

NB: The graded, first version of the report must be returned if you hand in a second time!

H2a: Binary Alloy

Andréas Sundström and Linnea Hesslow

December 5, 2018

Task N ^o	Points	Avail. points
Σ		

Introduction

We study a simple model of a binary alloy of copper and zinc, commonly known as brass. Some properties of the system can be determined analytically in the mean field theory, but a numerical simulation of the Ising model gives more accurate results at low dimensionality. Both the mean field theory and the Ising model play an important role in statistical mechanics. We use these two models to compute various statistical properties and compare the results.

Task 1: mean field theory

In the mean field theory (MFT) model, every individual atom is assumed to interact only with an average of the whole system, and the system is also assumed to be in equilibrium. All microscopic variations are therefore neglected.

In this binary alloy model, we have two sub-lattices (one Cu sub-lattice and one Zn sub-lattice) and we can define an order parameter

$$P = 2\frac{\tilde{N}}{N} - 1, \quad (1)$$

where \tilde{N} is the number of Cu atoms in the Cu lattice and N is the total number of Cu atoms—equivalently, we could say that N is the number of lattice sites in the Cu sub-lattice and that \tilde{N}/N is the fraction of the atoms in the sub-lattice which are Cu. This order parameter can now be used to define the mean field theory of this system.

At first, we need a connection between the order parameter, P , and the temperature, T . To get to this we note that the equilibrium of the system is given by minimizing the Helmholtz's free energy, $F_{\text{MFT}} = U_{\text{MFT}} - TS_{\text{MFT}}$, where $U_{\text{MFT}} = E_{\text{MFT}}(P)$ is the system energy and $S_{\text{MFT}} = k_B \ln \omega_{\text{MFT}}$ is the entropy, where $\omega_{\text{MFT}} = \omega_{\text{MFT}}(P)$ is the number of possible micro-states. In the Cu sub-lattice, there are $\tilde{N} = (1 + P)N/2$ Cu atoms; therefore, the number of micro-states in the Cu sub-lattice is

$$\omega'_{\text{MFT}} = \binom{N}{\tilde{N}} = \frac{N!}{\tilde{N}!(N - \tilde{N})!} = \frac{N!}{[(1 + P)N/2]! [(1 - P)N/2]!} \quad (2)$$

and the entropy of the Cu sub-lattice is

$$\begin{aligned} S'_{\text{MFT}} &= k_B \ln \left(\frac{N!}{[(1 + P)N/2]! [(1 - P)N/2]!} \right) \\ &\approx Nk_B \ln(2) - k_B \frac{N}{2} [(1 + P) \ln(1 + P) + (1 - P) \ln(1 - P)], \end{aligned} \quad (3)$$

where Stirling's formula has been used to arrive at the last result. Now, the Zn sub-lattice is equivalent to the Cu sub-lattice but with the number Zn atoms and Cu atoms interchanged. The two lattices must therefore have the same entropies, and the full system entropy is just the sum of its parts; hence

$$S_{\text{MFT}} = 2Nk_B \ln(2) - Nk_B [(1 + P) \ln(1 + P) + (1 - P) \ln(1 - P)]. \quad (4)$$

Next, we need to find $E(P)$. Using the mean field approximation that every atom only interacts with the system average, we can derive the number of the different types of bonds. The number Cu-Cu bonds, $N_{\text{CuCu}}^{(\text{MFT})}$, are given by the number of Cu atoms in the Cu sub-lattice, \tilde{N} , times the number of bonds each atom has, 8, times the probability that another Cu atom is located in the Zn sub-lattice¹, $(N - \tilde{N})/N = (1 - P)/2$. This gives

$$N_{\text{CuCu}}^{(\text{MFT})} = 8 \frac{(1 + P)N}{2} \frac{1 - P}{2} = 2N(1 - P^2). \quad (5)$$

For the Zn-Zn bonds the number has to be the same, since the Zn and Cu atoms are interchangeable:

$$N_{\text{ZnZn}}^{(\text{MFT})} = N_{\text{CuCu}}^{(\text{MFT})} = 2N(1 - P^2). \quad (6)$$

¹This is because bonds can only be made between atoms in different sub-lattices.

Then we know that the total number of bonds in this system has to be $8N$, and therefore the remaining $8N - N_{\text{ZnZn}} - N_{\text{CuCu}}$ bonds has to be inter-species bonds:

$$N_{\text{CuZn}}^{(\text{MFT})} = 4N(1 + P^2). \quad (7)$$

The energy of the system is now given by

$$E_{\text{MFT}} = E_{\text{CuZn}}N_{\text{CuZn}}^{(\text{MFT})} + E_{\text{ZnZn}}N_{\text{ZnZn}}^{(\text{MFT})} + E_{\text{CuCu}}N_{\text{CuCu}}^{(\text{MFT})}, \quad (8)$$

which can be simplified to

$$E_{\text{MFT}}(P) = (E_0 - 2P^2\Delta E)N \quad (9)$$

where

$$\begin{aligned} E_0 &= 2(E_{\text{CuCu}} + E_{\text{ZnZn}} + 2E_{\text{CuZn}}), \\ \Delta E &= (E_{\text{CuCu}} + E_{\text{ZnZn}} - 2E_{\text{CuZn}}), \end{aligned} \quad (10)$$

and where $E_{\text{CuZn}} = -294$ meV, $E_{\text{ZnZn}} = -113$ meV, and $E_{\text{CuCu}} = -436$ meV are the different bond energies. We can now find the equilibrium $P = P_{\text{eq}}$ by minimizing the free energy

$$\begin{aligned} F_{\text{MFT}}(P, T) &= NE_0 - 2NP^2\Delta E \\ &\quad - 2Nk_{\text{B}}T \ln(2) + Nk_{\text{B}}T \left[(P+1) \ln(1+P) + (1-P) \ln(1-P) \right] \\ &= NE_0 - N\Delta E \left(2P^2 + 2\tilde{T} \ln(2)\tilde{T} \left[(P+1) \ln(1+P) + (1-P) \ln(1-P) \right] \right), \end{aligned} \quad (11)$$

where $\tilde{T} = k_{\text{B}}T/\Delta E$.

Numerical calculations

To actually minimize (11) with respect to P for a given T , we need to employ numerical methods. We implemented this in MATLAB and used the `fminbnd` function to find the minimum in the range $P \in [0, 1]$. This would then give us P as a function of temperature, $P(T)$. With that, we can easily numerically calculate the system energy $U(T) = E(P(T))$ and heat capacity

$$C(T) = \frac{\partial U}{\partial T} = \frac{\partial E}{\partial P} \frac{\partial P}{\partial T}. \quad (12)$$

Results and discussion

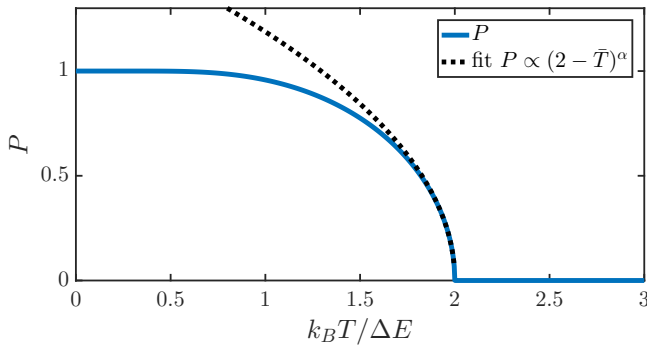


Figure 1: The mean field theory value of the order parameter, P , as a function of temperature, $\tilde{T} = k_{\text{B}}T/\Delta E$. There is a clear critical temperature, $\tilde{T}_{\text{crit}} = 2$ ($T_{\text{crit}} = 441$ K), above which P becomes constantly zero. Close below the critical temperature, there is a power law for $P(\tilde{T}) \propto (\tilde{T}_{\text{crit}} - \tilde{T})^\alpha$, with $\alpha = 0.494$; this is shown as the black dotted line.

From the numerical minimization of $F_{\text{MFT}}(P, T)$, we obtained $P(T)$ as shown in figure 1. There, we clearly see that there is a critical temperature at $\tilde{T}_{\text{crit}} = 2$ or, equivalently,

$$T_{\text{crit}} = \frac{2\Delta E}{k_{\text{B}}} = 441 \text{ K} = 168^\circ\text{C}. \quad (13)$$

Above this temperature the mean field theory predict that $P(T > T_{\text{crit}}) = 0$ is constant, which corresponds to a maximally disordered system. Below the critical temperature P quickly rises to $P(0) = 1$, which is a maximally ordered system. Note, however, that the sign of P could just as well be flipped since the system is symmetric under the transformation $P \rightarrow -P$ (just switch label on which sub-lattice is which). There is a symmetry break at $T = T_{\text{crit}}$, below which the system will spontaneously order itself into an asymmetric state: $P < 0$ or $P > 0$.

We can also find an approximating power law near the critical temperature:

$$\hat{P}(T) \propto (\bar{T}_{\text{crit}} - \bar{T})^\beta = (2 - \bar{T})^\beta, \quad (14)$$

with a so called *critical exponent*, β . We used a log-log fit to find $\beta = 0.494$, and the corresponding power relation is shown as the dotted black line in figure 1.

