Homework 2:
Monte Carlo & Ising Model

Jonatan Haraldsson
jonhara@chalmers.se

Oscar Stommendal
oscarsto@chalmers.se

Task Nº	Points	Avail. points
Σ		

December 28th 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 ($Cu_{0.5}Zn_{0.5}$) 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 $Cu_{0.5}Zn_{0.5}$, which for Zn 50 at% corresponds to the β' phase for $T \lesssim 468\,^{\circ}\mathrm{C} = 741\,\mathrm{K}$ and the β phase for $T \gtrsim 741\,\mathrm{K}$. Crystalographically, the β' phase is in an ordered BCC lattice, while β' phase is in a disordered state [5].

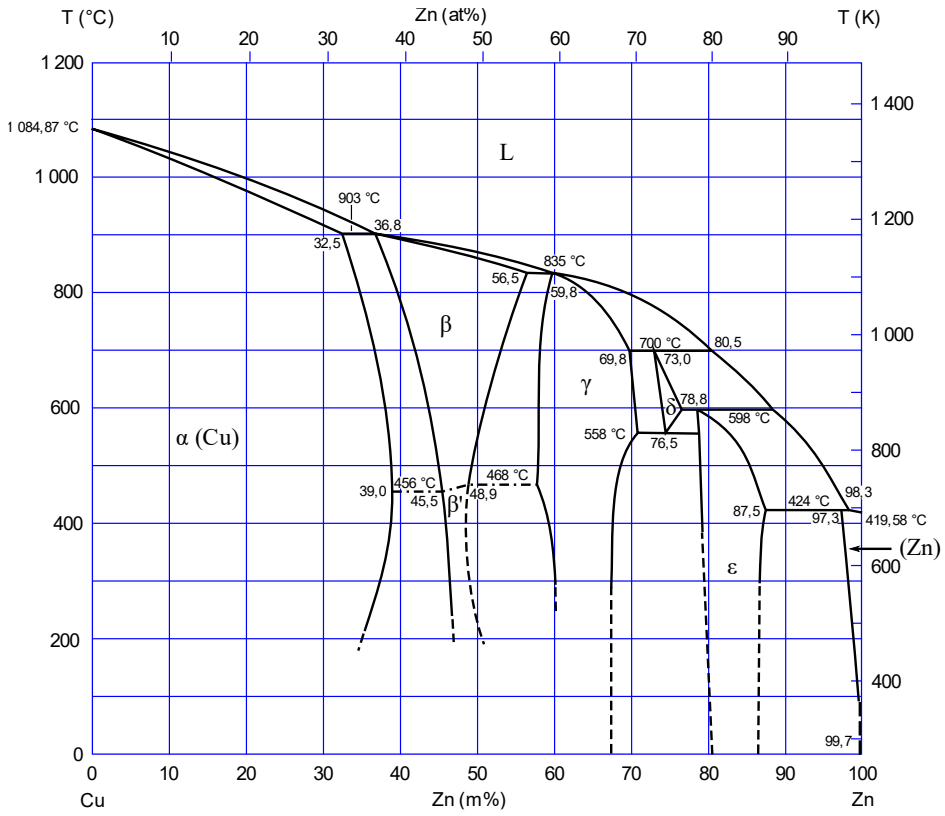

Figure 1: CuZn phase diagram with both Zn in mass % and at%. With the current set-up, $Cu_{0.5}Zn_{0.5}$, the system is in the β' and β phases depending on temperature. Diagram is taken from Wikipedia Commons.

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 $T = 0\,\mathrm{K}$, the Cu and Zn atoms are aligned perfectly in a BCC lattice, consisting of two simple cubic sublattices (a_{sub} and b_{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

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

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 $T_c = 2\Delta E/k_B = 905\,\mathrm{K} = 632\,^{\circ}\mathrm{C}$, 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 $T \approx 200\,\mathrm{K}$ to complete disorder at $T = T_c$. This is an expected behavior, since the β' phase, at $T < T_c$, is characterized by complete order, while the β phase, at $T > T_c$, instead is completely disordered.

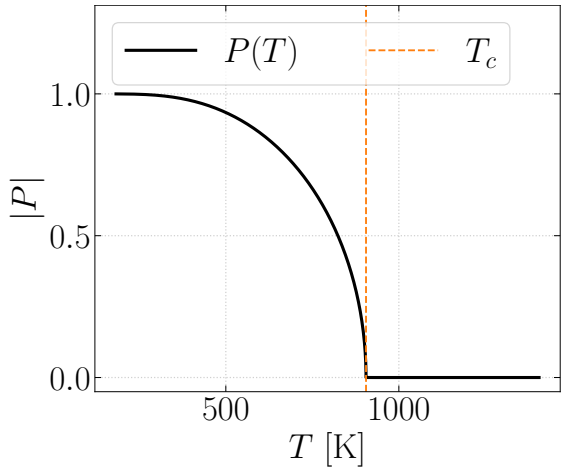

Figure 2: Mean-field solution for $P(T)$, using the relation in Equation 1. In addition, the orange line denotes T_c.

From Equation 1, a relation for $U(T)$ of the $N_{atoms} = 2\,000$ atoms was found to be

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

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

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

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 $U(T) \equiv 2E_0 = 2.274\,\mathrm{keV}\ \forall\ T > T_c$. 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.

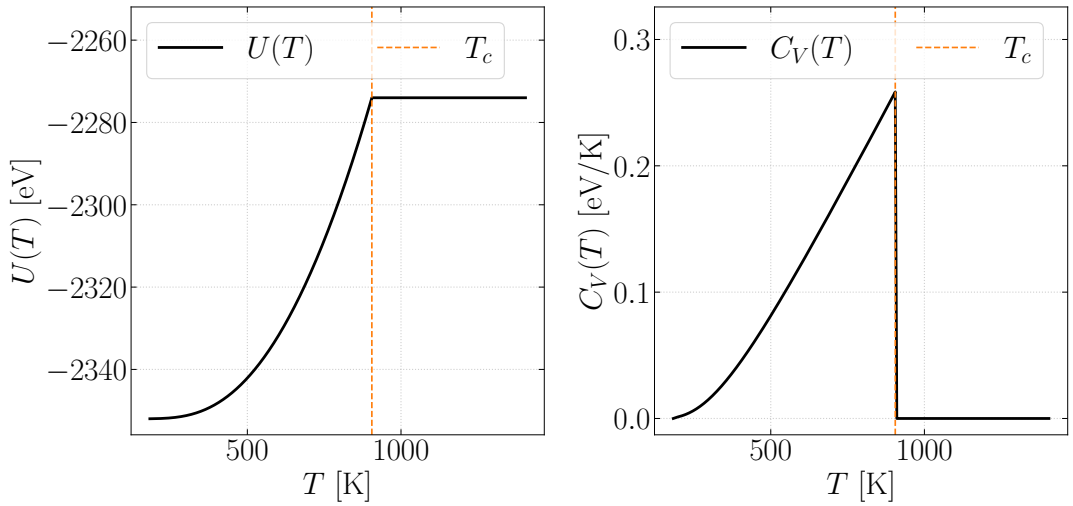

Figure 3: Mean-field solution for $U(T)$, using the relation in Equation 2 and heat capacity $C_V(T)$, retrieved using Equation 3. The orange line denotes the critical temperature T_c.

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_{sub} and all Zn atoms occupy b_{sub}, where each sublattice has 1 000 atoms ($N_{atoms} = 2\,000$). This set-up ensures complete order, i.e. $|P| = 1$. Thus, the initial energy was set to

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

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_{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 $a_0 = 2.969\,\text{Å}$ [6].

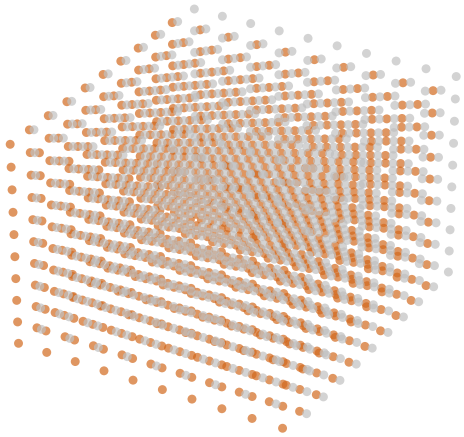

Figure 4: Three-dimensional plot of all 1 000 Cu (brown) and 1 000 Zn (silver) atoms in the initialized lattice.

To monitor the different binding energies between Cu–Cu, Zn–Zn or Cu–Zn as the atoms are interchanged, a matrix ($\boldsymbol{M_N}$) was created. Each row i in $\boldsymbol{M}$ corresponds to an atom position ($i \leq 1000$ for a_{sub} and $i > 1000$ for b_{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_{sub} are all Cu in a_{sub} and vice versa, as illustrated in Figure 5. Furthermore, an additional array, $\boldsymbol{A}$, 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 $\boldsymbol{A}$ a (2000×1) array. Note that index i here directly corresponds to the row index of the $\boldsymbol{M_N}$ matrix. For the atom at site x, this enabled us to efficiently

a) check which atom type occupied a certain lattice position, by calling $\boldsymbol{A}[x]$, and

b) retrieve the nearest neighbors and their atom type through applying the indices provided by $\boldsymbol{M_N}[x]$ to $\boldsymbol{A}$.

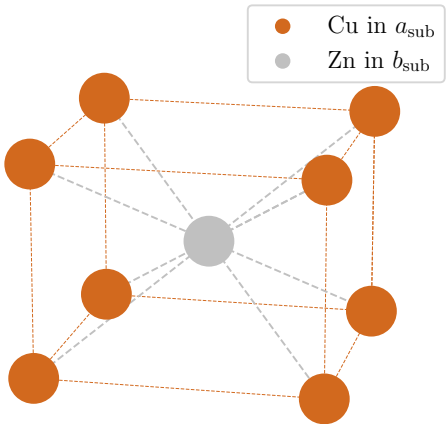

Figure 5: In a cold, perfectly ordered starting state, the nearest neighbors to the Zn atom in the center (b_{sub}) are all Cu atoms (a_{sub}).

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

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

Code Snippet Task 2, swappy

```
idx swappy(int *N, gsl_rng *r)
{
    idx index;
    int idx_1 = (int)(1999 * gsl_rng_uniform(r)); int A = N[idx_1];
    int idx_2 = (int)(1999 * gsl_rng_uniform(r)); int B = N[idx_2];
    while (A == B)
    {
        idx_2 = (int)(1999 * gsl_rng_uniform(r));
        B = N[idx_2];
    }
    index.alpha = idx_1; index.beta = idx_2;
    index.valueA = A; index.valueB = B;
    return index;
}
```

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 $\boldsymbol{M_N}$ 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 $\boldsymbol{M_N}$. 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 $1000\,\mathrm{K}$. 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 $1000\,\mathrm{K}$, the system was considered equilibrated after $\sim 100\,000$ iterations. In the case of $T = 400\,\mathrm{K}$, 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 α, as it is scaled with temperature according to Equation 5. Put differently, an increased temperature promotes more atom swaps, and thus faster equilibration times.

