H2a: Binary Alloy

Andréas Sundström and Linnea Hesslow

December 6, 2018

Task №	Points	Avail. points
Σ		

Introduction

Both the mean field theory and the Ising model play an important role in statistical mechanics. We use these two models to study a simple model of a binary alloy of 50 % copper and 50 % zink, commonly known as brass. Some properties of the system can be easily determined in the mean field theory, but a numerical simulation of the Ising model gives more accurate results. Here, we compute various statistical properties and compare the results of the Ising model and the mean field theory.

In both the mean field theory and the Ising model, we model the binary alloy with a static, three-dimensional bcc lattice consisting of Cu and Zn atoms. Each atom has eight bonds to each nearest neighbor, with bond energies

$$
\begin{aligned}
E_{\mathrm{CuZn}} &= -294\,\mathrm{meV}, \\
E_{\mathrm{CuCu}} &= -436\,\mathrm{meV}, \\
E_{\mathrm{ZnZn}} &= -113\,\mathrm{meV}.
\end{aligned}
\tag{1}
$$

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

$$
P = 2\frac{\tilde{N}}{N} - 1,
\tag{2}
$$

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 Helmholz's free energy, $F_{\mathrm{MFT}} = U_{\mathrm{MFT}} - TS_{\mathrm{MFT}}$, where $U_{\mathrm{MFT}} = E_{\mathrm{MFT}}(P)$ is the system energy and $S_{\mathrm{MFT}} = k_{\mathrm{B}} \ln \omega_{\mathrm{MFT}}$ is the entropy, where $\omega_{\mathrm{MFT}} = \omega_{\mathrm{MFT}}(P)$ 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

$$
\omega'_{\mathrm{MFT}} = \binom{N}{\tilde{N}} = \frac{N!}{\tilde{N}!\,(N - \tilde{N})!} = \frac{N!}{[(1 + P)N/2]!\,[(1 - P)N/2]!}
\tag{3}
$$

and the entropy of the Cu sub-lattice is

$$
\begin{aligned}
S'_{\mathrm{MFT}} =& k_{\mathrm{B}} \ln\!\left(\frac{N!}{[(P + 1)N/2]!\,[(P - 1)N/2]!} \right) \\
\approx& Nk_{\mathrm{B}} \ln(2) - k_{\mathrm{B}} \frac{N}{2}\Big[(1 + P)\ln(1 + P) + (1 - P)\ln(1 - P)\Big],
\end{aligned}
\tag{4}
$$

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 ust the sum of its parts; hence

$$
S_{\mathrm{MFT}} = 2Nk_{\mathrm{B}} \ln(2) - Nk_{\mathrm{B}}\Big[(P + 1)\ln(1 + P) + (1 - P)\ln(1 - P)\Big].
\tag{5}
$$

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, $N_{\mathrm{CuCu}}^{(\mathrm{MFT})}$, 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 atoms is located in the Zn sub-lattice[1], $(N - \tilde{N})/N = (1 - P)/2$. This gives

$$
N_{\mathrm{CuCu}}^{(\mathrm{MFT})} = 8\frac{(1 + P)N}{2} \frac{1 - P}{2} = 2N(1 - P^2).
\tag{6}
$$

---

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

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

$$N_{\text{ZnZn}}^{\text{(MFT)}} = N_{\text{CuCu}}^{\text{(MFT)}} = 2N(1 - P^2). \tag{7}$$

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:

$$N_{\text{CuZn}}^{\text{(MFT)}} = 4N(1 + P^2). \tag{8}$$

The energy of the system is now given by

$$E_{\text{MFT}} = E_{\text{CuZn}} N_{\text{CuZn}}^{\text{(MFT)}} + E_{\text{ZnZn}} N_{\text{ZnZn}}^{\text{(MFT)}} + E_{\text{CuCu}} N_{\text{CuCu}}^{\text{(MFT)}}, \tag{9}$$

which can be simplified to

$$E_{\text{MFT}}(P) = (E_0 - 2P^2 \Delta E) N \tag{10}$$

where

$$\begin{aligned} E_0 &= 2(E_{\text{CuCu}} + E_{\text{ZnZn}} + 2E_{\text{CuZn}}) = -2.274 \,\text{eV}, \\ \Delta E &= (E_{\text{CuCu}} + E_{\text{ZnZn}} - 2E_{\text{CuZn}}) = 39 \,\text{meV}, \end{aligned} \tag{11}$$

and where the bond energies are given in equation (1). We can now find the equilibrium $P = P_{\text{eq}}$ by minimizing the free energy

$$\begin{aligned} F_{\text{MFT}}(P, T) =& NE_0 - 2NP^2 \Delta E \\ & - 2Nk_{\text{B}} T \ln(2) + Nk_{\text{B}} T \big[(P + 1) \ln(1 + P) + (1 - P) \ln(1 - P)\big] \\ =& NE_0 - N\Delta E \big(2P^2 + 2\bar{T} \ln(2) - \bar{T} \big[(P + 1) \ln(1 + P) + (1 - P) \ln(1 - P)\big]\big), \end{aligned} \tag{12}$$

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

The critical temperature

The critical temperature can be found by differentiating (12) with respect to P:

$$\begin{aligned} 0 &= \frac{\partial F_{\text{MFT}}(P, T)}{\partial P} \\ &= -4PN + N\bar{T} \left[\ln(1 + P) + \frac{1 + P}{1 + P} - \ln(1 - P) - \frac{1 - P}{1 - P}\right]. \end{aligned} \tag{13}$$

