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 №	Points	Avail. points
$\sum$		

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.

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

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

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 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

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

and the entropy of the Cu sub-lattice is

$$\begin{aligned} S'_{\mathrm{MFT}} =& k_{\mathrm{B}} \ln\!\left(\frac{N!}{[(P+1)N/2]!\,[(P-1)N/2]!}\right) \\ \approx& Nk_{\mathrm{B}} \ln(2) - k_{\mathrm{B}}\frac{N}{2}\Big[(1+P)\ln(1+P) + (1-P)\ln(1-P)\Big], \end{aligned} \tag{3}$$

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 ust the sum of its parts; hence

$$S_{\mathrm{MFT}} = 2Nk_{\mathrm{B}} \ln(2) - Nk_{\mathrm{B}}\Big[(P+1)\ln(1+P) + (1-P)\ln(1-P)\Big]. \tag{4}$$

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, $N^{(\mathrm{MFT})}_{\mathrm{CuCu}}$, 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

$$N^{(\mathrm{MFT})}_{\mathrm{CuCu}} = 8\frac{(1+P)N}{2}\,\frac{1-P}{2} = 2N(1-P^2). \tag{5}$$

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

$$N^{(\mathrm{MFT})}_{\mathrm{ZnZn}} = N^{(\mathrm{MFT})}_{\mathrm{CuCu}} = 2N(1-P^2). \tag{6}$$

[1]This 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:

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

The energy of the system is now given by

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

which can be simplified to

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

where

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

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

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

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

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

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

Results and discussion

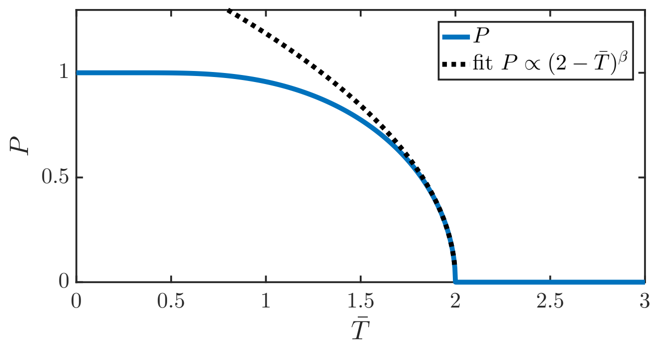

Figure 1: The mean field theory value of the order parameter, P, as a function of temperature, $\bar{T} = k_{\text{B}}T/\Delta E$. There is a clear critical temperature, $\bar{T}_{\text{crit}} = 2$ ($T_{\text{crit}} = 441$ K), above which P becomes constantly zero. Close below the critical temperature,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

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

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:

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

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.

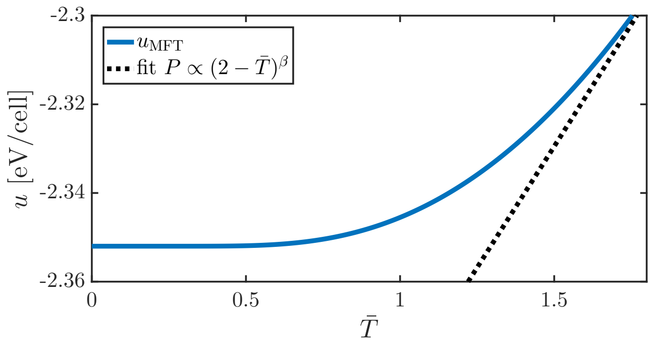

Figure 2: The mean field theory energy per unit cell, $u_{\text{MFT}} = U_{\text{MFT}}/N$, as a function of temperature, $\bar{T} = k_B T$. The energy rises from $u(\bar{T} = 0) = E_0 - 2\Delta E = -2.352\,\text{eV}$ to $u(\bar{T} = 0) = E_0 = -2.314\,\text{eV}$ per unit cell.