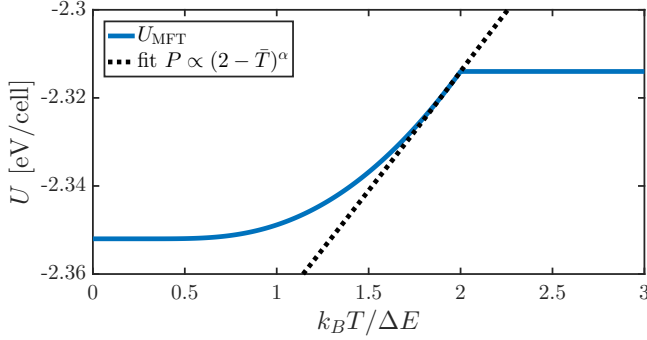


Figure 2: The mean field theory energy per unit cell, $u_{\text{MFT}} = U_{\text{MFT}}/N$, as a function of temperature, $\bar{T} = k_B T$. The energy rises from $u(\bar{T} = 0) = E_0 - 2\Delta E = -2.352$ eV to $u(\bar{T} = 0) = E_0 = -2.314$ eV per unit cell.

With $P(T)$ found, we can easily use (9) to find $U_{\text{MFT}}(T) = E_{\text{MFT}}(P(T))$, which is plotted in figure 2. There, we see that the energy rises with temperature, until we reach $\bar{T} = \bar{T}_{\text{crit}} = 2$ where, since P becomes constant $P = 0$, $U_{\text{MFT}}(T > T_{\text{crit}}) = NE_0 = -N \times 2.31$ eV becomes constant. We also see that using the corresponding power law (black dotted line) in $E(\hat{P})$ also agrees quite well.

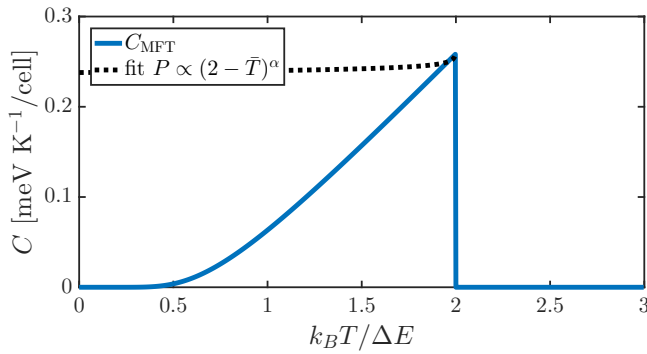


Figure 3: The mean field theory heat capacity, C_{MFT} , as function of temperature, $\bar{T} = k_B T / \Delta E$. The heat capacity rises until $\bar{T} = \bar{T}_{\text{crit}}$ to a maximum value of about $C_{\text{MFT}}^{(\text{max})} = 0.26$ meV/K per unit cell, above which C_{MFT} immediately drops to 0.

Lastly, we can calculate the MFT heat capacity of the system by numerically differentiate U from before, the result of which is shown in figure 3. Here, we see an almost linear rise in heat capacity as T approaches T_{crit} . Then, above the critical temperature, the mean field theory heat capacity drops to $C_{\text{MFT}}(T > T_{\text{crit}}) = 0$. This is clearly not physical since that would mean that there is no cost in energy to change the temperature of the system above the critical temperature.

There is also a critical exponent for the heat capacity, $\hat{C} \propto (\bar{T}_{\text{crit}} - \bar{T})^{-\alpha}$. Using (12), we can easily show that

$$\hat{C} \propto (\bar{T}_{\text{crit}} - \bar{T})^{2\beta-1}, \quad (15)$$

which corresponds to $\alpha = 1 - 2\beta = 0.012$. This power law is also plotted in figure 3, but the agreement is much worse than in the previous two cases.

The first critical exponent we found, β , is very close to $1/2$ and it is likely that the value we got deviate from $1/2$ due to numerical errors. If $\beta = 1/2$ exactly, then that would correspond to $\alpha = 0$, but that is not really what we see in figure 3.

By now it is clear that the mean field theory model does not provide a very good representation of the physical system and better methods are required, such as the numerical simulation in the next task.

Task 2: Ising model

We model the binary alloy with a static bcc lattice consisting of Cu and Zn atoms. The system size was $10 \times 10 \times 10$ cells and had periodic boundary conditions. Each atoms has eight bonds to each nearest neighbors, with energies

$$\begin{aligned} E_{\text{CuZn}} &= -294 \text{ meV} \\ E_{\text{CuCu}} &= -436 \text{ meV} \\ E_{\text{ZnZn}} &= -133 \text{ meV} \end{aligned} \quad (16)$$

We use the Metropolis algorithm to estimate statistical properties of the system. In each simulation step, we swap two randomly selected atoms in the lattice, and determine the energy change ΔE . If $\Delta E \leq 0$, or if $(\exp[-\Delta E/(k_B T)]) > \xi$, where ξ is a random number between 0 and 1, the change is accepted; otherwise the lattice remains in the previous state for another timestep. In this way, the Metropolis algorithm allows us to sample the state space according to the probability function $p \propto \exp[-E/(k_B T)]$, and thus favor the most probable configurations.

Equilibration

To equilibrate the system, we started with an ordered system and performed $N_{\text{eq,long}} = 10^6$ Monte Carlo steps to equilibrate the system at $T = -200^\circ \text{C}$. At higher temperatures, we started with the final lattice state of the previous temperature, and therefore the number of equilibration steps was reduced to $N_{\text{eq,short}} = 5 \cdot 10^5$. For all temperatures, we used 10^7 Monte Carlo steps in the production run.

Figure 4 shows the equilibration of the energy at three different temperatures: significantly below, close to and significantly above the critical temperature $T = 440^\circ \text{C}$. By plotting the energy per bond, i.e. $E/(8N_{\text{Cu}})$, we can compare the energies to the binding energies in equation (16). We note that the energy per bond is in the range $E_{\text{CuZn}} \leq E \leq E_{\text{max}}$, where

$$E_{\text{max}} \equiv \frac{1}{2}(E_{\text{CuCu}} + E_{\text{ZnZn}}) = -284.5 \text{ meV}. \quad (17)$$

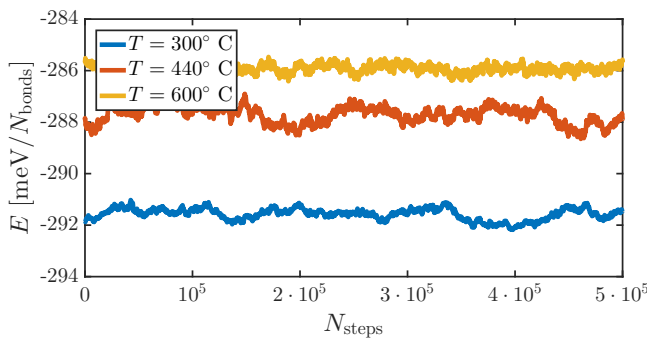


Figure 4: The energy normalized to the number of bonds in the system during the equilibration process.

Statistical properties

Figure 5 shows the equilibrium energy per cell as a function of temperature, and compared to the mean field theory. We also show the error bars of two standard deviations using the statistical inefficiency as calculated from the correlation function in section .

The metropolis simulation differs significantly from the mean free theory prediction; the critical temperature is significantly higher in the simulation and the mean energy continues to increase with temperature beyond the transition.

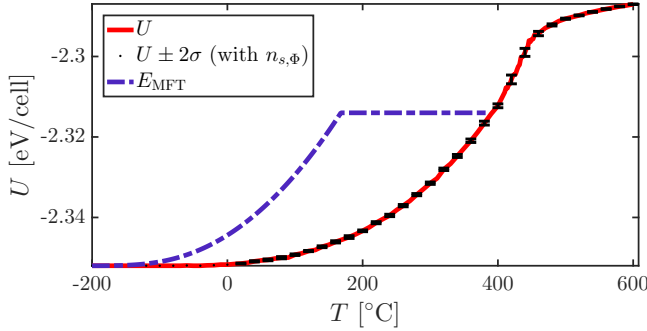


Figure 5: The average energy of the system normalized to the number of cells, as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dash-dotted blue: mean field theory prediction.

In mean field theory, the energy per cell never exceeds $8E_0 = -2.31$ eV, but in the Monte Carlo simulation the system is allowed to develop clusters of Cu and Zn atoms, which gives a higher energy than the completely randomly ordered system. In the high temperature limit of an infinite system, the theoretical maximum energy is $8E_{\max} \approx -2.276$ eV (defined in equation (17)). With a limited system size with 10 cells in each direction, we estimate that the maximum energy should be approximately

$$8(0.9 \cdot E_{\max} + 0.1E_{\text{CuZn}}) \approx 2.284 \text{ eV} \quad (18)$$

Here we obtain $E \approx -2.287$ eV at 600 °C, which is slightly below this limit.

The heat capacity can be determined either by

$$C = \frac{dU}{dT}, \quad (19)$$

or as the variance in the energy:

$$C = \frac{1}{k_B T^2} (\langle E^2 \rangle - \langle E \rangle^2). \quad (20)$$

Since the latter method does not depend on the derivatives, it gives less noisy data. This can be seen from figure 6 by comparing the gray and the black lines. Again, we note that the mean field theory gives a lower critical temperature than the simulation.

The order parameter P is shown in figure 7. Close to the phase transition, the data has high uncertainty, which is also reflected in the large error bars. Note that $P < 0$ at some temperature above the critical temperature – the system oscillates between a majority of the Cu atoms in the Cu sublattice, and a majority in the Zn sublattice.

Finally, the short range order parameter r is determined by the number of nearest-neighbor Cu-Zn bonds q as follows

$$r = \frac{1}{4N}(q - 4N) \rightarrow \begin{cases} 1, & \text{perfect order} \\ 0, & \text{no order, homogeneous system} \\ -1, & \text{fully separated system} \end{cases} \quad (21)$$

In the mean field theory, $r = P^2$. Figure 8 therefore shows not only the simulation and the MFT prediction, but also the curve P^2 . Until the transition, $r \approx P^2$ is a good estimation, but at higher temperatures r remains non-zero despite $P \approx 0$. We speculate that this is a sign that there are still more Cu-Zn bonds than the homogeneous system without order. Just like there are clusters of Cu atoms and Zn atoms, there could be clusters of order, which could possibly explain this behavior.

