0*i_T%.0fK.csv"↩
                    , ((int)log10(n_T))+1, i, T[i]);
                break;
            }
        }
        perform_simulation(T[i], n_eq_steps, n_steps,
                        atype, pos, nn_idxs, idx_by_pos,
                        save, save_step_file_path, rng, n_skip_saves,
                        E+i, P+i, r+i, C+i,
                        E_std+i, P_std+i, r_std+i,
                        s_E+i, s_P+i, s_r+i);
        printf("%i/%i done. (T = %.2f K)\n", i+1,n_T,T[i]);
    }

    // save the results
    FILE* file = fopen("data/H2a.csv", "w");
    fprintf(file, "# {\"n_eq_steps\": %i, \"n_steps\": %i}\n", n_eq_steps, n_steps);
    fprintf(file, "# T[K], E[eV], E_std[eV], s_E, P, P_std, s_P, r, r_std, s_r, C[eV/K]\n");
    for (int i=0; i<n_T; i++) {
        fprintf(file, "%.5f, %.10f, %.10f, %i, %.10f, %.10f, %i, %.10f, %.10f, %i, %.10f\n",
                T[i],
                E[i], E_std[i], s_E[i],
                P[i], P_std[i], s_P[i],
                r[i], r_std[i], s_r[i],
                C[i]);
    }
    fclose(file);

    // tidy up
    free(T);
    free(E);
    free(P);
    free(r);
    free(C);
    free(E_std);
    free(P_std);
```

```
    free(r_std);
    free(s_E);
    free(s_P);
    free(s_r);
    gsl_rng_free(rng);
    free(atype);
    destroy_2D_int_array(pos);
    destroy_2D_int_array(nn_idxs);
    return 0;
}
```

Functions for calculations and initialization of the lattice: `lattice_tools.c`

```
#include <stdio.h>
#include <stdlib.h>

// parameters of the simulation
const double E_AA = -436e-3; // eV E_CuCu
const double E_BB = -113e-3; // eV E_ZnZn
const double E_AB = -294e-3; // eV E_CuZn

/*
 * Constructs the arrays that describe a binary alloy with perfect ordering
 * @n_cells - number of unit cells per direction -> N_atoms = 2*n_cells^3
 * @atype - array of shape (N_atoms,) to be filled with the atom type
 * @pos - array of shape (N_atoms,4) to be filled with the atom coordinates (x,y,z,w)
 * x,y,z determine the unit cell and w=0,1 determines if its at the corner or the center
 * @nn_idxs - array of shape (N_atoms,8) to be filled with the indices of the nearest neighbors
 * @idx_by_pos - array of shape (n_cells, n_cells, n_cells, 2) to be filled
 * with the index of a specific position (x,y,z,w)
 */
void construct_bcc_binary_alloy(int n_cells, int* atype, int** pos,
                                int** nn_idxs, int idx_by_pos[n_cells][n_cells][n_cells][2]) {
    int N_atoms = 2*n_cells*n_cells*n_cells;

    int i = 0;

    // construct the lattice
    for (int x=0; x<n_cells; x++) {
    for (int y=0; y<n_cells; y++) {
    for (int z=0; z<n_cells; z++) {
    for (int w=0; w<2; w++) {
        // w==0 -> corner atom
        // w==1 -> center atom
        atype[i] = w;
        pos[i][0] = x;
        pos[i][1] = y;
        pos[i][2] = z;
        pos[i][3] = w;
        idx_by_pos[x][y][z][w] = i;
        i++;
    }
    }
    }
    }

    // assign the nearest neighbors
    // loop over all lattice positions
    for (int x=0; x<n_cells; x++) {
    for (int y=0; y<n_cells; y++) {
    for (int z=0; z<n_cells; z++) {
    for (int w=0; w<2; w++) {
        i = idx_by_pos[x][y][z][w];
```

```c
        int j = 0;