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

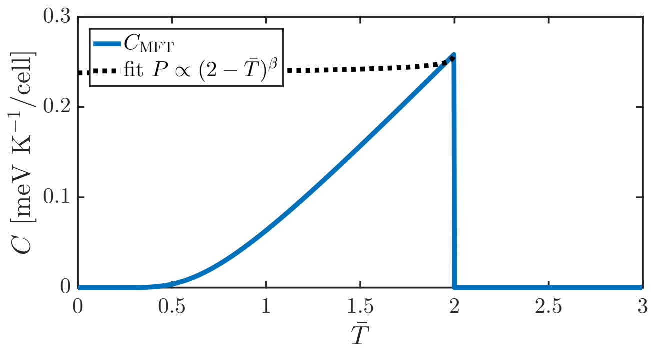

Figure 3: The mean field theory heat capacity, C_{MFT}, as function of temperature, $\bar{T} = k_B T/\Delta E$. The heat capacity rises until $\bar{T} = \bar{T}_{\text{crit}}$ to a maximum value of about $C^{(\text{max})}_{\text{MFT}} = 0.26\,\text{meV/K}$ per unit cell, above which C_{MFT} immediately 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 (12), we can easily show that

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

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.

Task 2: Ising model

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

$$\begin{aligned} E_{\mathrm{CuZn}} &= -294\,\mathrm{meV}, \\ E_{\mathrm{CuCu}} &= -436\,\mathrm{meV}, \\ E_{\mathrm{ZnZn}} &= -113\,\mathrm{meV}. \end{aligned} \tag{16}$$

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

$$\Delta E \leq 0, \tag{17}$$

or if

$$\exp[-\Delta E/(k_{\mathrm{B}}T)] > \xi, \tag{18}$$

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 $p \propto \exp[-E/(k_{\mathrm{B}}T)]$, and thus favor the most probable configurations.

Equilibration

To equilibrate the system, we started with an ordered system and performed $N_{\mathrm{eq,long}} = 10^6$ Monte Carlo steps to equilibrate the system at $T = -200\,°\mathrm{C}$. 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 $N_{\mathrm{eq,short}} = 5 \cdot 10^5$. 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 $T \approx 430\,°\mathrm{C}$. By plotting the energy per bond, i.e. $E/(8N_{\mathrm{Cu}})$, we can compare the energies to the binding energies in equation (16). We note that the energy per bond is in the range $E_{\mathrm{CuZn}} \leq E \leq E_{\mathrm{max}}$, where

$$E_{\mathrm{max}} \equiv \frac{1}{2}(E_{\mathrm{CuCu}} + E_{\mathrm{ZnZn}}) = -274.5\,\mathrm{meV}. \tag{19}$$

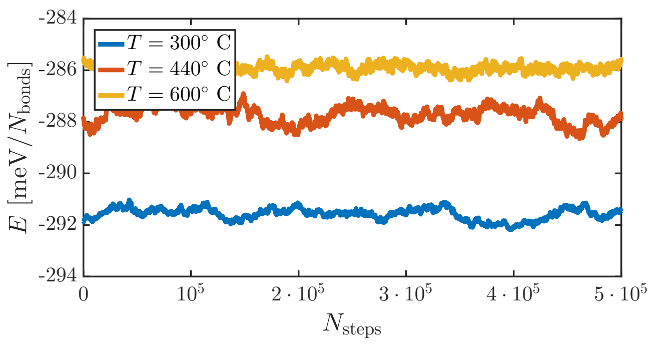

Figure 4: The energy normalized to the number of bonds in the system during the equilibration process.

Statistical properties

Figure 5 shows the equilibrium energy per 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 .

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.

