

# Homework 2:

## Monte Carlo & Ising Model

Jonatan Haraldsson  
[jonhara@chalmers.se](mailto:jonhara@chalmers.se)

Oscar Stommendal  
[oscarsto@chalmers.se](mailto:oscarsto@chalmers.se)

Task N <sup>o</sup>	Points	Avail. points
$\Sigma$		

December 28<sup>th</sup> 2024

Course: *Computational Physics*, 7.5 hp

Course Code: *FKA 122*

*Physics, MSc.*

CHALMERS UNIVERSITY OF TECHNOLOGY



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY

# Introduction

Ferromagnetism, or the fact the some metals give rise to attractive/repelling forces depending on their alignment, has been known to mankind since ancient times. Furthermore, materials with magnetic properties enabled the first compasses, arguably the most important navigation tool to date. In the modern world, materials with magnetic properties have countless applications from power generators and electric motors, to speakers and microphones, to the magnetic strip on credit cards or the Chalmers Kårkort.

To gain a deeper theoretical understanding of ferromagnetic materials, Ernest Ising and Wilhelm Lenz took a mathematical approach and labeled each position in the atomic lattice with a corresponding spin value  $\sigma = \pm 1$  (up or down) [1] [2]. This approach captures the magnetic properties to some extent, as each spin is a contributor to the total magnetic moment. Furthermore, all interactions are considered only by each lattice site's nearest neighbors [3]. Despite its simplicity, the *Ising model* has been used to describe numerous systems beyond ferromagnets. Originally, the model was analytically solved on a 1D lattice, and it was not until the mid 1940s, 20 years after the model was developed, when Norwegian physicist Lars Onsager provided an analytical solution in the 2D case. Since then, an exact analytical solution to a 3D lattice remains a mystery, yet to be found.

In this report, we do unfortunately not produce an exact analytical solution in 3D. Instead, a three-dimensional brass lattice ( $\text{Cu}_{0.5}\text{Zn}_{0.5}$ ) was firstly described analytically with a mean-field approach, and secondly, numerical simulations were done using the *Metropolis algorithm* at varying temperatures. With both methods, the internal energy,  $U$ , heat capacity  $C_V$  and long- and short-range order  $P$  and  $r$  were computed. Lastly, the statistical inefficiency for the simulation was calculated with an autocorrelation function and a block averaging method. For reference, the source code for all tasks is available in Appendix C and D and via [GitHub](#).

## The Binary Alloys CuZn

CuZn alloys, often referred to as *Brass* or *Bronze* for higher Cu content, has been greatly used by throughout human history [4]. Due to its low melting point, brass has a high castability, making brass tools easy to manufacture. Although brass is often used in pipe connectors, the most famous modern example of brass products is *Brass instruments*, i.e. trumpets, saxophones, trombones, etc. In Figure 1, the CuZn phase diagram is shown. The simulations in this report was done on  $\text{Cu}_{0.5}\text{Zn}_{0.5}$ , which for Zn 50at% corresponds to the  $\beta'$  phase for  $T \lesssim 468^\circ\text{C} = 741\text{ K}$  and the  $\beta$  phase for  $T \gtrsim 741\text{ K}$ . Crystallographically, the  $\beta'$  phase is in an ordered BCC lattice, while  $\beta$  phase is in a disordered state [5].

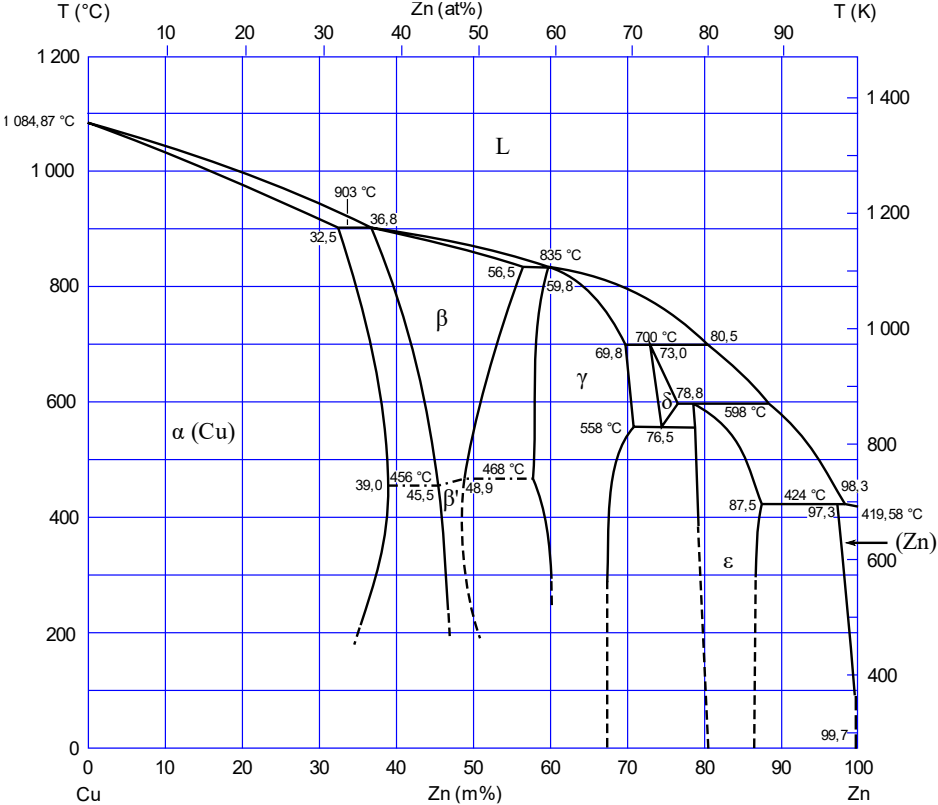


Figure 1: CuZn phase diagram with both Zn in mass % and at%. With the current set-up,  $\text{Cu}_{0.5}\text{Zn}_{0.5}$ , the system is in the  $\beta'$  and  $\beta$  phases depending on temperature. Diagram is taken from [Wikipedia Commons](#).

## Task 1 – Mean-Field Approximation

In the first task, the mean-field approximation for the system's energy  $U$  was used to find the temperature dependence on the long-range order parameter  $P(T)$ . In addition to plotting  $P(T)$ , plots for the energy,  $U(T)$ , and the heat capacity  $C_V(T)$  was also created with the mentioned approximation.

For starters, the long-range order parameter  $P \in [-1, 1]$  is obtained by counting the number of atoms in their corresponding sublattice. If  $|P| = 1$ , the system is in complete order, and if  $P = 0$ , the system is completely mixed. At  $T = 0$  K, the Cu and Zn atoms are aligned perfectly in a BCC lattice, consisting of two simple cubic sublattices ( $a_{\text{sub}}$  and  $b_{\text{sub}}$ ) with a shift  $\vec{\delta} = \frac{a_0}{2}(\hat{x} + \hat{y} + \hat{z})$ , where  $a_0$  is the lattice parameter. In that case,  $P = 1$ , however, as  $T$  increases, thermal energy may cause a decrease in order.

Mathematically, a relation between  $T$  and  $P$  was obtained by minimizing the free energy  $F = U - TS$ , where  $S$  is the entropy. In other words,  $P$  such that  $\frac{\partial F}{\partial P} = 0$  was found. After some algebraic massaging, available in Appendix A, we obtained

$$T = \frac{4P\Delta E}{k_B \log\left(\frac{1+P}{1-P}\right)}, \quad (1)$$

where  $\Delta E = E_{\text{CuCu}} + E_{\text{ZnZn}} - 2E_{\text{CuZn}}$ . In addition, the critical temperature  $T_c$  of the phase transition, was found to be  $T_c = 2\Delta E/k_B = 905 \text{ K} = 632^\circ\text{C}$ , slightly higher than suggested by the phase diagram in Figure 1.

Although Equation 1 gives  $T(P)$ , a plot for  $P(T)$  was created in Figure 2 by letting  $|P| \in [0,1]$  and solving for  $T$ . The shape of  $P(T)$  in Figure 2 suggest that the long-range order decreases steadily from complete order at  $T \approx 200$  K to complete disorder at  $T = T_c$ . This is an expected behavior, since the  $\beta'$  phase, at  $T < T_c$ , is characterized by complete order, while the  $\beta$  phase, at  $T > T_c$ , instead is completely disordered.

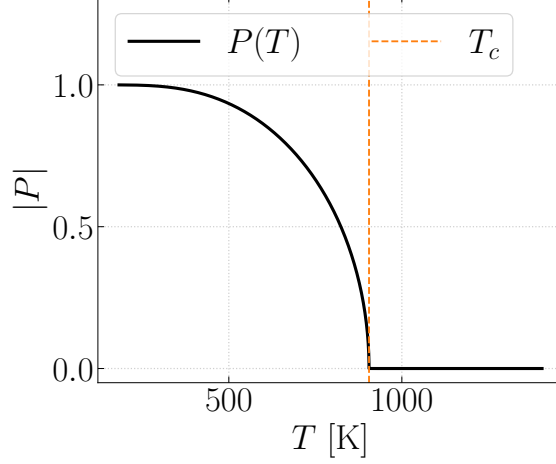


Figure 2: Mean-field solution for  $P(T)$ , using the relation in Equation 1. In addition, the orange line denotes  $T_c$ .

From Equation 1, a relation for  $U(T)$  of the  $N_{atoms} = 2000$  atoms was found to be

$$U(T) = N_{atoms}(E_0 - \Delta E P(T)^2), \quad (2)$$

where  $E_0 = E_{CuCu} + E_{ZnZn} + 2E_{CuZn} = 1.137$  keV. Lastly, the heat capacity was given by

$$C_V(T) = \left( \frac{\partial U(T)}{\partial T} \right)_V. \quad (3)$$

The corresponding plots for  $U(T)$  and  $C(T)$  are given in Figure 3. The mean-field solution for  $U(T)$  suggests, somewhat non-physically, that the internal energy of the system stays constant for  $T > T_c$ . However, considering that  $P(T) \equiv 0$  for  $T > T_c$ , it follows directly from Equation 2 that  $U(T) \equiv 2E_0 = 2.274$  keV  $\forall T > T_c$ . Due to this, its derivative,  $C_V(T)$ , too has a somewhat non-physical shape with a sharp peak at  $T_c$  followed by zeros.

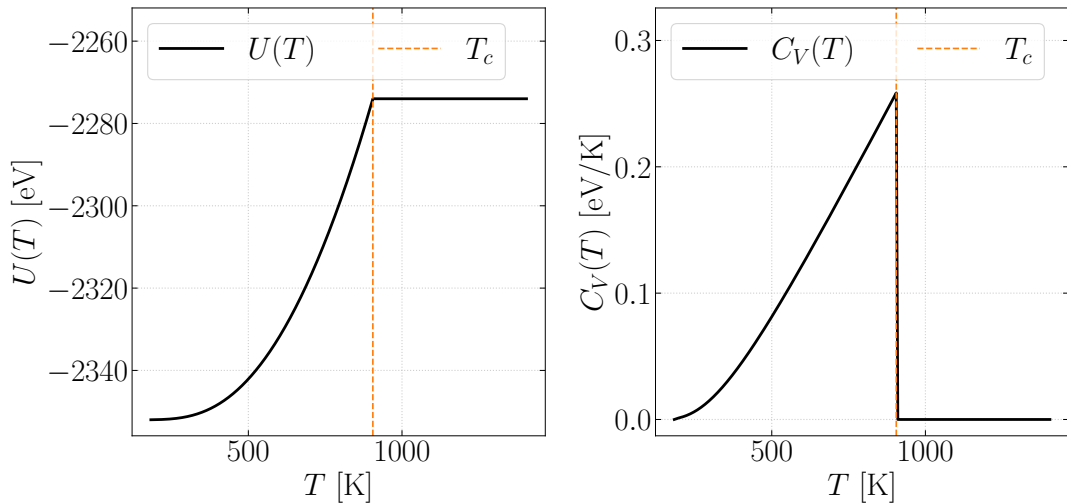


Figure 3: Mean-field solution for  $U(T)$ , using the relation in Equation 2 and heat capacity  $C_V(T)$ , retrieved using Equation 3. The orange line denotes the critical temperature  $T_c$ .

## Task 2 – Numerical Set-up and MC Simulations

In this task, the first step was to initialize a BCC lattice as a  $10 \times 10 \times 10$  supercell. This was done in the beginning of each MC simulation by letting all Cu atoms occupy the  $a_{\text{sub}}$  and all Zn atoms occupy  $b_{\text{sub}}$ , where each sublattice has 1 000 atoms ( $N_{\text{atoms}} = 2\,000$ ). This set-up ensures complete order, i.e.  $|P| = 1$ . Thus, the initial energy was set to

$$E_{\text{initial}} = \langle n \rangle \cdot \frac{N_{\text{atoms}}}{2} \cdot E_{\text{CuZn}} = 8 \cdot \frac{(N_a + N_b)}{2} \cdot (-0.294 \text{ eV}) = -2352 \text{ eV}, \quad (4)$$

where  $N_a$  and  $N_b$  are the total number of atoms in each sublattice,  $\langle n \rangle$  the number of nearest neighbors for one atom and  $E_{\text{CuZn}}$  the binding energy between a Cu and Zn atom. The factor of  $1/2$  ensures that each bond is counted only once, as every atom bonds with all of each nearest neighbors and vice versa. Not adjusting this would double-count each of these bonds. Subsequently, to verify the initialization, the lattice was plotted in 3D with `Python`, see Figure 4. The lattice parameter was set to  $a_0 = 2.969 \text{ \AA}$  [6].

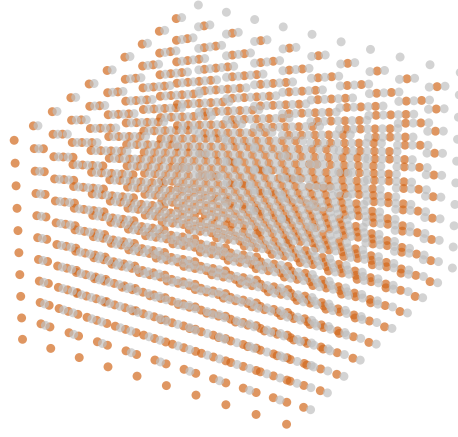


Figure 4: Three-dimensional plot of all 1 000 Cu (brown) and 1 000 Zn (silver) atoms in the initialized lattice.

To monitor the different binding energies between Cu–Cu, Zn–Zn or Cu–Zn as the atoms are interchanged, a matrix ( $\mathbf{M}_N$ ) was created. Each row  $i$  in  $\mathbf{M}$  corresponds to an atom position ( $i \leq 1000$  for  $a_{\text{sub}}$  and  $i > 1000$  for  $b_{\text{sub}}$ ) and each column value (in total 8, since we have 8 nearest neighbors) describe the indices of the nearest neighbors' positions. Note that in a cold start, i.e.  $P = 1$ , the eight nearest neighbors to a Zn atom in  $b_{\text{sub}}$  are all Cu in  $a_{\text{sub}}$  and vice versa, as illustrated in Figure 5. Furthermore, an additional array,  $\mathbf{A}$ , to keep track of which atom type occupied each position in the lattice was constructed accordingly

$$A_i = \begin{cases} 1, & \text{if position } i \text{ is occupied by Cu } \forall i \in [1, N_{\text{atoms}}] \\ 0, & \text{otherwise (if position } i \text{ is occupied by Zn),} \end{cases}$$

making  $\mathbf{A}$  a  $(2000 \times 1)$  array. Note that index  $i$  here directly corresponds to the row index of the  $\mathbf{M}_N$  matrix. For the atom at site  $x$ , this enabled us to efficiently

- a) check which atom type occupied a certain lattice position, by calling  $\mathbf{A}[x]$ , and
- b) retrieve the nearest neighbors and their atom type through applying the indices provided by  $\mathbf{M}_N[x]$  to  $\mathbf{A}$ .

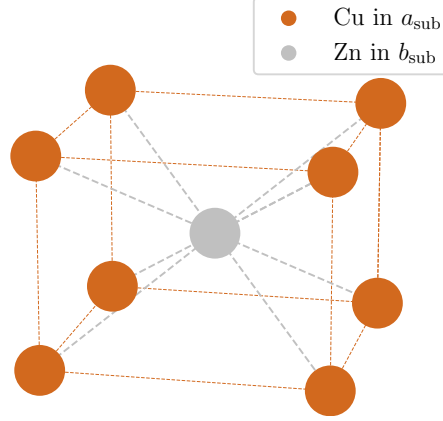


Figure 5: In a cold, perfectly ordered starting state, the nearest neighbors to the Zn atom in the center ( $b_{\text{sub}}$ ) are all Cu atoms ( $a_{\text{sub}}$ ).