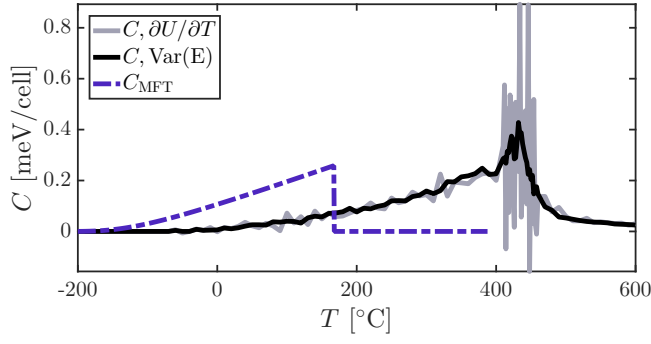


Figure 6: The specific heat of the system normalized to the number of cells, as a function of temperature. Solid gray: simulation using the derivative of U directly, black: simulation using the variance of E , dash-dotted blue: mean field theory prediction.

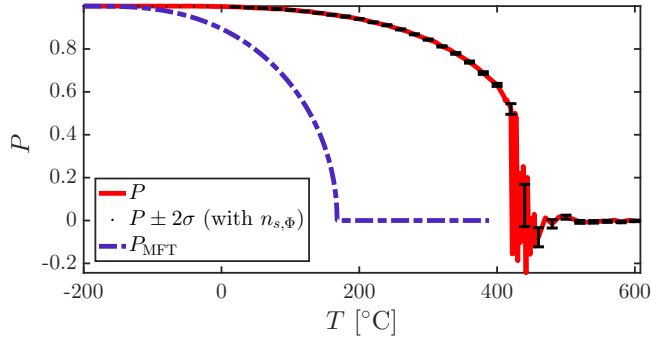


Figure 7: The order parameter P as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dash-dotted blue: mean field theory prediction.

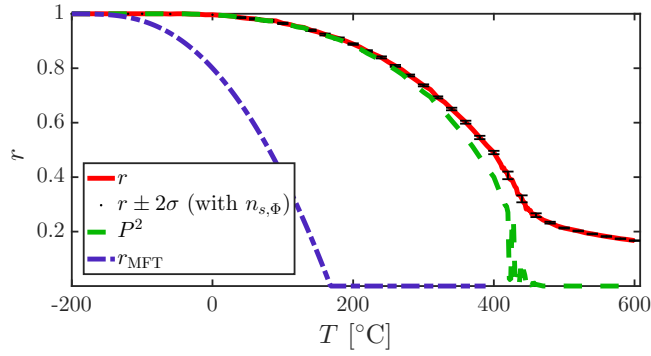


Figure 8: The short range order parameter as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dashed green: estimate $r \approx P^2$, dash-dotted blue: mean field theory prediction.

Statistical inefficiency

As described in the Lecture notes, the statistical inefficiency can be used to obtain error estimates of correlated data.

Suppose we want to measure a quantity I , as an average of $N \gg 1$ measurements:

$$I = \langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^N f_i. \quad (22)$$

The variance is then given by

$$\text{Var}[I] = \frac{n_s}{N} \text{Var}[f], \quad \text{Var}[f] = \langle f^2 \rangle - \langle f \rangle^2, \quad (23)$$

where n_s is the statistical inefficiency. The statistical inefficiency can be determined either from the decay of the correlation function,

$$\Phi_{k=n_s} = e^{-2} \approx 0.1, \quad \frac{\langle f_i f_{i+k} \rangle - \langle f \rangle^2}{\langle f^2 \rangle - \langle f \rangle^2}, \quad (24)$$

or from block averaging

$$n_s = \lim_{B \rightarrow \infty} \frac{B \text{Var}[F]}{\text{Var}[f]}, \quad F_j = \frac{1}{B} \sum_{i=1}^B f_{i+(j-1)B}, \quad j \in [1, N_{\text{blocks}}]. \quad (25)$$

The two methods in equations (24) and (25) should give similar estimates of n_s , which they do in the simulations here. The obtained statistical inefficiency is shown in figures 9 and 10 at three different temperatures, calculated with the correlation function and block average respectively.

In the case of block average, we used a moving average of 100 points, as the data become noisy when the block size become comparable to the total number of steps. Alternatively, we could have made more blocks of the largest sizes by also using shifted blocks of data, but the results obtained here were considered accurate enough.

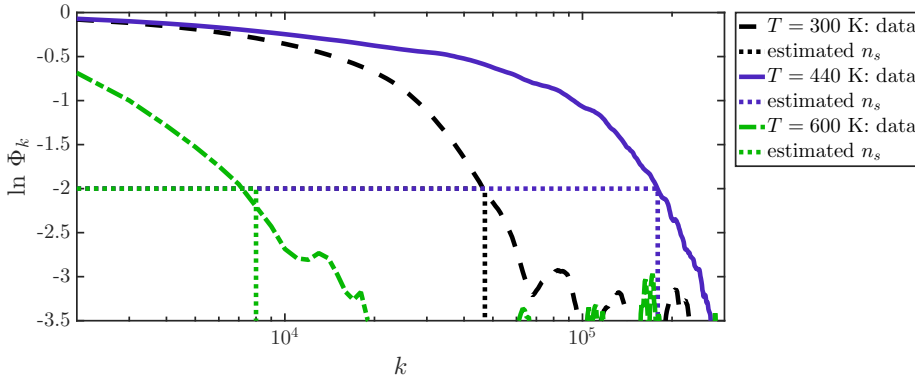


Figure 9: The logarithm of the correlation function $\Phi_k(k)$ for three different temperatures. Dotted lines mark the estimated value of $n_s = k(\ln \Phi_k = -2)$.

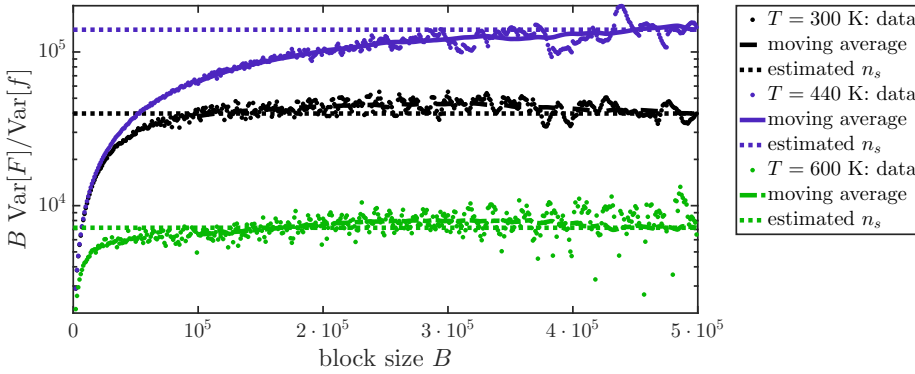


Figure 10: The statistical inefficiency determined with block averages for three different temperatures. Raw data is shown with dots, solid line show a moving average with 100 points, and the dotted lines show the estimated values of the statistical inefficiency.

Note in figures 9 and 10 that the statistical inefficiency is larger close to the phase transition at $T \approx 440^\circ\text{C}$ than at the lower and higher temperatures $T = 300^\circ\text{C}$ and $T = 600^\circ\text{C}$. We speculate that this is related to the diverging property of the correlation length close to the phase transition.

This peak in the statistical inefficiency close to the phase transition can be clearly identified also in figure 11, where n_s is plotted as a function of temperature using the two methods described above. We note that both methods give similar estimates of

n_s , but the correlation function give larger fluctuations than the block average method. Moreover, we note that the statistical inefficiency diverges as $T \rightarrow 0$ K. This is because very few changes in the lattice will be accepted at low temperatures, which give highly correlated data. At low temperatures, the equilibrium system is almost completely ordered, and we note that the uncertainty of the quantities U, P and r is still small at low temperatures as their variance decrease rapidly with decreasing temperature.

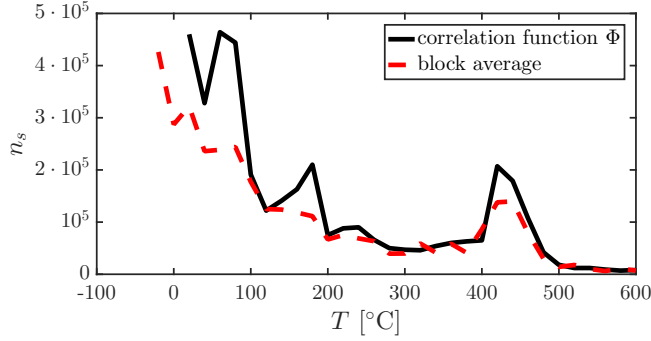


Figure 11: The statistical inefficiency n_s as a function of temperature using both the correlation function and block averages to determine n_s .

Concluding discussion

We study a binary alloy of brass. We compare semi-analytical results from mean-field theory with a Monte Carlo simulation using the Metropolis algorithm, and determine the energy, heat capacity, the order parameter as well as the short range order parameter. We find that the mean field theory underestimates the critical temperature and also fails to explain the behavior of the system above this phase transition.