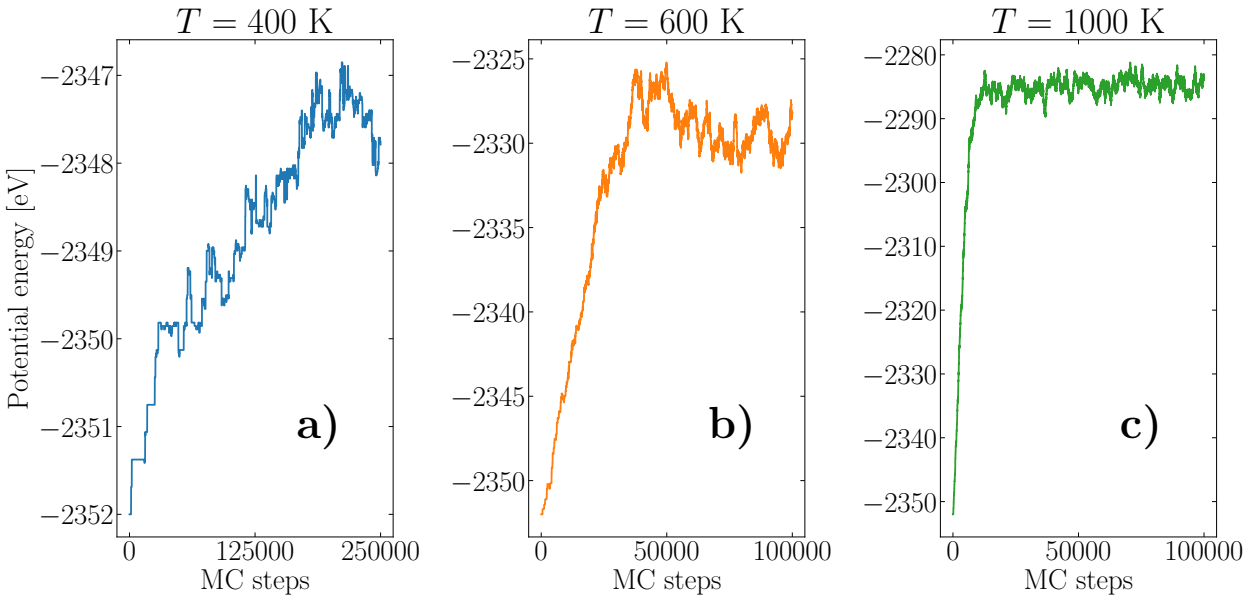

Figure 6: The potential energy at each iteration during the MC simulation for $T =$ a) 300, b) 600 and c) $1000\,\mathrm{K}$.

Additionally, Table 1 shows the number of *burn-in* iterations along with the resulting energy and acceptance ratio Γ for the three cases $T = 400, 600$ and $1000\,\mathrm{K}$. As expected, a temperature closer to 0 has a lower acceptance ratio and hence, a lower energy since we initialize the lattice at $0\,\mathrm{K}$ (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 Γ**
400	2.5×10^5	– 2348	0.020
600	1×10^5	– 2327	0.130
1 000	1×10^5	– 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 $T \in [300\,\mathrm{K}, 1000\,\mathrm{K}]$, with step-size $\Delta T = 25\,\mathrm{K}$. 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 $T < 600\,\mathrm{K}$ 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),

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

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

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

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

where $N_{a,\mathrm{Cu}}$ is the number of Cu atoms in a_{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 Φ_k,

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

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

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

where σ_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 $T = 750\,\text{K}$; 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 $T = 750\,\text{K}$ in C_V aligns with the inflection point in $U(T)$.

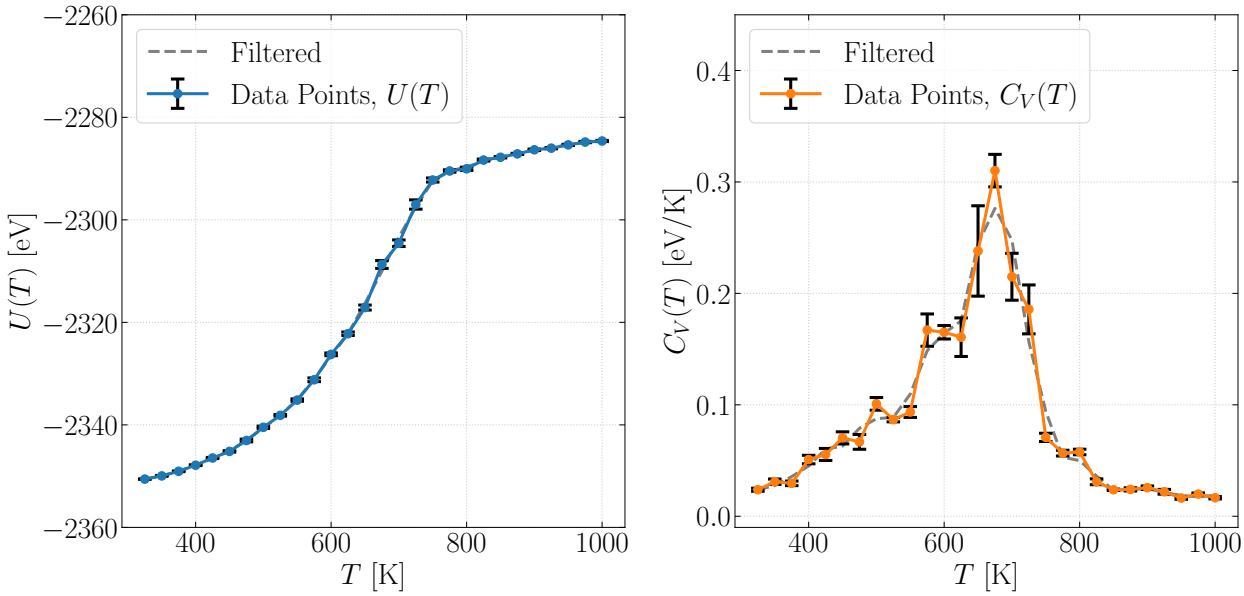

Figure 7: Sampled data points for $U(T)$ (left) and $C_V(T)$ (right) along with a filtered curve. The error bars denote the mean error using both autocorrelation and block averaging for every sampled T.

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 β phase) at $\sim 750\,\mathrm{K}$, which in more in-line with the face diagram in Figure 1. Similar to $P(T)$, $r(T)$ also reaches its minimum close to $\sim 750\,\mathrm{K}$, 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 indicate that there is some local order in the system.

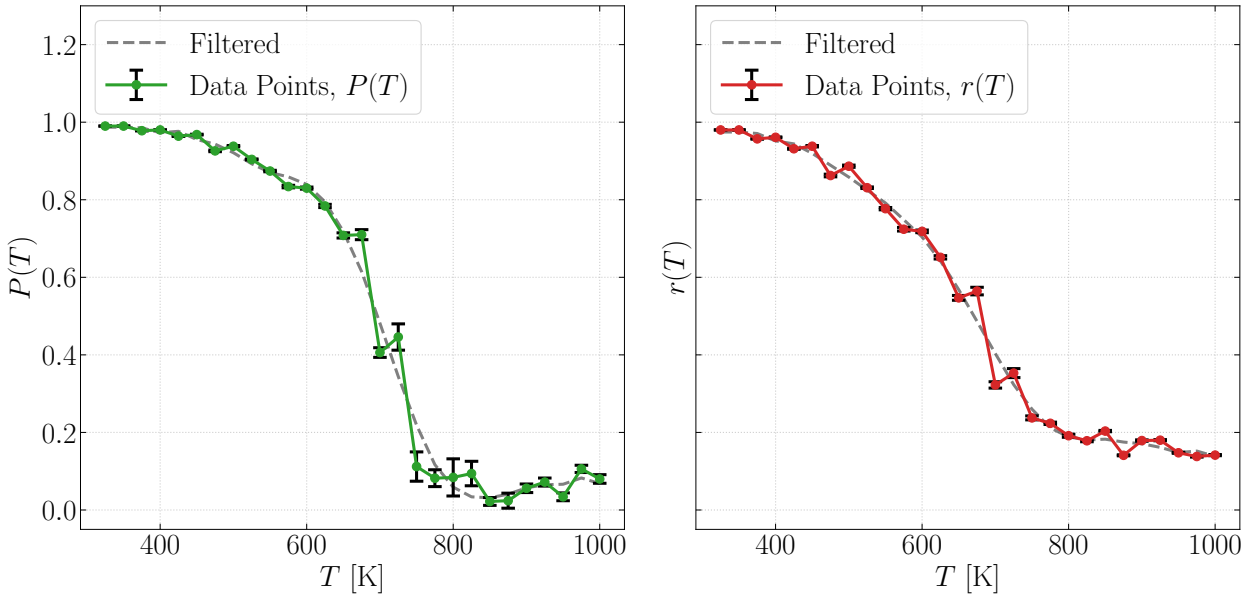

Figure 8: Sampled data points for $P(T)$ (left) and $r(T)$ (right) along with a filtered curve. The error bars denote the mean error using both autocorrelation and block averaging for every sampled T.

Regarding the error estimation, Figure 9 below shows the calculated errors ϵ for all quantities as a function of temperature. Each ϵ 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 ($T \approx 741\,\mathrm{K}$) 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.