Next, the *Metropolis algorithm* was implemented by first generating two unique indices  $i \in [1, N_{\text{atoms}}]$  randomly using **swappy** provided in the code snippet below. Subsequently, the two atoms corresponding to the generated indices through **A**, are swapped if they the change's energy difference ( $\Delta E$ ) satisfied

$$\alpha = \frac{e^{-E'/k_B T}}{e^{-E/k_B T}} = e^{-\Delta E/k_B T} \geq 1. \quad (5)$$

#### Code Snippet Task 2, swappy

```

1  idx swappy(int *N, gsl_rng *r)
2  {
3      idx index;
4      int idx_1 = (int)(1999 * gsl_rng_uniform(r)); int A = N[idx_1];
5      int idx_2 = (int)(1999 * gsl_rng_uniform(r)); int B = N[idx_2];
6      while (A == B)
7      {
8          idx_2 = (int)(1999 * gsl_rng_uniform(r));
9          B = N[idx_2];
10     }
11     index.alpha = idx_1; index.beta = idx_2;
12     index.valueA = A; index.valueB = B;
13     return index;
14 }
```

In Equation 5,  $E$  and  $E'$  are the potential energies within the lattice before and after the swap.  $\Delta E = E' - E$  is thus the difference in energy caused by the swap. Note that our algorithm only chose to propose swaps that actually resulted in an energy difference, i.e. proposals involving two atoms of the same type was not considered at all. The energies were obtained by summing the bond energies  $E_i$  to the nearest neighbors at each lattice position  $i$  using the  $\mathbf{M}_N$  matrix. In other words,  $E = \sum E_i$ . As an example,  $E_i$  for a Cu at atom site  $i$  is calculated accordingly

$$E_i = N_{\text{Cu}}^{(i)} E_{\text{CuCu}} + N_{\text{Zn}}^{(i)} E_{\text{CuZn}},$$

where  $N_{\text{Cu}}^{(i)}$  and  $N_{\text{Zn}}^{(i)}$  are the number of Cu and Zn atoms amongst the nearest neighbors provided by row  $i$  in  $\mathbf{M}_N$ . Lastly, if  $\alpha < 1$  in Equation 5, the proposed swap was accepted when  $\alpha > u$ , where  $u \in [0, 1]$  is a randomly generated number. Otherwise, the swap proposal was rejected, leaving all atoms un-swapped.

The aforementioned procedure of swapping atoms was repeated  $N = 1 \times 10^6$  times at temperatures  $T = 400, 600$  and  $1000$  K. However, to ensure the system had reached an equilibrium prior to the  $N$  iterations,  $N_{eq}$  *burn-in* iterations were run. The values for  $N_{eq}$  was obtained by examining the energy in each iteration of the *Metropolis algorithm*, as illustrated in Figure 6. For  $T = 600$  and  $1000$  K, the system was considered equilibrated after  $\sim 100\,000$  iterations. In the case of  $T = 400$  K, however, the system tended to need an extra  $150\,000$  burn-in steps, giving in total  $250\,000$  in this case. This is arguably a direct consequence of the acceptance  $\alpha$ , as it is scaled with temperature according to Equation 5. Put differently, an increased temperature promotes more atom swaps, and thus faster equilibration times.

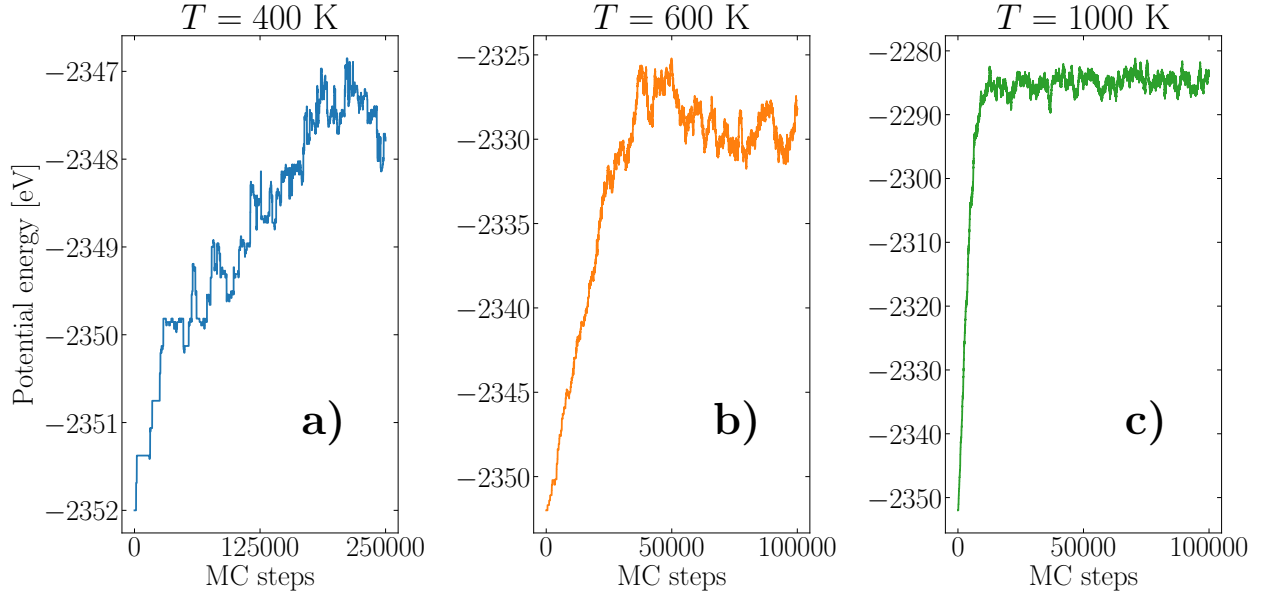


Figure 6: The potential energy at each iteration during the MC simulation for  $T =$  a) 300, b) 600 and c) 1000 K.

Additionally, Table 1 shows the number of *burn-in* iterations along with the resulting energy and acceptance ratio  $\Gamma$  for the three cases  $T = 400, 600$  and  $1000$  K. As expected, a temperature closer to 0 has a lower acceptance ratio and hence, a lower energy since we initialize the lattice at 0 K (perfectly ordered,  $P = 1$ ). Analogously, a higher temperature yields a higher acceptance ratio and energy, since the system moves towards a more chaotic form (randomly placed Cu and Zn atoms), causing more accepted steps in the *Metropolis algorithm*.

Table 1: Results from the MC simulations in task 2.

Temperature $T$ [K]	Burn-in $N_{eq}$	Energy $E_{pot}$ [eV]	Acceptance $\Gamma$
400	$2.5 \times 10^5$	- 2348	0.020
600	$1 \times 10^5$	- 2327	0.130
1 000	$1 \times 10^5$	- 2285	0.622

### Task 3 – Full Sampling of $U$ , $C_V$ , $P$ and $r$ using MC

In the third task, statistical averages of  $U(T)$ ,  $C_V(T)$ ,  $P(T)$  and in addition, the short-range order parameter  $r(T)$  was obtained by running the *Metropolis algorithm* from Task 2. Similar to  $P$ , the short-range order parameter  $r$  reflects the local order in the lattice, as it depends on the number of nearest neighbors being different atom types (see Equation 8 below). In addition, the statistical inefficiency,  $s$ , of the Metropolis algorithm was determined using both autocorrelation and block averaging.

#### Statistical Averages and Temperature Dependence

To find the temperature dependence for  $P$ ,  $U$ ,  $C$  and  $r$ , the average of  $N$  iterations was calculated for  $T \in [300 \text{ K}, 1000 \text{ K}]$ , with step-size  $\Delta T = 25 \text{ K}$ . The number of iterations  $N$  was chosen such that the number of accepted steps (i.e. the samples from which the average quantities were calculated) was  $N_{acc} = 100\,000$ , see Table 2 in Appendix B. This yielded different number of iterations  $N$  for every temperature due to the acceptance ratio being different at each  $T$  (which is reasonable considering the physics of the system, see the reasoning in Task 2) and was done in order to retrieve a better comparison of the statistical inefficiency. Again, a number of *burn-in* iterations were run for every  $T$ , where we used  $N_{eq} = 250\,000$  for  $T < 600 \text{ K}$  and  $N_{eq} = 100\,000$  otherwise.

Subsequently, plots for  $U(T)$ ,  $C(T)$ ,  $P(T)$  and  $r(T)$  were created. The internal energy  $U$  was calculated as in Task 2 by summing the bond energies to the nearest neighbors at each lattice position. The heat capacity  $C_V(T)$  was calculated from  $U(T)$ , this time using energy fluctuations (which is more handy when calculating  $U$  through sampling at constant  $T$ ),

$$C_V(T) = \frac{\langle U^2 \rangle - \langle U \rangle^2}{k_B T^2}. \quad (6)$$

The long- and short-range order parameters  $P(T)$  and  $r(T)$  were calculated using

$$P(T) = 4 \frac{N_{a,\text{Cu}}}{N_{atoms}} - 1 \quad \text{and} \quad (7)$$

$$r(T) = \frac{2}{4N_{atoms}}(q - 2N_{atoms}), \quad (8)$$

where  $N_{a,\text{Cu}}$  is the number of Cu atoms in  $a_{\text{sub}}$  and  $q$  is the number of nearest neighbors that have Cu-Zn bonds.

#### Statistical Inefficiency

The statistical inefficiency was found, firstly using the autocorrelation function  $\Phi_k$ ,

$$\Phi_k = \frac{\langle X_{i+k} X_i \rangle - \langle X_i \rangle^2}{\langle X_i^2 \rangle - \langle X_i \rangle^2}, \quad (9)$$

where  $X_i$  resembles a sampled data point for quantity  $X$ . From Equation 9, the statistical inefficiency,  $s$ , for  $P$ ,  $U$ ,  $C$  and  $r$  was obtained by substituting  $X$  and then finding  $k = s$  such that  $\Phi_s = e^{-2}$ . Secondly,  $s$  was also estimated by constructing data blocks according to

$$F_j = \frac{1}{B} \sum_{i=1}^B X_{i+B(j-1)}.$$



From this,  $s$  was found by  $s = B\text{Var}[F]/\text{Var}[X]$  for sufficiently large  $B$ . This was done for different block sizes up to  $\sim 5\%$  of the total sample size. Afterwards, this was plotted and an average value of  $s$  was calculated when this had properly converged. Finally, an error for  $X$  was estimated for both methods according to

$$\epsilon_X = \sqrt{\frac{s \cdot \sigma_X^2}{N_X}}, \quad (10)$$

where  $\sigma_X$  is the standard deviation of  $X$  and  $N_X$  the total amount of  $X$  samples.

During the MC simulations,  $s$  was estimated using both autocorrelation and block average for every sampled  $T$  using the methodology described above.

## Results and Discussions

Firstly, the internal energy,  $U(T)$  and the heat capacity  $C_V(T)$  are given in Figure 7 along with a filtered curve and estimated errors corresponding to the mean calculated error using both autocorrelation and block averaging. The filtering was done with a *Savitzky-Golay filter* implemented according to `SciPy Cookbook` [7]. In both cases, the curve shapes are somewhat similar to the mean-field curves in Figure 3, however, with a slightly shifted phase transition temperature near  $T = 750$  K; closer to the phase transition in Figure 1. In contrast to the mean-field solution, the MC sampled  $U(T)$  still increases with temperature for  $T$  above the phase transition, which is a more physical behavior. With this approach, calculating  $U(T)$  by summing all bonding energies,  $U(T)$  should, however, reach a plateau as the system transforms into the disordered state. In other words, swapping atoms in an already completely disordered system cannot cause an increased disorder, and thus no increase in  $U(T)$ . Lastly, considering that  $C_V = \frac{\partial U}{\partial T}$ , Figure 7 resembles the derivative to  $U(T)$  quite well; especially the peak at  $T = 750$  K in  $C_V$  aligns with the inflection point in  $U(T)$ .

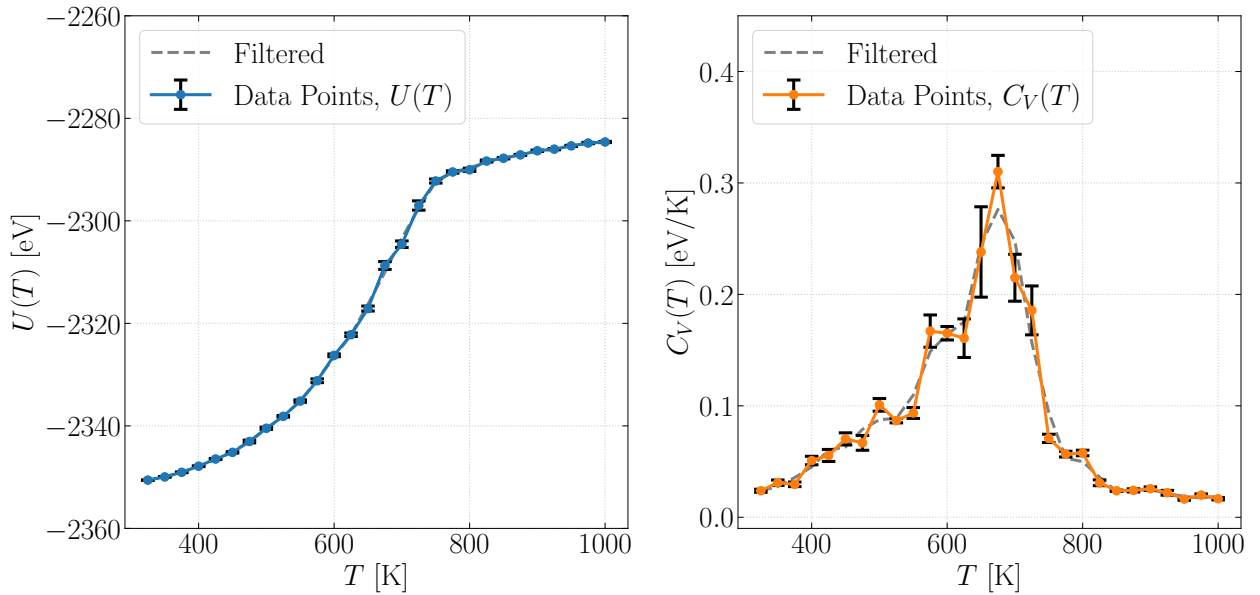


Figure 7: Sampled data points for  $U(T)$  (left) and  $C_V(T)$  (right) along with a filtered curve. The error bars denote the mean error using both autocorrelation and block averaging for every sampled  $T$ .

Secondly, sampled data points of  $P(T)$  and  $r(T)$  are given in Figure 8. When comparing  $P(T)$  in Figure 8 with the mean-field solution in Figure 1, the MC sampled  $P(T)$  reaches close to zero (i.e. the disordered  $\beta$  phase) at  $\sim 750$  K, which is more in-line with the phase diagram in Figure 1. Similar to  $P(T)$ ,  $r(T)$  also reaches its minimum close to  $\sim 750$  K, close to the phase transition temperature suggested in Figure 1. Although this,  $r(T)$  reaches a minimum of slightly less than 0.2, which still indicates that there is some local order in the system.