A Source Code

A.1 Main program task 2: main.T2.c

```
1  /*
2   H2a, Task 2
3  */
4  #include <stdio.h>
5  #include <math.h>
6  #include <stdlib.h>
7
8  #include "funcs.h"
9
10 #define Nc 10 //number of cells
11 #define N_neigh 8
12 #define degC_to_K 273.15
13 #define kB 8.61733e-5
14
15 /* Main program */
16 int main(){
17     int N_Cu = Nc*Nc*Nc;
18     int N_atoms = 2*N_Cu;
19     int N_bonds = 8*N_Cu;
20     double Etot, E_Var, r, P; // Macro parameters
21     gsl_rng *q = init_random(); // initialize random number generator
22
23     /* done for all saved steps: */
24     int N_MCsteps = 1e7;
25     int N_eq = 1e6;
26     int N_eq_short = 5e5;
27     double *E_equilibration = malloc(sizeof(double[N_eq]));
28     double *P_equilibration = malloc(sizeof(double[N_eq]));
29     double *E_production = malloc(sizeof(double[N_MCsteps]));
30
31     /* statistical inefficiency */
32     int N_k = 500;
33     int N_skip = 1000; // k_Max = N_k * N_skip;
34     double *phi = malloc(sizeof(double[N_k]));
35     double *var_F = malloc(sizeof(double[N_k]));
36
37     /* set Temperature steps */
38     double beta;
39     double dT_small = 2;
40     double dT_large = 10;
41     double T_start = -200;
42     double T_end = 600;
43     double T_start_fine = 410;
44     double T_end_fine = 460;
45     int nT;
46     double *T_degC = init_temps(&nT, dT_small, dT_large, T_start, T_end,
47                                T_start_fine, T_end_fine);
48     // save equilibration data and stat inefficiency at T%20 =0
49     int T_save_step = 20;
50     /* done for all temps: */
51     double *E_mean = malloc(sizeof(double[nT]));
52     double *E_mean_approx = malloc(sizeof(double[nT]));
53     double *E_sq_mean = malloc(sizeof(double[nT]));
54     double *P_mean = malloc(sizeof(double[nT]));
55     double *P_sq_mean = malloc(sizeof(double[nT]));
56     double *r_mean = malloc(sizeof(double[nT]));
57     double *r_sq_mean = malloc(sizeof(double[nT]));
58
59     /* allocate and initialize lattice and nearest neighbors */
60     int *lattice = malloc(sizeof(int[N_atoms]));
61     init_ordered_lattice(N_atoms, N_Cu, lattice);
62     int (*nearest)[N_neigh] = malloc(sizeof(int[N_atoms][N_neigh]));
63     init_nearestneighbor(Nc, nearest);
64
65     /* initialize macro parameters */
66     Etot = get_Etot(lattice, N_Cu, nearest);
67     P = get_order_parameter(lattice, N_Cu);
68     r = get_short_range_order_parameter(lattice, nearest, N_Cu);
69
70
71     /* ***** start simulation ***** */
72     for (int iT=0; iT<nT; iT++){
73         /* Loop over all temperatures */
74         printf("Now running T = %.0f degC\n", T_degC[iT]);
75         beta = 1/(kB*(T_degC[iT] + degC_to_K));
76
77         /* ***** Equilibration run ***** */
78         if (iT!=0){// First run needs longer equilibration
79             N_eq=N_eq_short;
80         }
81         /* Do the Monte Carlo stepping */
82         for( int i=0; i<N_eq; i++){
83             MC_step(&Etot, &r, &P, q, lattice, nearest, beta, N_Cu);
```

```

84 // Save the energy `Etot` and orerparameter `P`
85 E_equilibration[i] = Etot;
86 P_equilibration[i]= P;
87 }
88 //Write the equilibration run to file
89 if ( ((int)T_degC[iT]) % T_save_step==0){
90     write_equil_to_file(T_degC[iT],
91         E_equilibration, N_bonds, P_equilibration, N_eq);
92 }
93
94 /* ***** Production run ***** */
95 /*
96 The saved energies are shifted by this (semi-arbitrary) amount.
97 This helps to increase the accuracy when calculating the
98 (needed for the heat capacity).
99 */
100 E_mean_approx[iT] = Etot;
101 /* initialize at temperature[iT] */
102 E_mean[iT] = 0; E_sq_mean[iT] = 0;
103 P_mean[iT] = 0; P_sq_mean[iT] = 0;
104 r_mean[iT] = 0; r_sq_mean[iT]=0;
105
106 /* Do the Monte Carlo stepping */
107 for( int i=0; i<N_MCsteps; i++){
108     MC_step( &Etot, &r, &P, q, lattice, nearest, beta, N_Cu);
109     E_production[i] = Etot- E_mean_approx[iT];
110     update_E_P_r(iT, Etot-E_mean_approx[iT], E_mean, E_sq_mean, P, P_mean,
111         P_sq_mean, r, r_mean,r_sq_mean, lattice, nearest, N_Cu);
112 }
113 /* Divide by number of Monte Carlo steps to get average */
114 E_mean[iT] *= 1/((double)N_MCsteps);
115 E_sq_mean[iT] *= 1/((double)N_MCsteps);
116 P_mean[iT] *= 1/((double)N_MCsteps);
117 P_sq_mean[iT] *= 1/((double)N_MCsteps);
118 r_mean[iT] *= 1/((double)N_MCsteps);
119 r_sq_mean[iT] *= 1/((double)N_MCsteps);
120
121 /*
122 We only calucluate the statistical inefficiency at some
123 temperatures to save on runtime.
124 */
125 if ( ((int)T_degC[iT]) % T_save_step==0 ){//calc stat ineff
126     // Calcualte the variance of the energy
127     E_Var = E_sq_mean[iT] - E_mean[iT]*E_mean[iT];
128
129     printf("Calculating statistical inefficiencies \n");
130     //Calcualte the auto-correlation
131     get_phi (phi, N_MCsteps, E_mean[iT], E_Var, E_production,N_k,N_skip);
132     //Calcualte the block-average variance
133     get_varF_block_average(var_F, N_MCsteps, E_mean[iT], E_Var,
134         E_production, N_k, N_skip);
135     //Write the stat ineff to file
136     write_stat_inefficiency_to_file(T_degC[iT], phi, var_F, N_k, N_skip);
137 }//END if calc stat ineff
138 }//END temp for
139
140 //Write the results of the production run to file
141 write_production(T_degC, nT, E_mean_approx, E_mean, E_sq_mean,
142     P_mean, P_sq_mean, r_mean, r_sq_mean);
143
144
145
146 //Don't forget to free all malloc's.
147 free(nearest); nearest = NULL;
148 free(lattice); lattice = NULL;
149 free(E_equilibration); E_equilibration = NULL;
150 free(P_equilibration); P_equilibration = NULL;
151 free(E_mean); E_mean = NULL;
152 free(E_mean_approx); E_mean_approx = NULL;
153 free(E_sq_mean); E_sq_mean = NULL;
154 free(P_mean); P_mean = NULL;
155 free(P_sq_mean); P_sq_mean = NULL;
156 free(r_mean); r_mean = NULL;
157 free(r_sq_mean); r_sq_mean = NULL;
158 free(E_production); E_production = NULL;
159 free(phi); phi = NULL;
160 free(var_F); var_F = NULL;
161 free(T_degC); T_degC = NULL;
162
163 gsl_rng_free(q); // deallocate rng
164 return 0;
165 }

```

A.2 Misc functions : funcs.c

```
1 #include "funcs.h"
```

```

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92

```

```

/***** get functions *****/
double get_bond_E(int site_1, int site_2){
    /*
    The bond can be one of three types: ZnZn, CuZn=ZnCu, or CuCu.
    With the lattice encoding Cu=1 and Zn=0, we get
    Zn+Zn = 0, Zn+Cu = Cu+Zn = 1, Cu+Cu = 2.
    Hence the switch over the three cases: 0, 1, and 2.
    */
    double Ebond=0;
    switch (site_1 + site_2){
    case 0:
        Ebond= -0.113;//E_ZnZn;
        break;
    case 1:
        Ebond= -0.294;//E_CuZn;
        break;
    case 2:
        Ebond= -0.436;//E_CuCu;
        break;
    }
    return Ebond;
}

double get_order_parameter(int *lattice, int N_Cu){
    /*
    The macro order parameter 'P' is given by the number of atoms in
    their respective sub-lattice (normalized and shifted to get a
    better physical interpretation), e.g. the number of Cu atoms in
    the Cu sub-lattice.
    */
    int N_Cu_in_Cu_lattice=0;
    for(int i=0;i<N_Cu;i++){
        /*
        Sum the atoms in the Cu sub-lattice (i=0,1,2,...,N_Cu-1), and
        with the encoding Cu=1 and Zn=0, we can simply add the values
        of the lattice encoding at each sub-lattice point.
        */
        N_Cu_in_Cu_lattice+=lattice[i];
    }
    return (double)N_Cu_in_Cu_lattice/N_Cu *2 -1;
}

double get_short_range_order_parameter(int *lattice, int(*nearest)[N_neigh],
    int N_Cu){
    /*
    The short range order parameter 'r' is given by the number of AB bonds
    (normalized and shifted to get a better physical interpretation).
    */
    int N_CuZnBonds=0;
    for(int i=0;i<N_Cu;i++){
        for( int j=0; j<N_neigh; j++){
            /*
            With the encoding Cu=1 and Zn=0, we know that in order for a
            bond to be a CuZn/ZnCu the sum of a lattice point with its
            neighbour must be 1 (see 'get_bond_E' for more detail).
            */
            N_CuZnBonds+= ((lattice[i] + lattice[nearest[i][j]]) == 1);
        }
    }
    return (double) N_CuZnBonds/(4*N_Cu)-1; // this is 'r'
}

double get_Etot(int *lattice, int N_Cu, int (*nearest)[N_neigh]){
    /*
    The total energy of the system is given by looping over every atom
    in one of the sub-lattices (Cu) and summing the energies of its
    bonds to every neighbour.
    We only need to sum over every atom in one sub-lattice since there
    are no bonds within a sub-lattice.
    */
    double Etot=0;
    for(int i=0; i<N_Cu; i++){ // loop over atoms
        for( int j=0; j<N_neigh; j++){ // loop over neighbours
            Etot+= get_bond_E(lattice[i], lattice[nearest[i][j]]);
        }
    }
    return Etot;
}

void get_phi (double *phi, int N_times, double f_mean,
    double f_var, double *data, int N_k, int N_skip){
    /*
    Function for calculating the auto-correlation in a data set. The
    rate at which the auto-correlation decay can be used to calculate
    the statistical inefficiency in the data set.
    Formula:
    
$$\phi_k = (\langle f_{i+k} f_i \rangle - \langle f_i \rangle^2) / (\langle f_i^2 \rangle - \langle f_i \rangle^2)$$

    Note that, by definition,  $\phi_0 = 1$ .
    */
}