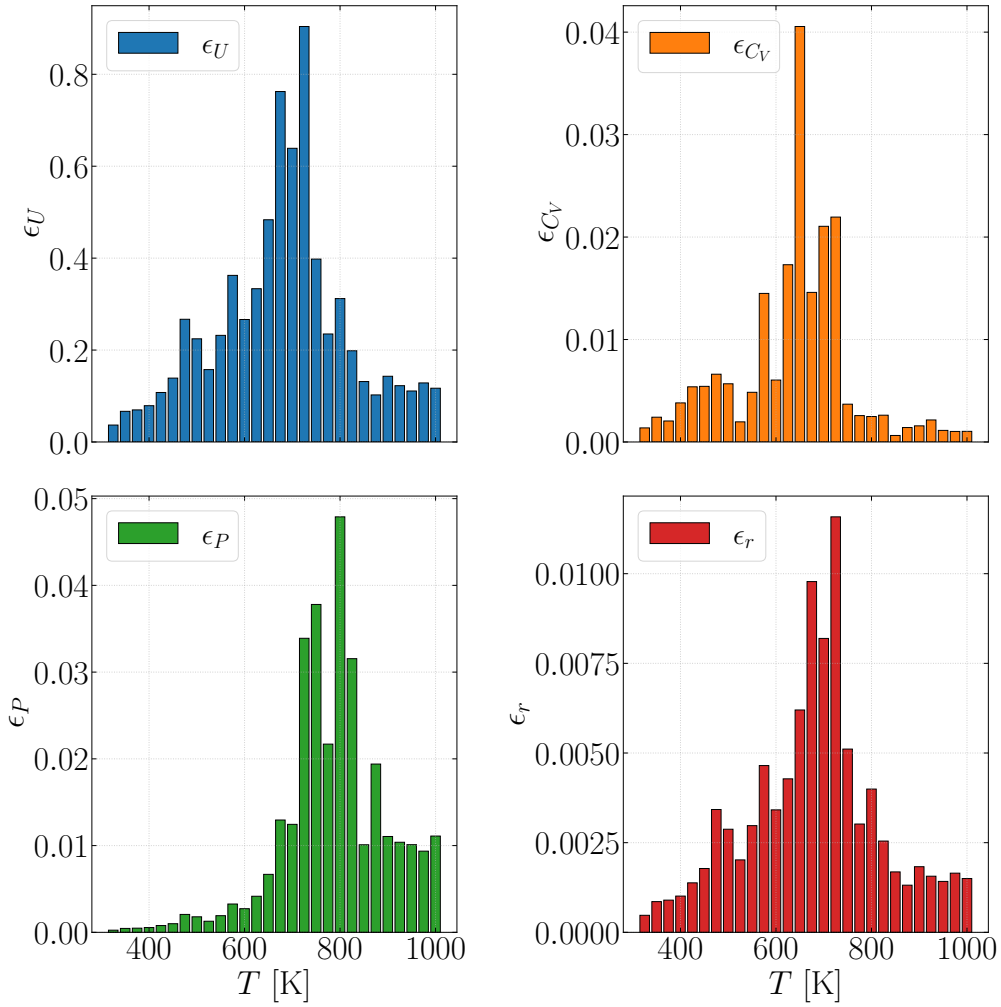

Figure 9: The error ϵ for the internal energy U, heat capacity C_V and long- and short-range order parameter P and r as a function of temperature.

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 $T = 450\,\mathrm{K}$. From the autocorrelation function, the values of s for the different quantities have been displayed as the intersection between the respective Φ_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 a exponential behavior of Φ_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.

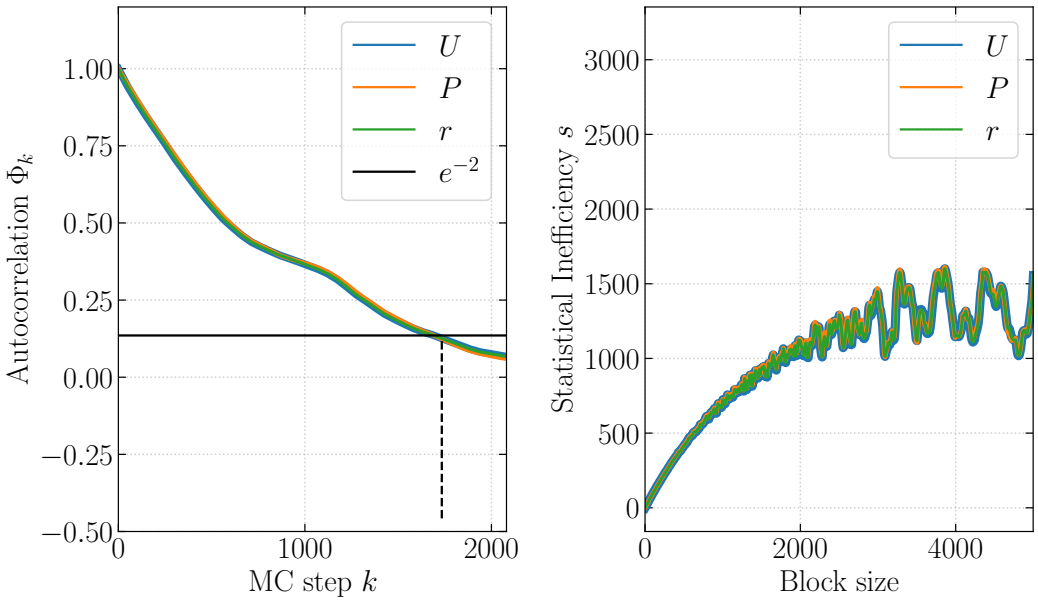

Figure 10: The autocorrelation function together with the statistical inefficiency from the block average method for the internal energy U and long- and short-range order parameters at $T = 450\,\mathrm{K}$. The dashed line in the left plot indicate the index of Φ_k where this is closest to the value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

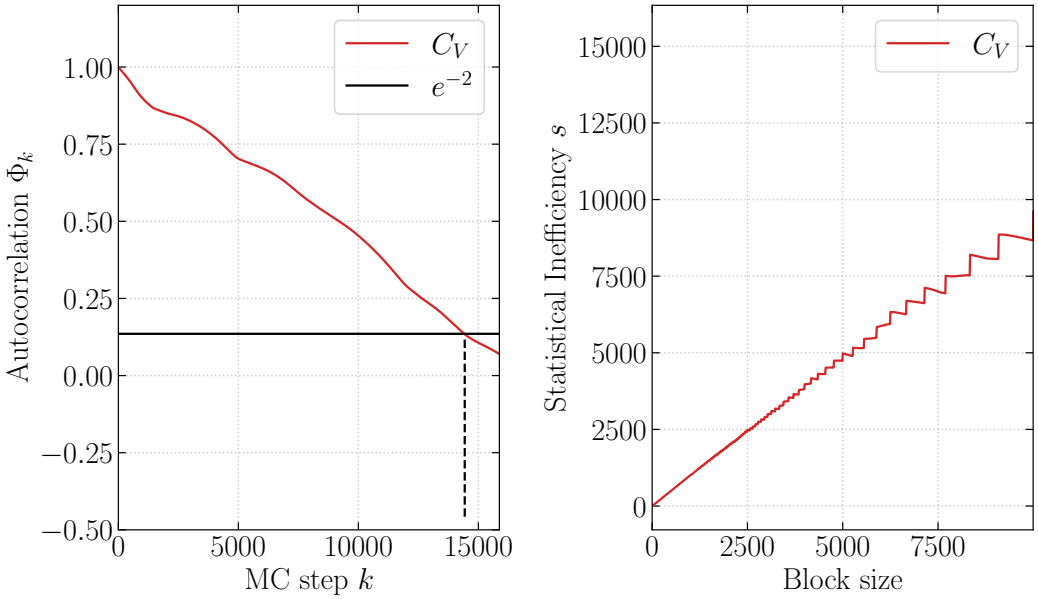

Figure 11: The autocorrelation function together with the statistical inefficiency from the block average method for the heat capacity C_V at $T = 450\,\mathrm{K}$. The dashed line in the left plot indicate the index of Φ_k where this is closest to the value of e^{-2}, i.e. the estimated value of s using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

Another interesting result found for $T > T_c$, was that Φ_k and the block averages for P tended to diverge from the corresponding values for U and r. To clarify, compare Φ_k and the block averaging for three quantities at $T = 450\,\mathrm{K}$ in Figure 10 to the same curves at $T = 1000\,\mathrm{K}$ 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.

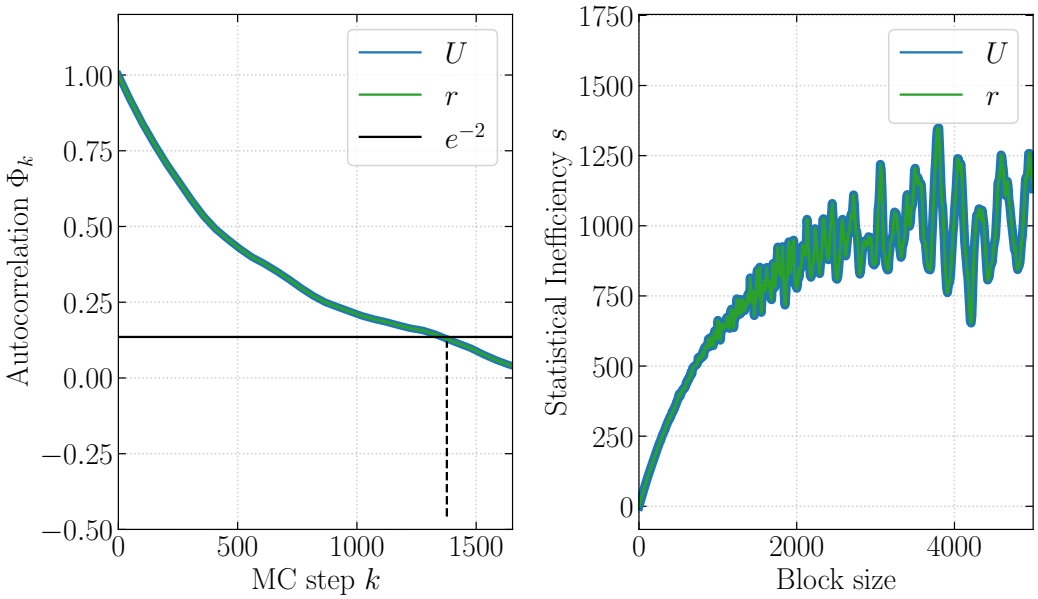

Figure 12: The autocorrelation function together with the statistical inefficiency from the block average method for the internal energy U and short-range order parameter at $T = 1000\,\mathrm{K}$. The dashed line in the left plot indicate the index of Φ_k where this is closest to the value of e^{-2},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

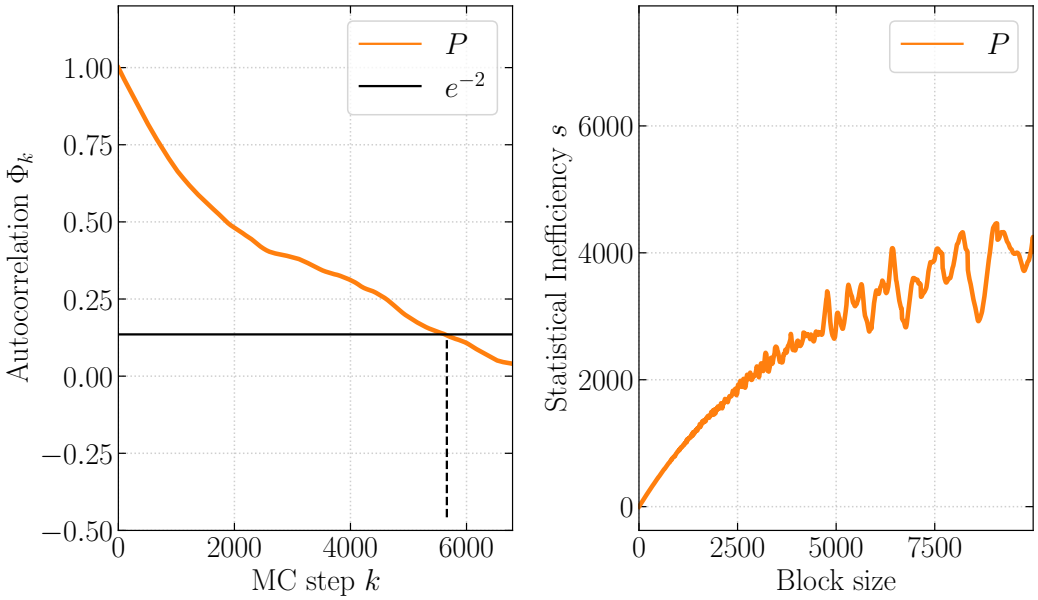

Figure 13: The autocorrelation function together with the statistical inefficiency from the block average method for the long-range order parameter P at $T = 1000\,\mathrm{K}$. The dashed line in the left plot indicate the index of Φ_k where this is closest to the value of e^{-2}, i.e. the estimated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

Conclusion

In summary, we have simulated a brass lattice consisting of 50% Cu and Zn, initialized in the β, 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 editted: 01-12-2024. (), [Online]. Available: 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.