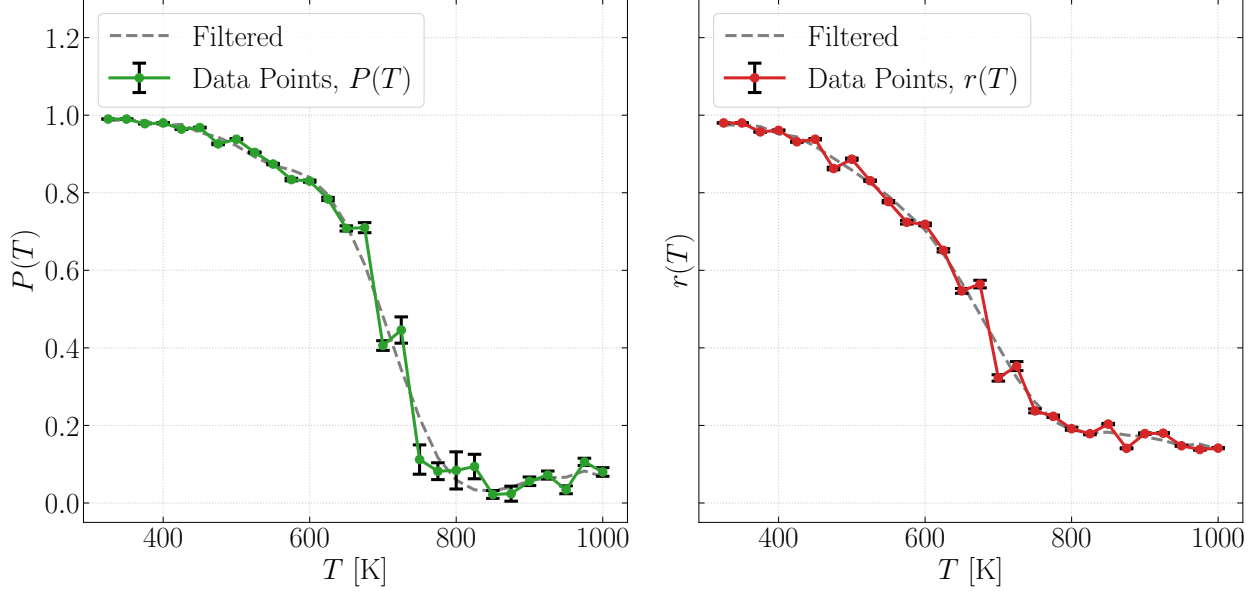


Figure 8: Sampled data points for  $P(T)$  (left) and  $r(T)$  (right) along with a filtered curve. The error bars denote the mean error using both autocorrelation and block averaging for every sampled  $T$ .

Regarding the error estimation, Figure 9 below shows the calculated errors  $\epsilon$  for all quantities as a function of temperature. Each  $\epsilon$  is a mean value of the errors calculated using autocorrelation and blocking averaging at every sampled  $T$ . We especially note that the errors increase at and around the critical temperature ( $T \approx 741$  K) for all quantities. This is expected and reasonable since fluctuations increase near the critical point, leading to larger uncertainties in the sampled quantities. Moreover, we also note that the errors for  $U$  and  $r$  more or less completely resemble each other. This could be explained by the fact that the *Metropolis algorithm*, and hence the determination/update of  $U$ , depends on the short-range order since we use the nearest neighbor approximation when computing  $U$ . As a final note,  $U$  and  $r$  have the smallest errors relative to the quantity-size, followed by  $P$  and last  $C_V$ . This is also expected since  $U$  and  $r$  are calculated “directly” as part of each MC step, while  $P$  and  $C_V$  needs further calculations (the number of Cu atoms and the variance of  $U$ , respectively), introducing more statistical uncertainty.

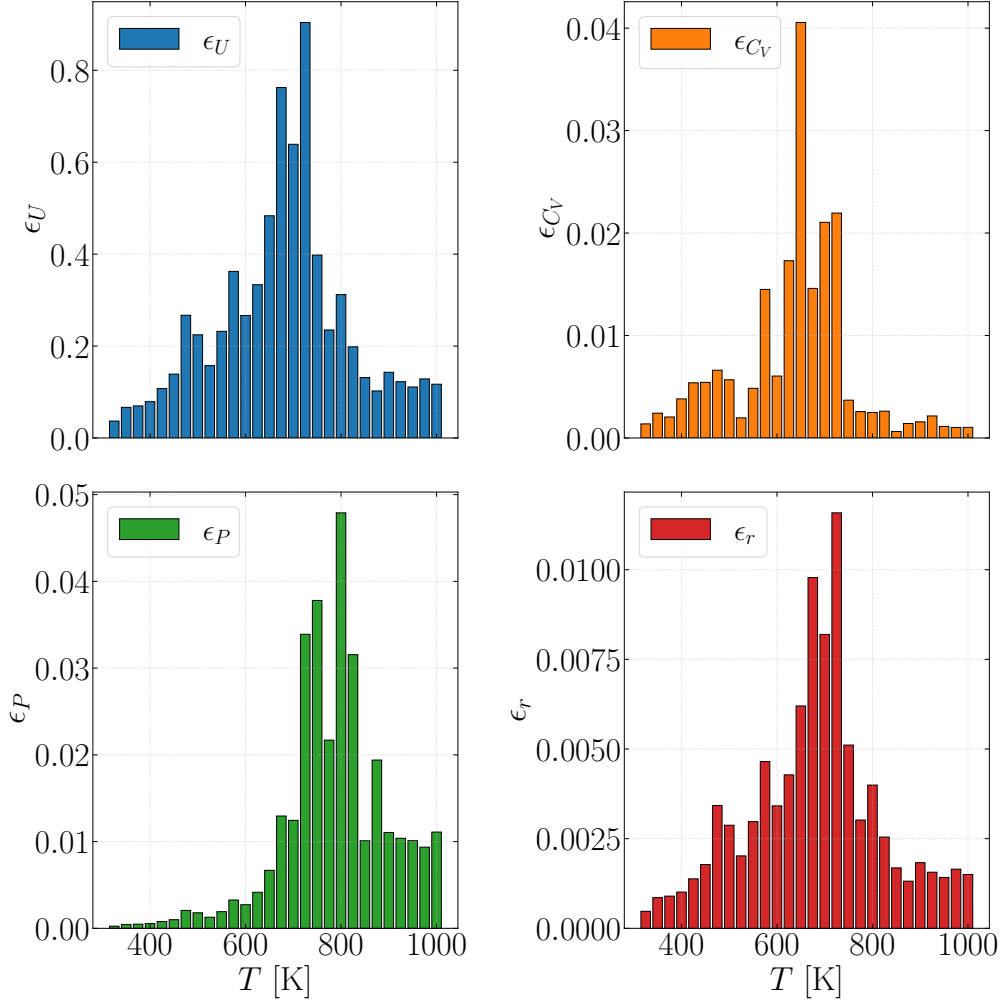


Figure 9: The error  $\epsilon$  for the internal energy  $U$ , heat capacity  $C_V$  and long- and short-range order parameter  $P$  and  $r$  as a function of temperature.

Moreover, Figures 10 and 11 shows the autocorrelation function and  $s$  using block average for the internal energy  $U$ , heat capacity  $C_V$  along with long- and short-range order parameters  $P$  and  $r$  at  $T = 450$  K. From the autocorrelation function, the values of  $s$  for the different quantities have been displayed as the intersection between the respective  $\Phi_k$  and the function  $y = e^{-2}$ . The estimated value of  $s$  using the block average is the average of this when it has converged. In all simulations, as mentioned earlier, a total of 100 000 samples were used, and block sizes up to 5% of the sample size were considered.

In Figure 10, the block averages obviously converge, however, for the heat capacity in Figure 11, we failed on receiving the converged behavior ( $s$  was then estimated as the mean for all block averages). This may be due to several reasons, the biggest and most likely being the need for more samples. Unfortunately, this would probably be a bit hard on our precious laptops, but it can definitely be something to investigate further. Another obvious reason can be that our explored block sizes simply is not enough for convergence, i.e. going to a higher block size might show more convergence tendencies. We also notice that we lack an exponential behavior of  $\Phi_k$  in this case. This suggests that the samples are highly correlated (resulting in the high estimation of  $s$ ), which also can be seen as the errors for  $C_V$  are relatively large compared to the other quantities.

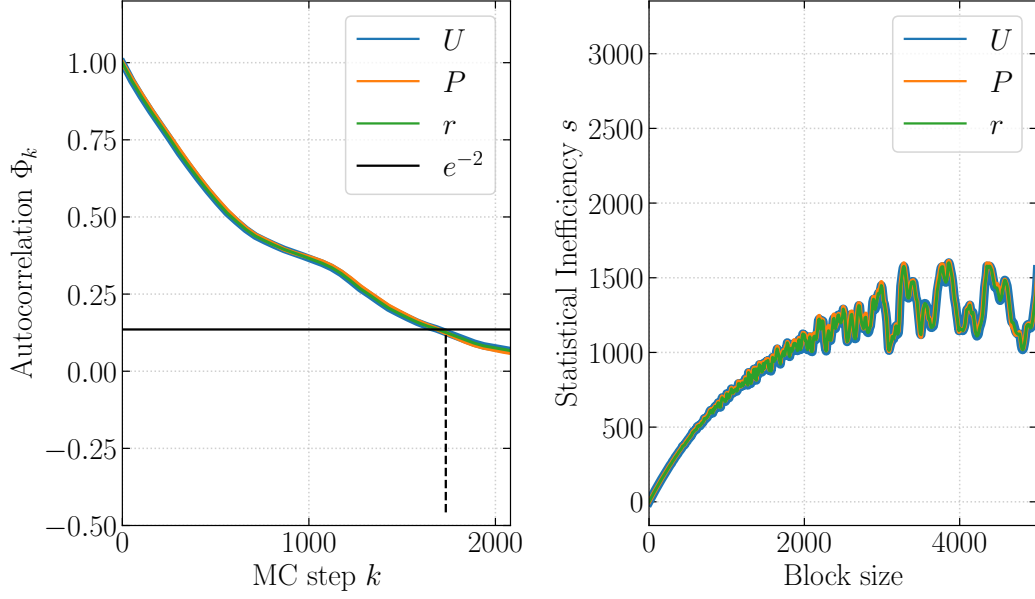


Figure 10: The autocorrelation function together with the statistical inefficiency from the block average method for the internal energy  $U$  and long- and short-range order parameters at  $T = 450$  K. The dashed line in the left plot indicate the index of  $\Phi_k$  where this is closest to the value of  $e^{-2}$ , i.e. the estimated value of  $s$  using this method.

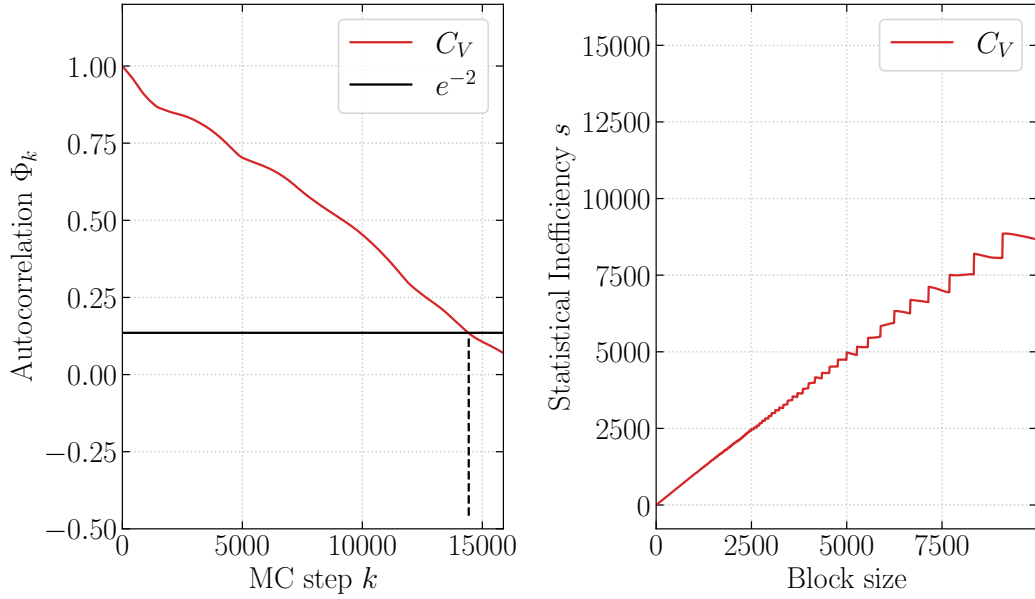


Figure 11: The autocorrelation function together with the statistical inefficiency from the block average method for the heat capacity  $C_V$  at  $T = 450$  K. The dashed line in the left plot indicate the index of  $\Phi_k$  where this is closest to the value of  $e^{-2}$ , i.e. the estimated value of  $s$  using this method.

Another interesting result found for  $T > T_c$ , was that  $\Phi_k$  and the block averages for  $P$  tended to diverge from the corresponding values for  $U$  and  $r$ . To clarify, compare  $\Phi_k$  and the block averaging for three quantities at  $T = 450$  K in Figure 10 to the same curves at  $T = 1000$  K in Figures 12 and 13. In the two latter Figures, the values for  $P$  has deviated more from those for  $U$  and  $r$ . This, again, reflects the results in Figure 9, where errors for  $U$  and  $r$  are more or less the same at every  $T$  while the errors for  $P$  are increasing for larger  $T$ . As discussed earlier,

it is somewhat expected that errors increase for larger  $T$ , since the fluctuations at this point become more pronounced. The deviation for  $P$  could arise from the reduced long-range order at high temperatures, leading to an increased sensitivity to statistical noise and fluctuations. Consequently, the autocorrelation and block averages for  $P$  should reflect these pronounced fluctuations, while  $U$  and  $r$ , being more directly linked to the short-range interactions and lattice properties, remain relatively stable and similar to each other.

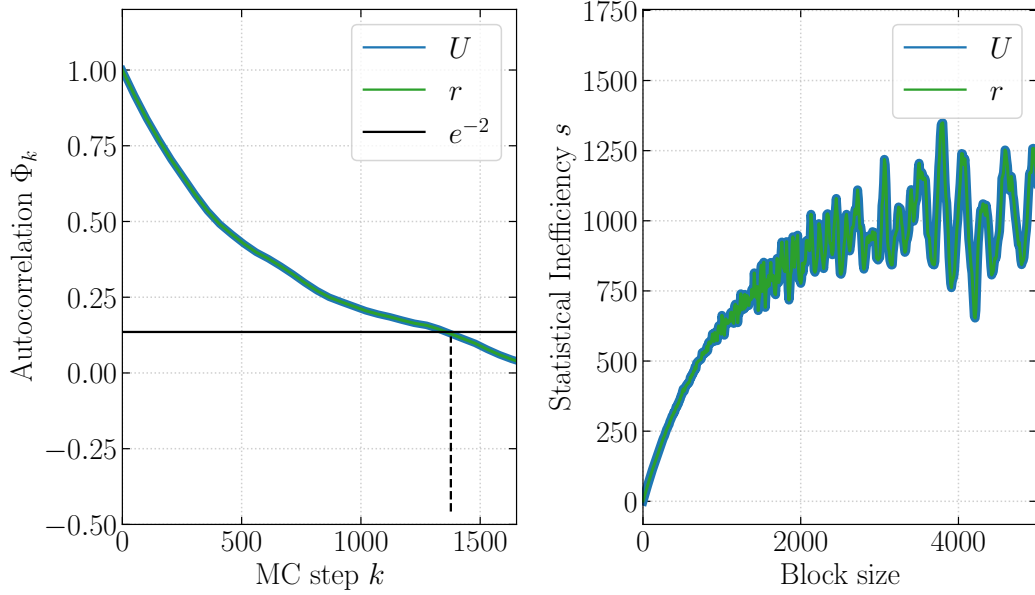


Figure 12: The autocorrelation function together with the statistical inefficiency from the block average method for the internal energy  $U$  and short-range order parameter at  $T = 1000$  K. The dashed line in the left plot indicate the index of  $\Phi_k$  where this is closest to the value of  $e^{-2}$ , i.e. the estimated value of  $s$  using this method.