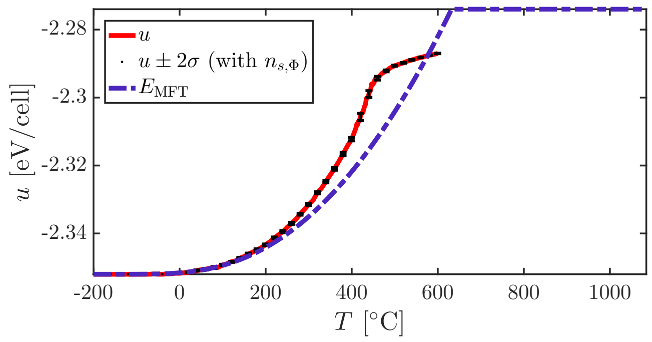

Figure 5: The average energy of the system normalized to the number of cells, as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dash-dotted blue: mean field theory prediction.

In mean field theory, the energy per unit 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 (19)). With a limited system size with 10 cells in each direction, we estimate that the maximum energy should be approximately

$$8(0.9 \cdot E_{\max} + 0.1E_{\mathrm{CuZn}}) \approx 2.284\,\mathrm{eV} \tag{20}$$

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

$$C = \frac{\partial u}{\partial T}, \tag{21}$$

or as the variance in the energy:

$$C = \frac{1}{k_B T^2}\left(\langle E^2\rangle - \langle E\rangle^2\right). \tag{22}$$

Since the latter method does not depend on the derivatives, it gives less noisy data. This is 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 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.

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

$$r = \frac{1}{4N}(q - 4N) \to \begin{cases} 1, & \text{perfect order} \\ 0, & \text{no order, homogeneous system} \\ -1, & \text{fully separated system} \end{cases} \tag{23}$$

In the mean field theory, $r = P^2$ by equation (7). 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 completely 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.

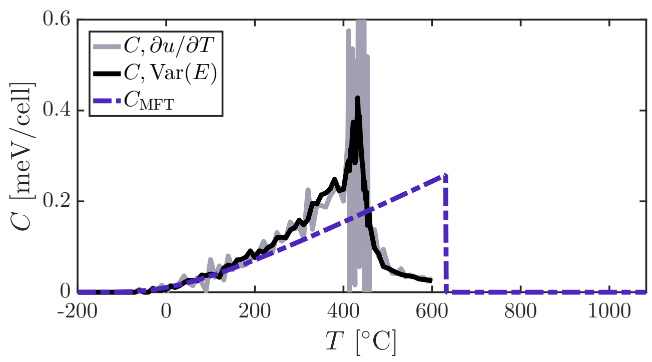

Figure 6: The specific heat of the system normalized to the number of cells, as a function of temperature. Solid gray: simulation using the derivative of U directly, black: simulation using the variance of E, dash-dotted blue: mean field theory prediction.

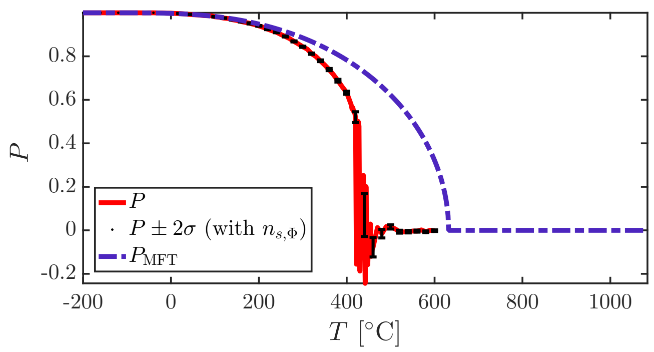

Figure 7: The order parameter P as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dash-dotted blue: mean field theory prediction.

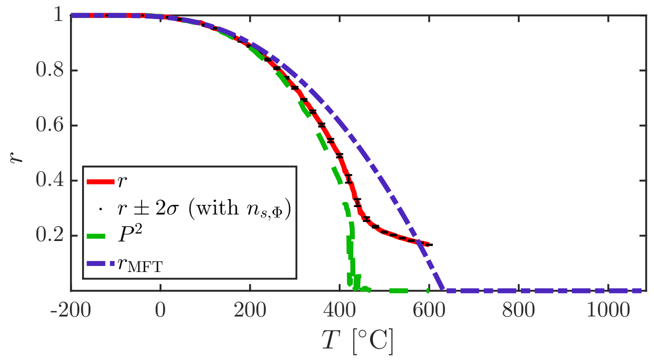

Figure 8: The short range order parameter as a function of temperature. Solid red: simulation, black: error bars at selected temperatures, dashed green: estimate $r \approx P^2$, dash-dotted blue: mean field theory prediction.