[4] B. Wikipedia. Last editted: 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/. 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. 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

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

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

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

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

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

int
run(
    int argc,
    char *argv[]
   )
{
    // ------------------------------ Constants -------------------------------- //

    double E_cucu = -0.436; // Bonding energy Cu-Cu [eV]
    double E_znzn = -0.113; // Bonding energy Zn-Zn [eV]
    double E_cuzn = -0.294; // Bonding energy Cu-Zn [eV]
    double k_B = 8.617333262145e-5; // Boltzmann constant [eV/K]
    double a = 2.949; // Lattice parameter [Å]
    double closest_distance_bcc = sqrt(3) * a / 2; // Closest distance between atoms in
    ↪  BCC [Å]
    int L = 10; // Number of unit cells in each direction
    int N_atoms = 2 * L * L * L; // Number of atoms
    double E_initial = N_atoms / 2 * E_cuzn * 8; // Initial energy
    char filename[100];
    gsl_rng *r = init_gsl_rng(19);

    // ------------------------------ Task 1 -------------------------------- //

    double step_size = 0.0001; // Step size for P
    int N = (int)(1 / step_size) + 100; // Number of points (add 100 for the end, where P
    ↪  = 0)
    FILE *file = fopen("data/task_1/data.csv", "w");
    double delta_E = E_cucu + E_znzn - 2 * E_cuzn;
    double *Us = (double *)malloc(N * sizeof(double)); // Energy vector
    double *Ts = (double *)malloc(N * sizeof(double)); // Temperature vector
    double *Ps = (double *)malloc((N + 1) * sizeof(double)); // Long-range order vector
    double C_V;
    Ps[0] = 1.; // Start at P = 1
    // Loop over P and calculate T, U and C_V
    for (int i = 1; i < N; i++)
    {
        if (i < (int)(1 / step_size))
        {
            Ps[i] = Ps[i-1] - step_size; // Update P
            Ts[i-1] = 4 * Ps[i] * delta_E / k_B / log((1 + Ps[i]) / (1 - Ps[i])); //
            ↪  Calculate T
            Us[i-1] = N_atoms * (E_cucu + E_znzn + 2 * E_cuzn) - N_atoms * Ps[i] * Ps[i]
            ↪  * delta_E; // Calculate U
            if (i >= 2)
            {
                double dU = Us[i-1] - Us[i-2]; // dU
```

```
                double dT = Ts[i-1] - Ts[i-2]; // dT
                C_V = dU / dT; // Calculate C_V
            }
            else
            {
                C_V = 0.; // Set C_V to 0 for the first iteration
            }
            fprintf(file, "%f, %f, %f, %f\n", Ps[i], Ts[i-1], Us[i-1], C_V);
        }
        // When P = 0, set P = 0 and continue to calculate T, U and C_V
        else if (i >= (int)(1 / step_size))
        {
            Ps[i] = 0.;
            Ts[i-1] = Ts[i-2] + 5; // Use a step size of 5 K
            Us[i-1] = N_atoms * (E_cucu + E_znzn + 2 * E_cuzn) - N_atoms * Ps[i] * Ps[i]
            ↪  * delta_E;
            double C_V = (Us[i-1] - Us[i-2]) / (Ts[i-1] - Ts[i-2]);
            fprintf(file, "%f, %f, %f, %f\n", Ps[i], Ts[i-1], Us[i-1], C_V);
        }
    }

    // Free memory and close files
    fclose(file);
    free(Us);
    free(Ts);
    free(Ps);

    // ---------------------------------- Task 2 ---------------------------------- //

    int its_eq = 250000; // Number of iterations for equilibrium
    double T = 400; // Temperature [K]
    metro metro_result;

    double **sub_A = create_2D_array(N_atoms / 2, 3); // Sublattice A
    double **sub_B = create_2D_array(N_atoms / 2, 3); // Sublattice B
    int **neighbors = (int **)create_2D_array(N_atoms, 8); // Neighbors matrix
    int *atoms = (int *)calloc(N_atoms, sizeof(int)); // Atoms vector
    // Initialize all atoms in sublattice A to be Cu (1) (= Cold start)
    for (int i = 0; i < N_atoms / 2; i++)
    {
        atoms[i] = 1;
    }

    init_sc(sub_A, L, a, (double[3]){0, 0, 0}); // Initialize sublattice A
    init_sc(sub_B, L, a, (double[3]){0.5, 0.5, 0.5 }); // Initialize sublattice B
    nearest_neighbors_bcc(sub_A, sub_B, N_atoms, neighbors, 10 * a, 0.001,
    ↪  closest_distance_bcc); // Find nearest neighbors

    // File for equilibrium
    sprintf(filename, "data/task_2/equilibrium_%i_%.0f.csv", its_eq, T);
    FILE *fp_eq = fopen(filename, "w");
    fprintf(fp_eq, "accepted, E_tot\n");

    // Run metropolis for equilibrium
    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);
    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
        ↪  equilibrium\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,
        ↪  E_tot, r, NULL, NULL, NULL, NULL, N_atoms, fp);
124     // Calculate lattice properties
125     lattice_to_files(fp_atoms, fp_neighbors, atoms, neighbors, N_atoms);
126 
127     // Free memory and close files
128     destroy_2D_array(sub_A);
129     destroy_2D_array(sub_B);
130     destroy_2D_array((double **)neighbors);
131     free(atoms);
132     fclose(fp);
133     fclose(fp_eq);
134     fclose(fp_atoms);
135     gsl_rng_free(r);
136 
137     // ---------------------------------- Task 3 ---------------------------------- //
138 
139     double T_start = 300; // T value at MC simulation start [K]
140     double T_end = 1000; // T value at MC simulation end [K]
141     int dt = 25; // T step size [K]
142     metro metro_result;
143     metro metro_result_eq;
144     atom_count lat_props;
145 
146     // File for data
147     sprintf(filename, "data/task_3/data.csv");
148     FILE *fp_data = fopen(filename, "w");
149     fprintf(fp_data, "T, U, C_V, P, r, Accepted ratio, Accepted ratio equilibrium,
        ↪  Iterations\n");
150 
151     int its_eq;
152     // Loop over T values
153     for (double T = T_start; T < T_end+1; T+=dt)
154     {
155         printf("T: %f\n", T); // Print T value, for debugging
156         // Let the number of iterations for equilibrium depend on T
157         if (T < 600)
```

```
        {
            its_eq = 250000;
        }
        else
        {
            its_eq = 100000;
        }
        double **sub_A = create_2D_array(N_atoms / 2, 3); // Sublattice A
        double **sub_B = create_2D_array(N_atoms / 2, 3); // Sublattice B
        int **neighbors = (int **)create_2D_array(N_atoms, 8); // Neighbors matrix
        int *atoms = (int *)calloc(N_atoms, sizeof(int)); // Atoms vector
        // Initialize all atoms in sublattice A to be Cu (1) (= Cold start)
        for (int i = 0; i < N_atoms / 2; i++)
        {
            atoms[i] = 1;
        }

        init_sc(sub_A, L, a, (double[3]){0, 0, 0}); // Initialize sublattice A
        init_sc(sub_B, L, a, (double[3]){0.5, 0.5, 0.5}); // Initialize sublattice B
        nearest_neighbors_bcc(sub_A, sub_B, N_atoms, neighbors, 10 * a, 0.001,
        ↪  closest_distance_bcc); // Find nearest neighbors

        // Run metropolis for equilibrium
        metro_result_eq = metropolis(its_eq, atoms, neighbors, k_B, T, E_cucu, E_znzn,
        ↪  E_cuzn, E_initial, r, NULL, NULL, NULL, NULL, N_atoms, NULL);

        int its = 100000; // Number of accepted MC iterations
        double *U = (double *)calloc(its, sizeof(double)); // Energy vector
        double *C_V = (double *)calloc(its, sizeof(double)); // Heat capacity vector
        double *P = (double *)calloc(its, sizeof(double)); // Long-range order vector
        double *R = (double *)calloc(its, sizeof(double)); // Short-range order vector

        // Run metropolis for MC iterations
        metro_result = metropolis(its, atoms, neighbors, k_B, T, E_cucu, E_znzn, E_cuzn,
        ↪  metro_result_eq.Etot, r, U, C_V, P, R, N_atoms, NULL);
        lat_props = lattice_props(atoms, neighbors, N_atoms); // Calculate lattice
        ↪  properties, e.g., N_Cu_A and N_CuZn
        its = metro_result.its; // Number of iterations

        double U_avg = average(U, metro_result.accepted); // Average energy
        double C_V_inst = variance(U, metro_result.accepted) / k_B / T / T; //
        ↪  Instantaneous heat capacity
        double p = (2. * lat_props.N_Cu_A / (N_atoms / 2) - 1); // Long-range order
        ↪  parameter
        double Rr = (lat_props.N_CuZn - 4. * N_atoms / 2) / (4 * N_atoms / 2); //
        ↪  Short-range order parameter
        fprintf(fp_data, "%f, %f, %f, %f, %f, %f, %f, %i\n", T, U_avg, C_V_inst, p, Rr,
        ↪  (double)metro_result.accepted / its, (double)metro_result_eq.accepted /
        ↪  its_eq, its);