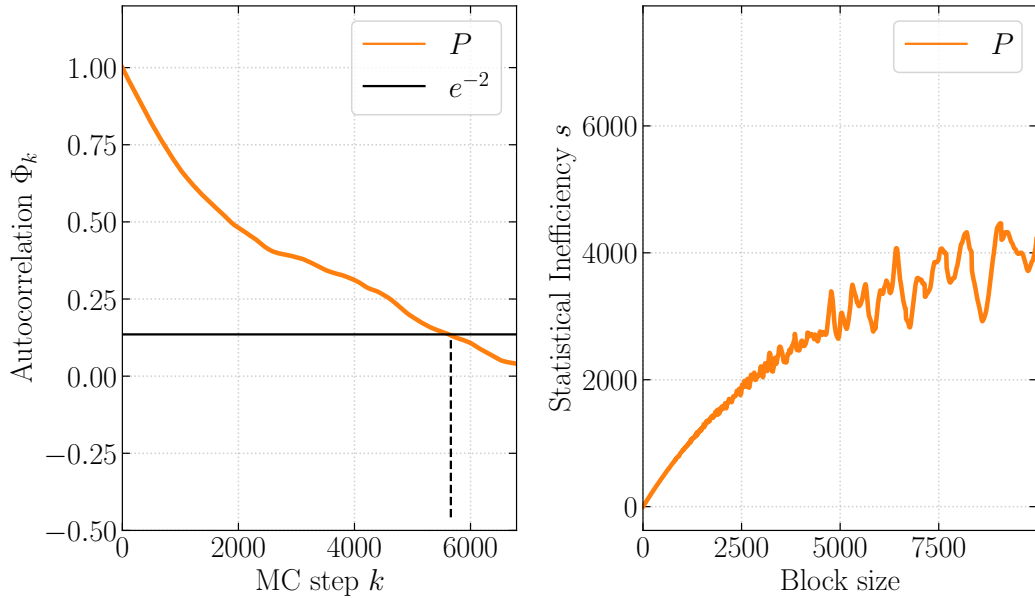


Figure 13: The autocorrelation function together with the statistical inefficiency from the block average method for the long-range order parameter  $P$  at  $T = 1000$  K. The dashed line in the left plot indicate the index of  $\Phi_k$  where this is closest to the value of  $e^{-2}$ , i.e. the estimated value of  $s$  using this method.

## Conclusion

In summary, we have simulated a brass lattice consisting of 50% Cu and Zn, initialized in the  $\beta$ , BCC, phase. From the simulations, using both a mean-field approximation and later the *Metropolis algorithm*, properties such as long- and short-range order  $P(T)$  and  $r(T)$  along with internal energy  $U(T)$  and heat capacity  $C_V(T)$  was calculated. Although the methods give somewhat similar temperature dependence, the *Metropolis algorithm*, provides more physical results with  $T_c$  closer to what is suggested by the phase diagram in Figure 1, which makes the *Metropolis algorithm* a superior method. From the error estimation, the approach used seemed to give largest errors for  $C_V$ , which is most likely due to the fact that  $C_V$  is not calculated directly within the MC sampling. Additionally, for larger  $T$ , the autocorrelation and block averaging for long-range order  $P$  deviates from the same curves for  $U$  and  $r$ . This is, however, somewhat expected, since  $U$  and  $r$  are related to short-range interactions (i.e. the nearest neighbors).

Finally, with these simulations we did not gain any further insights into solving the *Ising model* analytically in 3D, but we still believe Ising, Lenz, and Onsager would consider it impressive that a three-dimensional lattice could be described with such great accuracy using simple laptops and a bit of code.

# References

- [1] Wikipedia. “Ising model”. Last edited: 01-12-2024. (), [Online]. Available: [https://en.wikipedia.org/wiki/Ising\\_model](https://en.wikipedia.org/wiki/Ising_model). accessed: 2024-12-12.
- [2] Stanford.edu. (), [Online]. Available: <https://stanford.edu/~jeffjar/statmech/intro4.html>. accessed: 2024-12-12.
- [3] J. Obermeyer. “The ising model in one and two dimensions”. (), [Online]. Available: [https://www.thphys.uni-heidelberg.de/~wolschin/statsem20\\_3s.pdf](https://www.thphys.uni-heidelberg.de/~wolschin/statsem20_3s.pdf).
- [4] B. Wikipedia. Last edited: 26-10-2024. (), [Online]. Available: <https://en.wikipedia.org/wiki/Brass#History>. accessed: 2024-12-12.
- [5] R. Ye. “What are the different types of brass: Specifications & properties”. (2023), [Online]. Available: [https://www.3erp.com/blog/brashttps://materials.springer.com/isp/crystallographic/docs/sd\\_0314456s-types/](https://www.3erp.com/blog/brashttps://materials.springer.com/isp/crystallographic/docs/sd_0314456s-types/). accessed: 2024-12-12.
- [6] Springer Materials. “CuZn Crystal Structure”. P. Villars and K. Cenzual, Eds. (2011), [Online]. Available: [https://materials.springer.com/isp/crystallographic/docs/sd\\_0314456](https://materials.springer.com/isp/crystallographic/docs/sd_0314456). accessed: 2024-12-10.
- [7] SciPy. “SciPy Cookbook / Savitzky Golay Filtering”. (), [Online]. Available: <https://scipy.github.io/old-wiki/pages/Cookbook/SavitzkyGolay>. accessed: 2024-12-11.

# Appendix

## A Mean-Field Equations

To begin, the free energy  $F = U - TS$ , where  $U$  is the internal energy,  $S$  the entropy and  $T$  temperature. From page 21 in H2a\_auxiliary.pdf,  $S$  is after some massage given by

$$S = -2Nk_B \log 2 + Nk_B [(1 + P) \log(1 + P) + (1 - P) \log(1 - P)],$$

and since  $U = E_0 - 2NP^2\Delta E$ ,  $F$  is given by

$$E_0 - 2NP^2\Delta E - 2Nk_B \log 2 + Nk_B [(1 + P) \log(1 + P) + (1 - P) \log(1 - P)].$$

Here,  $E_0 = N(E_{\text{CuCu}} + E_{\text{ZnZn}} + 2E_{\text{CuZn}})$  and  $\Delta E = E_{\text{CuCu}} + E_{\text{ZnZn}} - 2E_{\text{CuZn}}$  Here,  $f \equiv F/N$  or free energy per atom is considered. To find the  $P$  that minimizes  $f$ ,

$$\frac{\partial f}{\partial P} = -4P\Delta E + k_B T \log \left( \frac{1 + P}{1 - P} \right) = 0$$

is solved for  $P$ . Finally,

$$T = \frac{4P\Delta E}{k_B \log \left( \frac{1+P}{1-P} \right)}.$$



## B Additional Results

Table 2: The number of accepted steps (samples of  $U$ )  $N_{samples}$ , total number of iterations  $N$ , equilibration iterations  $N_{eq}$  and acceptance ratio for every sampled  $T$ .

$T$ [K]	$N_{samples}$	$N$	$N_{eq}$	Acceptance $N_{samples}/N$ (%)
300	100 000	23 409 400	250 000	0.43
325	100 000	13 508 300	250 000	0.74
350	100 000	9 142 670	250 000	1.09
375	100 000	6 235 580	250 000	1.60
400	100 000	4 577 910	250 000	2.18
425	100 000	3 442 300	250 000	2.91
450	100 000	2 845 360	250 000	3.51
475	100 000	2 184 610	250 000	4.58
500	100 000	1 731 530	250 000	5.78
525	100 000	1 437 240	250 000	6.96
550	100 000	1 180 420	250 000	8.47
575	100 000	956 838	250 000	10.45
600	100 000	761 226	100 000	13.14
625	100 000	639 335	100 000	15.64
650	100 000	525 613	100 000	19.03
675	100 000	396 587	100 000	25.22
700	100 000	338 243	100 000	29.56
725	100 000	263 062	100 000	38.01
750	100 000	222 414	100 000	44.96
775	100 000	207 469	100 000	48.20
800	100 000	201 814	100 000	49.55
825	100 000	190 183	100 000	52.58
850	100 000	184 851	100 000	54.10
875	100 000	180 285	100 000	55.47
900	100 000	174 321	100 000	57.37
925	100 000	170 807	100 000	58.55
950	100 000	166 573	100 000	60.03
975	100 000	162 960	100 000	61.36
1000	100 000	160 843	100 000	62.17

## C Source code in C

### C.1 run.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #include "tools.h"
5  #include "lattice.h"
6  #include "run.h"
7
8  int
9  run(
10     int argc,
11     char *argv[]
12 )
13 {
14     // ----- Constants ----- //
15
16     double E_cucu = -0.436; // Bonding energy Cu-Cu [eV]
17     double E_znzn = -0.113; // Bonding energy Zn-Zn [eV]
18     double E_cuzn = -0.294; // Bonding energy Cu-Zn [eV]
19     double k_B = 8.617333262145e-5; // Boltzmann constant [eV/K]
20     double a = 2.949; // Lattice parameter [Å]
21     double closest_distance_bcc = sqrt(3) * a / 2; // Closest distance between atoms in
22     ↪ BCC [Å]
23     int L = 10; // Number of unit cells in each direction
24     int N_atoms = 2 * L * L * L; // Number of atoms
25     double E_initial = N_atoms / 2 * E_cuzn * 8; // Initial energy
26     char filename[100];
27     gsl_rng *r = init_gsl_rng(19);
28
29     // ----- Task 1 ----- //
30
31     double step_size = 0.0001; // Step size for P
32     int N = (int)(1 / step_size) + 100; // Number of points (add 100 for the end, where P
33     ↪ = 0)
34     FILE *file = fopen("data/task_1/data.csv", "w");
35     double delta_E = E_cucu + E_znzn - 2 * E_cuzn;
36     double *Us = (double *)malloc(N * sizeof(double)); // Energy vector
37     double *Ts = (double *)malloc(N * sizeof(double)); // Temperature vector
38     double *Ps = (double *)malloc((N + 1) * sizeof(double)); // Long-range order vector
39     double C_V;
40     Ps[0] = 1.; // Start at P = 1
41     // Loop over P and calculate T, U and C_V
42     for (int i = 1; i < N; i++)
43     {
44         if (i < (int)(1 / step_size))
45         {
46             Ps[i] = Ps[i-1] - step_size; // Update P
47             Ts[i-1] = 4 * Ps[i] * delta_E / k_B / log((1 + Ps[i]) / (1 - Ps[i])); //
48             ↪ Calculate T
49             Us[i-1] = N_atoms * (E_cucu + E_znzn + 2 * E_cuzn) - N_atoms * Ps[i] * Ps[i]
50             ↪ * delta_E; // Calculate U
51             if (i >= 2)
52             {
53                 double dU = Us[i-1] - Us[i-2]; // dU
```

```

50         double dT = Ts[i-1] - Ts[i-2]; // dT
51         C_V = dU / dT; // Calculate C_V
52     }
53     else
54     {
55         C_V = 0.; // Set C_V to 0 for the first iteration
56     }
57     fprintf(file, "%f, %f, %f, %f\n", Ps[i], Ts[i-1], Us[i-1], C_V);
58 }
59 // When P = 0, set P = 0 and continue to calculate T, U and C_V
60 else if (i >= (int)(1 / step_size))
61 {
62     Ps[i] = 0.;
63     Ts[i-1] = Ts[i-2] + 5; // Use a step size of 5 K
64     Us[i-1] = N_atoms * (E_cucu + E_znzn + 2 * E_cuzn) - N_atoms * Ps[i] * Ps[i]
        ↪ * delta_E;
65     double C_V = (Us[i-1] - Us[i-2]) / (Ts[i-1] - Ts[i-2]);
66     fprintf(file, "%f, %f, %f, %f\n", Ps[i], Ts[i-1], Us[i-1], C_V);
67 }
68 }
69
70 // Free memory and close files
71 fclose(file);
72 free(Us);
73 free(Ts);
74 free(Ps);
75
76 // ----- Task 2 ----- //
77
78 int its_eq = 250000; // Number of iterations for equilibrium
79 double T = 400; // Temperature [K]
80 metro metro_result;
81
82 double **sub_A = create_2D_array(N_atoms / 2, 3); // Sublattice A
83 double **sub_B = create_2D_array(N_atoms / 2, 3); // Sublattice B
84 int **neighbors = (int **)create_2D_array(N_atoms, 8); // Neighbors matrix
85 int *atoms = (int *)calloc(N_atoms, sizeof(int)); // Atoms vector
86 // Initialize all atoms in sublattice A to be Cu (1) (= Cold start)
87 for (int i = 0; i < N_atoms / 2; i++)
88 {
89     atoms[i] = 1;
90 }
91
92 init_sc(sub_A, L, a, (double[3]){0, 0, 0}); // Initialize sublattice A
93 init_sc(sub_B, L, a, (double[3]){0.5, 0.5, 0.5}); // Initialize sublattice B
94 nearest_neighbors_bcc(sub_A, sub_B, N_atoms, neighbors, 10 * a, 0.001,
    ↪ closest_distance_bcc); // Find nearest neighbors
95
96 // File for equilibrium
97 sprintf(filename, "data/task_2/equilibrium_%i_%.0f.csv", its_eq, T);
98 FILE *fp_eq = fopen(filename, "w");
99 fprintf(fp_eq, "accepted, E_tot\n");
100
101 // Run metropolis for equilibrium
102 metro_result = metropolis(its_eq, atoms, neighbors, k_B, T, E_cucu, E_znzn, E_cuzn,
    ↪ E_initial, r, NULL, NULL, NULL, NULL, N_atoms, fp_eq);
103 double E_tot = metro_result.Etot;

```

```

104     int accepted = metro_result.accepted;
105     printf("Acceptance rate from equilibrium: %f\n", (double)accepted / its_eq);
106
107     int its = 1000000; // Number of MC iterations after equilibrium
108
109     // File for MC iterations
110     sprintf(filename, "data/task_2/energy_%i_%i%.0f.csv", its_eq, its, T);
111     FILE *fp = fopen(filename, "w");
112     fprintf(fp, "Equilibrium iterations, Accepted ratio equilibrium, E_tot\n");
113     fprintf(fp, "%i, %f, %f\n", its_eq, (double)accepted / its_eq, E_tot);
114     fprintf(fp, "accepted, E_tot\n");
115
116     // File for atoms and neighbors
117     sprintf(filename, "data/task_2/lattice/atoms_%i_%i%.0f.csv", its_eq, its, T);
118     FILE *fp_atoms = fopen(filename, "w");
119     sprintf(filename, "data/task_2/lattice/neighbors.csv");
120     FILE *fp_neighbors = fopen(filename, "w");
121
122     // Run metropolis for MC iterations
123     metro_result = metropolis(its, atoms, neighbors, k_B, T, E_cucu, E_znzn, E_cuzn,
124     ↪ E_tot, r, NULL, NULL, NULL, NULL, N_atoms, fp);
125     // Calculate lattice properties
126     lattice_to_files(fp_atoms, fp_neighbors, atoms, neighbors, N_atoms);
127
128     // Free memory and close files
129     destroy_2D_array(sub_A);
130     destroy_2D_array(sub_B);
131     destroy_2D_array((double **)neighbors);
132     free(atoms);
133     fclose(fp);
134     fclose(fp_eq);
135     fclose(fp_atoms);
136     gsl_rng_free(r);
137
138     // ----- Task 3 ----- //
139
140     double T_start = 300; // T value at MC simulation start [K]
141     double T_end = 1000; // T value at MC simulation end [K]
142     int dt = 25; // T step size [K]
143     metro metro_result;
144     metro metro_result_eq;
145     atom_count lat_props;
146
147     // File for data
148     sprintf(filename, "data/task_3/data.csv");
149     FILE *fp_data = fopen(filename, "w");
150     fprintf(fp_data, "T, U, C_V, P, r, Accepted ratio, Accepted ratio equilibrium,\n");
151     ↪ Iterations\n");
152
153     int its_eq;
154     // Loop over T values
155     for (double T = T_start; T < T_end+1; T+=dt)
156     {
157         printf("T: %f\n", T); // Print T value, for debugging
158         // Let the number of iterations for equilibrium depend on T
159         if (T < 600)

```