Equation (13) is always fulfilled for $P = 0$, which means that the free energy has a local extremum point at $P = 0 \,\forall \bar{T}$. By expanding (13) at small P, we find

$$0 = -4P + 2\bar{T}_{\text{crit}} P \rightarrow \bar{T}_{\text{crit}} = 2. \tag{14}$$

This means that $\bar{T}_{\text{crit}} = 2$ is the only extremum point at small P. We note that

$$\frac{\partial^2 F_{\text{MFT}}(P, T)}{\partial P^2}\bigg|_{P=0} = 2N(\bar{T} - 2) \Rightarrow \begin{cases} P = 0 \text{ minimum if } \bar{T} > 2 \\ P = 0 \text{ maximum if } \bar{T} < 2. \end{cases} \tag{15}$$

Consequently, there is a phase transition $\bar{T} = \bar{T}_{\text{crit}} = 2$: the equilibrium order parameter $P \neq 0$ for $\bar{T} < 2$, whereas $P = 0$ for $\bar{T} > 2$.

Numerical calculations

To actually minimize (12) 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 the easily numerically calculated the system energy $U(T) = E(P(T))$ and heat capacity

$$C(T) = \frac{\partial U}{\partial T} = \frac{\partial E}{\partial P} \frac{\partial P}{\partial T}. \tag{16}$$

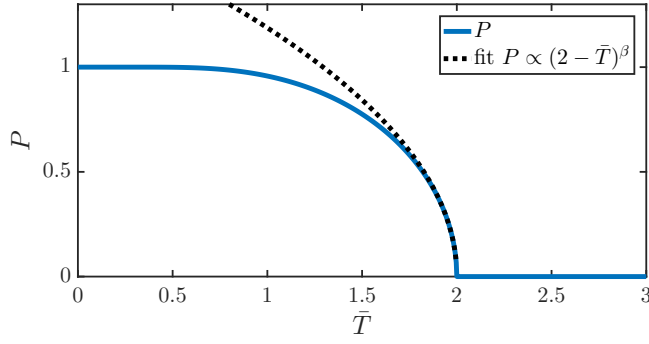Figure 1: The mean field theory value of the order parameter, P, as a function of temperature, $\bar{T} = k_B T / \Delta E$. There is a clear critical temperature, $\bar{T}_{\text{crit}} = 2$ ($T_{\text{crit}} = 905\,\text{K}$), above which P becomes constantly zero. Close below the critical temperature, there is a power law for $P(\bar{T}) \propto (\bar{T}_{\text{crit}} - \bar{T})^{\alpha}$, with $\alpha = 0.494$; this is shown as the black dotted line.

Results and discussion

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 $\bar{T}_{\text{crit}} = 2$, which is consistent with the prediction (14). Numerically,

$$T_{\text{crit}} = \frac{2\Delta E}{k_B} = 905\,\text{K} = 632\,°\text{C}. \tag{17}$$

Above this temperature the mean field theory predicts 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 $P \to -P$ (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:

$$\hat{P}(T) \propto (\bar{T}_{\text{crit}} - \bar{T})^{\beta} = (2 - \bar{T})^{\beta}, \tag{18}$$

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.
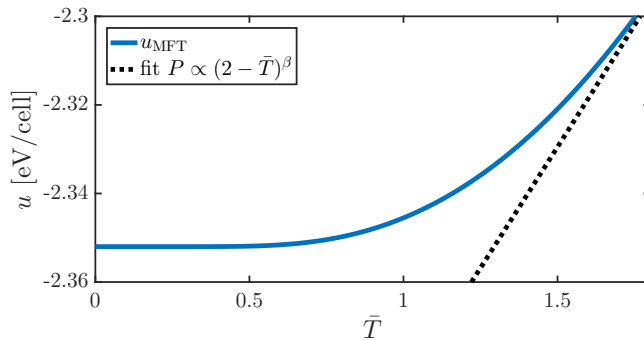

Figure 2: The mean field theory energy per unit cell, $u_{\text{MFT}} = U_{\text{MFT}}/N$, as a function of temperature, $\bar{T} = k_B T$. The energy rises from $u(\bar{T} = 0) = E_0 - 2\Delta E = -2.352\,\text{eV}$ to $u(\bar{T} = 0) = E_0 = -2.314\,\text{eV}$ per unit cell.

With $P(T)$ found, we can easily use (10) 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 $P = 0$, $U_{\text{MFT}}(T > T_{\text{crit}}) = NE_0 = -N \times 2.31\,\text{eV}$ becomes constant. We also see that using the corresponding power law (black dotted line) in $E(\hat{P})$ also agrees quite well.

3
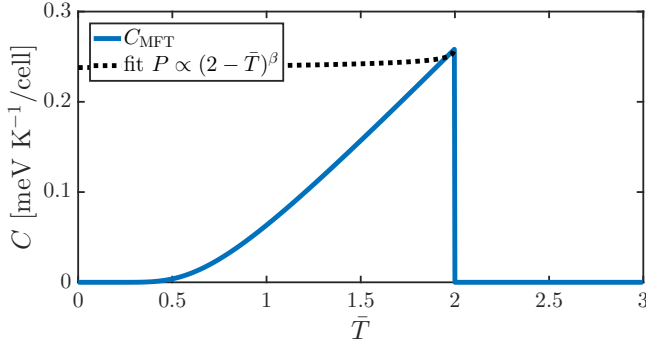
Figure 3: The mean field theory heat capacity, C_{MFT}, as function of temperature, $\bar{T} = k_B T/\Delta E$. The heat capacity rises until $\bar{T} = \bar{T}_{\text{crit}}$ to a maximum value of about $C_{\text{MFT}}^{(\text{max})} = 0.26\,\text{meV/K}$ per unit cell, above which C_{MFT} immediately 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 (16), we can easily show that

$$\hat{C} \propto (\bar{T}_{\text{crit}} - \bar{T})^{-\alpha} = (\bar{T}_{\text{crit}} - \bar{T})^{2\beta-1}, \tag{19}$$