        // Files for autocorrelation and blocking
        sprintf(filename, "data/task_3/auto_corr_%i.csv", (int)T);
        FILE *fp_auto_corr = fopen(filename, "w");
        fprintf(fp_auto_corr, "N, Var_U, Var_CV, Var_P, Var_R\n");
        fprintf(fp_auto_corr, "%i, %f, %f, %f, %f\n", metro_result.accepted, variance(U,
        ↪  metro_result.accepted), variance(C_V, metro_result.accepted), variance(P,
        ↪  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,
                ↪  metro_result.accepted, b), block_average(C_V, metro_result.accepted, b),
                ↪  block_average(P, metro_result.accepted, b), block_average(R,
                ↪  metro_result.accepted, b));
214         }
215         // Loop over lags
216         for (int i = 0; i < metro_result.accepted*0.5;
            ↪  i+=(metro_result.accepted*0.5/1000+1))
217         {
218             // Subtract the average value from the data
219             addition_with_constant(U, U, -average(U, metro_result.accepted),
                ↪  metro_result.accepted);
220             addition_with_constant(C_V, C_V, -average(C_V, metro_result.accepted),
                ↪  metro_result.accepted);
221             addition_with_constant(P, P, -average(P, metro_result.accepted),
                ↪  metro_result.accepted);
222             addition_with_constant(R, R, -average(R, metro_result.accepted),
                ↪  metro_result.accepted);
223             fprintf(fp_auto_corr, "%i, %f, %f, %f, %f\n", i, autocorrelation(U,
                ↪  metro_result.accepted, i), autocorrelation(C_V, metro_result.accepted,
                ↪  i), autocorrelation(P, metro_result.accepted, i), autocorrelation(R,
                ↪  metro_result.accepted, i));
224         }
225 
226         // Free memory and close files
227         free(U);
228         free(C_V);
229         free(P);
230         free(R);
231         destroy_2D_array(sub_A);
232         destroy_2D_array(sub_B);
233         destroy_2D_array((double **)neighbors);
234         free(atoms);
235     }
236 
237     // Free memory and close files
238     fclose(fp_data);
239     gsl_rng_free(r);
240 
241     return 0;
242 }
243 
244 gsl_rng *
245 init_gsl_rng(
246              int seed
247             )
248 {
249     const gsl_rng_type * T;
```

```
    gsl_rng * r;
    gsl_rng_env_setup();
    T = gsl_rng_default; // default random number generator
    r = gsl_rng_alloc(T); // allocate memory for the random number generator

    if (!r) {
        fprintf(stderr, "Error: Could not allocate memory for RNG.\n");
        exit(EXIT_FAILURE); // Exit if allocation fails
    }

    // Set the seed
    gsl_rng_set(r, seed);

    return r;
}

double
boundary_distance_between_vectors(
                                  double *v1,
                                  double *v2,
                                  int dim,
                                  double box_length
                                 )
{
    double r = 0.0;
    double delta;
    for (int d = 0; d < dim; d++)
    {
        delta = v1[d] - v2[d];
        // Apply boundary conditions
        delta -= round(delta / box_length) * box_length;
        r += delta * delta;
    }

    return sqrt(r);
}

void
nearest_neighbors_bcc(
                      double **pos_A,
                      double **pos_B,
                      int N_atoms,
                      int **neighbors,
                      double box_length,
                      double cutoff,
                      double closest_distance
                     )
{
    double r; // Distance between atoms
    // Loop over all atoms in sublattice A
    for (int i = 0; i < N_atoms / 2; i++)
    {
        int count = 0; // Counter for neighbors
        for (int j = 0; j < N_atoms / 2; j++)
        {
            r = boundary_distance_between_vectors(pos_A[i], pos_B[j], 3, box_length);
            // If the distance between atoms is within the cutoff distance, store the
```

```
            // index of the atom in sublattice B (index > 1000 in atoms vector)
            //
            if (fabs(closest_distance - r) < cutoff)
            {
                neighbors[i][count] = j + 1000;
                count++;
            }
        }
    }
    // Loop over all atoms in sublattice B
    for (int i = 0; i < N_atoms / 2; i++)
    {
        int count = 0; // Counter for neighbors
        for (int j = 0; j < N_atoms / 2; j++)
        {
            r = boundary_distance_between_vectors(pos_B[i], pos_A[j], 3, box_length);
            // If the distance between atoms is within the cutoff distance, store the
            // index of the atom in sublattice A (index < 1000 in atoms vector)
            if (fabs(closest_distance - r) < cutoff)
            {
                neighbors[i+1000][count] = j;
                count++;
            }
        }
    }
}

metro
metropolis(
           int its,
           int *atoms,
           int **neighbors,
           double k_B,
           double T,
           double E_cucu,
           double E_znzn,
           double E_cuzn,
           double E_tot,
           gsl_rng *r,
           double *U,
           double *C_V,
           double *P,
           double *R,
           int N_atoms,
           FILE *fp
          )
{
    metro metro_result;
    atom_count lat_props;
    idx index;
    double E; // Energy before swap
    double E_prime; // Energy after swap
    double delta_E; // Energy difference
    double alpha; // Acceptance probability
    int accepted = 0; // Number of accepted swaps
    int i = 0; // Number of iterations
    // Loop over the number of iterations (task 2) or until its swaps are accepted (task
    ↪  3)
```

```
    // for (int i = 0; i < its; i++)
    while (accepted < its)
    {
        index = swappy(atoms, r); // Get the indices and values of the atoms to swap
        int A = index.alpha; // Index of the atom in sub_A
        int B = index.beta; // Index of the atom in sub_B
        int value_A = index.valueA; // Value of the atom in sub_A
        int value_B = index.valueB; // Value of the atom in sub_B
        E = energy_bond(index, atoms, neighbors, E_cucu, E_znzn, E_cuzn); // Energy before swap
        atoms[A] = value_B; // Swap the atoms
        atoms[B] = value_A; // Swap the atoms
        E_prime = energy_bond(index, atoms, neighbors, E_cucu, E_znzn, E_cuzn); // Energy after swap
        delta_E = E_prime - E; // Energy difference
        alpha = exp(-delta_E / k_B / T); // Acceptance probability
        // If the swap is accepted, increment the number of accepted swaps and update the energy
        if (gsl_rng_uniform(r) < alpha)
        {
            accepted++;
            E_tot += delta_E;
            // Files for task 2
            if (fp != NULL)
            {
                fprintf(fp, "%i, %f\n", accepted, E_tot);
            }
            // For task 3, store the quantity values in the vectors (for autocorrelation and blocking)
            if (U != NULL)
            {
                U[accepted-1] = E_tot;
                double U_fluct = variance(U, accepted);
                C_V[accepted-1] = U_fluct / k_B / T / T;
                lat_props = lattice_props(atoms, neighbors, N_atoms);
                P[accepted-1] = (2. * lat_props.N_Cu_A / (N_atoms / 2) - 1);
                R[accepted-1] = (lat_props.N_CuZn - 4. * N_atoms / 2) / (4 * N_atoms / 2);
            }
        }
        else
        {
            // File for task 2
            if (fp != NULL)
            {
                fprintf(fp, "%i, %f\n", accepted, E_tot);
            }
            // If the swap is not accepted, swap the atoms back
            atoms[A] = value_A;
            atoms[B] = value_B;
        }
        i++; // Increment the number of iterations
    }
    // Store the values in the struct and return it
    metro_result.accepted = accepted;
    metro_result.Etot = E_tot;
    metro_result.its = i;
```

```
416
417     return metro_result;
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         {
```

```
            E += E_znzn;
        }
        else
        {
            E += E_cuzn;
        }
    }
    // Loop over the neighbors of atom B
    for (int i = 0; i < 8; i++)
    {
        // Add the energy of the bond between atom B and its neighbors
        if (atoms[neighbors_2[i]] == 1 && atom_2 == 1)
        {
            E += E_cucu;
        }
        else if (atoms[neighbors_2[i]] == 0 && atom_2 == 0)
        {
            E += E_znzn;
        }
        else
        {
            E += E_cuzn;
        }
    }
    return E;
}

atom_count
lattice_props(
              int *atoms,
              int **neighbors,
              int N_atoms
             )
{
    atom_count count;
    int N_Cu_A = 0; // Number of Cu atoms in sublattice A
    int N_CuZn = 0; // Number of Cu-Zn bonds
    // Loop over all atoms in sublattice A
    for (int i = 0; i < N_atoms / 2; i++)
    {
        // If the atom is Cu, increment N_Cu_A
        if (atoms[i] == 1)
        {
            N_Cu_A++;
        }
        int atom = atoms[i]; // Value of the atom
        // Loop over the neighbors of the atom
        for (int j = 0; j < 8; j++)
        {
            // If the atom is Cu and the neighbor is Zn or vice versa, increment N_CuZn
            if (atom == 1 && atoms[neighbors[i][j]] == 0)
            {
                N_CuZn++;
            }
            else if (atom == 0 && atoms[neighbors[i][j]] == 1)
            {
                N_CuZn++;
```

```
            }
        }
    }
    // Store the values in the struct and return it
    count.N_Cu_A = N_Cu_A;
    count.N_CuZn = N_CuZn;

    return count;
}

void
lattice_to_files(
                 FILE *fp_atoms,
                 FILE *fp_neighbors,
                 int *atoms,
                 int **neighbors,
                 int N_atoms
                )
{
    for (int i = 0; i < N_atoms; i++)
    {
        fprintf(fp_atoms, "%i\n", atoms[i]);
        for (int j = 0; j < 8; j++)
        {
            if (j == 7)
            {
                fprintf(fp_neighbors, "%i\n", neighbors[i][j]);
            }
            else
            {
                fprintf(fp_neighbors, "%i, ", neighbors[i][j]);
            }
        }
    }
}
}
```

C.2 tools.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_fft_real.h>
#include <gsl/gsl_fft_halfcomplex.h>
#include <complex.h>
#include "tools.h"

void
elementwise_addition(
                     double *res,
                     double *v1,
                     double *v2,
                     unsigned int len
                    )
```

```
{
    for (int i = 0; i < len; i++) {
        res[i] = v1[i] + v2[i];
    }
}

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

void
addition_with_constant(
                       double *res,
                       double *v,
                       double constant,
                       unsigned int len
                       )
{
    for (int i = 0; i < len; i++) {
        res[i] = v[i] + constant;
    }
}

void
multiplication_with_constant(
                             double *res,
                             double *v,
                             double constant,
                             unsigned int len
                             )
{
    for (int i = 0; i < len; i++) {
        res[i] = v[i] * constant;
    }
}

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

```
}

double **
create_2D_array(
                unsigned int row_size,
                unsigned int column_size
               )
{
    // Allocate memory for row pointers
    double **array = malloc(row_size * sizeof(double *));
    if (array == NULL) {
        fprintf(stderr, "Memory allocation failed for row pointers.\n");
        return NULL;
    }

    // Allocate a single contiguous block for all elements, i.e. all matrix elements
    // get stored in a single block of memory, array[0] is the start of the block
    // i.e. here we have array[0][index] as the way to access the elements
    array[0] = malloc(row_size * column_size * sizeof(double));
    if (array[0] == NULL) {
        fprintf(stderr, "Memory allocation failed for data block.\n");
        free(array);
        return NULL;
    }