        // the nearest neighbors have oposite w
        // the positions differ by (-1 or 0) if w==0 or by (0 or 1) if w==1
        // loop over all 8 nearest neighbors
        for (int dx=w-1; dx<=w; dx++) {
        for (int dy=w-1; dy<=w; dy++) {
        for (int dz=w-1; dz<=w; dz++) {
            // position of this particular nearest neighbor
            int x_nn = x+dx;
            int y_nn = y+dy;
            int z_nn = z+dz;
            int w_nn;
            if (w==0) w_nn = 1;
            if (w==1) w_nn = 0;
            // periodic boundary conditions:
            if (x_nn==-1) x_nn = n_cells-1;
            if (y_nn==-1) y_nn = n_cells-1;
            if (z_nn==-1) z_nn = n_cells-1;
            if (x_nn==n_cells) x_nn = 0;
            if (y_nn==n_cells) y_nn = 0;
            if (z_nn==n_cells) z_nn = 0;
            nn_idxs[i][j] = idx_by_pos[x_nn][y_nn][z_nn][w_nn];
            // sanity check:
            if (nn_idxs[i][j] >= N_atoms || nn_idxs[i][j]<0) {
                printf("ERROR: nn_idxs[%i][%i] = %i even though N_atoms = %i\n",i,j,nn_idxs[i][j]],N_atoms);
                exit(1);
            }
            j++;
        }
        }
        }
    }
    }
    }
    }
}



void count_bonds(int N_atoms, int* atype, int** nn_idxs,
                int* N_AA, int* N_BB, int* N_AB) {
    // count the types of neares neighbor pairs
    (*N_AA) = 0;
    (*N_BB) = 0;
    (*N_AB) = 0;

    for (int i=0; i<N_atoms; i++) {
        int atype_a = atype[i];
        for (int j=0; j<8; j++) {
            int atype_b = atype[nn_idxs[i][j]];
            if (atype_a == 0 && atype_b == 0) {
                (*N_AA)++;
            } else if (atype_a == 1 && atype_b == 1) {
                (*N_BB)++;
            } else {
                (*N_AB)++;
            }
        }
    }
    // but now we counted each bond twice
    if ((*N_AA)%2 == 1 || (*N_BB)%2 == 1 || (*N_AB)%2 == 1) {
        perror("ERROR: number of pairs should be even");
        exit(1);
    }
    (*N_AA) /= 2;
    (*N_BB) /= 2;
    (*N_AB) /= 2;
}

```

```c
double calc_energy(int N_AA, int N_BB, int N_AB) {
    return N_AA*E_AA + N_BB*E_BB + N_AB*E_AB;
}

double calc_P(int N_atoms, int N_Aa) {
    // long-range order parameter P
    int N_A = N_atoms/2;
    return 2.*((double)N_Aa)/N_A - 1.;
}

double calc_r(int N_atoms, int N_AB) {
    // short-range order parameter r
    int N = N_atoms/2;
    return (N_AB-4.*N)/(4.*N);
}
```

Functions for calculations of the statistical inefficiency: `statistical_ineff.c`

```c
#include <stdio.h>
#include <math.h>
#include "tools.h"

#include "statistical_ineff.h"

double calc_corr_func(double* f, double f2_mean, double f_mean2, int k, int n_f) {
    // f2_mean = <f^2> and f_mean2 = <f>^2
    // calculate <f_{i+k}*f_i> (each product is saved in ff)
    int n_ff = n_f - k;
    double *ff = (double*)malloc(n_f*sizeof(double));
    elementwise_multiplication(ff, f+k, f, n_ff);
    double ff_mean = average(ff, n_ff);
    free(ff);

    return (ff_mean - f_mean2)/(f2_mean - f_mean2);
}

void calc_all_corr_func(double* f, int n_f, int* k, double* Phi, int n_k) {
    // calculate corr func for all given k

    // first shift f
    double f_mean = average(f, n_f);
    double* f_shifted = (double*)malloc(n_f*sizeof(double));
    for (int i=0; i<n_f; i++) {
        f_shifted[i] = f[i] - f_mean;
    }
    // calculate <f^2> and <f>^2
    double* f_squared = (double*)malloc(n_f*sizeof(double));
    elementwise_multiplication(f_squared, f_shifted, f_shifted, n_f);
    double f2_mean = average(f_squared, n_f);
    double f_mean2 = 0;
    free(f_squared);

    print_progress(0,0,n_k,true);
    for (int i=0; i<n_k; i++) {
        Phi[i] = calc_corr_func(f_shifted, f2_mean, f_mean2, k[i], n_f);
        print_progress(i+1,0,n_k,false);
    }

    free(f_shifted);
}

double calc_s_corr(double* f, int n_f) {
    // calculate the statistical inefficiency through the correlation function
```