```

158     {
159         its_eq = 250000;
160     }
161     else
162     {
163         its_eq = 100000;
164     }
165     double **sub_A = create_2D_array(N_atoms / 2, 3); // Sublattice A
166     double **sub_B = create_2D_array(N_atoms / 2, 3); // Sublattice B
167     int **neighbors = (int **)create_2D_array(N_atoms, 8); // Neighbors matrix
168     int *atoms = (int *)calloc(N_atoms, sizeof(int)); // Atoms vector
169     // Initialize all atoms in sublattice A to be Cu (1) (= Cold start)
170     for (int i = 0; i < N_atoms / 2; i++)
171     {
172         atoms[i] = 1;
173     }
174
175     init_sc(sub_A, L, a, (double[3]){0, 0, 0}); // Initialize sublattice A
176     init_sc(sub_B, L, a, (double[3]){0.5, 0.5, 0.5}); // Initialize sublattice B
177     nearest_neighbors_bcc(sub_A, sub_B, N_atoms, neighbors, 10 * a, 0.001,
178         ↪ closest_distance_bcc); // Find nearest neighbors
179
180     // Run metropolis for equilibrium
181     metro_result_eq = metropolis(its_eq, atoms, neighbors, k_B, T, E_cucu, E_znzn,
182         ↪ E_cuzn, E_initial, r, NULL, NULL, NULL, NULL, N_atoms, NULL);
183
184     int its = 100000; // Number of accepted MC iterations
185     double *U = (double *)calloc(its, sizeof(double)); // Energy vector
186     double *C_V = (double *)calloc(its, sizeof(double)); // Heat capacity vector
187     double *P = (double *)calloc(its, sizeof(double)); // Long-range order vector
188     double *R = (double *)calloc(its, sizeof(double)); // Short-range order vector
189
190     // Run metropolis for MC iterations
191     metro_result = metropolis(its, atoms, neighbors, k_B, T, E_cucu, E_znzn, E_cuzn,
192         ↪ metro_result_eq.Etot, r, U, C_V, P, R, N_atoms, NULL);
193     lat_props = lattice_props(atoms, neighbors, N_atoms); // Calculate lattice
194     ↪ properties, e.g., N_Cu_A and N_CuZn
195     its = metro_result.its; // Number of iterations
196
197     double U_avg = average(U, metro_result.accepted); // Average energy
198     double C_V_inst = variance(U, metro_result.accepted) / k_B / T / T; //
199     ↪ Instantaneous heat capacity
200     double p = (2. * lat_props.N_Cu_A / (N_atoms / 2) - 1); // Long-range order
201     ↪ parameter
202     double Rr = (lat_props.N_CuZn - 4. * N_atoms / 2) / (4 * N_atoms / 2); //
203     ↪ Short-range order parameter
204     fprintf(fp_data, "%f, %f, %f, %f, %f, %f, %f, %i\n", T, U_avg, C_V_inst, p, Rr,
205         ↪ (double)metro_result.accepted / its, (double)metro_result_eq.accepted /
206         ↪ its_eq, its);
207
208     // Files for autocorrelation and blocking
209     sprintf(filename, "data/task_3/auto_corr%i.csv", (int)T);
210     FILE *fp_auto_corr = fopen(filename, "w");
211     fprintf(fp_auto_corr, "N, Var_U, Var_CV, Var_P, Var_R\n");
212     fprintf(fp_auto_corr, "%i, %f, %f, %f, %f\n", metro_result.accepted, variance(U,
213         ↪ metro_result.accepted), variance(C_V, metro_result.accepted), variance(P,
214         ↪ metro_result.accepted), variance(R, metro_result.accepted));

```

```

204     fprintf(fp_auto_corr, "Lag, U, C_V, P, R\n");
205
206     sprintf(filename, "data/task_3/blocking_%i.csv", (int)T);
207     FILE *fp_blocking = fopen(filename, "w");
208     fprintf(fp_blocking, "Block_size, U, C_V, P, R\n");
209
210     // Loop over block sizes
211     for (int b = 1; b < metro_result.accepted*0.2; b+=10)
212     {
213         fprintf(fp_blocking, "%i, %f, %f, %f, %f\n", b, block_average(U,
214             ↪ metro_result.accepted, b), block_average(C_V, metro_result.accepted, b),
215             ↪ block_average(P, metro_result.accepted, b), block_average(R,
216             ↪ metro_result.accepted, b));
217     }
218     // Loop over lags
219     for (int i = 0; i < metro_result.accepted*0.5;
220         ↪ i+=(metro_result.accepted*0.5/1000+1))
221     {
222         // Subtract the average value from the data
223         addition_with_constant(U, U, -average(U, metro_result.accepted),
224             ↪ metro_result.accepted);
225         addition_with_constant(C_V, C_V, -average(C_V, metro_result.accepted),
226             ↪ metro_result.accepted);
227         addition_with_constant(P, P, -average(P, metro_result.accepted),
228             ↪ metro_result.accepted);
229         addition_with_constant(R, R, -average(R, metro_result.accepted),
230             ↪ metro_result.accepted);
231         fprintf(fp_auto_corr, "%i, %f, %f, %f, %f\n", i, autocorrelation(U,
232             ↪ metro_result.accepted, i), autocorrelation(C_V, metro_result.accepted,
233             ↪ i), autocorrelation(P, metro_result.accepted, i), autocorrelation(R,
234             ↪ metro_result.accepted, i));
235     }
236
237     // Free memory and close files
238     free(U);
239     free(C_V);
240     free(P);
241     free(R);
242     destroy_2D_array(sub_A);
243     destroy_2D_array(sub_B);
244     destroy_2D_array((double **)neighbors);
245     free(atoms);
246 }
247
248 // Free memory and close files
249 fclose(fp_data);
250 gsl_rng_free(r);
251
252 return 0;
253 }
254
255 gsl_rng *
256 init_gsl_rng(
257     int seed
258 )
259 {
260     const gsl_rng_type * T;

```

```

250     gsl_rng * r;
251     gsl_rng_env_setup();
252     T = gsl_rng_default; // default random number generator
253     r = gsl_rng_alloc(T); // allocate memory for the random number generator
254
255     if (!r) {
256         fprintf(stderr, "Error: Could not allocate memory for RNG.\n");
257         exit(EXIT_FAILURE); // Exit if allocation fails
258     }
259
260     // Set the seed
261     gsl_rng_set(r, seed);
262
263     return r;
264 }
265
266 double
267 boundary_distance_between_vectors(
268     double *v1,
269     double *v2,
270     int dim,
271     double box_length
272 )
273 {
274     double r = 0.0;
275     double delta;
276     for (int d = 0; d < dim; d++)
277     {
278         delta = v1[d] - v2[d];
279         // Apply boundary conditions
280         delta -= round(delta / box_length) * box_length;
281         r += delta * delta;
282     }
283
284     return sqrt(r);
285 }
286
287 void
288 nearest_neighbors_bcc(
289     double **pos_A,
290     double **pos_B,
291     int N_atoms,
292     int **neighbors,
293     double box_length,
294     double cutoff,
295     double closest_distance
296 )
297 {
298     double r; // Distance between atoms
299     // Loop over all atoms in sublattice A
300     for (int i = 0; i < N_atoms / 2; i++)
301     {
302         int count = 0; // Counter for neighbors
303         for (int j = 0; j < N_atoms / 2; j++)
304         {
305             r = boundary_distance_between_vectors(pos_A[i], pos_B[j], 3, box_length);
306             // If the distance between atoms is within the cutoff distance, store the

```

```

307         // index of the atom in sublattice B (index > 1000 in atoms vector)
308         //
309         if (fabs(closest_distance - r) < cutoff)
310         {
311             neighbors[i][count] = j + 1000;
312             count++;
313         }
314     }
315 }
316 // Loop over all atoms in sublattice B
317 for (int i = 0; i < N_atoms / 2; i++)
318 {
319     int count = 0; // Counter for neighbors
320     for (int j = 0; j < N_atoms / 2; j++)
321     {
322         r = boundary_distance_between_vectors(pos_B[i], pos_A[j], 3, box_length);
323         // If the distance between atoms is within the cutoff distance, store the
324         // index of the atom in sublattice A (index < 1000 in atoms vector)
325         if (fabs(closest_distance - r) < cutoff)
326         {
327             neighbors[i+1000][count] = j;
328             count++;
329         }
330     }
331 }
332 }
333
334 metro
335 metropolis(
336     int its,
337     int *atoms,
338     int **neighbors,
339     double k_B,
340     double T,
341     double E_cucu,
342     double E_znzn,
343     double E_cuzn,
344     double E_tot,
345     gsl_rng *r,
346     double *U,
347     double *C_V,
348     double *P,
349     double *R,
350     int N_atoms,
351     FILE *fp
352 )
353 {
354     metro metro_result;
355     atom_count lat_props;
356     idx index;
357     double E; // Energy before swap
358     double E_prime; // Energy after swap
359     double delta_E; // Energy difference
360     double alpha; // Acceptance probability
361     int accepted = 0; // Number of accepted swaps
362     int i = 0; // Number of iterations
363     // Loop over the number of iterations (task 2) or until its swaps are accepted (task
    ↪ 3)

```



```

364 // for (int i = 0; i < its; i++)
365 while (accepted < its)
366 {
367     index = swappy(atoms, r); // Get the indices and values of the atoms to swap
368     int A = index.alpha; // Index of the atom in sub_A
369     int B = index.beta; // Index of the atom in sub_B
370     int value_A = index.valueA; // Value of the atom in sub_A
371     int value_B = index.valueB; // Value of the atom in sub_B
372     E = energy_bond(index, atoms, neighbors, E_cucu, E_znzn, E_cuzn); // Energy
    ↪ before swap
373     atoms[A] = value_B; // Swap the atoms
374     atoms[B] = value_A; // Swap the atoms
375     E_prime = energy_bond(index, atoms, neighbors, E_cucu, E_znzn, E_cuzn); // Energy
    ↪ after swap
376     delta_E = E_prime - E; // Energy difference
377     alpha = exp(-delta_E / k_B / T); // Acceptance probability
378     // If the swap is accepted, increment the number of accepted swaps and update the
    ↪ energy
379     if (gsl_rng_uniform(r) < alpha)
380     {
381         accepted++;
382         E_tot += delta_E;
383         // Files for task 2
384         if (fp != NULL)
385         {
386             fprintf(fp, "%i, %f\n", accepted, E_tot);
387         }
388         // For task 3, store the quantity values in the vectors (for autocorrelation
    ↪ and blocking)
389         if (U != NULL)
390         {
391             U[accepted-1] = E_tot;
392             double U_fluct = variance(U, accepted);
393             C_V[accepted-1] = U_fluct / k_B / T / T;
394             lat_props = lattice_props(atoms, neighbors, N_atoms);
395             P[accepted-1] = (2. * lat_props.N_Cu_A / (N_atoms / 2) - 1);
396             R[accepted-1] = (lat_props.N_CuZn - 4. * N_atoms / 2) / (4 * N_atoms /
    ↪ 2);
397         }
398     }
399     else
400     {
401         // File for task 2
402         if (fp != NULL)
403         {
404             fprintf(fp, "%i, %f\n", accepted, E_tot);
405         }
406         // If the swap is not accepted, swap the atoms back
407         atoms[A] = value_A;
408         atoms[B] = value_B;
409     }
410     i++; // Increment the number of iterations
411 }
412 // Store the values in the struct and return it
413 metro_result.accepted = accepted;
414 metro_result.Etot = E_tot;
415 metro_result.its = i;

```

```

416     return metro_result;
417 }
418
419
420 idx
421 swappy(
422     int *atoms,
423     gsl_rng *r
424 )
425 {
426     idx index;
427     int idx_1 = (int)(1999 * gsl_rng_uniform(r)); // Random index for atom A
428     int A = atoms[idx_1]; // Value of atom A
429     int idx_2 = (int)(1999 * gsl_rng_uniform(r)); // Random index for atom B
430     int B = atoms[idx_2]; // Value of atom B
431     // Make sure that A and B are not the same atom type
432     while (A == B)
433     {
434         idx_2 = (int)(1999 * gsl_rng_uniform(r));
435         B = atoms[idx_2];
436     }
437     // Store the indices and values of the atoms and return the resulting struct
438     index.alpha = idx_1;
439     index.beta = idx_2;
440     index.valueA = A;
441     index.valueB = B;
442
443     return index;
444 }
445
446 double
447 energy_bond(
448     idx index,
449     int *atoms,
450     int **neighbors,
451     double E_cucu,
452     double E_znzn,
453     double E_cuzn
454 )
455 {
456     double E = 0.; // Energy
457     int atom_1_idx = index.alpha; // Index of atom A
458     int atom_2_idx = index.beta; // Index of atom B
459     int atom_1 = atoms[atom_1_idx]; // Value of atom A
460     int atom_2 = atoms[atom_2_idx]; // Value of atom B
461     int *neighbors_1 = neighbors[atom_1_idx]; // Neighbors of atom A
462     int *neighbors_2 = neighbors[atom_2_idx]; // Neighbors of atom B
463     // Loop over the neighbors of atom A
464     for (int i = 0; i < 8; i++)
465     {
466         // Add the energy of the bond between atom A and its neighbors
467         if (atoms[neighbors_1[i]] == 1 && atom_1 == 1)
468         {
469             E += E_cucu;
470         }
471         else if (atoms[neighbors_1[i]] == 0 && atom_1 == 0)
472         {

```

```

473         E += E_znzn;
474     }
475     else
476     {
477         E += E_cuzn;
478     }
479 }
480 // Loop over the neighbors of atom B
481 for (int i = 0; i < 8; i++)
482 {
483     // Add the energy of the bond between atom B and its neighbors
484     if (atoms[neighbors_2[i]] == 1 && atom_2 == 1)
485     {
486         E += E_cucu;
487     }
488     else if (atoms[neighbors_2[i]] == 0 && atom_2 == 0)
489     {
490         E += E_znzn;
491     }
492     else
493     {
494         E += E_cuzn;
495     }
496 }
497 return E;
498 }
499
500 atom_count
501 lattice_props(
502     int *atoms,
503     int **neighbors,
504     int N_atoms
505 )
506 {
507     atom_count count;
508     int N_Cu_A = 0; // Number of Cu atoms in sublattice A
509     int N_CuZn = 0; // Number of Cu-Zn bonds
510     // Loop over all atoms in sublattice A
511     for (int i = 0; i < N_atoms / 2; i++)
512     {
513         // If the atom is Cu, increment N_Cu_A
514         if (atoms[i] == 1)
515         {
516             N_Cu_A++;
517         }
518         int atom = atoms[i]; // Value of the atom
519         // Loop over the neighbors of the atom
520         for (int j = 0; j < 8; j++)
521         {
522             // If the atom is Cu and the neighbor is Zn or vice versa, increment N_CuZn
523             if (atom == 1 && atoms[neighbors[i][j]] == 0)
524             {
525                 N_CuZn++;
526             }
527             else if (atom == 0 && atoms[neighbors[i][j]] == 1)
528             {
529                 N_CuZn++;

```

```

530     }
531 }
532 }
533 // Store the values in the struct and return it
534 count.N_Cu_A = N_Cu_A;
535 count.N_CuZn = N_CuZn;
536
537 return count;
538 }
539
540 void
541 lattice_to_files(
542     FILE *fp_atoms,
543     FILE *fp_neighbors,
544     int *atoms,
545     int **neighbors,
546     int N_atoms
547 )
548 {
549     for (int i = 0; i < N_atoms; i++)
550     {
551         fprintf(fp_atoms, "%i\n", atoms[i]);
552         for (int j = 0; j < 8; j++)
553         {
554             if (j == 7)
555             {
556                 fprintf(fp_neighbors, "%i\n", neighbors[i][j]);
557             }
558             else
559             {
560                 fprintf(fp_neighbors, "%i, ", neighbors[i][j]);
561             }
562         }
563     }
564 }
565 }

```

## C.2 tools.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <gsl/gsl_fft_real.h>
5  #include <gsl/gsl_fft_halfcomplex.h>
6  #include <complex.h>
7  #include "tools.h"
8
9  void
10 elementwise_addition(
11     double *res,
12     double *v1,
13     double *v2,
14     unsigned int len
15 )

```

```

16 {
17     for (int i = 0; i < len; i++) {
18         res[i] = v1[i] + v2[i];
19     }
20 }
21
22 void
23 elementwise_multiplication(
24     double *res,
25     double *v1,
26     double *v2,
27     unsigned int len
28 )
29 {
30     for (int i = 0; i < len; i++) {
31         res[i] = v1[i] * v2[i];
32     }
33 }
34
35 void
36 addition_with_constant(
37     double *res,
38     double *v,
39     double constant,
40     unsigned int len
41 )
42 {
43     for (int i = 0; i < len; i++) {
44         res[i] = v[i] + constant;
45     }
46 }
47
48 void
49 multiplication_with_constant(
50     double *res,
51     double *v,
52     double constant,
53     unsigned int len
54 )
55 {
56     for (int i = 0; i < len; i++) {
57         res[i] = v[i] * constant;
58     }
59 }
60
61 double
62 dot_product(
63     double *v1,
64     double *v2,
65     unsigned int len
66 )
67 {
68     double res = 0.;
69     for (int i = 0; i < len; i++) {
70         res += v1[i] * v2[i];
71     }
72     return res;