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

Task 2: Ising model

We use the Metropolis algorithm to estimate statistical properties of the system, which has a size of $10 \times 10 \times 10$ unit cells and periodic boundary conditions. In each simulation step, we swap two randomly selected atoms in the lattice, and determine the energy change ΔE. If

$$\Delta E \leq 0, \tag{20}$$

or if

$$\exp[-\Delta E/(k_B T)] > \xi, \tag{21}$$

where ξ is a random number between 0 and 1, the change is accepted; otherwise the lattice remains in the previous state for another step. In this way, the Metropolis algorithm allows us to sample the state space according to a probability $p \propto \exp[-E/(k_B T)]$, and thus favor the most probable configurations. However, this also poses a risk that the system gets stuck in some configuration due to a high rejection rate of chages. This means that the system needs a large number, much larger than the number of particles, of iterations to reach an equlibrium given the temperature. Then, when the system is equlibrated, we also need a large number of iterations in the production run, where the data is collected.

Equilibration

To equilibrate the system, we started with a fully ordered system and performed $N_{\text{eq,long}} = 10^6$ Monte Carlo steps to equilibrate the system at $T = -200\,°\text{C}$. At higher temperatures, we started with the final lattice state of the previous temperature run, 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 $T_{\text{crit}} \approx$

4

430 °C. We note that the energy per unit cell is in the range

$$E_0 - 2\Delta E \approx -2.35 \, \text{eV} \leq E \leq E_0 \approx -2.27 \, \text{eV}, \tag{22}$$

which is consistent with the mean field theory prediction (10); and the energy is equal to or lower than the completely random system where $E = E_0$.
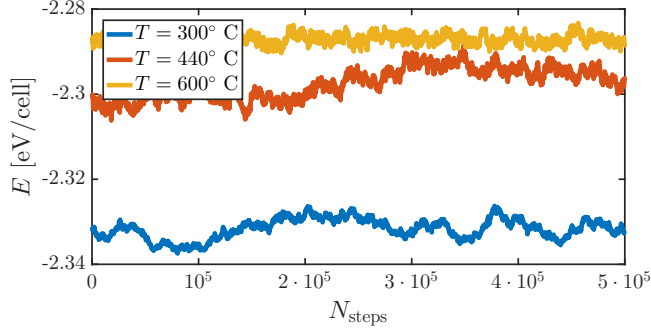

Figure 4: The energy per unit cell in the system during the equilibration process.

Statistical properties

Figure 5 shows the equilibrium energy per unit 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 .
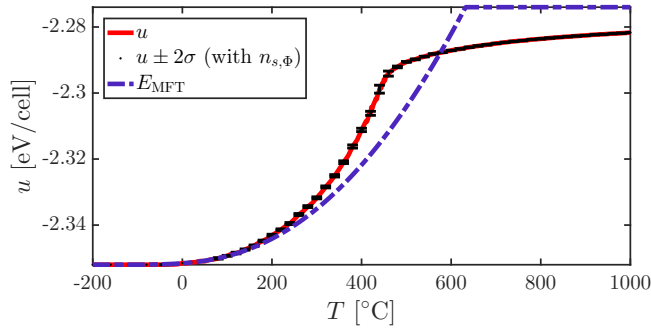

Figure 5: The average energy of the system normalized to the number of cells, as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dash-dotted blue: mean field theory prediction.

The metropolis simulation differs significantly from the mean free theory prediction: the critical temperature is significantly lower in the simulation, and the mean energy continues to increase with temperature beyond the transition. In mean field theory, the energy per unit in the range $u \in [E_0 - 2\Delta E, E_0]$ (see equation (22). This is fulfilled also in the Monte Carlo simulation, which means that it did not develop clusters of Cu and Zn atoms, in which case the energy per bond would approach the average of the Cu-Cu and Zn-Zn bond energies. In contrast, the simulation does not reach the limit of $u = E_0$ even significantly above T_{crit}.

The heat capacity can be determined either by

$$C = \frac{\partial u}{\partial T}, \tag{23}$$

or as the variance in the energy:

$$C = \frac{1}{k_B T^2} \left(\langle E^2 \rangle - \langle E \rangle^2 \right). \tag{24}$$

5

Since the latter method does not depend on numerical 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 higher critical temperature than the simulation. Moreover, the heat capacity remains non-zero above the critical temperature. Thus, the Ising model does not share the unphysical artifact of the mean field theory (where there is no cost in energy to change the temperature of the system above the critical temperature). This is because the energy $u < E_0$, even at high temperatures in the simulation.
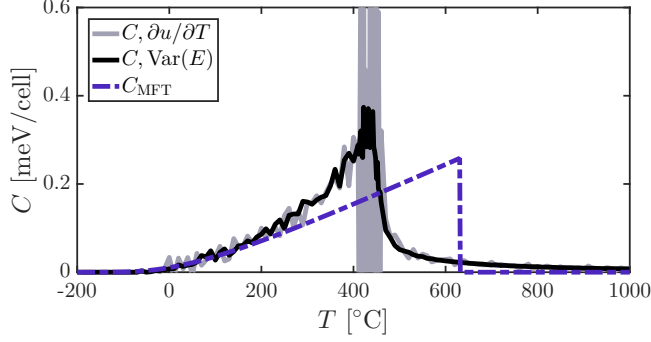