```c
    // use binary search to find the k so that the correlation function is at exp(-2)

    // first shift f
    double f_mean = average(f, n_f);
    double* f_shifted = (double*)malloc(n_f*sizeof(double));
    for (int i=0; i<n_f; i++) {
        f_shifted[i] = f[i] - f_mean;
    }
    // calculate <f^2> and <f>^2
    double* f_squared = (double*)malloc(n_f*sizeof(double));
    elementwise_multiplication(f_squared, f_shifted, f_shifted, n_f);
    double f2_mean = average(f_squared, n_f);
    double f_mean2 = 0;
    free(f_squared);

    // do binary search to find k0
    // for which phi(k0) is just below exp(-2)
    // and phi(k0-1) is just above exp(-2)
    // assumption: phi(k<k0)>= exp(-2) and phi(k>k0)< exp(-2)
    // (hopefully true)
    double Phi_wanted = exp(-2);

    // start with a rough search going for higher values until its lower
    double rough_step_scaler = 2.0;
    double temp_k = 1;
    double temp_Phi = calc_corr_func(f_shifted, f2_mean, f_mean2, (int)(temp_k), n_f);
    int n_steps_rough = 1;
    while (temp_Phi > Phi_wanted && (temp_k*rough_step_scaler) < n_f) {
        temp_k *= rough_step_scaler;
        temp_Phi = calc_corr_func(f_shifted, f2_mean, f_mean2, (int)(temp_k), n_f);
        n_steps_rough++;
    }
    int k_low = (int)(temp_k/rough_step_scaler);
    int k_high = (int)temp_k;
    double Phi_k_low = calc_corr_func(f_shifted, f2_mean, f_mean2, k_low, n_f);
    double Phi_k_high = temp_Phi;
    int n_steps_binary = 1;
    while (k_high-k_low > 1) {
        if (Phi_k_low < Phi_wanted || Phi_k_high >= Phi_wanted) {
            // assumption probably does not hold
            // Phi_wanted is not in the range k_low, k_high anymore
            printf("ERROR: Binary search failed. (Phi_wanted = %.5f)\n", Phi_wanted);
            printf("ERROR: k_low = %i, Phi_k_low = %.5f\n", k_low, Phi_k_low);
            printf("ERROR: k_high = %i, Phi_k_high = %.5f\n", k_high, Phi_k_high);
            exit(1);
        }
        int k_mid = (k_high+k_low)/2;
        double Phi_k_mid = calc_corr_func(f_shifted, f2_mean, f_mean2, k_mid, n_f);
        if (Phi_k_mid >= Phi_wanted) {
            k_low = k_mid;
            Phi_k_low = Phi_k_mid;
        } else {
            k_high = k_mid;
            Phi_k_high = Phi_k_mid;
        }
        //printf("k_low = %i; k_high = %i ; Phi_k_low = %f; Phi_k_high = %f\n",k_low,k_high,Phi_k_low,Phi_k_high);
        n_steps_binary++;
    }
    // check if Phi wanted is still in the found range
    if (Phi_k_low < Phi_wanted || Phi_k_high >= Phi_wanted) {
        // assumption probably does not hold
        // Phi_wanted is not in the range k_low, k_high anymore
        printf("ERROR: Binary search found the wrong range. (Phi_wanted = %.5f)\n", Phi_wanted);
        printf("ERROR: k_low = %i, Phi_k_low = %.5f\n", k_low, Phi_k_low);
        printf("ERROR: k_high = %i, Phi_k_high = %.5f\n", k_high, Phi_k_high);
        exit(1);
    }

    //printf("n_steps_rough = %i; n_steps_binary_search = %i; s_corr = %i\n", n_steps_rough, n_steps_binary, k_high);
```