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:

$$I = \langle f \rangle \equiv \frac{1}{N} \sum_{i=1}^{N} f_i. \tag{24}$$

The variance is then given by

$$\mathrm{Var}[I] = \frac{n_s}{N}\mathrm{Var}[f], \quad \mathrm{Var}[f] = \langle f^2\rangle - \langle f\rangle^2, \tag{25}$$

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

$$\Phi_{k=n_s} = e^{-2} \approx 0.1, \quad \frac{\langle f_i f_{i+k}\rangle - \langle f\rangle^2}{\langle f^2\rangle - \langle f\rangle^2}, \tag{26}$$

or from block averaging

$$n_s = \lim_{B\to\infty} \frac{B\mathrm{Var}[F]}{\mathrm{Var}[f]}, \quad F_j = \frac{1}{B}\sum_{i=1}^{B} f_{i+(j-1)B}, \quad j \in [1, N_{\text{blocks}}]. \tag{27}$$

The two methods in equations (26) and (27) 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.

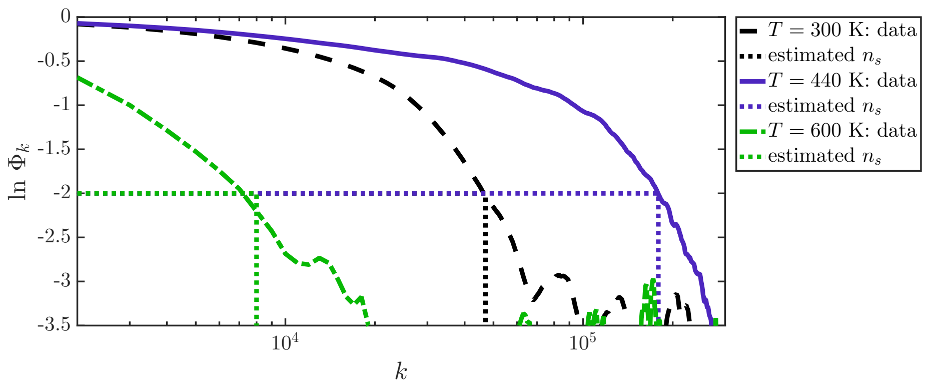

Figure 9: The logarirhm of the correlation function $\Phi_k(k)$ for three different temperatures. Dotted lines mark the estimated value of $n_s = k(\ln \Phi_k = -2)$.

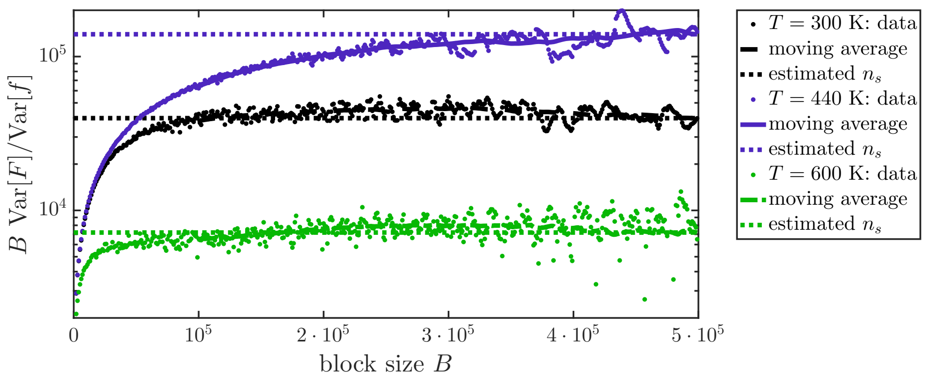

Figure 10: The statistical inefficiency determined with block averages for three different temperatures. Raw data is shown with dots, solid line show a moving average with 100 points, and the dotted lines show the estimated values of the statistical inefficiency.