```

```

93 int N_terms_in_avg; // helper variable
94 for (int k=0; k<N_k; k++){
95     /*
96     We loop over `k` in the formula above to get the auto-correlation
97     at the different times.
98     `phi[k]` is used to hold intermediary values, and only becomes the
99     auto-correlation at the last step in this loop.
100    */
101    phi[k] = 0;
102
103    /*
104    The number of terms in the sum to get  $\langle f_{i+k} f_i \rangle$  must be such
105    that i fulfills the relation:
106    `(i+k)*N_skip < N_times`,
107    which is equivalent to saying that
108    `i < N_times/N_skip - k`.
109    */
110    N_terms_in_avg = N_times/N_skip - k;
111    for (int i=0; i<N_terms_in_avg; i++){
112        /*
113        Add the products of the off-setted data points to get:
114        sum_{i} f_{i+k} f_i
115        */
116        phi[k] += data[i*N_skip]*data[(i+k)*N_skip];
117    }
118    /*
119    First:
120     $\langle f_{i+k} f_i \rangle = (1/N\_avg) \sum_{i} \{N\_avg\} f_{i+k} f_i$ ,
121    then we get the auto-correlation by subtracting `f_mean`^2
122    and dividing by the variance.
123    */
124    phi[k] = (phi[k]/N_terms_in_avg - f_mean*f_mean)/f_var;
125 }
126 }
127
128 void get_varF_block_average(double *var_F, int N_times, double f_mean,
129                             double f_var, double *data, int N_k, int N_skip){
130     /*
131     Function for calculating the variances of the blockaverages for `N_k`
132     different block sizes. This variance can then be used to calculate the
133     statistical inefficiency in the data set.
134     */
135     int block_size;
136     double Fj; // help variable, holding each block average
137     int number_of_blocks; // The number of blocks depends on the block size
138
139     for (int k=0; k<N_k; k++) { // block size loop
140         /*
141         For every block size, we need to loop over every block,
142         and every element in that block
143         */
144         block_size = N_skip * (k+1);
145         number_of_blocks = N_times/block_size;
146
147         var_F[k] = 0; // start
148         for (int j=0; j<number_of_blocks; j++) { // loop over all blocks
149             /* For every block, we loop over all elements in it to take average. */
150             Fj = 0; // reset to 0
151             for (int i=0; i<block_size; i++) { // internal block loop
152                 /* Adding all elems in the block to get the average */
153                 Fj += data[j*block_size + i];
154             }
155             Fj *= 1/(double)block_size; // divide by block size to get average
156             var_F[k] += Fj*Fj; // will become the variance soon
157         }
158         /*
159         To get the variance of F we use:
160          $Var[F] = \langle F^2 \rangle - \langle F \rangle^2 = \langle F^2 \rangle - \langle f \rangle^2$ ,
161         where f is the data set the block averages were taken from.
162         */
163         var_F[k] = var_F[k]/number_of_blocks - f_mean*f_mean;
164         var_F[k] *= block_size/f_var;
165     } // end block size loop
166 }
167
168 /***** Monte Carlo step functions *****/
169 void MC_step( double *Etot, double *r, double *P, gsl_rng *q,
170              int *lattice, int (*nearest)[N_neigh], double beta, int N_Cu){
171     /*
172     Function that takes a Monte Carlo step and updates the lattice points,
173     `Etot`, `r`, and `P` accordingly.
174     It is important to utilize the _chage_ in energy, `r` and `P` when
175     updating them as to not have to do a costly full calculation of either
176     every step in the Monte Carlo loop.
177     */
178     // Picks two random sites in the whole lattice.
179     int i1 = (int)(2*N_Cu*gsl_rng_uniform(q));
180     int i2 = (int)(2*N_Cu*gsl_rng_uniform(q));
181     // saves the original values
182     int old_1 = lattice[i1];
183     int old_2 = lattice[i2];

```

```

184 // Used to calculate the change in `Etot` and `r`
185 double dr = 0;
186 double dE = 0;
187 // We only need to do something if the two atoms are different
188 if (old_1 != old_2){
189     for( int j=0; j<N_neigh; j++){
190         /*
191          The change in `Etot` and `r` are first _minus_ the old energies and `r`
192          contributions.
193          */
194         dE -= get_bond_E(lattice[i1], lattice[nearest[i1][j]])
195             + get_bond_E(lattice[i2], lattice[nearest[i2][j]]);
196
197         dr -= ((lattice[i1] + lattice[nearest[i1][j]]) == 1)
198             + ((lattice[i2] + lattice[nearest[i2][j]]) == 1);
199     }
200     /* Then we do the change of the two atoms */
201     lattice[i1] = old_2;
202     lattice[i2] = old_1;
203     for( int j=0; j<N_neigh; j++){
204         /*
205          And _add_ the contributions to `Etot` and `r` from the updated lattice.
206          */
207         dE += +get_bond_E(lattice[i1], lattice[nearest[i1][j]])
208             + get_bond_E(lattice[i2], lattice[nearest[i2][j]]);
209
210         dr += ((lattice[i1] + lattice[nearest[i1][j]]) == 1)
211             + ((lattice[i2] + lattice[nearest[i2][j]]) == 1);
212     }
213
214     if ( (dE<=0) || (exp(-beta * dE) > gsl_rng_uniform(q)) ){
215         /*
216          The test is accepted if dE < 0 (accept immediately), OR
217          otherwise it's accepted with a probability of `exp(-beta * dE)`
218          */
219         // Updates P
220         if (i1 < N_Cu)
221             *P += (double)(lattice[i1] - old_1 )/N_Cu *2;
222         if (i2 < N_Cu)
223             *P += (double)(lattice[i2] - old_2 )/N_Cu *2;
224     }else{
225         /*
226          If the test failed, we change back to the old lattice configuration
227          and no change happens to `Etot` or `r`
228          */
229         lattice[i1] = old_1;
230         lattice[i2] = old_2;
231         dE = 0;
232         dr = 0;
233     } // end if step is accepted
234     *Etot += dE;
235     *r += dr/(4*N_Cu);
236 } // end if atoms are different
237 }
238
239 void update_E_P_r(int iT, double E_dev, double *E_mean, double *E_sq_mean,
240                 double P, double *P_mean, double *P_sq_mean,
241                 double r, double *r_mean, double *r_sq_mean,
242                 int *lattice, int (*nearest)[N_neigh], int N_Cu){
243     /*
244      Updates the macro parameters `E`, `P`, and `r`, as well as their squares.
245      Runs in every Monte Carlo step during the production run.
246      */
247     E_mean[iT] += E_dev;
248     E_sq_mean[iT] += E_dev * E_dev;
249
250     P_mean[iT] += P;
251     P_sq_mean[iT] += P*P;
252
253     r_mean[iT] += r;
254     r_sq_mean[iT] += r*r;
255 }
256
257 /***** initializing functions *****/
258 double * init_temps( int *nT, double dT_small, double dT_large,
259                     double T_start, double T_end, double T_start_fine,
260                     double T_end_fine){
261     /*
262      Creates an array `T_degC` with the temperatures to loop over in the main
263      function, given the fine temperature step range and the sizes of the
264      temperature steps.
265      */
266     *nT = (int) ((T_end_fine - T_start_fine)/dT_small
267                 + (T_start_fine-T_start + T_end-T_end_fine)/dT_large +1);
268     double *T_degC = malloc(sizeof(double)*nT);
269     T_degC[0] = T_start;
270     for (int iT=1; iT<*nT; iT++){ // loop over all temps
271         if (T_degC[iT-1]>=T_start_fine && T_degC[iT-1]<T_end_fine){
272             T_degC[iT] = T_degC[iT-1] + dT_small;
273         }else{
274             T_degC[iT] = T_degC[iT-1] + dT_large;