Figure 6: The specific heat of the system normalized to the number of cells, as a function of temperature. Solid gray: simulation using the derivative of U directly, black: simulation using the variance of E, dash-dotted blue: mean field theory prediction.
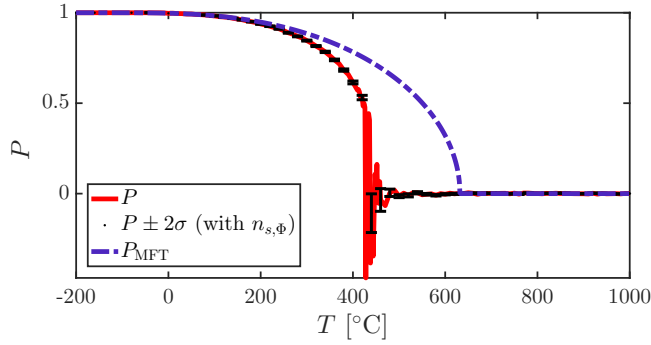

Figure 7: The order parameter P as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dash-dotted blue: mean field theory prediction.

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 temperatures above the critical temperature – the system oscillates between a majority of the Cu atoms in the Cu sub-lattice, and a majority in the Zn sub-lattice. Physically, however, there is no real difference between a system where $P < 0$ and $P > 0$, so the large fluctuations should not be seen as a major concern.

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

$$r = \frac{1}{4N}(N_{CuZn} - 4N) \rightarrow \begin{cases} 1, & \text{perfect order} \\ 0, & \text{no order, homogeneous system} \\ -1, & \text{fully separated system} \end{cases} \tag{25}$$

In the mean field theory,

$$r^{MFT} = \frac{1}{4N}[4N(1 + P^2) - 4N] = P^2 \tag{26}$$

by equation (8). Therefore, figure 8 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

6

at higher temperatures r remains non-zero despite $P \approx 0$. This is a sign that there are still more Cu-Zn bonds than the completely random system. This is also consistent with the energy in figure 5, since

$$U = E_{\text{CuZn}} N_{\text{CuZn}} + \tfrac{1}{2}(8N - N_{\text{CuZn}})(E_{\text{ZnZn}} + E_{\text{CuCu}}) \tag{27}$$

$$= E_{\text{CuZn}} 4N(r + 1) + 2N[1 - r](E_{\text{ZnZn}} + E_{\text{CuCu}}) \tag{28}$$

$$= NE_0 - 2Nr\Delta E. \tag{29}$$

Accordingly, $r = 1$ corresponds to the low temperature limit $u = E_0 - 2\Delta E$; the average energy then approaches $u \to E_0$ at high temperatures (c.f. equation (22)).
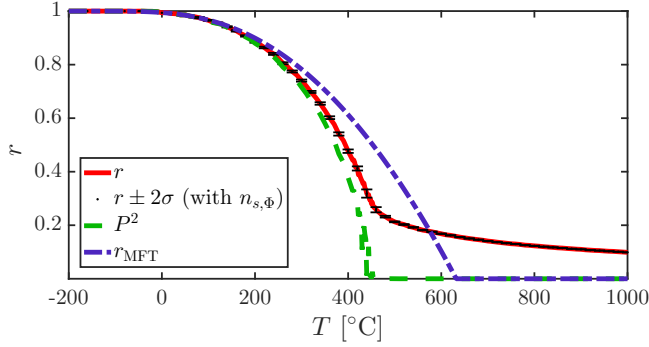

Figure 8: The short range order parameter as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dashed green: estimate $r \approx P^2$, dash-dotted blue: mean field theory prediction.

Finally a note on the critical exponents. Similarly to the MFT analysis, we performed a power law fit of the near-critical behavior of $P \propto (\bar{T}_{\text{crit}} - \bar{T})^\beta$ and $C \propto (\bar{T}_{\text{crit}} - \bar{T})^{-\alpha}$, resulting in

$$\beta \approx 0.1, \quad \alpha \approx 0.2. \tag{30}$$

Although these values are highly uncertain, we note that the MFT prediction $\alpha_{\text{MFT}} = 1 - 2\beta_{\text{MFT}}$ does not generalize to the Ising model simulation.

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:

$$I = \langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f_i. \tag{31}$$

The variance is then given by

$$\text{Var}[I] = \frac{n_s}{N} \text{Var}[f], \quad \text{Var}[f] = \langle f^2 \rangle - \langle f \rangle^2, \tag{32}$$

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

$$\Phi_{k=n_s} = e^{-2} \approx 0.1, \quad \frac{\langle f_i f_{i+k} \rangle - \langle f \rangle^2}{\langle f^2 \rangle - \langle f \rangle^2}, \tag{33}$$

or from block averaging

$$n_s = \lim_{B \to \infty} \frac{B\text{Var}[F]}{\text{Var}[f]}, \quad F_j = \frac{1}{B} \sum_{i=1}^{B} f_{i+(j-1)B}, \quad j \in [1, N_{\text{blocks}}]. \tag{34}$$

The two methods in equations (33) and (34) should give similar estimates of n_s, which they do in the simulations here. The obtained statistical inefficiency is shown in