    // Set the row pointers to the appropriate positions in the block, in order to
    // allow for the array[i][j] syntax. Now array[i] points to the start of the
    // i-th row.
    for (int i = 1; i < row_size; i++) {
        array[i] = array[0] + i * column_size;
    }

    return array;
}

void
destroy_2D_array(
                 double **array
                )
{
    if (array != NULL) {
        free(array[0]);  // Free the contiguous block
        free(array);      // Free the row pointers
    }
}

void
matrix_vector_multiplication(
                             double *result,
                             double **A,
                             double *b,
                             unsigned int n,
                             unsigned int m
                            )
{
    for (int i = 0; i < n; i++) {
        result[i] = 0.;
```

```
        for (int j = 0; j < m; j++) {
            result[i] += A[i][j] * b[j];
        }
    }
}

void
matrix_matrix_multiplication(
                             double **result,
                             double **A,
                             double **B,
                             unsigned int n,
                             unsigned int m,
                             unsigned int k
                            )
{
    for (int i = 0; i < n; i++) {
        for (int kappa = 0; kappa < k; kappa++) {
            result[i][kappa] = 0.;
            for (int j = 0; j < m; j++) {
                result[i][kappa] += A[i][j] * B[j][kappa];
            }
        }
    }
}

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


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

double
average(
        double *v,
        unsigned int len
       )
{
    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
```

```
            )
{
    int num_blocks = data_len / block_size;
    double *block_averages = (double *)calloc(num_blocks, sizeof(double));
    for (int i = 0; i < num_blocks; i++) {
        double sum = 0.0;
        for (int j = 0; j < block_size; j++) {
            sum += data[i * block_size + j];
        }
        block_averages[i] = sum / block_size;
    }
    double block_avg = block_size * variance(block_averages, num_blocks) / variance(data,
    ↪  data_len);
    free(block_averages);

    return block_avg;
}

double
distance_between_vectors(
                         double *v1,
                         double *v2,
                         unsigned int len
                        )
{
    double res = 0.;
    // With previous defined functions
    double *diff = malloc(len * sizeof(double));
    multiplication_with_constant(v1, v1, -1., len);
    elementwise_addition(diff, v1, v2, len);
    res = vector_norm(diff, len);
    free(diff);
    return res;
}

void
cumulative_integration(
                       double *res,
                       double *v,
                       double dx,
                       unsigned int v_len
                      )
{
    double sum = 0.;
    res[0] = 0.;
    for (int i = 1; i < v_len; i++) {
        sum = 0.5 * (v[i - 1] + v[i]) * dx;
        res[i] = res[i - 1] + sum;
    }
}

void
write_xyz(
          FILE *fp,
          char *symbol,
          double **positions,
          double **velocities,
```

```
          double alat,
          int natoms
         )
{
    fprintf(fp, "%i\nLattice=\"%f 0.0 0.0 0.0 %f 0.0 0.0 0.0 %f\" ", natoms, alat, alat,
    ↪  alat);
    fprintf(fp, "Properties=species:S:1:pos:R:3:vel:R:3 pbc=\"T T T\"\n");
    for(int i = 0; i < natoms; ++i){
        fprintf(fp, "%s %f %f %f %f %f %f\n",
                symbol, positions[i][0], positions[i][1], positions[i][2],
                velocities[i][0], velocities[i][1], velocities[i][2]);
    }
}

void fft_freq(
              double *res,
              int n,
              double timestep
             )
{
    for (int i = 0; i < n; i++) {
        if (i < n / 2) {
            res[i] = 2 * M_PI * i / (n * timestep);
        }
        else {
            res[i] = 2 * M_PI * (i - n) / (n * timestep);
        }
    }
}

/* Freely given functions */
void
skip_line(
          FILE *fp
         )
{
    int c;
    while (c = fgetc(fp), c != '\n' && c != EOF);
}

void
read_xyz(
         FILE *fp,
         char *symbol,
         double **positions,
         double **velocities,
         double *alat
        )
{
    int natoms;
    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){
        perror("Error");
    }
    skip_line(fp);
    for(int i = 0; i < natoms; ++i){
        fscanf(fp, "%s %lf %lf %lf ",
```

```
                symbol, &positions[i][0], &positions[i][1], &positions[i][2]);
        fscanf(fp, "%lf %lf %lf\n",
                &velocities[i][0], &velocities[i][1], &velocities[i][2]);
    }
}

void powerspectrum(
                   double *res,
                   double *signal,
                   int n,
                   double timestep
                  )
{
    /* Declaration of variables */
    double *complex_coefficient = malloc(sizeof(double) * 2*n); // array for the complex
    ↪  fft data
    double *data_cp = malloc(sizeof(double) * n);

    /*make copy of data to avoid messing with data in the transform*/
    for (int i = 0; i < n; i++) {
    data_cp[i] = signal[i];
    }

    /* Declare wavetable and workspace for fft */
    gsl_fft_real_wavetable *real;
    gsl_fft_real_workspace *work;

    /* Allocate space for wavetable and workspace for fft */
    work = gsl_fft_real_workspace_alloc(n);
    real = gsl_fft_real_wavetable_alloc(n);

    /* Do the fft*/
    gsl_fft_real_transform(data_cp, 1, n, real, work);

    /* Unpack the output into array with alternating real and imaginary part */
    gsl_fft_halfcomplex_unpack(data_cp, complex_coefficient,1,n);

    /*fill the output powspec_data with the powerspectrum */
    for (int i = 0; i < n; i++) {
    res[i] =
    ↪  (complex_coefficient[2*i]*complex_coefficient[2*i]+complex_coefficient[2*i+1]*complex_coeffici
    res[i] *= timestep / n;
    }

    /* Free memory of wavetable and workspace */
    gsl_fft_real_wavetable_free(real);
    gsl_fft_real_workspace_free(work);
    free(complex_coefficient);
    free(data_cp);
}
```

D Source Code in Python

```python
# Libraries
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants as K
import pandas as pd
from IPython.display import display, Markdown

# Latex style
plt.style.use('default')
plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rc('font', size=20)
plt.rcParams['text.latex.preamble'] = r'\usepackage{amsmath}'

# Set ticks on both sides
plt.rcParams['xtick.direction'] = 'in'
plt.rcParams['ytick.direction'] = 'in'
plt.rcParams['xtick.major.size'] = 5
plt.rcParams['ytick.major.size'] = 5
plt.rcParams['xtick.top'] = True
plt.rcParams['ytick.right'] = True

# Constants
k_B = K.Boltzmann
e = K.elementary_charge
k_B /= e
n_atoms = 2000

# Functions
def read_data(task, T, its_eq=0, its=0):
    '''
    Read data from csv files.

    Args:
        task (int): task number.
        T (int): temperature.
        its_eq (int): number of equilibrium iterations.
        its (int): number of MC iterations.

    Returns:
        if task == 2:
            eq_data (np.array): equilibrium data.
            eq_facts (np.array): equilibrium facts, i.e. acceptance ratio, its, etc.
            atoms_data (np.array): atoms data, showing which atom is at what position.
            energy_data (np.array): energy data.
            energy_facts (np.array): energy facts, i.e. acceptance ratio, its, etc.
        if task == 3 & T:
            N (int): number of samples.
            auto_corr_data (np.array): autocorrelation data.
            block_data (np.array): block averaging data.
            error_dict_corr (dict): errors from autocorrelation.
            error_dict_block (dict): errors from block averaging.
            var_dict (dict): variance of U, C_V, P, r.
        if task == 3 & not T:
            data (np.array): data from task 3.
```

```
    '''
    if task == 2:
        if its_eq:
            eq_facts = np.genfromtxt(f'data/task_2/equilibrium_{its_eq}_{T}.csv',
            ↪  max_rows=1, delimiter=',', dtype=np.float64)
            eq_data = np.genfromtxt(f'data/task_2/equilibrium_{its_eq}_{T}.csv',
            ↪  delimiter=',', dtype=np.float64)[1:, :]
        else:
            eq_facts = None
            eq_data = None
        if its:
            atoms_data =
            ↪  np.genfromtxt(f'data/task_2/lattice/atoms_{its_eq}_{its}_{T}.csv')
            energy_facts = np.genfromtxt(f'data/task_2/energy_{its_eq}_{its}_{T}.csv',
            ↪  max_rows=2, delimiter=',', dtype=np.float64)[1]
            energy_data = np.genfromtxt(f'data/task_2/energy_{its_eq}_{its}_{T}.csv',
            ↪  delimiter=',', dtype=np.float64, skip_header=3)
        else:
            atoms_data = None
            energy_facts = None
            energy_data = None
        return eq_data, eq_facts, atoms_data, energy_data, energy_facts

    if task == 3:
        if T:
            filename_corr = f'data/task_{task}/auto_corr_{T}.csv'
            filename_block = f'data/task_{task}/blocking_{T}.csv'

            N, var_U, var_CV, var_P, var_r = np.genfromtxt(filename_corr, delimiter=',',
            ↪  dtype=np.float64)[1, :]
            auto_corr_data = np.genfromtxt(filename_corr, delimiter=',',
            ↪  dtype=np.float64)[3:, :]
            block_data = np.genfromtxt(filename_block, delimiter=',',
            ↪  dtype=np.float64)[1:, :]

            # Find the first index where the autocorrelation is less than exp(-2)
            s_corr_U = auto_corr_data[:, 0][np.where(auto_corr_data[:, 1] <
            ↪  np.exp(-2))[0][0]]
            s_corr_CV = auto_corr_data[:, 0][np.where(auto_corr_data[:, 2] <
            ↪  np.exp(-2))[0][0]]
            s_corr_P = auto_corr_data[:, 0][np.where(auto_corr_data[:, 3] <
            ↪  np.exp(-2))[0][0]]
            s_corr_r = auto_corr_data[:, 0][np.where(auto_corr_data[:, 4] <
            ↪  np.exp(-2))[0][0]]