```c
    free(f_shifted);
    return k_high;
}


double calc_s_block_avg(double* f, double f_variance, int B, int n_f) {
    int n_F = n_f/B;
    // the last block might not be full
    //if (n_f%B!=0) {
    //    n_F++;
    //}
    double* F = (double*)malloc(n_F*sizeof(double));
    for (int i=0; i<n_F; i++) {
        F[i] = average(f+i*B, B);
    }
    double F_variance = standard_deviation(F, n_F);
    F_variance *= F_variance;
    free(F);
    return B*(F_variance/f_variance);
}
```

Utility functions for C: `tools.c`

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

#include "tools.h"

void constant_multiplication(double* res,
                             double* v1,
                             double a1,
                             unsigned int len)
{
    for(int i=0;i<len;i++){
        res[i] = a1*v1[i];
    }
}

void
elementwise_addition(
             double *res,
             double *v1,
             double *v2,
             unsigned int len
                )
{
    for(int i=0;i<len;i++){
        res[i] = v1[i] + v2[i];
    }
}

void
elementwise_multiplication(
               double *res,
               double *v1,
               double *v2,
               unsigned int len
                      )
{
    for(int i=0;i<len;i++){
        res[i] = v1[i] * v2[i];
    }
```

```
}

double
dot_product(
        double *v1,
        double *v2,
        unsigned int len
       )
{
    double res = 0;
    for(int i=0;i<len;i++){
        res += v1[i] * v2[i];
    }
    return res;
}

double**
create_2D_array(
        unsigned int nrows,
        unsigned int ncols
           )
{
    // allocate 1D array of doubles containing the whole matrix
    double* linear_array = (double*)malloc(nrows*ncols*sizeof(double));
    // allocate 1D array of pointers to doubles containing the pointers to each row starting point
    double** array = (double**)malloc(nrows*sizeof(double*));
    // let each row pointer point to the correct address
    for(int row=0;row<nrows;row++){
        array[row] = linear_array + row*ncols;
    }
    return array;
}

void
destroy_2D_array(
         double **array
        )
{
    // free the linear_array
    free(array[0]);
    // free the pointers array
    free(array);
}

int**
create_2D_int_array(
        unsigned int nrows,
        unsigned int ncols
           )
{
    // allocate 1D array of doubles containing the whole matrix
    int* linear_array = (int*)malloc(nrows*ncols*sizeof(int));
    // allocate 1D array of pointers to doubles containing the pointers to each row starting point
    int** array = (int**)malloc(nrows*sizeof(int*));
    // let each row pointer point to the correct address
    for(int row=0;row<nrows;row++){
        array[row] = linear_array + row*ncols;
    }
    return array;
}

void
destroy_2D_int_array(
         int **array
        )
{
    // free the linear_array
    free(array[0]);
    // free the pointers array
```

```
    free(array);
}

void
matrix_multiplication(
              double **result,
              double **m1,
              double **m2,
              unsigned int m,
              unsigned int n
             )
{
    // https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication#Schoolbook_algorithm
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            // calculate matrix element in row i, col j
            result[i][j] = 0;
            for(int k=0;k<m;k++){
                result[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }
}

double
vector_norm(
        double *v1,
        unsigned int len
       )
{
    return sqrt(dot_product(v1, v1, len));
}


void
normalize_vector(
         double *v1,
         unsigned int len
        )
{
    double norm = vector_norm(v1, len);
    constant_multiplication(v1, v1, 1./norm, len);
}

double
average(
    double *v1,