```

```

275     }
276 }
277 return T_degC;
278 }
279
280
281 void init_ordered_lattice(int N_atoms, int N_Cu, int *lattice){
282     /*
283      Initialize lattice with Cu atoms (1) in Cu lattice (i=0:N_Cu-1)
284      and Zn (0) in Zn lattice (i=N_Cu:N_atoms-1):
285     */
286     for( int i=0; i<N_Cu; i++){
287         lattice[i] = 1;
288     }
289     for( int i=N_Cu; i<N_atoms; i++){
290         lattice[i] = 0;
291     }
292 }
293
294 void init_random_lattice(int N_atoms, int N_Cu, int *lattice, gsl_rng *q){
295     /*
296      Initialize lattice with Cu and Zn atoms randomly distributed:
297     */
298     for( int i=0; i<N_Cu; i++){
299         lattice[i] = (int)(gsl_rng_uniform(q)+0.5);
300         lattice[i+N_Cu] = 1-lattice[i];
301     }
302 }
303
304
305 void init_nearestneighbor(int Nc, int (*nearest)[N_neigh]){
306     /*
307      Create a matrix `nearest[i][j]` with the index of the `j`th nearest
308      neighbors to site `i`.
309      N.B. Each site has `N_neigh` (8) nearest neighbors.
310     */
311     int i_atom;
312     int N_Cu = Nc*Nc*Nc;
313     for( int i=0; i<Nc; i++){
314         for( int j=0; j<Nc; j++){
315             for( int k=0; k<Nc; k++){
316                 i_atom = k + Nc*j + Nc*Nc*i;
317                 // k i j in one lattice <=> "k-0.5" "i-0.5" "j-0.5" in the other lattice
318                 // use mod to handle periodic boundary conditions
319                 nearest[i_atom][0] = k + Nc*j + Nc*Nc*i + N_Cu;
320                 nearest[i_atom][1] = k + Nc*j + Nc*Nc*((i+1)%Nc) + N_Cu;
321                 nearest[i_atom][2] = k + Nc*((j+1)%Nc) + Nc*Nc*i + N_Cu;
322                 nearest[i_atom][3] = k + Nc*((j+1)%Nc) + Nc*Nc*((i+1)%Nc) + N_Cu;
323                 nearest[i_atom][4] = (k+1)%Nc + Nc*j + Nc*Nc*i + N_Cu;
324                 nearest[i_atom][5] = (k+1)%Nc + Nc*j + Nc*Nc*((i+1)%Nc) + N_Cu;
325                 nearest[i_atom][6] = (k+1)%Nc + Nc*((j+1)%Nc) + Nc*Nc*i + N_Cu;
326                 nearest[i_atom][7] = (k+1)%Nc + Nc*((j+1)%Nc) + Nc*Nc*((i+1)%Nc) + N_Cu;
327
328                 // k i j in one lattice <=> "k+0.5" "i+0.5" "j+0.5" in the other lattice
329                 // use mod to handle periodic boundary conditions
330                 // note that mod([negative])<0 :
331                 i_atom += N_Cu;
332                 nearest[i_atom][0] = k + Nc*j + Nc*Nc*i;
333                 nearest[i_atom][1] = k + Nc*j + Nc*Nc*((i-1+Nc)%Nc)↵
334
335                 nearest[i_atom][2] = k + Nc*((j-1+Nc)%Nc) + Nc*Nc*i;
336                 nearest[i_atom][3] = k + Nc*((j-1+Nc)%Nc) + Nc*Nc*((i-1+Nc)%Nc)↵
337
338                 nearest[i_atom][4] = (k-1+Nc)%Nc + Nc*j + Nc*Nc*i;
339                 nearest[i_atom][5] = (k-1+Nc)%Nc + Nc*j + Nc*Nc*((i-1+Nc)%Nc)↵
340
341                 nearest[i_atom][6] = (k-1+Nc)%Nc + Nc*((j-1+Nc)%Nc) + Nc*Nc*i;
342                 nearest[i_atom][7] = (k-1+Nc)%Nc + Nc*((j-1+Nc)%Nc) + Nc*Nc*((i-1+Nc)%Nc)↵
343
344             }
345         }
346     }
347 }
348
349 gsl_rng* init_random(){
350     /*
351      Initializes a GSL random nubner generator, and returns the pointer.
352     */
353     gsl_rng *q;
354     const gsl_rng_type *rng_T; // static info about rngs
355     gsl_rng_env_setup(); // setup the rngs
356     rng_T = gsl_rng_default; // specify default rng
357     q = gsl_rng_alloc(rng_T); // allocate default rng
358     gsl_rng_set(q,time(NULL)); // Initialize rng
359     return q;
360 }
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

362  /*
363   * Writes the energy per bond `E_equilibration`/`N_bonds` and order
364   * parameter `P`, at each Monte Carlo step during the equilibration runs.
365   */
366   FILE *file_pointer;
367   char file_name[256];
368   sprintf(file_name, "../data/E_equilibration-T%d.tsv", (int) T_degC);
369   file_pointer = fopen(file_name, "w");
370   for (int i=0; i<N_eq; i++){
371     fprintf(file_pointer, "%.8f\t%.8f \n", E_equilibration[i]/N_bonds, P[i]);
372   }
373   fclose(file_pointer);
374 }
375
376 void write_production(double *T_degC, int nT, double *E_mean_approx,
377                     double *E_mean, double *E_sq_mean,
378                     double *P_mean, double *P_sq_mean,
379                     double *r_mean, double *r_sq_mean){
380   /*
381    * Writes the macro parameters `E_mean_approx`, `E_mean`, `E_sq_mean`,
382    * `P_mean`, `P_sq_mean`, `r_mean`, and `r_sq_mean` for each temperature
383    * to file.
384    */
385   FILE *file_pointer;
386   char file_name[256];
387   sprintf(file_name, "../data/E_production.tsv");
388   file_pointer = fopen(file_name, "w");
389   fprintf(file_pointer, "%s T[degC]\t E_approx\t<E-E_approx>\t<(E-E_approx)^2>\t<
390     tP\ttr\n");
391   for (int iT=0; iT<nT; iT++){
392     fprintf(file_pointer, "%.2f\t%.8e\t%.8e\t%.8e\t%.8f\t%.8f\t %.8f\t%.8f \n",
393             T_degC[iT], E_mean_approx[iT], E_mean[iT], E_sq_mean[iT], P_mean[iT],
394             P_sq_mean[iT], r_mean[iT], r_sq_mean[iT]);
395   }
396   fclose(file_pointer);
397 }
398
399 void write_stat_inefficiency_to_file(double T_degC, double *phi, double *var_F,
400                                     int N_k, int N_skip){
401   /*
402    * Writes the auto-correlation `phi` and block varaiances `var_F` for each
403    * tested temperature to file.
404    */
405   FILE *file_pointer;
406   char file_name[256];
407   sprintf(file_name, "../data/stat_inefficiency-T%d.tsv", (int) T_degC);
408   file_pointer = fopen(file_name, "w");
409   for (int i=0; i<N_k; i++){
410     fprintf(file_pointer, "%d\t%.8f\t%.8f \n", i*N_skip, phi[i], var_F[i]);
411   }
412   fclose(file_pointer);
413 }

```

B Auxiliary

B.1 Makefile

```

1
2 CC = gcc
3 CFLAGS = -O3 -Wall
4
5 LIBS = -lm -lgsl -lgslcblas
6
7 HEADERS = funcs.h
8 OBJECTS = funcs.o
9
10
11 %.o: %.c $(HEADERS)
12     $(CC) -c -o $@ $< $(CFLAGS)
13
14 all: Task2
15
16
17
18 Task2: $(OBJECTS) main_T2.c
19     $(CC) -o $@ $^ $(CFLAGS) $(LIBS)
20
21 # $(PROGRAMS): $(OBJECTS) main_T1.c
22 #     $(CC) -o $@ $^ $(CFLAGS) $(LIBS)
23
24 clean:
25     rm -f *.o
26     touch *.c