```

```

73 }
74
75 double **
76 create_2D_array(
77     unsigned int row_size,
78     unsigned int column_size
79 )
80 {
81     // Allocate memory for row pointers
82     double **array = malloc(row_size * sizeof(double *));
83     if (array == NULL) {
84         fprintf(stderr, "Memory allocation failed for row pointers.\n");
85         return NULL;
86     }
87
88     // Allocate a single contiguous block for all elements, i.e. all matrix elements
89     // get stored in a single block of memory, array[0] is the start of the block
90     // i.e. here we have array[0][index] as the way to access the elements
91     array[0] = malloc(row_size * column_size * sizeof(double));
92     if (array[0] == NULL) {
93         fprintf(stderr, "Memory allocation failed for data block.\n");
94         free(array);
95         return NULL;
96     }
97
98     // Set the row pointers to the appropriate positions in the block, in order to
99     // allow for the array[i][j] syntax. Now array[i] points to the start of the
100    // i-th row.
101    for (int i = 1; i < row_size; i++) {
102        array[i] = array[0] + i * column_size;
103    }
104
105    return array;
106 }
107
108 void
109 destroy_2D_array(
110     double **array
111 )
112 {
113     if (array != NULL) {
114         free(array[0]); // Free the contiguous block
115         free(array); // Free the row pointers
116     }
117 }
118
119 void
120 matrix_vector_multiplication(
121     double *result,
122     double **A,
123     double *b,
124     unsigned int n,
125     unsigned int m
126 )
127 {
128     for (int i = 0; i < n; i++) {
129         result[i] = 0.;

```

```

130         for (int j = 0; j < m; j++) {
131             result[i] += A[i][j] * b[j];
132         }
133     }
134 }
135
136 void
137 matrix_matrix_multiplication(
138     double **result,
139     double **A,
140     double **B,
141     unsigned int n,
142     unsigned int m,
143     unsigned int k
144 )
145 {
146     for (int i = 0; i < n; i++) {
147         for (int kappa = 0; kappa < k; kappa++) {
148             result[i][kappa] = 0.;
149             for (int j = 0; j < m; j++) {
150                 result[i][kappa] += A[i][j] * B[j][kappa];
151             }
152         }
153     }
154 }
155
156 double
157 vector_norm(
158     double *v1,
159     unsigned int len
160 )
161 {
162     double res = 0.;
163     for (int i = 0; i < len; i++) {
164         res += v1[i] * v1[i];
165     }
166     return sqrt(res);
167 }
168
169 void
170 normalize_vector(
171     double *v,
172     unsigned int len
173 )
174 {
175     double norm = vector_norm(v, len);
176     multiplication_with_constant(v, v, 1. / norm, len);
177 }
178
179 double
180 average(
181     double *v,
182     unsigned int len
183 )
184 {
185     double res = 0.;

```

```

187     for (int i = 0; i < len; i++) {
188         res += v[i];
189     }
190     return res / len;
191 }
192
193
194 double
195 standard_deviation(
196     double *v,
197     unsigned int len
198 )
199 {
200     double avg = average(v, len);
201     double res = 0.;
202     for (int i = 0; i < len; i++) {
203         res += (v[i] - avg) * (v[i] - avg);
204     }
205     return sqrt(res / len);
206 }
207
208 double
209 variance(
210     double *v,
211     unsigned int len
212 )
213 {
214     double var;
215     var = pow(standard_deviation(v, len), 2);
216
217     return var;
218 }
219
220 double
221 autocorrelation(
222     double *data,
223     int data_len,
224     int time_lag_ind
225 )
226 {
227     double corr = 0.;
228     double avg = average(data, data_len);
229     double var = variance(data, data_len);
230
231     for (int i = 0; i < data_len - time_lag_ind; i++)
232     {
233         corr += (data[i]*data[i+time_lag_ind] - avg*avg) / var;
234     }
235
236     return corr / (data_len - time_lag_ind);
237 }
238
239 double
240 block_average(
241     double *data,
242     int data_len,
243     int block_size

```



```

244         )
245     {
246         int num_blocks = data_len / block_size;
247         double *block_averages = (double *)calloc(num_blocks, sizeof(double));
248         for (int i = 0; i < num_blocks; i++) {
249             double sum = 0.0;
250             for (int j = 0; j < block_size; j++) {
251                 sum += data[i * block_size + j];
252             }
253             block_averages[i] = sum / block_size;
254         }
255         double block_avg = block_size * variance(block_averages, num_blocks) / variance(data,
↵ data_len);
256         free(block_averages);
257
258         return block_avg;
259     }
260
261     double
262     distance_between_vectors(
263         double *v1,
264         double *v2,
265         unsigned int len
266     )
267     {
268         double res = 0.;
269         // With previous defined functions
270         double *diff = malloc(len * sizeof(double));
271         multiplication_with_constant(v1, v1, -1., len);
272         elementwise_addition(diff, v1, v2, len);
273         res = vector_norm(diff, len);
274         free(diff);
275         return res;
276     }
277
278     void
279     cumulative_integration(
280         double *res,
281         double *v,
282         double dx,
283         unsigned int v_len
284     )
285     {
286         double sum = 0.;
287         res[0] = 0.;
288         for (int i = 1; i < v_len; i++) {
289             sum = 0.5 * (v[i - 1] + v[i]) * dx;
290             res[i] = res[i - 1] + sum;
291         }
292     }
293
294     void
295     write_xyz(
296         FILE *fp,
297         char *symbol,
298         double **positions,
299         double **velocities,

```

```

300         double alat,
301         int natoms
302     )
303 {
304     fprintf(fp, "%i\nLattice=\"%f 0.0 0.0 0.0 %f 0.0 0.0 0.0 %f\" ", natoms, alat, alat,
↪      alat);
305     fprintf(fp, "Properties=species:S:1:pos:R:3:vel:R:3 pbc=\"T T T\"\n");
306     for(int i = 0; i < natoms; ++i){
307         fprintf(fp, "%s %f %f %f %f %f %f\n",
308             symbol, positions[i][0], positions[i][1], positions[i][2],
309             velocities[i][0], velocities[i][1], velocities[i][2]);
310     }
311 }
312
313 void fft_freq(
314     double *res,
315     int n,
316     double timestep
317 )
318 {
319     for (int i = 0; i < n; i++) {
320         if (i < n / 2) {
321             res[i] = 2 * M_PI * i / (n * timestep);
322         }
323         else {
324             res[i] = 2 * M_PI * (i - n) / (n * timestep);
325         }
326     }
327 }
328
329 /* Freely given functions */
330 void
331 skip_line(
332     FILE *fp
333 )
334 {
335     int c;
336     while (c = fgetc(fp), c != '\n' && c != EOF);
337 }
338
339 void
340 read_xyz(
341     FILE *fp,
342     char *symbol,
343     double **positions,
344     double **velocities,
345     double *alat
346 )
347 {
348     int natoms;
349     if(fscanf(fp, "%i\nLattice=\"%lf 0.0 0.0 0.0 %lf 0.0 0.0 0.0 %lf\" ", &natoms, alat,
↪      alat, alat) == 0){
350         perror("Error");
351     }
352     skip_line(fp);
353     for(int i = 0; i < natoms; ++i){
354         fscanf(fp, "%s %lf %lf %lf ",

```

```

355         symbol, &positions[i][0], &positions[i][1], &positions[i][2]);
356         fscanf(fp, "%lf %lf %lf\n",
357             &velocities[i][0], &velocities[i][1], &velocities[i][2]);
358     }
359 }
360
361 void powerspectrum(
362     double *res,
363     double *signal,
364     int n,
365     double timestep
366 )
367 {
368     /* Declaration of variables */
369     double *complex_coefficient = malloc(sizeof(double) * 2*n); // array for the complex
370     ↪ fft data
371     double *data_cp = malloc(sizeof(double) * n);
372
373     /*make copy of data to avoid messing with data in the transform*/
374     for (int i = 0; i < n; i++) {
375         data_cp[i] = signal[i];
376     }
377
378     /* Declare wavetable and workspace for fft */
379     gsl_fft_real_wavetable *real;
380     gsl_fft_real_workspace *work;
381
382     /* Allocate space for wavetable and workspace for fft */
383     work = gsl_fft_real_workspace_alloc(n);
384     real = gsl_fft_real_wavetable_alloc(n);
385
386     /* Do the fft*/
387     gsl_fft_real_transform(data_cp, 1, n, real, work);
388
389     /* Unpack the output into array with alternating real and imaginary part */
390     gsl_fft_halfcomplex_unpack(data_cp, complex_coefficient, 1, n);
391
392     /*fill the output powspec_data with the powerspectrum */
393     for (int i = 0; i < n; i++) {
394         res[i] =
395             ↪ (complex_coefficient[2*i]*complex_coefficient[2*i]+complex_coefficient[2*i+1]*complex_coefficient[2*i+1])
396         res[i] *= timestep / n;
397     }
398
399     /* Free memory of wavetable and workspace */
400     gsl_fft_real_wavetable_free(real);
401     gsl_fft_real_workspace_free(work);
402     free(complex_coefficient);
403     free(data_cp);
404 }

```

## D Source Code in Python

```
1  # Libraries
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import scipy.constants as K
5  import pandas as pd
6  from IPython.display import display, Markdown
7
8  # Latex style
9  plt.style.use('default')
10 plt.rc('text', usetex=True)
11 plt.rc('font', family='serif')
12 plt.rc('font', size=20)
13 plt.rcParams['text.latex.preamble'] = r'\usepackage{amsmath}'
14
15 # Set ticks on both sides
16 plt.rcParams['xtick.direction'] = 'in'
17 plt.rcParams['ytick.direction'] = 'in'
18 plt.rcParams['xtick.major.size'] = 5
19 plt.rcParams['ytick.major.size'] = 5
20 plt.rcParams['xtick.top'] = True
21 plt.rcParams['ytick.right'] = True
22
23 # Constants
24 k_B = K.Boltzmann
25 e = K.elementary_charge
26 k_B /= e
27 n_atoms = 2000
28
29 # Functions
30 def read_data(task, T, its_eq=0, its=0):
31     '''
32     Read data from csv files.
33
34     Args:
35         task (int): task number.
36         T (int): temperature.
37         its_eq (int): number of equilibrium iterations.
38         its (int): number of MC iterations.
39
40     Returns:
41         if task == 2:
42             eq_data (np.array): equilibrium data.
43             eq_facts (np.array): equilibrium facts, i.e. acceptance ratio, its, etc.
44             atoms_data (np.array): atoms data, showing which atom is at what position.
45             energy_data (np.array): energy data.
46             energy_facts (np.array): energy facts, i.e. acceptance ratio, its, etc.
47         if task == 3 & T:
48             N (int): number of samples.
49             auto_corr_data (np.array): autocorrelation data.
50             block_data (np.array): block averaging data.
51             error_dict_corr (dict): errors from autocorrelation.
52             error_dict_block (dict): errors from block averaging.
53             var_dict (dict): variance of U, C_V, P, r.
54         if task == 3 & not T:
55             data (np.array): data from task 3.
```

```

56
57
58 if task == 2:
59     if its_eq:
60         eq_facts = np.genfromtxt(f'data/task_2/equilibrium_{its_eq}_{T}.csv',
61                                 ↪ max_rows=1, delimiter=',', dtype=np.float64)
62         eq_data = np.genfromtxt(f'data/task_2/equilibrium_{its_eq}_{T}.csv',
63                                 ↪ delimiter=',', dtype=np.float64)[1:, :]
64     else:
65         eq_facts = None
66         eq_data = None
67     if its:
68         atoms_data =
69             ↪ np.genfromtxt(f'data/task_2/lattice/atoms_{its_eq}_{its}_{T}.csv')
70         energy_facts = np.genfromtxt(f'data/task_2/energy_{its_eq}_{its}_{T}.csv',
71                                     ↪ max_rows=2, delimiter=',', dtype=np.float64)[1]
72         energy_data = np.genfromtxt(f'data/task_2/energy_{its_eq}_{its}_{T}.csv',
73                                     ↪ delimiter=',', dtype=np.float64, skip_header=3)
74     else:
75         atoms_data = None
76         energy_facts = None
77         energy_data = None
78     return eq_data, eq_facts, atoms_data, energy_data, energy_facts
79
80 if task == 3:
81     if T:
82         filename_corr = f'data/task_{task}/auto_corr_{T}.csv'
83         filename_block = f'data/task_{task}/blocking_{T}.csv'
84
85         N, var_U, var_CV, var_P, var_r = np.genfromtxt(filename_corr, delimiter=',',
86                                                         ↪ dtype=np.float64)[1, :]
87         auto_corr_data = np.genfromtxt(filename_corr, delimiter=',',
88                                         ↪ dtype=np.float64)[3:, :]
89         block_data = np.genfromtxt(filename_block, delimiter=',',
90                                   ↪ dtype=np.float64)[1:, :]
91
92         # Find the first index where the autocorrelation is less than exp(-2)
93         s_corr_U = auto_corr_data[:, 0][np.where(auto_corr_data[:, 1] <
94                                                    ↪ np.exp(-2))[0][0]]
95         s_corr_CV = auto_corr_data[:, 0][np.where(auto_corr_data[:, 2] <
96                                                    ↪ np.exp(-2))[0][0]]
97         s_corr_P = auto_corr_data[:, 0][np.where(auto_corr_data[:, 3] <
98                                                    ↪ np.exp(-2))[0][0]]
99         s_corr_r = auto_corr_data[:, 0][np.where(auto_corr_data[:, 4] <
100                                                    ↪ np.exp(-2))[0][0]]
101
102         # Calculate the statistical inefficiency from the blocking data
103         s_block_U = np.mean(block_data[:, 1])
104         s_block_CV = np.mean(block_data[:, 2])
105         s_block_P = np.mean(block_data[:, 3])
106         s_block_r = np.mean(block_data[:, 4])
107
108         var_dict = {'U': var_U, 'C_V': var_CV, 'P': var_P, 'r': var_r}
109         error_dict_corr = {'U': s_corr_U, 'C_V': s_corr_CV, 'P': s_corr_P, 'r':
110                            ↪ s_corr_r}
111         error_dict_block = {'U': s_block_U, 'C_V': s_block_CV, 'P': s_block_P, 'r':
112                             ↪ s_block_r}

```

```

99         return N, auto_corr_data, block_data, error_dict_corr, error_dict_block,
100             ↪ var_dict
101     else:
102         data = np.genfromtxt('data/task_3/data.csv', delimiter=',', dtype=np.float64,
103             ↪ skip_header=1)
104
105         return data
106
107 def savitzky_golay(y, window_size, order, deriv=0, rate=1):
108     '''
109     Code for applying a filter to the sampled data.
110     Found here: https://scipy.github.io/old-wiki/pages/Cookbook/SavitzkyGolay.
111     '''
112     from math import factorial
113     try:
114         window_size = np.abs(int(window_size))
115         order = np.abs(int(order))
116     except ValueError:
117         raise ValueError("window_size and order have to be of type int")
118     if window_size % 2 != 1 or window_size < 1:
119         raise TypeError("window_size size must be a positive odd number")
120     if window_size < order + 2:
121         raise TypeError("window_size is too small for the polynomials order")
122     order_range = range(order+1)
123     half_window = (window_size -1) // 2
124     # precompute coefficients
125     b = np.asmatrix([[k**i for i in order_range] for k in range(-half_window,
126         ↪ half_window+1)])
127     m = np.linalg.pinv(b).A[deriv] * rate**deriv * factorial(deriv)
128     # pad the signal at the extremes with
129     # values taken from the signal itself
130     firstvals = y[0] - np.abs( y[1:half_window+1][::-1] - y[0] )
131     lastvals = y[-1] + np.abs(y[-half_window-1:-1][::-1] - y[-1])
132     y = np.concatenate((firstvals, y, lastvals))
133     return np.convolve( m[::-1], y, mode='valid')
134
135 def plot_task_1(quantity, T_c, save=False):
136     '''
137     Plot data for task 1.
138
139     Args:
140         quantity (str): 'UC' or 'P'.
141         T_c (float): critical temperature.
142         save (bool): whether to save the plot or not.
143     '''
144     # Read data and define the temperature
145     data = np.loadtxt('data/task_1/data.csv', delimiter=',')
146     T = data[:, 1]
147     if quantity == 'UC':
148         data_U = data[:, 2] # Potential energy
149         data_C = data[:, 3] # Heat capacity
150         fact_U = 0.008 # Factor to increase the y-axis
151         fact_C = 0.2 # Factor to increase the y-axis
152
153     # Plot the data
154     fig, axs = plt.subplots(1, 2, figsize=(14, 7))