            # Calculate the statistical inefficiency from the blocking data
            s_block_U = np.mean(block_data[:, 1])
            s_block_CV = np.mean(block_data[:, 2])
            s_block_P = np.mean(block_data[:, 3])
            s_block_r = np.mean(block_data[:, 4])

            var_dict = {'U': var_U, 'C_V': var_CV, 'P': var_P, 'r': var_r}
            error_dict_corr = {'U': s_corr_U, 'C_V': s_corr_CV, 'P': s_corr_P, 'r':
            ↪  s_corr_r}
            error_dict_block = {'U': s_block_U, 'C_V': s_block_CV, 'P': s_block_P, 'r':
            ↪  s_block_r}
```

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

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

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

        plt.tight_layout()
    elif quantity == 'P':
        data = abs(data[:, 0]) # Long-range order parameter
        y_label = '$\\vert P\\vert$'
        fact = 0.25 # Factor to increase the y-axis

        # Plot the data
        fig = plt.figure(figsize=(7, 6))
        plt.plot(T, data, 'k', lw=2.5, label='$P(T)$')
        plt.axvline(x=T_c, color='tab:orange', linestyle='--', label='$T_c$')
        plt.xlabel('$T$ [K]')
        plt.ylabel(y_label)
        plt.ylim(plt.ylim()[0], plt.ylim()[1] + abs(plt.ylim()[1] * fact))
        plt.grid(linestyle=':', linewidth=1, alpha=0.6)
        plt.legend(loc='upper left', ncol=2)
        plt.tight_layout()

    if save:
        save_fig(fig, f'{quantity}_T', 1)
    plt.show()

def analyzer_task_2(T, its_eq, its, changed=True, plot=False, save=False):
    '''
    Analyze and plot task 2 data.

    Args:
        T (list): temperatures to plot.
        its_eq (list): equilibrium iterations.
        its (list): MC iterations.
        changed (bool): whether to keep only accepted steps or not.
        plot (bool): whether to plot or not (if its).
        save (bool): whether to save the plot or not.

    Returns:
        if its is None:
            None.
        if its is not None and changed:
            U_changed (np.array): only the accepted steps of the potential energy.
            acceptance equilibration: acceptance ratio during equilibration.
```

```
        acceptance (float): acceptance ratio.
    '''
    # Plot equilibrium data
    if its_eq and its is None:
        fig, axs = plt.subplots(1, len(T), figsize=(20, 10))
        colors = ['tab:blue', 'tab:orange', 'tab:green']
        labels = ['\\textbf{(a)}', '\\textbf{(b)}', '\\textbf{(c)}', '\\textbf{(d)}',
        ↪  '\\textbf{(e)}', '\\textbf{(f)}']
        if len(T) == 3:
            axs = np.array([[axs[0], axs[1], axs[2]]])
        for i, t in enumerate(T):
            data, eq_facts, _, _, _ = read_data(2, t, its_eq=its_eq[i])
            axs[0, i].plot(data[:, 1], lw=2, color=colors[i])
            # axs[1, i].plot(data[:, 1], lw=2, color=colors[i])
            # axs[0, 0].set_ylabel('Accepted steps')
            axs[0, 0].set_ylabel('Potential energy [eV]')
            axs[-1, i].set_xlabel('MC steps')
            axs[0, i].set_title(f'$T = {t}$ K', fontsize=40)
            axs[0, i].set_xticks(np.linspace(0, len(data[:, 0]), 3))
            axs[0, i].text(0.65, 0.2, labels[i], transform=axs[0, i].transAxes,
            ↪  fontsize=50)
            # axs[1, i].text(0.65, 0.2, labels[2*i + 1], transform=axs[1, i].transAxes,
            ↪  fontsize=50)
            # axs[0, i].set_xticks([])
        if save:
            plt.tight_layout()
            save_fig(fig, 'burnin', 2)
    # Plot MC data
    elif its and its_eq:
        _, _, atoms_data, energy_data, energy_facts = read_data(2, T, its_eq=its_eq,
        ↪  its=its)
        accept_diff = np.diff(energy_data[:, 0]) # Difference in accepted steps
        accept_diff = np.insert(accept_diff, 0, True) # Insert True at the beginning
        U_changed = energy_data[:, 1][accept_diff > 0] # Only the accepted steps of the
        ↪  potential energy

        # Plot only the accepted steps
        if plot and changed:
            fig, axs = plt.subplots(1, 2, figsize=(9, 6))
            axs[0].plot(energy_data[:, 0])
            axs[1].plot(U_changed)
            axs[0].set_ylabel('Accepted steps')
            axs[1].set_ylabel('Potential energy [eV]')
            axs[0].set_xlabel('MC steps')
            axs[1].set_xlabel('MC steps')
            plt.tight_layout()
        # Plot all steps
        elif plot and not changed:
            fig, axs = plt.subplots(1, 2, figsize=(9, 6))
            axs[0].plot(energy_data[:, 0])
            axs[1].plot(energy_data[:, 1])
            axs[0].set_ylabel('Accepted steps')
            axs[1].set_ylabel('Potential energy [eV]')
            axs[0].set_xlabel('MC steps')
            axs[1].set_xlabel('MC steps')
            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
            ↪ block averaging
295         error_1 = (np.array(errors_corr['U'], dtype=np.float64) +
            ↪ np.array(errors_block['U'], dtype=np.float64)) / 2
296         error_2 = (np.array(errors_corr['C_V'], dtype=np.float64) +
            ↪ np.array(errors_block['C_V'], dtype=np.float64)) / 2
297     elif quantity == 'Pr':
298         fig, axs = plt.subplots(1, 2, figsize=(16, 8), sharey=True)
299         Y_1 = np.abs(data[1:, 3])
300         Y_2 = data[1:, 4]
301         y_label_1 = '$P(T)$'
302         y_label_2 = '$r(T)$'
303         legend_loc = 'upper left'
304         ylim_1 = [-0.05, 1.3]
305         ylim_2 = [-0.05, 1.3]
306         colors = ['tab:green', 'tab:red']
307         # Calculate the errors as the average of the errors from autocorrelation and
            ↪ block averaging
308         error_1 = (np.array(errors_corr['P'], dtype=np.float64) +
            ↪ np.array(errors_block['P'], dtype=np.float64)) / 2
309         error_2 = (np.array(errors_corr['r'], dtype=np.float64) +
            ↪ np.array(errors_block['r'], dtype=np.float64)) / 2
310 
```

```
    # Plot the data with the errors
    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])
    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])
    # Plot the filtered data
    if quantity == 'UC':
        axs[0].plot(T, savitzky_golay(Y_1, 5, 3), color='k', linewidth = 2.7, label='Filtered', alpha=0.5, linestyle='--')
        axs[1].plot(T, savitzky_golay(Y_2, 5, 3), color='k', linewidth = 2.7, label='Filtered', alpha=0.5, linestyle='--')
    else:
        axs[0].plot(T, savitzky_golay(Y_1, 25, 9), color = 'k',linewidth = 2.7, label='Filtered', alpha=0.5, linestyle='--')
        axs[1].plot(T, savitzky_golay(Y_2, 25, 9), color = 'k',linewidth = 2.7, label='Filtered', alpha=0.5, linestyle='--')
    # Grid, labels and legends
    for ax in axs:
        ax.grid(linestyle=':', linewidth=1, alpha=0.6)
        ax.set_xlabel('$T$ [K]')
        ax.legend(fontsize = 26, loc=legend_loc, ncol=1)
        if ax == axs[0]:
            ax.set_ylabel(y_label_1)
            ax.set_ylim(ylim_1)
        else:
            ax.set_ylabel(y_label_2)
            ax.set_ylim(ylim_2)
    if save:
        plt.tight_layout()
        plt.savefig(f'figs/task_3/{quantity}_T_metro.pdf', bbox_inches='tight')

    # Plot the errors
    if error_plot:
        # Calculate the errors as the average of the errors from autocorrelation and block averaging
        e_U = (np.array(errors_corr['U'], dtype=np.float64) + np.array(errors_block['U'], dtype=np.float64)) / 2
        e_CV = (np.array(errors_corr['C_V'], dtype=np.float64) + np.array(errors_block['C_V'], dtype=np.float64)) / 2
        e_P = (np.array(errors_corr['P'], dtype=np.float64) + np.array(errors_block['P'], dtype=np.float64)) / 2
        e_r = (np.array(errors_corr['r'], dtype=np.float64) + np.array(errors_block['r'], dtype=np.float64)) / 2

        fig, axs = plt.subplots(2, 2, figsize=(16, 16), sharex=True)
        axs[0, 0].bar(T, e_U, color='tab:blue', label='$\\epsilon_{U}$', width=20, edgecolor='k')
        axs[0, 1].bar(T, e_CV, color='tab:orange', label='$\\epsilon_{C_V}$', width=20, edgecolor='k')
        axs[1, 0].bar(T, e_P, color='tab:green', label='$\\epsilon_{P}$', width=20, edgecolor='k')
        axs[1, 1].bar(T, e_r, color='tab:red', label='$\\epsilon_{r}$', width=20, edgecolor='k')
        labels = ['$\\epsilon_{U}$', '$\\epsilon_{C_V}$', '$\\epsilon_{P}$', '$\\epsilon_{r}$']
```

```
        for i, ax in enumerate(axs.flatten()):
            ax.grid(linestyle=':', linewidth=1, alpha=0.6)
            ax.legend(loc='upper left', fontsize=32)
            ax.set_ylabel(labels[i])
            ax.set_xticks([400, 600, 800, 1000])
        axs[1, 0].set_xlabel('$T$ [K]')
        axs[1, 1].set_xlabel('$T$ [K]')
        plt.tight_layout()

        if save_error_plot:
            plt.savefig(f'figs/task_3/errors.pdf', bbox_inches='tight')

    plt.show()

def plot_task_3_error(T, autocorr, block, s_corr_U, s_corr_P, s_corr_CV, save=False,
                      xlim_factor=1., xlim_factor_CVP=1., P_sep=False):
    '''
    Plot the autocorrelation and block averaging data for task 3.

    Args:
        T (int): temperature for which to plot the data.
        autocorr (np.array): autocorrelation data.
        block (np.array): block averaging data.
        s_corr_U (float): statistical inefficiency for the potential energy.
        s_corr_P (float): statistical inefficiency for the long-range order parameter.
        s_corr_CV (float): statistical inefficiency for the heat capacity.
        save (bool): whether to save the plot or not.
        xlim_factor (float): factor to increase the x-axis limit for the potential
                             energy/long/short-range order parameter.
        xlim_factor_CVP (float): factor to increase the x-axis limit for the
                                 heat capacity/long-range order parameter.
        P_sep (bool): whether to plot the long-range order parameter separately or not.
    '''
    fig, axs = plt.subplots(1, 2, figsize=(10, 6))
    # Plot U, P and r data in one plot and C_V data in the other
    if not P_sep:
        # Autocorrelation data for U, P and r
        axs[0].plot(autocorr[:, 0], autocorr[:, 1], 'tab:blue', lw=5)