Note in figures 9 and 10 that the statistical inefficiency is larger close to the phase transition at $T \approx 440\,^\circ$C than at the lower and higher temperatures $T = 300\,^\circ$C and

$T = 600\,°\mathrm{C}$. 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 $T \to 0\,\mathrm{K}$. 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.

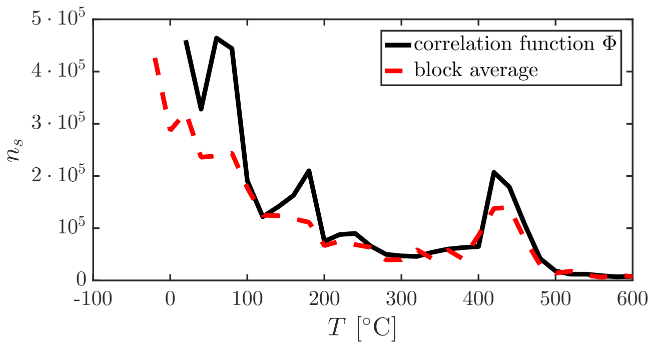

Figure 11: The statistical inefficiency n_s as a function of temperature using both the correlation function and block averages to determine n_s.

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

```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        = 600;
  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);
      // Save the energy `Etot` and orerparameter `P`
      E_equilibration[i] = Etot;
      P_equilibration[i]= P;
    }
    //Write the equlibration run to file
    if ( ((int)T_degC[iT]) % T_save_step==0){
      write_equil_to_file(T_degC[iT],
              E_equilibration, N_bonds, P_equilibration, N_eq);
    }

    /* ********************* Production run ********************** */
    /*
      The saved energies are shifted by this (semi-arbitrary) amount.
      This helps to increase the accuracy when calcuating the
      (needed for the heat capacity).
    */
    E_mean_approx[iT] = Etot;
    /* initialize at temperature[iT] */
    E_mean[iT] = 0;   E_sq_mean[iT] = 0;
    P_mean[iT] = 0;   P_sq_mean[iT] = 0;
    r_mean[iT] = 0;   r_sq_mean[iT]=0;

    /* Do the Monte Carlo stepping */
    for( int i=0; i<N_MCsteps; i++){
      MC_step( &Etot, &r, &P, q, lattice, nearest, beta, N_Cu);
      E_production[i] = Etot- E_mean_approx[iT];
      update_E_P_r(iT, Etot-E_mean_approx[iT], E_mean, E_sq_mean, P, P_mean,
           P_sq_mean, r, r_mean,r_sq_mean, lattice, nearest, N_Cu);
    }
    /* Divide by number of Monte Carlo steps to get average */
    E_mean[iT]    *= 1/(double)N_MCsteps;
    E_sq_mean[iT] *= 1/(double)N_MCsteps;
    P_mean[iT]    *= 1/(double)N_MCsteps;
    P_sq_mean[iT] *= 1/(double)N_MCsteps;
    r_mean[iT]    *= 1/(double)N_MCsteps;
    r_sq_mean[iT] *= 1/(double)N_MCsteps;

    /*
       We only calucluate the statistical inefficiency at some
       temperatures to save on runtime.
    */
    if ( ((int)T_degC[iT]) % T_save_step==0 ){//calc stat ineff
      // Calcualte the variance of the energy
      E_Var = E_sq_mean[iT] - E_mean[iT]*E_mean[iT];

      printf("Calculating statistical inefficiencies \n");
      //Calcualte the auto-correlation
      get_phi (phi, N_MCsteps, E_mean[iT], E_Var, E_production,N_k,N_skip);
      //Calcualte the block-average variance
      get_varF_block_average(var_F, N_MCsteps, E_mean[iT], E_Var,
                 E_production, N_k, N_skip);
      //Write the stat ineff to file
      write_stat_inefficiency_to_file(T_degC[iT], phi, var_F, N_k, N_skip);
    }//END if calc stat ineff
  }//END temp for

  //Write the results of the production run to file
  write_production(T_degC, nT, E_mean_approx, E_mean, E_sq_mean,
           P_mean, P_sq_mean, r_mean, r_sq_mean);



  //Don't forget to free all malloc's.
  free(nearest);          nearest = NULL;
  free(lattice);          lattice = NULL;
  free(E_equilibration);  E_equilibration = NULL;
  free(P_equilibration);  P_equilibration = NULL;
  free(E_mean);           E_mean = NULL;
  free(E_mean_approx);    E_mean_approx = NULL;
  free(E_sq_mean);        E_sq_mean = NULL;
  free(P_mean);           P_mean = NULL;
  free(P_sq_mean);        P_sq_mean = NULL;
  free(r_mean);           r_mean = NULL;
  free(r_sq_mean);        r_sq_mean = NULL;
  free(E_production);     E_production = NULL;
  free(phi);              phi = NULL;
  free(var_F);            var_F = NULL;
  free(T_degC);           T_degC = NULL;

  gsl_rng_free(q); // deallocate  rng
  return 0;
}
```