```


C MATLAB scripts

C.1 Task 1 and analysis scripts for Task 2

```
1 %% initial
2
3 tmp = matlab.desktop.editor.getActive; %% cd to current path
4 cd(fileparts(tmp.Filename));
5 set(0,'DefaultFigureWindowStyle','docked');
6 warning('off','MATLAB:handle_graphics:exceptions:SceneNode'); % interpreter
7 GRAY = 0.7*[0.9 0.9 1];
8 kB = 8.61733e-5;
9 %% task 1: MFT
10 doSave = 0;
11 clc
12
13 Pmin = 0;
14 Pmax = 1;
15
16 E_CuCu = -.436;
17 E_ZnZn = -.133;
18 E_CuZn = -.294;
19
20 E0=2*(E_CuCu+E_ZnZn+2*E_CuZn);
21 Delta_E=(E_CuCu+E_ZnZn-2*E_CuZn);
22
23 E0_bar=E0/Delta_E;
24 E_MFT=@(P) E0 - 2*P.^2*Delta_E;
25 E_MFT_bar=@(P) E0_bar - 2*P.^2;
26 dE_MFTdP =@(P) - 4*P*Delta_E;
27
28 F_MFT = @(P,Tbar) E_MFT_bar(P) + Tbar*(-2*log(2) + (1+P).*log(1+P)+(1-P).*log(1-↵
    P));
29 P_eq=@(Tbar) fminbnd(@(P)F_MFT(P, Tbar), Pmin, Pmax, optimset('TolX',1e-9));
30
31 Tbar = linspace(0,3,1000)';
32 T_MFT=Tbar*Delta_E/kB;
33 T_MFT_degC = T_MFT - 273.15;
34 Peq = zeros(size(Tbar));
35 for iT = 1:numel(Tbar)
36     Peq(iT) = P_eq(Tbar(iT));
37 end
38
39 % plot P(T) and make a fit
40 figure(1);clf
41 plot(Tbar, Peq);hold on
42
43 dT=2-Tbar(Tbar<2);
44 Peq_nonzero = Peq(Tbar<2);
45
46 I_good = (dT<0.1);
47 log_dT = log(dT(I_good));
48 log_P = log(Peq_nonzero(I_good));
49 A=[ones(size(log_dT)), log_dT]\log_P;
50 b = exp(A(1));
51 alpha = A(2);
52 fprintf('alpha = %.3f\n', alpha)
53
54 P_approx = @(alpha,b,Tbar) b*(2-Tbar).^alpha;
55 plot(Tbar(Tbar<2),P_approx(alpha,b,Tbar(Tbar<2)),'k:')
56 xlabel('$k_B T / \Delta E$')
57 ylabel('$P$')
58 legend('$P$', 'fit $P \propto (2-\bar{T})^\beta$', 'location', '↵
    NorthWest');
59 ylim([0 1.3]);
60 if doSave; setFigureSize(gcf, 300, 600); end
61
62 % plot E_MFT and the fit
63 figure(2);clf
64 plot(Tbar,E_MFT(Peq)); hold on
65 plot(Tbar,E_MFT(P_approx(alpha,b,Tbar)),'k:')
66 xlabel('$k_B T / \Delta E$')
67 ylabel('$U$ [eV/cell]')
68 legend('$U_{\rm MFT}$', 'fit $P \propto (2-\bar{T})^\beta$', 'location', '↵
    NorthWest');
69 ylim([-2.36 -2.3]);
70 if doSave; setFigureSize(gcf, 300, 600); end
71
72 figure(3);clf
73 C_MFT=diff(E_MFT(Peq))./diff(T_MFT);
74 plot(Tbar(1:end-1), C_MFT*1e3); hold on
75 C_approx=4*b^2*kB*alpha*(2-Tbar).^(2*alpha-1);
76 plot(Tbar(Tbar<2),1e3*C_approx(Tbar<2),'k:')
77 xlabel('$k_B T / \Delta E$')
78 ylabel('$C$ [meV K$^{-1}$]/cell]')
79 legend('$C_{\rm MFT}$', 'fit $P \propto (2-\bar{T})^\beta$', 'location', '↵
    NorthWest');
80 ylim([0 0.3])
```

```

81 if doSave; setFigureSize(gcf, 300, 600); end
82
83 ImproveFigureCompPhys()
84 if doSave
85     saveas(1, '../figures/P_MFT.eps', 'epsc');
86     saveas(2, '../figures/E_MFT.eps', 'epsc');
87     saveas(3, '../figures/C_MFT.eps', 'epsc');
88 end
89
90
91 %% task 2: equilibration and statistical inefficiency
92 clc;
93 doSave = 0;
94 Ts = [-200:20:600]';
95 TsToPlot = [300 440 600]';
96 t_eq=0;
97
98 figure(1);clf;
99
100 for i=1:numel(TsToPlot)
101     data = load(sprintf('../data/E_equilibration-T%d.tsv',TsToPlot(i)));
102     E = data(:,1);
103     steps = 1:length(E);
104     %P = data(:,2);
105     plot(steps, E*1000); hold on
106 end
107 legstr = strcat({'$T='}, num2str(TsToPlot), '\circ C');
108 legend(legstr, 'location', 'NorthWest');
109 ylabel('$E$ [meV/$N_{\rm bonds}$]')
110 xlabel('$N_{\rm steps}$')
111 ax = gca;
112 ax.XTickLabel = {'0', '$10^5$', '$2\cdot 10^5$', '$3\cdot 10^5$', '$4\cdot 10^5$', '$5\cdot 10^5$'};
113
114 ImproveFigureCompPhys(1)
115
116 figure(3); clf;figure(2); clf;
117 [ns_Phi,ns_block] = deal(nan(size(Ts)));
118 Nskip = 10; % did not use all k's when calculating block averages
119 N_avg = 100; % moving average
120 for i=1:numel(Ts)
121     data = load(sprintf('../data/stat_inefficiency-T%d.tsv',Ts(i)));
122     k = data(:,1);
123     block_size = k+Nskip;
124     phi = data(:,2);
125     VarF_norm = data(:,3);
126     kstar = k(find(log(phi)<=-2, 1, 'first'));
127     if ~isempty(kstar)
128         ns_Phi(i) = kstar;
129     end
130
131     filtereddata = movmean(VarF_norm,N_avg);
132     ns_block(i) = filtereddata(end);
133
134     if any(Ts(i) == TsToPlot)
135         figure(2)
136
137         semilogx(k, log(phi));hold on;
138         plot([0.1 kstar kstar], [-2 -2 -6],':k')
139
140         figure(3)
141         semilogy(block_size, VarF_norm, '.'); hold on;
142         plot(block_size(N_avg:end), filtereddata(N_avg:end));
143         plot(block_size, filtereddata(end)*ones(size(block_size)), ':k');
144
145     end
146 end
147
148 figure(4); clf;
149 plot(Ts, ns_Phi, 'k',Ts, ns_block, '--r')
150 ax = gca;
151 ax.YTickLabel = {'0', '$10^5$', '$2\cdot 10^5$', '$3\cdot 10^5$', '$4\cdot 10^5$', '$5\cdot 10^5$'};
152 ylabel('$n_s$');
153 legend('correlation function $\Phi$', 'block average');
154 xlabel('$T$ [$\circ C$]');
155 ImproveFigureCompPhys(gcf)
156
157 legs_Phi = cell(6,1);
158 legs_block = cell(9,1);
159 for i = 1:numel(TsToPlot)
160     tt = ['$T=' num2str(TsToPlot(i)) '$ K: '];
161     legs_Phi{1 + 2*(i-1)} = [tt 'data'];
162     legs_Phi{2 + 2*(i-1)} = 'estimated $n_s$';
163     legs_block{1 + 3*(i-1)} = [tt 'data'];
164     legs_block{2 + 3*(i-1)} = 'moving average';
165     legs_block{3 + 3*(i-1)} = 'estimated $n_s$';
166 end
167
168 figure(2);
169

```

```

170 legend(legs_Phi, 'location', 'northeastoutside');
171 xlabel('$k$'); ylabel('ln $\Phi_k$');
172 ylim([-3.5 0]);
173 xlim([2e3 3e5])
174 %ax = gca; ax.XTick = [3e3 1e4 3e4 1e5 3e5];
175 %ax.XTickLabel = {'$3\cdot 10^3$', '$10^4$', '$3\cdot 10^4$', '$10^5$', '$3\cdot 10^5$'}';
176 figure(3);
177 ax = gca;
178 [ax.Children(:).MarkerSize] = deal(12);
179 legend(legs_block, 'location', 'northeastOutside');
180 xlabel('block size $B$');
181 ylabel('$B$ Var[$F$]/Var[$f$]');
182 ylim([2e3 2e5])
183 ax = gca;
184 ax.XTickLabel = {'0', '$10^5$', '$2\cdot 10^5$', '$3\cdot 10^5$', '$4\cdot 10^5$', '$5\cdot 10^5$'}';
185
186 ImproveFigureCompPhys(2, 'LineColor', {'LINNEAGREEN', 'LINNEAGREEN', 'GERIBLUE', 'GERIBLUE', 'k', 'k'}',...
187 'LineStyle', {':', '-.', ':', '-', ':', '--'}')
188 ImproveFigureCompPhys(3, 'LineColor', {'LINNEAGREEN', 'LINNEAGREEN', 'LINNEAGREEN', 'GERIBLUE', 'GERIBLUE', 'GERIBLUE', 'k', 'k', 'k'}',...
189 'LineStyle', {':', '-.', 'none', ':', '-', 'none', ':', '--', 'none'}');
190
191 if doSave
192     figure(1);
193     setFigureSize(gcf, 300, 600);
194     saveas(gcf, '../figures/equilibration.eps', 'eps');
195     figure(2);
196     setFigureSize(gcf, 350, 900);
197     saveas(gcf, '../figures/stat_inefficiency_Phi.eps', 'eps');
198     figure(3);
199     setFigureSize(gcf, 350, 900);
200     saveas(gcf, '../figures/stat_inefficiency_block.eps', 'eps');
201     figure(4);
202     setFigureSize(gcf, 300, 600);
203     saveas(gcf, '../figures/stat_inefficiency_both.eps', 'eps');
204 end
205
206
207 %% task 2: U, C, P and r
208
209 doSave = 0;
210
211 data = load('../data/E_production.tsv');
212 T_degC = data(:,1);
213 N_Cu = 1e3;
214 N_timeSteps = 1e7;
215
216 Emean_approx = data(:,2)/N_Cu; % divide by N_Cu to get energy and Cv per cell
217 Emean_shifted = data(:,3)/N_Cu;
218 E_sq_mean_shifted = data(:,4)/N_Cu^2;
219
220 E_Var = (E_sq_mean_shifted - Emean_shifted.^2);
221
222 Cv = 1./(kB * (T_degC+273.15).^2).*E_Var*N_Cu;
223 U = (Emean_shifted + Emean_approx);
224 U_std = sqrt(E_Var/N_timeSteps);
225 P = data(:,5);
226 P_std = sqrt((data(:,6)-P.^2)/N_timeSteps); % without ns so far
227 r = data(:,7);
228 r_std = sqrt((data(:,8)- r.^2)/N_timeSteps);
229
230 ind = zeros(size(Ts));
231 for i = 1:numel(Ts)
232     ind(i) = find(Ts(i) == T_degC);
233 end
234
235 figure(1);clf;
236 plot(T_degC, U); hold on;
237 errorbar(Ts, U(ind), 2*U_std(ind).*sqrt(ns_Phi), 'k', 'linewidth', 2.5); hold on;
238 plot(T_MFT_degC, E_MFT(Peq), '-.'); hold on
239 ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'r'}');
240 legend('$U$', '$U_{\rm pm 2 \sigma}$ (with $n_{\rm s}$, $\rm \Phi$)$', '$E_{\rm MFT}$', 'Location', 'NorthWest');
241 ylabel('$U$ [eV/cell]')
242 axis tight
243
244 figure(2); clf;
245 plot(T_degC(2:end), 1e3*diff(U)./diff(T_degC)); hold on;
246 plot(T_degC, 1e3*Cv);
247 plot(T_MFT_degC(1:end-1), 1e3*C_MFT, '-.');
248 ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'k', GRAY});
249 legend('$C$', '{\partial U}/{\partial T}$', '$C_{\rm Var}(E)$', '$C_{\rm MFT}$', 'Location', 'NorthWest');
250 ylabel('$C$ [meV/cell]')
251 ylim([0 0.6])
252
253 figure(3);clf;