        axs[0].plot(autocorr[:, 0], autocorr[:, 3], 'tab:orange', lw=3)
        axs[0].plot(autocorr[:, 0], autocorr[:, 4], 'tab:green', lw=2)
        axs[0].plot([], [], 'tab:blue', label='$U$')
        axs[0].plot([], [], 'tab:orange', label='$P$')
        axs[0].plot([], [], 'tab:green', label='$r$')
        axs[0].axhline(np.exp(-2), color='k', linestyle='-', label='$e^{-2}$')
        axs[0].axvline(s_corr_U, ymin=0.025, ymax=0.365, color='k', linestyle='--')
        axs[0].set_ylim(-0.5, 1.2)
        axs[0].set_xlim(0, s_corr_P * 1.2)
        axs[0].set_ylabel('Autocorrelation $\\Phi_k$')
        axs[0].set_xlabel('MC step $k$')
        axs[0].grid(linestyle=':', linewidth=1, alpha=0.6)
        axs[0].legend()
        plt.tight_layout()

        # Block averaging data for U, P and r
        axs[1].plot(block[:, 0], block[:, 1], 'tab:blue', lw=6)
        axs[1].plot(block[:, 0], block[:, 3], 'tab:orange', lw=3)
        axs[1].plot(block[:, 0], block[:, 4], 'tab:green', lw=2)
```

```
        axs[1].plot([], [], 'tab:blue', label='$U$')
        axs[1].plot([], [], 'tab:orange', label='$P$')
        axs[1].plot([], [], 'tab:green', label='$r$')
        axs[1].set_ylabel('Statistical Inefficiency $s$')
        axs[1].set_xlabel('Block size')
        axs[1].set_xlim(0, np.max(block[:, 0]) * xlim_factor)
        axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
        axs[1].legend()
        plt.tight_layout()

        if save:
            save_fig(fig, 'auto_corr_block_UPr'+str(T), 3)

        fig, axs = plt.subplots(1, 2, figsize=(10, 6))
        # Autocorrelation data for C_V
        axs[0].plot(autocorr[:, 0], autocorr[:, 2], 'tab:red')
        axs[0].plot([], [], 'tab:red', label='$C_V$')
        axs[0].axhline(np.exp(-2), color='k', linestyle='-', label='$e^{-2}$')
        axs[0].axvline(s_corr_CV, ymin=0.025, ymax=0.365, color='k', linestyle='--')
        axs[0].set_ylim(-0.5, 1.2)
        axs[0].set_xlim(0, s_corr_CV * 1.1)
        axs[0].set_ylabel('Autocorrelation $\\Phi_k$')
        axs[0].set_xlabel('MC step $k$')
        axs[0].grid(linestyle=':', linewidth=1, alpha=0.6)
        axs[0].legend()
        plt.tight_layout()

        # Block averaging data for C_V
        axs[1].plot(block[:, 0], block[:, 2], 'tab:red')
        axs[1].plot([], [], 'tab:red', label='$C_V$')
        axs[1].set_ylabel('Statistical Inefficiency $s$')
        axs[1].set_xlabel('Block size')
        axs[1].set_xlim(0, np.max(block[:, 0]) * xlim_factor_CVP)
        axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
        axs[1].legend()
        plt.tight_layout()

        if save:
            save_fig(fig, 'auto_corr_block_CV'+str(T), 3)
    else:
        # Autocorrelation data for U and r
        axs[0].plot(autocorr[:, 0], autocorr[:, 1], 'tab:blue', lw=5)
        axs[0].plot(autocorr[:, 0], autocorr[:, 4], 'tab:green', lw=2)
        axs[0].plot([], [], 'tab:blue', label='$U$')
        axs[0].plot([], [], 'tab:green', label='$r$')
        axs[0].axhline(np.exp(-2), color='k', linestyle='-', label='$e^{-2}$')
        axs[0].axvline(s_corr_U, ymin=0.025, ymax=0.365, color='k', linestyle='--')
        axs[0].set_ylim(-0.5, 1.2)
        axs[0].set_xlim(0, s_corr_U * 1.2)
        axs[0].set_ylabel('Autocorrelation $\\Phi_k$')
        axs[0].set_xlabel('MC step $k$')
        axs[0].grid(linestyle=':', linewidth=1, alpha=0.6)
        axs[0].legend()
        plt.tight_layout()

        # Block averaging data for U and r
        axs[1].plot(block[:, 0], block[:, 1], 'tab:blue', lw=6)
```

```
        axs[1].plot(block[:, 0], block[:, 4], 'tab:green', lw=2)
        axs[1].plot([], [], 'tab:blue', label='$U$')
        axs[1].plot([], [], 'tab:green', label='$r$')
        axs[1].set_ylabel('Statistical Inefficiency $s$')
        axs[1].set_xlabel('Block size')
        axs[1].set_xlim(0, np.max(block[:, 0]) * xlim_factor)
        axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
        axs[1].legend()
        plt.tight_layout()

        if save:
            save_fig(fig, 'auto_corr_block_Ur'+str(T), 3)

        # Autocorrelation data for P
        fig, axs = plt.subplots(1, 2, figsize=(10, 6))
        axs[0].plot(autocorr[:, 0], autocorr[:, 3], 'tab:orange', lw=3)
        axs[0].plot([], [], 'tab:orange', label='$P$')
        axs[0].axhline(np.exp(-2), color='k', linestyle='-', label='$e^{-2}$')
        axs[0].axvline(s_corr_P, ymin=0.025, ymax=0.365, color='k', linestyle='--')
        axs[0].set_ylim(-0.5, 1.2)
        axs[0].set_xlim(0, s_corr_P * 1.2)
        axs[0].set_ylabel('Autocorrelation $\\Phi_k$')
        axs[0].set_xlabel('MC step $k$')
        axs[0].grid(linestyle=':', linewidth=1, alpha=0.6)
        axs[0].legend()
        plt.tight_layout()

        # Block averaging data for P
        axs[1].plot(block[:, 0], block[:, 3], 'tab:orange', lw=3)
        axs[1].plot([], [], 'tab:orange', label='$P$')
        axs[1].set_ylabel('Statistical Inefficiency $s$')
        axs[1].set_xlabel('Block size')
        axs[1].set_xlim(0, np.max(block[:, 0]) * xlim_factor_CVP)
        axs[1].grid(linestyle=':', linewidth=1, alpha=0.6)
        axs[1].legend()
        plt.tight_layout()

        if save:
            save_fig(fig, 'auto_corr_block_P'+str(T), 3)

    plt.show()

def get_errors(Ts):
    '''
    Calculate the errors for task 3.

    Args:
        Ts (list): temperatures for which to calculate the errors.

    Returns:
        errors_corr (dict): errors from autocorrelation.
        errors_block (dict): errors from block averaging.
        facts (dict): facts about the simulations, i.e. number of samples, iterations
                      and acceptance ratio.
    '''
    # Initialize the dictionaries
    errors_corr = {
```

```
    'U': [],
    'C_V': [],
    'P': [],
    'r': []
    }
    errors_block = {
        'U': [],
        'C_V': [],
        'P': [],
        'r': []
    }
    facts = {
        'samples': [],
        'its': [],
        'acceptance': []
    }
    # The number of iterations for each temperature
    N_its = np.genfromtxt('data/task_3/data.csv', delimiter=',', dtype=np.float64,
    ↪  skip_header=1)[:, 7]

    # Iterate over the temperatures
    for T in Ts:
        # Read the data
        N, autocorr, block, error_dict_corr, error_dict_block, var_dict = read_data(3,
        ↪  int(T))
        facts['samples'].append(N)
        facts['its'].append(N_its[Ts.index(T)])
        facts['acceptance'].append(N / N_its[Ts.index(T)])

        # Statistical inefficiencies from autocorrelation
        s_U_corr = error_dict_corr['U']
        s_CV_corr = error_dict_corr['C_V']
        s_P_corr = error_dict_corr['P']
        s_r_corr = error_dict_corr['r']

        # Statistical inefficiencies from block averaging
        s_U_block = error_dict_block['U']
        s_CV_block = error_dict_block['C_V']
        s_P_block = error_dict_block['P']
        s_r_block = error_dict_block['r']

        # Calculate the errors from the statistical inefficiency
        error_U_corr = np.sqrt(var_dict['U'] * s_U_corr / N)
        error_CV_corr = np.sqrt(var_dict['C_V'] * s_CV_corr / N)
        error_P_corr = np.sqrt(var_dict['P'] * s_P_corr / N)
        error_r_corr = np.sqrt(var_dict['r'] * s_r_corr / N)

        error_U_block = np.sqrt(var_dict['U'] * s_U_block / N)
        error_CV_block = np.sqrt(var_dict['C_V'] * s_CV_block / N)
        error_P_block = np.sqrt(var_dict['P'] * s_P_block / N)
        error_r_block = np.sqrt(var_dict['r'] * s_r_block / N)

        # Append the errors to the dictionaries
        errors_corr['U'].append(f'{error_U_corr:.2e}')
        errors_corr['C_V'].append(f'{error_CV_corr:.2e}')
        errors_corr['P'].append(f'{error_P_corr:.2e}')
        errors_corr['r'].append(f'{error_r_corr:.2e}')
```

```
576
577         errors_block['U'].append(f'{error_U_block:.2e}')
578         errors_block['C_V'].append(f'{error_CV_block:.2e}')
579         errors_block['P'].append(f'{error_P_block:.2e}')
580         errors_block['r'].append(f'{error_r_block:.2e}')
581
582     return errors_corr, errors_block, facts
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,
    ↪  plot=True, save=True)
616
617 # Display MC data
618 U_400, accept_400_eq, accept_400 = analyzer_task_2(400, its_eq_400, its, changed=False,
    ↪  plot=False, save=False)
619 U_600, accept_600_eq, accept_600 = analyzer_task_2(600, its_eq, its, changed=False,
    ↪  plot=False, save=False)
620 U_1000, accept_1000_eq, accept_1000 = analyzer_task_2(1000, its_eq, its, changed=False,
    ↪  plot=False, save=False)
621
622 df = pd.DataFrame({'$T$': [400, 600, 1000], '$E_{pot}$': [np.mean(U_400), np.mean(U_600),
    ↪  np.mean(U_1000)],
623                    '$N_{eq}$': [its_eq_400, its_eq, its_eq], '$\\text{Acceptance}_{eq}$':
                       ↪  [accept_400_eq, accept_600_eq, accept_1000_eq],
624                    '$\\text{Acceptance}$': [accept_400, accept_600, accept_1000]})
625 markdown = df.to_markdown(index=False)
626 display(Markdown(markdown))
```

```
627
628 ######################################## Task 3 ########################################
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'$\\textbf{{{np.mean(np.array(errors_block["P"],
    ↪  dtype=np.float64)):.2e}}}$')
```

```
# errors_block['r'].append(f'$\\textbf{{{np.mean(np.array(errors_block["r"], dtype=np.float64)):.2e}}}$')
# 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']})
# df_block['$N_{\\text{its}}$'][:-1] = df_block['$N_{\\text{its}}$'][:-1].apply(lambda x: f'{x:.0f}')
# df_block['$\\text{Acceptance}$'][:-1] = df_block['$\\text{Acceptance}$'][:-1].apply(lambda x: f'{100*x:.2f} %')
# md_table_block = df_block.to_markdown(index=False)
# display(Markdown('### Blocking Errors \n' + md_table_block))

# Constants
T = 1000
save = True

# Read data
N, autocorr, block, error_dict_corr, error_dict_block, var_dict = read_data(3, T)
print(N)
s_corr_U, s_corr_CV = error_dict_corr['U'], error_dict_corr['C_V']
s_corr_P, s_corr_r = error_dict_corr['P'], error_dict_corr['r']

# Plot the error data
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)

```