7

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.
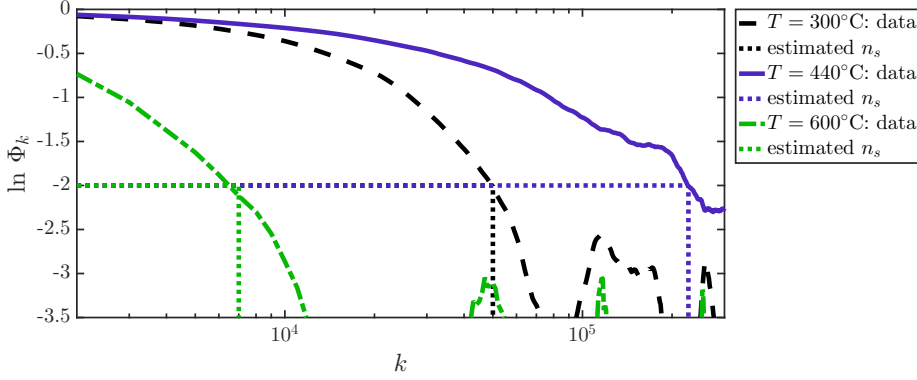

Figure 9: The logarirhm of the correlation function $\Phi_k(k)$ for three different temperatures. Dotted lines mark the estimated value of $n_s = k(\ln \Phi_k = -2)$.
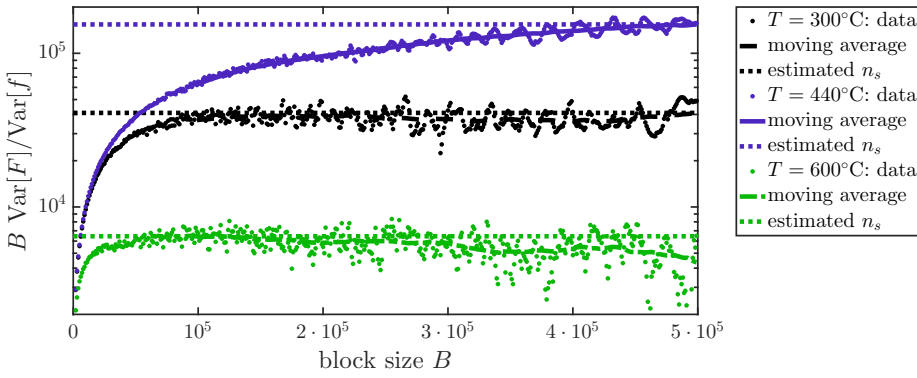

Figure 10: The statistical inefficiency determined with block averages for three different temperatures. Raw data is shown with dots, solid line show a moving average with 100 points, and the dotted lines show the estimated values of the statistical inefficiency.

Note in figures 9 and 10 that the statistical inefficiency is larger close to the phase transition at $T \approx 440\,°C$ than at the lower and higher temperatures $T = 300\,°C$ and $T = 600\,°C$. 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 consistent estimates of n_s, but the correlation function give larger fluctuations than the block average method. Moreover, that the statistical inefficiency diverges as $T \to 0\,K$. This is because very few changes in the lattice will be accepted at low temperatures, resulting in highly correlated data. At low temperatures, the equilibrium system is almost completely ordered, and therefore the uncertainty of the quantities U, P and r is still small at low temperatures; note the vanishing 2σ error bars in figures 5-8.

Concluding discussion

We study a binary alloy of 50 % copper and 50 % zinc. We compare semi-analytical results from mean-field theory with a Monte Carlo simulation using the Metropolis
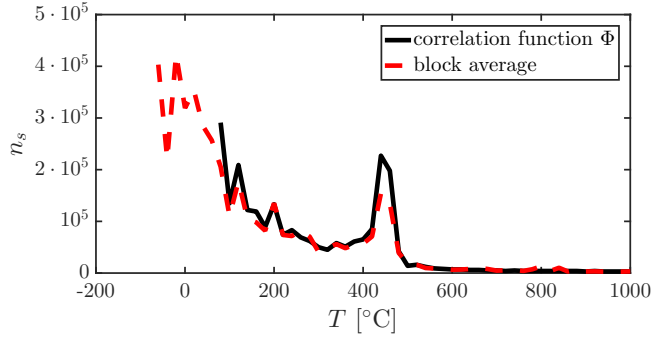
Figure 11: The statistical inefficiency n_s as a function of temperature using both the correlation function and block averages to determine n_s.

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 overestimates the critical temperature. Furthermore, it fails to explain the behavior of the system above this phase transition, as the simulations show a slower convergence to the high-temperature behavior. This results in a non-zero heat capacity at above-critical temperatures, as well as a non-zero short range order parameter r. We also determine the critical exponents associated with the order parameter P and the heat capacity from the simulations. Although these are uncertain, it is clear that they differ significantly from the mean field theory prediction.

Finally, we consider the statistical inefficiency. The block average method is consistent with the results using the correlation function, and we observe a peak in close to the phase transition, which is possibly connected to the longer correlation length of a system close to the phase transition.