```

```

254 plot(T_degC, P, 'r'); hold on;
255 errorbar(Ts, P(ind), 2*P_std(ind).*sqrt(ns_Phi), '.k', 'linewidth', 2.5); hold ←
    on;
256 plot(T_MFT_degC, Peq, '-.k');
257 ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'r'});
258 legend('$P$', '$P\pm 2 \sigma$ (with $n_{s, \rm \Phi}$)', '$P_{\rm MFT}$', '←
    Location', 'SouthWest');
259 ylabel('$P$ ')
260 axis tight
261
262 figure(4);clf;
263 plot(T_degC, r, 'r');hold on;
264 errorbar(Ts, r(ind), 2*r_std(ind).*sqrt(ns_Phi), '.k','linewidth', 1.5);hold on;
265 plot(T_degC, P.^2, '--',T_MFT_degC, Peq.^2, '-.');
266 ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'LINNEAGREEN','r'});
267 legend('$r$', '$r\pm 2 \sigma$ (with $n_{s, \rm \Phi}$)', '$P^2$', '$r_{\rm MFT}$'←
    ', 'Location', 'SouthWest');
268 ylabel('$r$ ')
269 axis tight
270 ImproveFigureCompPhys((2:4), 'linewidth', 2)
271
272 if doSave
273     for ifig = 1:4;
274         figure(ifig)
275         setFigureSize(gcf, 300, 600);
276         xlabel('$T$ [$^\circ$C]');
277         xlim([-200 Inf])
278     end
279     ImproveFigureCompPhys(1:4);
280     saveas(1, '../figures/U.eps', 'eps');
281     saveas(2, '../figures/C.eps', 'eps');
282     saveas(3, '../figures/P.eps', 'eps');
283     saveas(4, '../figures/r.eps', 'eps');
284 end
285
286 %%
287 Tcrit = 430;
288 dT=Tcrit-T_degC(T_degC<Tcrit);
289 P_nonzero = abs(P(T_degC<Tcrit));
290
291 I_good = (dT<30 & P_nonzero>0.4);
292 log_dT = log(dT(I_good));
293 log_P = log(P_nonzero(I_good));
294 A=[ones(size(log_dT)), log_dT]\log_P;
295 b = exp(A(1));
296 alpha = A(2);
297 fprintf('P: alpha = %.3f\n', alpha)
298 P_approx = @(alpha,b,T) b*(Tcrit-T).^alpha;
299
300 %figure(5);clf;
301 %loglog(dT,P_nonzero) ; hold on;
302 %plot(dT, P_approx(alpha, b, Tcrit-dT), 'g')
303
304 figure(3)
305 Tvec = linspace(300,Tcrit);
306 plot(Tvec, P_approx(alpha, b, Tvec), ':k')
307 ImproveFigureCompPhys(gcf)
308
309
310
311 Cv_good = abs(Cv(T_degC<Tcrit));
312 I_good = (dT<150);
313 log_dT = log(dT(I_good));
314 log_C = log(Cv_good(I_good));
315 A=[ones(size(log_dT)), log_dT]\log_C;
316 b = exp(A(1));
317 alpha = A(2);
318 fprintf('Cv: alpha = %.3f\n', alpha)
319 C_approx = @(alpha,b,T) b*(Tcrit-T).^alpha;
320
321 %figure(6);clf;
322 %loglog(dT,Cv_good) ; hold on;
323 %plot(dT, C_approx(alpha, b, Tcrit-dT), 'g')
324
325 figure(2);
326 plot(Tvec, 1e3*C_approx(alpha, b, Tvec), ':r')
327 ImproveFigureCompPhys(gcf)

```

C.2 Improve figure appearance: ImproveFigureCompPhys.m

```

1 function ImproveFigureCompPhys(varargin)
2 %ImproveFigureCompPhys Improves the figures of supplied handles
3 % Input:
4 % - none (improve all figures) or handles to figures to improve
5 % - optional:
6 %     LineWidth int

```

```

7 %      LineStyle column vector cell, e.g. {'-','--'}',
8 %      LineColor column vector cell, e.g. {'k',[0 1 1], 'MYBLUE'}'
9 %      colors: MYBLUE,MYORANGE,MYGREEN,MYPURPLE, MYYELLOW,
10 %      MYLIGHTBLUE, MYRED
11 %      Marker column vector cell, e.g. {'.', 'o', 'x'}'
12
13 % ImproveFigure was originally written by Adam Stahl, but has been heavily
14 % modified by Linnea Hesslow
15
16
17 %%% Handle inputs
18 % If no inputs or if the first argument is a string (a property rather than
19 % a handle), use all open figures
20 if nargin == 0 || ischar(varargin{1})
21     %Get all open figures
22     figHs = findobj('Type','figure');
23     nFigs = length(figHs);
24 else
25     % Check the supplied figure handles
26     figHs = varargin{1};
27     figHs = figHs(ishandle(figHs) == 1); %Keep only those handles that are ←
        proper graphics handles
28     nFigs = length(figHs);
29 end
30
31 % Define desired properties
32 titleSize = 24;
33 interpreter = 'latex';
34 lineWidth = 4;
35 axesWidth = 1.5;
36 labelSize = 22;
37 textSize = 20;
38 legTextSize = 18;
39 tickLabelSize = 18;
40 LineColor = {};
41 LineStyle = {};
42 Marker = {};
43
44 % define colors
45 co = [ 0      0.4470    0.7410
46       0.8500    0.3250    0.0980
47       0.9290    0.6940    0.1250
48       0.4940    0.1840    0.5560
49       0.4660    0.6740    0.1880
50       0.3010    0.7450    0.9330
51       0.6350    0.0780    0.1840 ];
52 colors = struct('MYBLUE', co(1,:),...
53     'MYORANGE', co(2,:),...
54     'MYYELLOW', co(3,:),...
55     'MYPURPLE', co(4,:),...
56     'MYGREEN', co(5,:),...
57     'MYLIGHTBLUE', co(6,:),...
58     'MYRED',co(7,:),...
59     'GERIBLUE', [0.3000    0.1500    0.7500],...
60     'GERIRED', [1.0000    0.2500    0.1500],...
61     'GERIYELLOW', [0.9000    0.7500    0.1000],...
62     'LIGHTGREEN', [0.4    0.85    0.4],...
63     'LINNEAGREEN', [7 184 4]/255);
64
65 % Loop through the supplied arguments and check for properties to set.
66 for i = 1:nargin
67     if ischar(varargin{i})
68         switch lower(varargin{i}) %Compare lower case strings
69             case 'linewidth'
70                 lineWidth = varargin{i+1};
71             case 'linestyle'
72                 LineStyle = varargin{i+1};
73             case 'linecolor'
74                 LineColor = varargin{i+1};
75                 for iLineColor = 1:numel(LineColor)
76                     if isfield(colors, LineColor{iLineColor})
77                         LineColor{iLineColor} = colors.(LineColor{iLineColor});
78                     end
79                 end
80             case 'marker'
81                 Marker = varargin{i+1};
82         end
83     end
84 end
85 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
86
87 %%% Improve the figure(s)
88
89 for iFig = 1:nFigs
90     fig = figHs(iFig);
91
92     lineObjects = findall(fig, 'Type', 'line');
93     textObjects = findall(fig, 'Type', 'text');
94     axesObjects = findall(fig, 'Type', 'axes');
95     legObjects = findall(fig, 'Type', 'legend');

```

```

97 contourObjects = findall(fig,'Type','contour'); % not counted as lines
98
99 %%% TEXT APPEARANCE: first set all to textSize and then change the ones
100 %%% that need to be changed again
101
102 %Change size of any text objects in the plot
103 set(textObjects,'FontSize',textSize);
104 set(legObjects,'FontSize',legTextSize);
105
106 %%% FIX LINESTYLE, COLOR ETC. FOR EACH PLOT SEPARATELY
107 for iAx = 1:numel(axesObjects)
108     lineObjInAx = findall(axesObjects(iAx), 'Type', 'line');
109
110     %set line style and color style (only works if all figs have some
111     %number of line plots..)
112     if ~isempty(LineStyle)
113         set(lineObjInAx, {'LineStyle'}, LineStyle)
114         set(contourObjects, {'LineStyle'}, LineStyle); %%%%%%%%%
115     end
116     if ~isempty(LineColor)
117         set(lineObjInAx, {'Color'}, LineColor)
118         set(contourObjects, {'LineColor'}, LineColor); %%%%%%%%%
119     end
120     if ~isempty(Marker)
121         set(lineObjInAx, {'Marker'}, Marker)
122         set(lineObjInAx, {'Markersize'}, num2cell(10+22*strcmp(Marker, '.'))↵
123         )
124     end
125
126     %%% change font sizes.
127     % Tick label size
128     xLim = axesObjects(iAx).XLim;
129     axesObjects(iAx).FontSize = tickLabelSize;
130     axesObjects(iAx).XLim = xLim;
131     %Change label size
132     axesObjects(iAx).XLabel.FontSize = labelSize;
133     axesObjects(iAx).YLabel.FontSize = labelSize;
134
135     %Change title size
136     axesObjects(iAx).Title.FontSize = titleSize;
137 end
138
139 %%% LINE APPEARANCE
140 %Change line thicknesses
141 set(lineObjects,'LineWidth',lineWidth);
142 set(contourObjects, 'LineWidth', lineWidth);
143 set(axesObjects, 'LineWidth', axesWidth)
144
145 % set interpreter: latex or tex
146 set(textObjects, 'interpreter', interpreter)
147 set(legObjects, 'Interpreter', interpreter)
148 set(axesObjects, 'TickLabelInterpreter', interpreter);
149 end
end

```

C.3 Change size of figures: setFigureSize.m

```

1 function [ fig ] = setFigureSize( fig, H, W )
2 fig.Units = 'points';
3 fig.WindowStyle = 'normal'; % undock
4 fig.Position(3:4) = [W H];
5 end

```