    unsigned int len
       )
{
    double res = 0;
    for(int i=0;i<len;i++){
        res += v1[i];
    }
    return res/len;
}


double
standard_deviation(
               double *v1,
               unsigned int len
              )
{
    /* https://numpy.org/doc/stable/reference/generated/numpy.std.html
     * The standard deviation is the square root of the average of the squared deviations from the mean,
     * i.e., std = sqrt(mean(x)), where x = abs(a - a.mean())**2.
     * std(v1) = sqrt(sum(v1 - v1_mean)^2 / len(v1)) */
    double mean = average(v1, len);
```

```
    double res = 0;
    for(int i=0;i<len;i++){
        res += (v1[i] - mean)*(v1[i] - mean);
    }
    return sqrt(res/len);
}

double
distance_between_vectors(
             double *v1,
             double *v2,
             unsigned int len
                    )
{
    // dist(v1,v2) = |v1 - v2|
    double res = 0;
    for(int i=0;i<len;i++){
        res += (v1[i] - v2[i])*(v1[i] - v2[i]);
    }
    return sqrt(res);
}



void print_vector(double* vec, int length){
    printf("[");
    for(int i=0; i < length; i++){
        printf("%.2f, ", vec[i]);
    }
    printf("\b\b]\n");
}
void print_vector_int(int* vec, int length){
    printf("[");
    for(int i=0; i < length; i++){
        printf("%i, ", vec[i]);
    }
    printf("\b\b]\n");
}

void fprint_vector(FILE* file, double* vec, int length){
    for(int i=0; i < length; i++){
        fprintf(file, "%.6f", vec[i]);
        if(i<length-1) fprintf(file, ", ");
    }
    fprintf(file, "\n");
}

void print_matrix(double** mat, int n, int m){
    printf("[");
    for(int i=0; i < n; i++){
        printf("[");
        for(int j=0; j < m; j++){
            printf("%.2f, ", mat[i][j]);
        }
        printf("\b\b],");
        if(i<n-1) printf("\n");
    }
    printf("\b]\n");
}

void print_matrix_int(int** mat, int n, int m){
    printf("[");
    for(int i=0; i < n; i++){
        printf("[");
        for(int j=0; j < m; j++){
            printf("%i, ", mat[i][j]);
        }
        printf("\b\b],");
        if(i<n-1) printf("\n");
    }
    printf("\b]\n");
```

```
}
void print_matrix_stack(int n, int m, double mat[][m]){
    printf("[");
    for(int i=0; i < n; i++){
        printf("[");
        for(int j=0; j < m; j++){
            printf("%.2f, ", mat[i][j]);
        }
        printf("\b\b],");
        if(i<n-1) printf("\n");
    }
    printf("\b]\n");
}

void fprint_matrix(FILE* file, double** mat, int n, int m){
    // write matrix to a file
    for(int i=0; i < n; i++){
        for(int j=0; j < m; j++){
            fprintf(file, "%.6f", mat[i][j]);
            if(j<m-1) fprintf(file, ", ");
            else fprintf(file, "\n");
        }
    }
}


void init_matrix_stack(int n, int m, double mat[n][m], double value){
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            mat[i][j] = value;
        }
    }
}

gsl_rng* init_rng(int seed){
    // seed = 0 means the seed is random
    // set up the random number generator from GSL
    gsl_rng_env_setup();
    const gsl_rng_type* T = gsl_rng_default;
    gsl_rng* rng = gsl_rng_alloc(T);
    if(seed != 0){
        gsl_rng_set(rng, seed);
    }

    return rng;
}

void print_progress(double current_value, double min_value, double max_value, bool start_new) {
    // shows the progress, but only at each round percentage, so that it does not slow down the computation
    static int percent;
    if (start_new) {
        percent = -1;
    }
    int curr_percent = (int)(100*(current_value-min_value)/(max_value-min_value));
    if (curr_percent != percent) {
        percent = curr_percent;
        printf("\33[2K\r%i %% (%.0f / %.0f)", percent, current_value, max_value);
        if(percent >= 100) {
            printf("\n");
        }
        fflush(stdout);
    }
}
```