A.2 Misc functions : funcs.c

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

```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.
  */
  int N_terms_in_avg; // helper variable
  for (int k=0; k<N_k; k++){
    /*
      We loop over `k` in the formula above to get the auto-correlation
      at the diffrent times.
      `phi[k]` is used to hold intermediary values, and only becomes the
      auto-correlation at the last step in this loop.
    */
    phi[k] = 0;

    /*
      The number of terms in the sum to get <f_{i+k}f_{i}> must be such
      that i fulfills the relation:
         `(i+k)*N_skip < N_times`,
      which is equivalent to saying that
         `i < N_times/N_skip - k'.
    */
    N_terms_in_avg = N_times/N_skip - k;
    for (int i=0; i<N_terms_in_avg; i++){
      /*
    Add the products of the off-setted data points to get:
    sum_{i} f_{i+k}f_{i}
      */
      phi[k] += data[i*N_skip]*data[(i+k)*N_skip];
    }
    /*
      First:
      <f_{i+k}f_{i}> = (1/N_avg) sum_{i}^{N_avg} f_{i+k}f_{i},
      then we get the auto-correlation by subtracting `f_mean`^2
      and divifing by the variance.
    */
    phi[k] = (phi[k]/N_terms_in_avg - f_mean*f_mean)/f_var;
  }
}

void get_varF_block_average(double *var_F, int N_times, double f_mean,
                double f_var, double *data, int N_k, int N_skip){
  /*
    Function for calcuating the variances of the blockaverages for `N_k`
    different block sizes. This varaince can then be used to calcuate the
    statistical inefficiency in the data set.
  */
  int block_size;
  double Fj; // help vaiable, holding each block average
  int number_of_blocks; // The number of blocks depends on the block size

  for (int k=0; k<N_k; k++) { // block size loop
    /*
       For every block size, we need to loop over every block,
       and every element in that block
    */
    block_size = N_skip * (k+1);
    number_of_blocks = N_times/block_size;

    var_F[k] = 0; // start
    for (int j=0; j<number_of_blocks; j++) {// loop over all blocks
      /* For every block, we loop over all elements in it to take average. */
      Fj = 0; // reset to 0
      for (int i=0; i<block_size; i++) {// internal block loop
    /* Adding all elemts in the block to get the average */
        Fj += data[j*block_size + i];
      }
      Fj *= 1/(double)block_size; // divide by block size to get average
      var_F[k] += Fj*Fj; // will become the variance soon
    }
    /*
      To get the varaince of F we use:
        Var[F] = <F^2> - <F>^2 = <F^2> - <f>^2,
      where f is the data set the block averages were taken from.
     */
    var_F[k] = var_F[k]/number_of_blocks - f_mean*f_mean;
    var_F[k] *= block_size/f_var;
  } // end block size loop
}

/************** Monte Carlo step functions ***********************************/
void MC_step( double *Etot, double *r, double *P, gsl_rng *q,
              int *lattice, int (*nearest)[N_neigh], double beta, int N_Cu){
  /*