9

A Source Code

A.1 Main program task 2: `main_T2.c`

```c
/*
  H2a, Task 2
*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "funcs.h"

#define Nc 10 //number of cells
#define N_neigh 8
#define degC_to_K 273.15
#define kB 8.61733e-5

/* Main program */
int main(){
  int N_Cu = Nc*Nc*Nc;
  int N_atoms = 2*N_Cu;
  int N_bonds = 8*N_Cu;
  double Etot, E_Var, r, P; // Macro parameters
  gsl_rng *q = init_random(); // initialize random number generator

  /* done for all saved steps: */
  int N_MCsteps = 1e7;
  int N_eq = 1e6;
  int N_eq_short = 5e5;
  double *E_equilibration = malloc(sizeof(double[N_eq]));
  double *P_equilibration = malloc(sizeof(double[N_eq]));
  double *E_production    = malloc(sizeof(double[N_MCsteps]));

  /* statistical inefficiency */
  int N_k     = 500;
  int N_skip = 1000; // k_Max = N_k * N_skip;
  double *phi   = malloc(sizeof(double[N_k]));
  double *var_F = malloc(sizeof(double[N_k]));

  /* set Temperature steps */
  double beta;
  double dT_small      = 2;
  double dT_large      = 10;
  double T_start       = -200;
  double T_end         = 1000;
  double T_start_fine = 410;
  double T_end_fine   = 460;
  int nT;
  double *T_degC = init_temps(&nT, dT_small, dT_large, T_start, T_end,
                  T_start_fine, T_end_fine);
  // save equilibration data and stat inefficiency at T%20 =0
  int T_save_step = 20;
  /* done for all temps: */
  double *E_mean        = malloc(sizeof(double[nT]));
  double *E_mean_approx = malloc(sizeof(double[nT]));
  double *E_sq_mean     = malloc(sizeof(double[nT]));
  double *P_mean        = malloc(sizeof(double[nT]));
  double *P_sq_mean     = malloc(sizeof(double[nT]));
  double *r_mean        = malloc(sizeof(double[nT]));
  double *r_sq_mean     = malloc(sizeof(double[nT]));

  /* allocate and initalize lattice and nearest neighbors */
  int *lattice = malloc(sizeof(int[N_atoms]));
  init_ordered_lattice(N_atoms, N_Cu, lattice);
  int (*nearest)[N_neigh] = malloc(sizeof(int[N_atoms][N_neigh]));
  init_nearestneighbor(Nc, nearest);

  /* initialize macro parameters */
  Etot = get_Etot(lattice, N_Cu, nearest);
  P = get_order_parameter(lattice, N_Cu);
  r = get_short_range_order_parameter(lattice, nearest, N_Cu);


  /* ******************** start simulation ********************* */
  for (int iT=0; iT<nT; iT++){
    /* Loop over all temperatures */
    printf("Now running T = %.0f degC\n",T_degC[iT]);
    beta = 1/(kB*(T_degC[iT] + degC_to_K));

    /* ******************* Equilibration run ****************** */
    if (iT!=0){// First run needs longer equlibration
      N_eq=N_eq_short;
    }
    /* Do the Monte Carlo stepping */
    for( int i=0; i<N_eq; i++){
      MC_step(&Etot, &r, &P, q, lattice, nearest, beta, N_Cu);
```

```c
84          // Save the energy `Etot` and orerparameter `P`
85          E_equilibration[i] = Etot;
86          P_equilibration[i]= P;
87        }
88        //Write the equlibration 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 calcuating 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`

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

11

```c
/********************* 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 tree 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 calcuating the austo-correlation in a data set. The
     rate at which the auto-correlation decay can be used to calcuate
     the statistical inefficiency in the data set.
     Formula:
       phi_k = (<f_{i+k}f_{i}> - <f_{i}>^2) / (<f_{i}^2> - <f_{i}>^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 differnt 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 <f_{i+k}f_{i}> 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       <f_{i+k}f_{i}> = (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 divifing 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 calcuating the variances of the blockaverages for `N_k`
132     different block sizes. This varaince can then be used to calcuate the
133     statistical inefficiency in the data set.
134   */
135   int block_size;
136   double Fj; // help vaiable, 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 elemts 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 varaince of F we use:
160         Var[F] = <F^2> - <F>^2 = <F^2> - <f>^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 clostly full calcualtion 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];
```

```c
184        // Used to clacluate 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 aer 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             contributtions.
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 contribtions 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 happes 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 producction 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;
```

```c
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 neares
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            nearest[i_atom][2] =k            + Nc*((j-1+Nc)%Nc) + Nc*Nc*i;
335            nearest[i_atom][3] =k            + Nc*((j-1+Nc)%Nc) + Nc*Nc*((i-1+Nc)%Nc)↩
                   ;
336            nearest[i_atom][4] =(k-1+Nc)%Nc + Nc*j             + Nc*Nc*i;
337            nearest[i_atom][5] =(k-1+Nc)%Nc + Nc*j             + Nc*Nc*((i-1+Nc)%Nc)↩
                   ;
338            nearest[i_atom][6] =(k-1+Nc)%Nc + Nc*((j-1+Nc)%Nc) + Nc*Nc*i;
339            nearest[i_atom][7] =(k-1+Nc)%Nc + Nc*((j-1+Nc)%Nc) + Nc*Nc*((i-1+Nc)%Nc)↩
                   ;
340          }
341        }
342      }
343    }
344
345    gsl_rng* init_random(){
346      /*
347         Initializes a GSL random nuber generator, and returns the pointer.
348      */
349      gsl_rng *q;
350      const  gsl_rng_type *rng_T;     // static  info  about  rngs
351      gsl_rng_env_setup ();           // setup  the  rngs
352      rng_T = gsl_rng_default;        // specify  default  rng
353      q = gsl_rng_alloc(rng_T);       // allocate  default  rng
354      gsl_rng_set(q,time(NULL));      // Initialize  rng
355      return q;
356    }
357
358
359    /************************ file I/O functions ***************************/
360    void write_equil_to_file(double T_degC, double *E_equilibration, int N_bonds,
361                double *P, int N_eq){
```