Plotting of task 1: `task1.py`

```python
# %%
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from tqdm.auto import tqdm

E_AA = -436e-3 # eV
E_BB = -113e-3 # eV
E_AB = -294e-3 # eV

k_B = 8.617333262e-5 # eV/K

N = 100000

E_0_per_N = 2*(E_AA+E_BB+2*E_AB)
dE = E_AA + E_BB - 2*E_AB

T_c = 2*dE/k_B
T_c_exp = 468 + 273.15

print(f"T_(c,mfa) = {T_c:.2f} K ; T_(c,exp) = {T_c_exp:.2f} K")

T = np.linspace(0,1200,1200) # K
#T = np.linspace(906,1200,1000) # K

def F_per_N(P,T):
    if P==1 or P==-1:
        return - 2*k_B*T*np.log(2)
    return E_0_per_N - 2*P**2*dE - 2*k_B*T*np.log(2) + k_B*T*((1+P)*np.log(1+P)+(1-P)*np.log(1-P)
        )

P_0 = 0.5
P = []
for T_i in tqdm(T):
    P_0 = minimize(F_per_N, P_0, args=T_i, bounds=((0,1),), tol=1e-10).x[0]
    P.append(P_0)
P = np.array(P)

N_AA_per_N = 2*(1-P**2)
N_BB_per_N = 2*(1-P**2)
N_AB_per_N = 4*(1+P**2)

E_per_N = N_AA_per_N*E_AA + N_BB_per_N*E_BB + N_AB_per_N*E_AB
E_MFA_per_N = E_0_per_N - 2*P**2*dE

C_per_N = np.gradient(E_per_N, T)

fig, axs = plt.subplots(2,3, figsize=(12,6))

#fig.suptitle(f"$T_c = {T_c:.3f} \, \mathrm{{K}}$")

plt.sca(axs[0][0])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.5)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.5)
plt.plot(T,P, color="C2")
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$P$")

plt.sca(axs[0][1])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.5)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.5)
plt.plot(T,E_per_N, color="C0")
#plt.plot(T,E_MFA_per_N)
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$E/N \:/\: \mathrm{eV}$")

plt.sca(axs[0][2])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.5)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.5)
plt.plot(T,C_per_N, color="C1")
```

```python
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$C/N \:/\: \mathrm{eV \, K^{-1}}$")

# log scale plots below
plt.sca(axs[1][0])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.5)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.5)
plt.plot(T,P, color="C2")
plt.yscale("log")
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$P$")

plt.sca(axs[1][1])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.5)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.5)
plt.plot(T,E_per_N[-1]-E_per_N, color="C0")
plt.yscale("log")
#plt.plot(T,E_MFA_per_N)
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(rf"$E(T={T[-1]:.0f}\mathrm{{K}})/N - E/N \:/\: \mathrm{{eV}}$")

plt.sca(axs[1][2])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.5)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.5)
plt.plot(T,C_per_N, color="C1")
plt.yscale("log")
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$C/N \:/\: \mathrm{eV \, K^{-1}}$")

plt.tight_layout()
plt.savefig("plots/task1.pdf")



# plot 3 different F(P) to visualize the minimization of F

def F_per_N(P,T):
    res = np.zeros_like(P)
    mask = (P==1)|(P==-1)
    res[mask] = - 2*k_B*T*np.log(2)
    P = P[~mask]
    res[~mask] = E_0_per_N - 2*P**2*dE - 2*k_B*T*np.log(2) + k_B*T*((1+P)*np.log(1+P)+(1-P)*np.log(1-P))
    return res

Ts = [100,500,750,10000] # K

P_lin = np.linspace(-1,1,1000)

fig, axs = plt.subplots(1,len(Ts),figsize=(len(Ts)*3,2.5))
for i,T_i in enumerate(Ts):
    F_ = F_per_N(P_lin, T_i)
    plt.sca(axs[i])
    plt.plot(P_lin, F_per_N(P_lin, T_i))
    plt.ylim(np.min(F_[F_<-1.5])*1.001, np.max(F_[F_<-1.5])*0.999)
    plt.xlabel(r"$P$")
    plt.ylabel(rf"$F (P, T={T_i:.0f}\mathrm{{K}}) \: / \: \mathrm{{eV}}$")

plt.tight_layout()
plt.savefig("plots/task1_F.pdf")
```