```

```

153     axs[0].plot(T, data_U, 'k', lw=2.5, label='$U(T)$')
154     axs[0].axvline(x=T_c, color='tab:orange', linestyle='--', label='$T_c$')
155     axs[0].set_xlabel('$T$ [K]')
156     axs[0].set_ylabel('$U(T)$ [eV]')
157     axs[0].grid(linestyle=':', linewidth=1, alpha=0.6)
158     axs[0].legend(loc='upper left', ncol=2)
159     axs[0].set_ylim(axs[0].get_ylim()[0], axs[0].get_ylim()[1] +
    ↪     abs(axs[0].get_ylim()[1] * fact_U))

160
161     axs[1].plot(T, data_C, 'k', lw=2.5, label='$C_V(T)$')
162     axs[1].axvline(x=T_c, color='tab:orange', linestyle='--', label='$T_c$')
163     axs[1].set_xlabel('$T$ [K]')
164     axs[1].set_ylabel('$C_V(T)$ [eV/K]')
165     axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
166     axs[1].legend(loc='upper left', ncol=2)
167     axs[1].set_ylim(axs[1].get_ylim()[0], axs[1].get_ylim()[1] +
    ↪     abs(axs[1].get_ylim()[1] * fact_C))

168
169     plt.tight_layout()
170     elif quantity == 'P':
171         data = abs(data[:, 0]) # Long-range order parameter
172         y_label = '$\\vert P\\vert$'
173         fact = 0.25 # Factor to increase the y-axis
174
175         # Plot the data
176         fig = plt.figure(figsize=(7, 6))
177         plt.plot(T, data, 'k', lw=2.5, label='$P(T)$')
178         plt.axvline(x=T_c, color='tab:orange', linestyle='--', label='$T_c$')
179         plt.xlabel('$T$ [K]')
180         plt.ylabel(y_label)
181         plt.ylim(plt.ylim()[0], plt.ylim()[1] + abs(plt.ylim()[1] * fact))
182         plt.grid(linestyle=':', linewidth=1, alpha=0.6)
183         plt.legend(loc='upper left', ncol=2)
184         plt.tight_layout()
185
186     if save:
187         save_fig(fig, f'{quantity}_T', 1)
188     plt.show()
189
190 def analyzer_task_2(T, its_eq, its, changed=True, plot=False, save=False):
191     '''
192     Analyze and plot task 2 data.
193
194     Args:
195         T (list): temperatures to plot.
196         its_eq (list): equilibrium iterations.
197         its (list): MC iterations.
198         changed (bool): whether to keep only accepted steps or not.
199         plot (bool): whether to plot or not (if its).
200         save (bool): whether to save the plot or not.
201
202     Returns:
203         if its is None:
204             None.
205         if its is not None and changed:
206             U_changed (np.array): only the accepted steps of the potential energy.
207             acceptance equilibration: acceptance ratio during equilibration.

```

```

208         acceptance (float): acceptance ratio.
209     """
210     # Plot equilibrium data
211     if its_eq and its is None:
212         fig, axs = plt.subplots(1, len(T), figsize=(20, 10))
213         colors = ['tab:blue', 'tab:orange', 'tab:green']
214         labels = ['\\textbf{a}', '\\textbf{b}', '\\textbf{c}', '\\textbf{d}',
215             ↪ '\\textbf{e}', '\\textbf{f}']
216         if len(T) == 3:
217             axs = np.array([axs[0], axs[1], axs[2]])
218         for i, t in enumerate(T):
219             data, eq_facts, _, _ = read_data(2, t, its_eq=its_eq[i])
220             axs[0, i].plot(data[:, 1], lw=2, color=colors[i])
221             # axs[1, i].plot(data[:, 1], lw=2, color=colors[i])
222             # axs[0, 0].set_ylabel('Accepted steps')
223             axs[0, 0].set_ylabel('Potential energy [eV]')
224             axs[-1, i].set_xlabel('MC steps')
225             axs[0, i].set_title(f'$T = {t}$ K', fontsize=40)
226             axs[0, i].set_xticks(np.linspace(0, len(data[:, 0]), 3))
227             axs[0, i].text(0.65, 0.2, labels[i], transform=axs[0, i].transAxes,
228                 ↪ fontsize=50)
229             # axs[1, i].text(0.65, 0.2, labels[2*i + 1], transform=axs[1, i].transAxes,
230                 ↪ fontsize=50)
231             # axs[0, i].set_xticks([])
232         if save:
233             plt.tight_layout()
234             save_fig(fig, 'burnin', 2)
235     # Plot MC data
236     elif its and its_eq:
237         _, _, atoms_data, energy_data, energy_facts = read_data(2, T, its_eq=its_eq,
238             ↪ its=its)
239         accept_diff = np.diff(energy_data[:, 0]) # Difference in accepted steps
240         accept_diff = np.insert(accept_diff, 0, True) # Insert True at the beginning
241         U_changed = energy_data[:, 1][accept_diff > 0] # Only the accepted steps of the
242             ↪ potential energy
243
244     # Plot only the accepted steps
245     if plot and changed:
246         fig, axs = plt.subplots(1, 2, figsize=(9, 6))
247         axs[0].plot(energy_data[:, 0])
248         axs[1].plot(U_changed)
249         axs[0].set_ylabel('Accepted steps')
250         axs[1].set_ylabel('Potential energy [eV]')
251         axs[0].set_xlabel('MC steps')
252         axs[1].set_xlabel('MC steps')
253         plt.tight_layout()
254     # Plot all steps
255     elif plot and not changed:
256         fig, axs = plt.subplots(1, 2, figsize=(9, 6))
257         axs[0].plot(energy_data[:, 0])
258         axs[1].plot(energy_data[:, 1])
259         axs[0].set_ylabel('Accepted steps')
260         axs[1].set_ylabel('Potential energy [eV]')
261         axs[0].set_xlabel('MC steps')
262         axs[1].set_xlabel('MC steps')
263         plt.tight_layout()

```



```

260         if save:
261             save_fig(fig, 'MC_steps', 2)
262
263         return U_changed, energy_facts[1], energy_data[-1, 0] / len(energy_data[:, 0])
264
265     plt.show()
266
267 def plot_task_3(quantity, n_atoms=2000, save=False, error_plot=False,
268               save_error_plot=False):
269     '''
270     Plot quantity data for task 3.
271
272     Args:
273         quantity (str): 'UC' or 'Pr'.
274         n_atoms (int): number of atoms.
275         save (bool): whether to save the plot or not.
276         error_plot (bool): whether to plot the errors or not.
277         save_error_plot (bool): whether to save the error plot.
278     '''
279     # Read data, define the temperature and retrieve the errors
280     data = read_data(3, None)
281     T = data[1:, 0]
282     errors_corr, errors_block, facts = get_errors(list(T))
283     # Initialize plot variables
284     if quantity == 'UC':
285         fig, axs = plt.subplots(1, 2, figsize=(16, 8))
286         Y_1 = data[1:, 1]
287         Y_2 = data[1:, 2]
288         y_label_1 = '$U(T)$ [eV]'
289         y_label_2 = '$C_V(T)$ [eV/K]'
290         legend_loc = 'upper left'
291         ylim_1 = [-2360, -2260]
292         ylim_2 = [-0.01, 0.45]
293         colors = ['tab:blue', 'tab:orange']
294         # Calculate the errors as the average of the errors from autocorrelation and
295         ↪ block averaging
296         error_1 = (np.array(errors_corr['U'], dtype=np.float64) +
297                   ↪ np.array(errors_block['U'], dtype=np.float64)) / 2
298         error_2 = (np.array(errors_corr['C_V'], dtype=np.float64) +
299                   ↪ np.array(errors_block['C_V'], dtype=np.float64)) / 2
300     elif quantity == 'Pr':
301         fig, axs = plt.subplots(1, 2, figsize=(16, 8), sharey=True)
302         Y_1 = np.abs(data[1:, 3])
303         Y_2 = data[1:, 4]
304         y_label_1 = '$P(T)$'
305         y_label_2 = '$r(T)$'
306         legend_loc = 'upper left'
307         ylim_1 = [-0.05, 1.3]
308         ylim_2 = [-0.05, 1.3]
309         colors = ['tab:green', 'tab:red']
310         # Calculate the errors as the average of the errors from autocorrelation and
311         ↪ block averaging
312         error_1 = (np.array(errors_corr['P'], dtype=np.float64) +
313                   ↪ np.array(errors_block['P'], dtype=np.float64)) / 2
314         error_2 = (np.array(errors_corr['r'], dtype=np.float64) +
315                   ↪ np.array(errors_block['r'], dtype=np.float64)) / 2

```

```

311 # Plot the data with the errors
312 axs[0].errorbar(T, Y_1, yerr=error_1, fmt='-o', ecolor='k', capsize=6, capthick=2.7,
    ↪ linewidth=2.7, label=f'Data Points, {y_label_1.replace(' [eV]', '')}',
    ↪ markersize=7.5, color=colors[0])
313 axs[1].errorbar(T, Y_2, yerr=error_2, fmt='-o', ecolor='k', capsize=6, capthick=2.7,
    ↪ linewidth=2.7, label=f'Data Points, {y_label_2.replace(' [eV/K]', '')}',
    ↪ markersize=7.5, color=colors[1])
314 # Plot the filtered data
315 if quantity == 'UC':
316     axs[0].plot(T, savitzky_golay(Y_1, 5, 3), color='k', linewidth = 2.7,
    ↪ label='Filtered', alpha=0.5, linestyle='--')
317     axs[1].plot(T, savitzky_golay(Y_2, 5, 3), color='k', linewidth = 2.7,
    ↪ label='Filtered', alpha=0.5, linestyle='--')
318 else:
319     axs[0].plot(T, savitzky_golay(Y_1, 25, 9), color = 'k', linewidth = 2.7,
    ↪ label='Filtered', alpha=0.5, linestyle='--')
320     axs[1].plot(T, savitzky_golay(Y_2, 25, 9), color = 'k', linewidth = 2.7,
    ↪ label='Filtered', alpha=0.5, linestyle='--')
321 # Grid, labels and legends
322 for ax in axs:
323     ax.grid(linestyle=':', linewidth=1, alpha=0.6)
324     ax.set_xlabel('$T$ [K]')
325     ax.legend(fontsize = 26, loc=legend_loc, ncol=1)
326     if ax == axs[0]:
327         ax.set_ylabel(y_label_1)
328         ax.set_ylim(ylim_1)
329     else:
330         ax.set_ylabel(y_label_2)
331         ax.set_ylim(ylim_2)
332 if save:
333     plt.tight_layout()
334     plt.savefig(f'figs/task_3/{quantity}_T_metro.pdf', bbox_inches='tight')
335
336 # Plot the errors
337 if error_plot:
338     # Calculate the errors as the average of the errors from autocorrelation and
    ↪ block averaging
339     e_U = (np.array(errors_corr['U'], dtype=np.float64) + np.array(errors_block['U'],
    ↪ dtype=np.float64)) / 2
340     e_CV = (np.array(errors_corr['C_V'], dtype=np.float64) +
    ↪ np.array(errors_block['C_V'], dtype=np.float64)) / 2
341     e_P = (np.array(errors_corr['P'], dtype=np.float64) + np.array(errors_block['P'],
    ↪ dtype=np.float64)) / 2
342     e_r = (np.array(errors_corr['r'], dtype=np.float64) + np.array(errors_block['r'],
    ↪ dtype=np.float64)) / 2
343
344     fig, axs = plt.subplots(2, 2, figsize=(16, 16), sharex=True)
345     axs[0, 0].bar(T, e_U, color='tab:blue', label='$\\epsilon_{U}$', width=20,
    ↪ edgecolor='k')
346     axs[0, 1].bar(T, e_CV, color='tab:orange', label='$\\epsilon_{C_V}$', width=20,
    ↪ edgecolor='k')
347     axs[1, 0].bar(T, e_P, color='tab:green', label='$\\epsilon_{P}$', width=20,
    ↪ edgecolor='k')
348     axs[1, 1].bar(T, e_r, color='tab:red', label='$\\epsilon_{r}$', width=20,
    ↪ edgecolor='k')
349     labels = ['$\\epsilon_{U}$', '$\\epsilon_{C_V}$', '$\\epsilon_{P}$',
    ↪ '$\\epsilon_{r}$']

```

```

350     for i, ax in enumerate(axes.flatten()):
351         ax.grid(linestyle=':', linewidth=1, alpha=0.6)
352         ax.legend(loc='upper left', fontsize=32)
353         ax.set_ylabel(labels[i])
354         ax.set_xticks([400, 600, 800, 1000])
355     axes[1, 0].set_xlabel('$T$ [K]')
356     axes[1, 1].set_xlabel('$T$ [K]')
357     plt.tight_layout()
358
359     if save_error_plot:
360         plt.savefig(f'figs/task_3/errors.pdf', bbox_inches='tight')
361
362 plt.show()
363
364 def plot_task_3_error(T, autocorr, block, s_corr_U, s_corr_P, s_corr_CV, save=False,
365                      xlim_factor=1., xlim_factor_CVP=1., P_sep=False):
366     '''
367     Plot the autocorrelation and block averaging data for task 3.
368
369     Args:
370         T (int): temperature for which to plot the data.
371         autocorr (np.array): autocorrelation data.
372         block (np.array): block averaging data.
373         s_corr_U (float): statistical inefficiency for the potential energy.
374         s_corr_P (float): statistical inefficiency for the long-range order parameter.
375         s_corr_CV (float): statistical inefficiency for the heat capacity.
376         save (bool): whether to save the plot or not.
377         xlim_factor (float): factor to increase the x-axis limit for the potential
378                             energy/long/short-range order parameter.
379         xlim_factor_CVP (float): factor to increase the x-axis limit for the
380                                heat capacity/long-range order parameter.
381         P_sep (bool): whether to plot the long-range order parameter separately or not.
382     '''
383     fig, axes = plt.subplots(1, 2, figsize=(10, 6))
384     # Plot U, P and r data in one plot and C_V data in the other
385     if not P_sep:
386         # Autocorrelation data for U, P and r
387         axes[0].plot(autocorr[:, 0], autocorr[:, 1], 'tab:blue', lw=5)
388         axes[0].plot(autocorr[:, 0], autocorr[:, 3], 'tab:orange', lw=3)
389         axes[0].plot(autocorr[:, 0], autocorr[:, 4], 'tab:green', lw=2)
390         axes[0].plot([], [], 'tab:blue', label='$U$')
391         axes[0].plot([], [], 'tab:orange', label='$P$')
392         axes[0].plot([], [], 'tab:green', label='$r$')
393         axes[0].axhline(np.exp(-2), color='k', linestyle='--', label='$e^{-2}$')
394         axes[0].axvline(s_corr_U, ymin=0.025, ymax=0.365, color='k', linestyle='--')
395         axes[0].set_ylim(-0.5, 1.2)
396         axes[0].set_xlim(0, s_corr_P * 1.2)
397         axes[0].set_ylabel('Autocorrelation $\Phi_k$')
398         axes[0].set_xlabel('MC step $k$')
399         axes[0].grid(linestyle=':', linewidth=1, alpha=0.6)
400         axes[0].legend()
401         plt.tight_layout()
402
403     # Block averaging data for U, P and r
404     axes[1].plot(block[:, 0], block[:, 1], 'tab:blue', lw=6)
405     axes[1].plot(block[:, 0], block[:, 3], 'tab:orange', lw=3)
406     axes[1].plot(block[:, 0], block[:, 4], 'tab:green', lw=2)