```c
362      /*
363        Writes the energy per bond `E_equilibration`/`N_bonds` and order
364        parameter `P`, at each Monte Carlo step during the equlibration 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, "%% T[degC]\t E_approx\t<E-E_approx>\t<(E-E_approx)^2>\↵
             tP\tr\n");
390      for (int iT=0; iT<nT; iT++){
391        fprintf(file_pointer, "%.2f\t%.8e\t%.8e\t%.8e\t%.8f\t%.8f\t %.8f\t%.8f \n",
392            T_degC[iT], E_mean_approx[iT], E_mean[iT], E_sq_mean[iT], P_mean[iT],
393            P_sq_mean[iT], r_mean[iT], r_sq_mean[iT]);
394      }
395      fclose(file_pointer);
396  }
397
398  void write_stat_inefficiency_to_file(double T_degC, double *phi, double *var_F,
399                      int N_k, int N_skip){
400      /*
401        Writes the auto-correlation `phi` and block varaiances `var_F` for each
402        tested temperature to file.
403      */
404      FILE *file_pointer;
405      char file_name[256];
406      sprintf(file_name,"../data/stat_inefficiency-T%d.tsv", (int) T_degC);
407      file_pointer = fopen(file_name, "w");
408      for (int i=0; i<N_k; i++){
409        fprintf(file_pointer, "%d\t%.8f\t%.8f \n", i*N_skip, phi[i],var_F[i]);
410      }
411      fclose(file_pointer);
412  }
```

B   Auxiliary

B.1   Makefile

```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

```matlab
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 = 1;
11 clc
12
13 Pmin = 0;
14 Pmax = 1;
15
16 E_CuCU = -.436;
17 E_ZnZn = -.113;
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 E_max = (E_CuCU + E_ZnZn)/2;
23
24 E0_bar=E0/Delta_E;
25 E_MFT=@(P) E0 - 2*P.^2*Delta_E;
26 E_MFT_bar=@(P) E0_bar - 2*P.^2;
27 dE_MFTdP =@(P) - 4*P*Delta_E;
28
29 F_MFT = @(P,Tbar) E_MFT_bar(P) + Tbar*(-2*log(2) + (1+P).*log(1+P)+(1-P).*log(1-↩
       P));
30 P_eq=@(Tbar)  fminbnd(@(P)F_MFT(P, Tbar), Pmin, Pmax, optimset('TolX',1e-9));
31
32 Tbar = linspace(0,3,1000)';
33 T_MFT=Tbar*Delta_E/kB;
34 T_MFT_degC = T_MFT - 273.15;
35 Peq = zeros(size(Tbar));
36 for iT = 1:numel(Tbar)
37     Peq(iT) = P_eq(Tbar(iT));
38 end
39
40 % plot P(T) and make a fit
41 figure(1);clf
42 plot(Tbar, Peq);hold on
43
44 dT=2-Tbar(Tbar<2);
45 Peq_nonzero = Peq(Tbar<2);
46
47 I_good = (dT<0.1);
48 log_dT = log(dT(I_good));
49 log_P  = log(Peq_nonzero(I_good));
50 A=[ones(size(log_dT)), log_dT]\log_P;
51 b      = exp(A(1));
52 alpha = A(2);
53 fprintf('alpha = %.3f\n', alpha)
54
55 P_approx = @(alpha,b,Tbar) b*(2-Tbar).^alpha;
56 plot(Tbar(Tbar<2),P_approx(alpha,b,Tbar(Tbar<2)),'k:')
57 xlabel('$\bar T$')
58 ylabel('$P$')
59 legend('$P$', 'fit $P \propto (2-\bar T)^\beta$')
60 ylim([0 1.3]);
61 if doSave; setFigureSize(gcf, 300, 600); end
62
63 % plot E_MFT and the fit
64 figure(2);clf
65 plot(Tbar,E_MFT(Peq)); hold on
66 plot(Tbar,E_MFT(P_approx(alpha,b,Tbar)),'k:')
67 xlabel('$\bar T$')
68 ylabel('$u$ [eV/cell]')
69 legend('$u_{\rm MFT}$', 'fit $P \propto (2-\bar T)^\beta$', 'location', '↩
       NorthWest');
70 ylim([-2.36 -2.3]);
71 if doSave; setFigureSize(gcf, 300, 600); end
72
73 figure(3);clf
74 C_MFT=diff(E_MFT(Peq))./diff(T_MFT);
75 plot(Tbar(1:end-1), C_MFT*1e3); hold on
76 C_approx=4*b^2*kB*alpha*(2-Tbar).^(2*alpha-1);
77 plot(Tbar(Tbar<2),1e3*C_approx(Tbar<2),'k:')
78 xlabel('$\bar T$')
79 ylabel('$C$ [meV K$^{-1}$/cell]')
80 legend('$C_{\rm MFT}$', 'fit $P \propto (2-\bar T)^\beta$', 'location', '↩
       NorthWest');
```

```matlab
81  ylim([0 0.3])
82  if doSave; setFigureSize(gcf, 300, 600); end
83
84  ImproveFigureCompPhys()
85  if doSave
86      saveas(1, '../figures/P_MFT.eps', 'epsc');
87      saveas(2, '../figures/E_MFT.eps', 'epsc');
88      saveas(3, '../figures/C_MFT.eps', 'epsc');
89  end
90
91
92  %% task 2: equilibration and statistical inefficiency
93  clc;
94  doSave = 1;
95  Ts=[-200:20:1000]';
96  TsToPlot = [300 440 600];
97  t_eq=0;
98
99  figure(1);clf;
100
101 for i=1:numel(TsToPlot)
102     data = load(sprintf('../data/E_equilibration-T%d.tsv',TsToPlot(i)));
103     E = data(:,1)*8; % convert from energy per bond to energy per cell
104     steps = 1:length(E);
105     plot(steps, E); hold on
106 end
107 legstr = strcat({'$T='}, num2str(TsToPlot), '^\circ$ C');
108 legend(legstr, 'location', 'NorthWest');
109 ylabel('$E$ [eV/cell]')
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) = max(filtereddata);
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, ns_block(i)*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 = ['$T=' num2str(TsToPlot(i)) '^\circ$C: '];
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
```