Plotting of task 2: `task2.py`

```python
# %%
import numpy as np
```

```python
import matplotlib.pyplot as plt
from pathlib import Path
import json


# plot the results
print(f"Plot final results...")
T, E, E_std, s_E, P, P_std, s_P, r, r_std, s_r, C = np.genfromtxt("data/H2a.csv", delimiter=",", unpack=True)

# get header metadata
with open("data/H2a.csv", "r") as file:
    metadata_str = "".join([file.readline() for i in range(1)])
    metadata_str = metadata_str.replace("# ","")
    metadata = json.loads(metadata_str)

n_eq_steps = metadata["n_eq_steps"]
n_steps = metadata["n_steps"]

# calculate T_c
k_B = 8.617333262e-5 # eV/K
E_AA = -436e-3 # eV
E_BB = -113e-3 # eV
E_AB = -294e-3 # eV
dE = E_AA + E_BB - 2*E_AB
T_c = 2*dE/k_B
T_c_exp = 468 + 273.15

E_err = np.sqrt(s_E*E_std**2/n_steps)
P_err = np.sqrt(s_P*P_std**2/n_steps)
r_err = np.sqrt(s_r*r_std**2/n_steps)

fig, axs = plt.subplots(4,1, figsize=(10,10))

#fig.suptitle(f"$T_{{c,task1}} = {T_c:.3f} \, \mathrm{{K}}$, $N_\mathrm{{eq}} = {n_eq_steps:.1e}$, $N = {n_steps:.1e}$")


plt.sca(axs[0])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.9)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.9)
plt.fill_between(T, E-E_err, E+E_err, color="C0", alpha=0.3)
plt.plot(T,E, color="C0")
plt.ylim(-2360,-2280)
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$E \:/\: \mathrm{eV}$")
plt.grid()

plt.sca(axs[1])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.9)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.9)
plt.plot(T,C, color="C1")
plt.ylim(0,0.5)
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$C \:/\: \mathrm{eV \, K^{-1}}$")
plt.grid()

plt.sca(axs[2])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.9)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.9)
plt.fill_between(T, np.abs(P)-P_err, np.abs(P)+P_err, color="C2", alpha=0.3)
plt.plot(T,P, color="C4", linestyle=":", alpha=0.8, label="$P$")
plt.plot(T,np.abs(P), color="C2", label="$|P|$")
plt.ylim(-0.5,1)
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$P$")
plt.grid()
plt.legend()

plt.sca(axs[3])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.9)
```

```
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.9)
plt.fill_between(T, r-r_err, r+r_err, color="C3", alpha=0.3)
plt.plot(T,r, color="C3")
plt.ylim(0,1)
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$r$")
plt.grid()

plt.tight_layout()
plt.savefig("plots/task2.pdf")

# plot the uncertainties
fig, axs = plt.subplots(3,1, figsize=(10,7.5))

plt.sca(axs[0])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.9)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.9)
plt.plot(T,E_err, color="C0")
plt.yscale("log")
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$\sigma_{\langle E \rangle} \:/\: \mathrm{eV}$")
plt.grid()

plt.sca(axs[1])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.9)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.9)
plt.plot(T,P_err, color="C2")
plt.yscale("log")
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$\sigma_{\langle P \rangle}$")
plt.grid()

plt.sca(axs[2])
plt.axvline(T_c, linestyle=":", color="k", alpha=0.9)
plt.axvline(T_c_exp, linestyle="--", color="k", alpha=0.9)
plt.plot(T,r_err, color="C3")
plt.yscale("log")
plt.xlabel(r"$T \:/\: \mathrm{K}$")
plt.ylabel(r"$\sigma_{\langle r \rangle}$")
plt.grid()

plt.tight_layout()
plt.savefig("plots/task2_uncertainties.pdf")


# plot the statistical inefficiency
plt.figure(figsize=(8,3))
plt.plot(T,s_E, "C0", label="$s(E)$ and \n$s(r)$")
plt.plot(T,s_P, "C2", label="$s(P)$")
#plt.plot(T,s_r, "C3", linestyle=":", label="$s(r)$")
plt.yscale("log")
plt.xlabel(r"$T \: / \: \mathrm{K}$")
plt.ylabel(r"$s$ (statistical inefficiency)")
plt.legend()
plt.tight_layout()
plt.savefig("plots/H2a_stat_ineff.pdf")


# Plot a few full simulations
full_simulations_dir = Path("data/full_simulations/")