```

```

407     axs[1].plot([], [], 'tab:blue', label='$U$')
408     axs[1].plot([], [], 'tab:orange', label='$P$')
409     axs[1].plot([], [], 'tab:green', label='$r$')
410     axs[1].set_ylabel('Statistical Inefficiency $s$')
411     axs[1].set_xlabel('Block size')
412     axs[1].set_xlim(0, np.max(block[:, 0]) * xlim_factor)
413     axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
414     axs[1].legend()
415     plt.tight_layout()
416
417     if save:
418         save_fig(fig, 'auto_corr_block_UPr'+str(T), 3)
419
420     fig, axs = plt.subplots(1, 2, figsize=(10, 6))
421     # Autocorrelation data for C_V
422     axs[0].plot(autocorr[:, 0], autocorr[:, 2], 'tab:red')
423     axs[0].plot([], [], 'tab:red', label='$C_V$')
424     axs[0].axhline(np.exp(-2), color='k', linestyle='-', label='$e^{-2}$')
425     axs[0].axvline(s_corr_CV, ymin=0.025, ymax=0.365, color='k', linestyle='--')
426     axs[0].set_ylim(-0.5, 1.2)
427     axs[0].set_xlim(0, s_corr_CV * 1.1)
428     axs[0].set_ylabel('Autocorrelation $\Phi_k$')
429     axs[0].set_xlabel('MC step $k$')
430     axs[0].grid(linestyle=':', linewidth=1, alpha=0.6)
431     axs[0].legend()
432     plt.tight_layout()
433
434     # Block averaging data for C_V
435     axs[1].plot(block[:, 0], block[:, 2], 'tab:red')
436     axs[1].plot([], [], 'tab:red', label='$C_V$')
437     axs[1].set_ylabel('Statistical Inefficiency $s$')
438     axs[1].set_xlabel('Block size')
439     axs[1].set_xlim(0, np.max(block[:, 0]) * xlim_factor_CVP)
440     axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
441     axs[1].legend()
442     plt.tight_layout()
443
444     if save:
445         save_fig(fig, 'auto_corr_block_CV'+str(T), 3)
446 else:
447     # Autocorrelation data for U and r
448     axs[0].plot(autocorr[:, 0], autocorr[:, 1], 'tab:blue', lw=5)
449     axs[0].plot(autocorr[:, 0], autocorr[:, 4], 'tab:green', lw=2)
450     axs[0].plot([], [], 'tab:blue', label='$U$')
451     axs[0].plot([], [], 'tab:green', label='$r$')
452     axs[0].axhline(np.exp(-2), color='k', linestyle='-', label='$e^{-2}$')
453     axs[0].axvline(s_corr_U, ymin=0.025, ymax=0.365, color='k', linestyle='--')
454     axs[0].set_ylim(-0.5, 1.2)
455     axs[0].set_xlim(0, s_corr_U * 1.2)
456     axs[0].set_ylabel('Autocorrelation $\Phi_k$')
457     axs[0].set_xlabel('MC step $k$')
458     axs[0].grid(linestyle=':', linewidth=1, alpha=0.6)
459     axs[0].legend()
460     plt.tight_layout()
461
462     # Block averaging data for U and r
463     axs[1].plot(block[:, 0], block[:, 1], 'tab:blue', lw=6)

```

```

464     axs[1].plot(block[:, 0], block[:, 4], 'tab:green', lw=2)
465     axs[1].plot([], [], 'tab:blue', label='$U$')
466     axs[1].plot([], [], 'tab:green', label='$r$')
467     axs[1].set_ylabel('Statistical Inefficiency $s$')
468     axs[1].set_xlabel('Block size')
469     axs[1].set_xlim(0, np.max(block[:, 0]) * xlim_factor)
470     axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
471     axs[1].legend()
472     plt.tight_layout()
473
474     if save:
475         save_fig(fig, 'auto_corr_block_Ur'+str(T), 3)
476
477     # Autocorrelation data for P
478     fig, axs = plt.subplots(1, 2, figsize=(10, 6))
479     axs[0].plot(autocorr[:, 0], autocorr[:, 3], 'tab:orange', lw=3)
480     axs[0].plot([], [], 'tab:orange', label='$P$')
481     axs[0].axhline(np.exp(-2), color='k', linestyle='-', label='$e^{-2}$')
482     axs[0].axvline(s_corr_P, ymin=0.025, ymax=0.365, color='k', linestyle='--')
483     axs[0].set_ylim(-0.5, 1.2)
484     axs[0].set_xlim(0, s_corr_P * 1.2)
485     axs[0].set_ylabel('Autocorrelation $\Phi_k$')
486     axs[0].set_xlabel('MC step $k$')
487     axs[0].grid(linestyle=':', linewidth=1, alpha=0.6)
488     axs[0].legend()
489     plt.tight_layout()
490
491     # Block averaging data for P
492     axs[1].plot(block[:, 0], block[:, 3], 'tab:orange', lw=3)
493     axs[1].plot([], [], 'tab:orange', label='$P$')
494     axs[1].set_ylabel('Statistical Inefficiency $s$')
495     axs[1].set_xlabel('Block size')
496     axs[1].set_xlim(0, np.max(block[:, 0]) * xlim_factor_CVP)
497     axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
498     axs[1].legend()
499     plt.tight_layout()
500
501     if save:
502         save_fig(fig, 'auto_corr_block_P'+str(T), 3)
503
504 plt.show()
505
506 def get_errors(Ts):
507     '''
508     Calculate the errors for task 3.
509
510     Args:
511         Ts (list): temperatures for which to calculate the errors.
512
513     Returns:
514         errors_corr (dict): errors from autocorrelation.
515         errors_block (dict): errors from block averaging.
516         facts (dict): facts about the simulations, i.e. number of samples, iterations
517                        and acceptance ratio.
518     '''
519     # Initialize the dictionaries
520     errors_corr = {

```

```

521     'U': [],
522     'C_V': [],
523     'P': [],
524     'r': []
525 }
526 errors_block = {
527     'U': [],
528     'C_V': [],
529     'P': [],
530     'r': []
531 }
532 facts = {
533     'samples': [],
534     'its': [],
535     'acceptance': []
536 }
537 # The number of iterations for each temperature
538 N_its = np.genfromtxt('data/task_3/data.csv', delimiter=',', dtype=np.float64,
539     ↪ skip_header=1)[: , 7]
540
541 # Iterate over the temperatures
542 for T in Ts:
543     # Read the data
544     N, autocorr, block, error_dict_corr, error_dict_block, var_dict = read_data(3,
545     ↪ int(T))
546     facts['samples'].append(N)
547     facts['its'].append(N_its[Ts.index(T)])
548     facts['acceptance'].append(N / N_its[Ts.index(T)])
549
550     # Statistical inefficiencies from autocorrelation
551     s_U_corr = error_dict_corr['U']
552     s_CV_corr = error_dict_corr['C_V']
553     s_P_corr = error_dict_corr['P']
554     s_r_corr = error_dict_corr['r']
555
556     # Statistical inefficiencies from block averaging
557     s_U_block = error_dict_block['U']
558     s_CV_block = error_dict_block['C_V']
559     s_P_block = error_dict_block['P']
560     s_r_block = error_dict_block['r']
561
562     # Calculate the errors from the statistical inefficiency
563     error_U_corr = np.sqrt(var_dict['U'] * s_U_corr / N)
564     error_CV_corr = np.sqrt(var_dict['C_V'] * s_CV_corr / N)
565     error_P_corr = np.sqrt(var_dict['P'] * s_P_corr / N)
566     error_r_corr = np.sqrt(var_dict['r'] * s_r_corr / N)
567
568     error_U_block = np.sqrt(var_dict['U'] * s_U_block / N)
569     error_CV_block = np.sqrt(var_dict['C_V'] * s_CV_block / N)
570     error_P_block = np.sqrt(var_dict['P'] * s_P_block / N)
571     error_r_block = np.sqrt(var_dict['r'] * s_r_block / N)
572
573     # Append the errors to the dictionaries
574     errors_corr['U'].append(f'{error_U_corr:.2e}')
575     errors_corr['C_V'].append(f'{error_CV_corr:.2e}')
576     errors_corr['P'].append(f'{error_P_corr:.2e}')
577     errors_corr['r'].append(f'{error_r_corr:.2e}')

```

```

576         errors_block['U'].append(f'{error_U_block:.2e}')
577         errors_block['C_V'].append(f'{error_CV_block:.2e}')
578         errors_block['P'].append(f'{error_P_block:.2e}')
579         errors_block['r'].append(f'{error_r_block:.2e}')
580
581     return errors_corr, errors_block, facts
582
583
584 def save_fig(fig, name, task):
585     fig.savefig(f'figs/task_{task}/{name}.pdf', bbox_inches='tight')
586
587 ##### Task 1 #####
588
589 # Constants
590 E_cucu = -0.436 # Bond energy for Cu-Cu [eV]
591 E_znzn = -0.113 # Bond energy for Zn-Zn [eV]
592 E_cuzn = -0.294 # Bond energy for Cu-Zn [eV]
593 delta_E = E_cucu + E_znzn - 2*E_cuzn
594
595 # Critical temperature [K]
596 T_c = 2*delta_E/k_B
597 print(f'T_c = {T_c}')
598
599 # Read data
600 data = np.loadtxt('data/task_1/data.csv', delimiter=',')
601
602 # Plot data
603 plot_task_1('P', T_c, save=False)
604 plot_task_1('UC', T_c, save=False)
605
606 ##### Task 2 #####
607
608 # Constants
609 its_eq_400 = 250000
610 its_eq = 100000
611 its = 1000000
612
613 # Plot equilibrium data
614 # task_2_analyzer(400, its_eq_400, its, changed=True, plot=True, save=False)
615 analyzer_task_2([400, 600, 1000], [its_eq_400, its_eq, its_eq], its=None, changed=False,
616 ↪ plot=True, save=True)
617
618 # Display MC data
619 U_400, accept_400_eq, accept_400 = analyzer_task_2(400, its_eq_400, its, changed=False,
620 ↪ plot=False, save=False)
621 U_600, accept_600_eq, accept_600 = analyzer_task_2(600, its_eq, its, changed=False,
622 ↪ plot=False, save=False)
623 U_1000, accept_1000_eq, accept_1000 = analyzer_task_2(1000, its_eq, its, changed=False,
624 ↪ plot=False, save=False)
625
626 df = pd.DataFrame({'T$': [400, 600, 1000], '$E_{pot}$': [np.mean(U_400), np.mean(U_600),
627 ↪ np.mean(U_1000)],
628 '$N_{eq}$': [its_eq_400, its_eq, its_eq], '$\\text{Acceptance}_{eq}$':
629 ↪ [accept_400_eq, accept_600_eq, accept_1000_eq],
630 '$\\text{Acceptance}$': [accept_400, accept_600, accept_1000]})
631
632 markdown = df.to_markdown(index=False)
633 display(Markdown(markdown))

```



```

627 ##### Task 3 #####
628
629
630 save = False
631
632 # Plot the data for task 3
633 plot_task_3('UC', save=save)
634 plot_task_3('Pr', save=save, error_plot=True, save_error_plot=False)
635 # plot_task_3('C_V', save=save)
636 # plot_task_3('P', save=save)
637 # plot_task_3('r', save=save)
638
639 # Constants
640 Ts = [300, 325, 350, 375, 400, 425, 450, 475, 500, 525, 550, 575, 600, 625, 650, 675,
641       700, 725, 750, 775, 800, 825, 850, 875, 900, 925, 950, 975, 1000] # Temperatures
642 eqs = [250000] * 12 + [100000] * 17 # Equilibrium iterations
643 errors_corr, errors_block, facts = get_errors(Ts) # Get the errors
644 # Ts.append('$\\textbf{Mean}$')
645
646 # Display the number of samples, iterations and acceptance ratio for each temperature
647 df = pd.DataFrame({'T$: Ts, '$N_{\\text{samples}}$: facts['samples'], '$N$:
    ↳ facts['its'], '$N_{\\text{eq}}$: eqs,
    ↳ '$\\text{Acceptance}\\hspace{0.2cm}N_{\\text{samples}} / N$: facts['acceptance']})
648 md_table = df.to_markdown(index=False)
649 display(Markdown('### Errors \\n' + md_table))
650
651 # errors_corr['U'].append(f'$\\textbf{{{np.mean(np.array(errors_corr["U"],
    ↳ dtype=np.float64)):.2e}}}$')
652 # errors_corr['C_V'].append(f'$\\textbf{{{np.mean(np.array(errors_corr["C_V"],
    ↳ dtype=np.float64)):.2e}}}$')
653 # errors_corr['P'].append(f'$\\textbf{{{np.mean(np.array(errors_corr["P"],
    ↳ dtype=np.float64)):.2e}}}$')
654 # errors_corr['r'].append(f'$\\textbf{{{np.mean(np.array(errors_corr["r"],
    ↳ dtype=np.float64)):.2e}}}$')
655 # facts['samples'].append(f'$\\textbf{{{np.mean(np.array(facts["samples"],
    ↳ dtype=np.float64)):.2e}}}$')
656 # facts['its'].append(f'$\\textbf{{{np.mean(np.array(facts["its"],
    ↳ dtype=np.float64)):.2e}}}$')
657 # facts['acceptance'].append(f'$\\textbf{{{100 * np.mean(np.array(facts["acceptance"],
    ↳ dtype=np.float64)):.2f}}}$ %')
658 # df_corr = pd.DataFrame({'T$: Ts, '$\\epsilon\\hspace{0.1cm}(U)$': errors_corr['U'],
    ↳ '$\\epsilon\\hspace{0.1cm}(C_V)$': errors_corr['C_V'],
    ↳ '$\\epsilon\\hspace{0.1cm}(P)$': errors_corr['P'], '$\\epsilon\\hspace{0.1cm}(r)$':
    ↳ errors_corr['r'], '$N$: facts['samples'], '$N_{\\text{its}}$: facts['its'],
    ↳ '$\\text{Acceptance}$': facts['acceptance']})
659 # df_corr['$N_{\\text{its}}$'][:-1] = df_corr['$N_{\\text{its}}$'][:-1].apply(lambda x:
    ↳ f'{x:.0f}')
660 # df_corr['$\\text{Acceptance}$'][:-1] =
    ↳ df_corr['$\\text{Acceptance}$'][:-1].apply(lambda x: f'{100*x:.2f} %')
661 # md_table_corr = df_corr.to_markdown(index=False)
662 # display(Markdown('### Autocorrelation Errors \\n' + md_table_corr))
663
664 # errors_block['U'].append(f'$\\textbf{{{np.mean(np.array(errors_block["U"],
    ↳ dtype=np.float64)):.2e}}}$')
665 # errors_block['C_V'].append(f'$\\textbf{{{np.mean(np.array(errors_block["C_V"],
    ↳ dtype=np.float64)):.2e}}}$')
666 # errors_block['P'].append(f'$\\textbf{{{np.mean(np.array(errors_block["P"],
    ↳ dtype=np.float64)):.2e}}}$')

```



```

667 # errors_block['r'].append(f'$\\textbf{{{np.mean(np.array(errors_block["r"],
↪ dtype=np.float64)):.2e}}}$')
668 # df_block = pd.DataFrame({'$T$: Ts, '$\\epsilon\\hspace{0.1cm}(U)$': errors_block['U'],
↪ '$\\epsilon\\hspace{0.1cm}(C_V)$': errors_block['C_V'],
↪ '$\\epsilon\\hspace{0.1cm}(P)$': errors_block['P'], '$\\epsilon\\hspace{0.1cm}(r)$':
↪ errors_block['r'], '$N$: facts['samples'], '$N_{\\text{its}}$: facts['its'],
↪ '$\\text{Acceptance}$': facts['acceptance']})
669 # df_block['$N_{\\text{its}}$'][: -1] = df_block['$N_{\\text{its}}$'][: -1].apply(lambda x:
↪ f'{x:.0f}')
670 # df_block['$\\text{Acceptance}$'][: -1] =
↪ df_block['$\\text{Acceptance}$'][: -1].apply(lambda x: f'{100*x:.2f} %')
671 # md_table_block = df_block.to_markdown(index=False)
672 # display(Markdown('### Blocking Errors \n' + md_table_block))
673
674 # Constants
675 T = 1000
676 save = True
677
678 # Read data
679 N, autocorr, block, error_dict_corr, error_dict_block, var_dict = read_data(3, T)
680 print(N)
681 s_corr_U, s_corr_CV = error_dict_corr['U'], error_dict_corr['C_V']
682 s_corr_P, s_corr_r = error_dict_corr['P'], error_dict_corr['r']
683
684 # Plot the error data
685 plot_task_3_error(T, autocorr, block, s_corr_U, s_corr_P, s_corr_CV, save=save,
↪ xlim_factor=0.25, xlim_factor_CVP=0.5, P_sep=True)
686

```