18

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

19

```matlab
254  ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'r'}');
255  legend('$P$', '$P\pm 2 \sigma$ (with $n_{s, \rm \Phi})$', '$P_{\rm MFT}$', '←
         Location', 'SouthWest');
256  ylabel('$P$ ')
257  axis tight
258
259  figure(4);clf;
260  plot(T_degC, r, 'r');hold on;
261  errorbar(Ts, r(ind), 2*r_std(ind).*sqrt(ns_Phi), '.k','linewidth', 1.5);hold on;
262  plot(T_degC, P.^2, '--',T_MFT_degC, Peq.^2, '-.');
263  ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'LINNEAGREEN','r'}');
264  legend('$r$', '$r\pm 2 \sigma$ (with $n_{s, \rm \Phi})$', '$P^2$','$r_{\rm MFT}$←
         ', 'Location', 'SouthWest');
265  ylabel('$r$ ')
266  axis tight
267  ImproveFigureCompPhys((2:4), 'linewidth', 2)
268
269  if doSave
270      for ifig = 1:4;
271          figure(ifig)
272          setFigureSize(gcf, 300, 600);
273          xlabel('$T$ [$^\circ$C]');
274          xlim([-200 1000])
275      end
276      ImproveFigureCompPhys(1:4);
277      saveas(1, '../figures/U.eps', 'epsc');
278      saveas(2, '../figures/C.eps', 'epsc');
279      saveas(3, '../figures/P.eps', 'epsc');
280      saveas(4, '../figures/r.eps', 'epsc');
281  end
282
283  %% test with critical exponents
284  Tcrit = 430;
285  dT=Tcrit-T_degC(T_degC<Tcrit);
286  P_nonzero = abs(P(T_degC<Tcrit));
287
288  I_good = (dT<30 & P_nonzero>0.4);
289  log_dT = log(dT(I_good));
290  log_P  = log(P_nonzero(I_good));
291  A=[ones(size(log_dT)), log_dT]\log_P;
292  b     = exp(A(1));
293  alpha = A(2);
294  fprintf('P: beta = %.3f\n', alpha)
295  P_approx = @(alpha,b,T) b*(Tcrit-T).^alpha;
296
297  %figure(5);clf;
298  %loglog(dT,P_nonzero) ; hold on;
299  %plot(dT, P_approx(alpha, b, Tcrit-dT), 'g')
300
301  figure(3)
302  Tvec = linspace(300,Tcrit);
303  plot(Tvec, P_approx(alpha, b, Tvec), ':k')
304  ImproveFigureCompPhys(gcf)
305
306
307
308  Cv_good = abs(Cv(T_degC<Tcrit));
309  I_good = (dT<150);
310  log_dT = log(dT(I_good));
311  log_C  = log(Cv_good(I_good));
312  A=[ones(size(log_dT)), log_dT]\log_C;
313  b     = exp(A(1));
314  alpha = A(2);
315  fprintf('Cv: alpha = %.3f\n', alpha)
316  C_approx = @(alpha,b,T) b*(Tcrit-T).^alpha;
317
318  %figure(6);clf;
319  %loglog(dT,Cv_good) ; hold on;
320  %plot(dT, C_approx(alpha, b, Tcrit-dT), 'g')
321
322  figure(2);
323  plot(Tvec, 1e3*C_approx(alpha, b, Tvec), ':r')
324  ImproveFigureCompPhys(gcf)
325  %% test U as a function of r
326  clf;
327  Ufunc = @(r) 4*(r+1)*E_CuZn + 2*(E_ZnZn+ E_CuCU) * (1-r);
328  plot(r, U, 'k',r, Ufunc(r), ':r')
329  ImproveFigureCompPhys(gcf)
```

C.2   Improve figure appearance: `ImproveFigureCompPhys.m`

```matlab
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:
```

```matlab
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
91      fig = figHs(iFig);
92
93      lineObjects = findall(fig, 'Type', 'line');
94      textObjects = findall(fig, 'Type', 'text');
95      axesObjects = findall(fig, 'Type', 'axes');
```

```
96      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         end
124
125         %%% change font sizes.
126         % Tick label size
127         xLim = axesObjects(iAx).XLim;
128         axesObjects(iAx).FontSize = tickLabelSize;
129         axesObjects(iAx).XLim = xLim;
130         %Change label size
131         axesObjects(iAx).XLabel.FontSize = labelSize;
132         axesObjects(iAx).YLabel.FontSize = labelSize;
133
134         %Change title size
135         axesObjects(iAx).Title.FontSize = titleSize;
136     end
137
138     %%% LINE APPEARANCE
139     %Change line thicknesses
140     set(lineObjects,'LineWidth',lineWidth);
141     set(contourObjects, 'LineWidth', lineWidth);
142     set(axesObjects, 'LineWidth',axesWidth)
143
144     % set interpreter: latex or tex
145     set(textObjects, 'interpreter', interpreter)
146     set(legObjects, 'Interpreter', interpreter)
147     set(axesObjects,'TickLabelInterpreter', interpreter);
148 end
149 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
```