fig, axs = plt.subplots(3,1, figsize=(10,8))

#fig.suptitle(f"$T = {T:2f} \,\mathrm{{K}}$, $N_\mathrm{{eq}} = {n_eq_steps:.1e}$, \n$<P> = {P_avg:.4f} \mathrm{{eV}}$, $<E> = {E_avg:.4f} \mathrm{{eV}}$, $<C> = {C:.4f} \mathrm{{eV/K}}$, $<r> = {r_avg:.4f}$")
for full_simulation_file in full_simulations_dir.iterdir():
    # get header metadata
    with open(full_simulation_file, "r") as file:
        metadata_str = "".join([file.readline() for i in range(2)])
        metadata_str = metadata_str.replace("# ","")
        metadata = json.loads(metadata_str)
```

```python

    T = metadata["T[K]"]
    n_eq_steps = metadata["n_eq_steps"]

    E_avg = metadata["E[eV]"]
    P_avg = metadata["P"]
    r_avg = metadata["r"]
    C = metadata["C[eV/K]"]

    print(f"Plot full simulation T={T:.0f} K ...")

    i_step, E, P, r = np.genfromtxt(full_simulation_file, delimiter=",", unpack=True)

    plt.sca(axs[0])
    plt.plot(i_step, P, label=f"$T = {T:.0f} \\mathrm{{K}}$", rasterized=True)

    plt.sca(axs[1])
    plt.plot(i_step, E, label=f"$T = {T:.0f} \\mathrm{{K}}$", rasterized=True)

    plt.sca(axs[2])
    plt.plot(i_step, r, label=f"$T = {T:.0f} \\mathrm{{K}}$", rasterized=True)


plt.sca(axs[0])
plt.axvline(n_eq_steps, linestyle="--", color="k", alpha=0.5, label=rf"$N_\mathrm{{eq}} = {n_eq_steps:.1e}$")
plt.xlabel("simulation step")
plt.ylabel("P")
plt.legend(loc="center right")

plt.sca(axs[1])
plt.axvline(n_eq_steps, linestyle="--", color="k", alpha=0.5, label=rf"$N_\mathrm{{eq}} = {n_eq_steps:.1e}$")
plt.xlabel("simulation step")
plt.ylabel("energy / eV")
plt.legend(loc="center right")

plt.sca(axs[2])
plt.axvline(n_eq_steps, linestyle="--", color="k", alpha=0.5, label=rf"$N_\mathrm{{eq}} = {n_eq_steps:.1e}$")
plt.xlabel("simulation step")
plt.ylabel("r")
plt.legend(loc="center right")

plt.tight_layout()
plt.savefig(f"plots/H2a_simsteps.pdf")


# Plot a few full simulation correlation functions
corr_simulations_dir = Path("data/corr_simulations/")
plt.figure(figsize=(4,3))

for i,corr_simulation_file in enumerate(corr_simulations_dir.iterdir()):
    # get header metadata
    with open(corr_simulation_file, "r") as file:
        metadata_str = "".join([file.readline() for i in range(1)])
        metadata_str = metadata_str.replace("# ","")
        metadata = json.loads(metadata_str)

    T = metadata["T[K]"]
    s_E = metadata["s_E"]

    print(f"Plot correlation function T={T:.0f} K ...")

    k, Phi = np.genfromtxt(corr_simulation_file, delimiter=",", unpack=True)

    plt.plot(k, Phi, color=f"C{i}", label=f"$T = {T:.0f} \\mathrm{{K}}$", rasterized=True)
    plt.axvline(s_E, linestyle="--", color=f"C{i}", alpha=0.5)

plt.axhline(np.exp(-2), linestyle=":", color="k", alpha=0.5, label=f"$\exp(-2)$")
plt.xscale("log")
```

```python
#plt.yscale("log")
plt.xlabel("$k$")
plt.ylabel(r"$\Phi_E(k)$")
plt.legend(loc="lower left")

plt.tight_layout()
plt.savefig(f"plots/H2a_corr_func.pdf")




# Plot a few simulation block averages
blok_simulations_dir = Path("data/blok_simulations/")
plt.figure(figsize=(4,3))

for i,blok_simulation_file in enumerate(blok_simulations_dir.iterdir()):
    # get header metadata
    with open(blok_simulation_file, "r") as file:
        metadata_str = "".join([file.readline() for i in range(1)])
        metadata_str = metadata_str.replace("# ","")
        metadata = json.loads(metadata_str)

    T = metadata["T[K]"]
    s_E = metadata["s_E"]

    print(f"Plot block averaging T={T:.0f} K ...")

    B, s = np.genfromtxt(blok_simulation_file, delimiter=",", unpack=True)

    plt.plot(B, s, color=f"C{i}", label=f"$T = {T:.0f} \\mathrm{{K}}$")
    plt.axhline(s_E, linestyle="--", color=f"C{i}", alpha=0.5)

plt.xscale("log")
plt.yscale("log")
plt.xlabel("$B$ (block size)")
plt.ylabel(r"$s$ (statistical inefficiency)")
plt.legend(loc="upper left")

plt.tight_layout()
plt.savefig(f"plots/H2a_block_avg.pdf")

# %%
```