     Function that takes a Monte Carlo step and updates the lattice points,
     `Etot`, `r`, and `P` accordingly.
     It is important to utilize the _chage_ in energy, `r` and `P` when
     updating them as to not have to do a clostly full calcualtion of either
     every step in the Monte Carlo loop.
  */
  // Picks two random sites in the whole lattice.
  int i1 = (int)(2*N_Cu*gsl_rng_uniform(q));
  int i2 = (int)(2*N_Cu*gsl_rng_uniform(q));
  // saves the original values
  int old_1 = lattice[i1];
  int old_2 = lattice[i2];
  // Used to clacluate the change in `Etot` and `r`
  double dr = 0;
  double dE = 0;
  // We only need to do something if the two atoms aer different
  if (old_1 != old_2){
    for( int j=0; j<N_neigh; j++){
      /*
     The change in `Etot` and `r` are first _minus_ the old energies and `r`
     contributtions.
      */
      dE-= get_bond_E(lattice[i1], lattice[nearest[i1][j]])
      +get_bond_E(lattice[i2], lattice[nearest[i2][j]]);

      dr -= ((lattice[i1] + lattice[nearest[i1][j]]) == 1)
       +((lattice[i2] + lattice[nearest[i2][j]]) == 1);
    }
    /* Then we do the change of the two atoms */
    lattice[i1] = old_2;
    lattice[i2] = old_1;
    for( int j=0; j<N_neigh; j++){
      /*
     And _add_ the contribtions to `Etot` and `r` from the updated lattice.
      */
      dE+= +get_bond_E(lattice[i1], lattice[nearest[i1][j]])
    +get_bond_E(lattice[i2], lattice[nearest[i2][j]]);

      dr += ((lattice[i1] + lattice[nearest[i1][j]]) == 1)
       +((lattice[i2] + lattice[nearest[i2][j]]) == 1);
    }

    if ( (dE<=0)|| (exp(-beta * dE) >  gsl_rng_uniform(q)) ){
      /*
     The test is accepted if dE < 0 (accept immediately), OR
     otherwise it's accepted with a probability of `exp(-beta * dE)`
      */
      // Updates P
      if (i1 < N_Cu)
        *P += (double)(lattice[i1] - old_1 )/N_Cu *2;
      if (i2 < N_Cu)
        *P += (double)(lattice[i2] - old_2 )/N_Cu *2;
    }else{
      /*
     If the test failed, we change back to the old lattice configuration
     and no change happes to `Etot` or `r`
      */
      lattice[i1] = old_1;
      lattice[i2] = old_2;
      dE = 0;
      dr = 0;
    }// end if step is accepted
    *Etot += dE;
    *r += dr/(4*N_Cu);
  }// end if atoms are different
}

void update_E_P_r(int iT, double E_dev, double *E_mean, double *E_sq_mean,
          double P, double *P_mean, double *P_sq_mean,
          double r, double *r_mean, double *r_sq_mean,
          int *lattice, int (*nearest)[N_neigh], int N_Cu){
  /*
    Updates the macro parameters `E`, `P`, and `r`, as well as their squares.
    Runs in every Monte Carlo step during the producction run.
  */
  E_mean[iT] += E_dev;
  E_sq_mean[iT] += E_dev * E_dev;

  P_mean[iT] += P;
  P_sq_mean[iT] += P*P;

  r_mean[iT] += r;
  r_sq_mean[iT] += r*r;
}

/************************* initializing functions*****************************/
double * init_temps( int *nT, double dT_small, double dT_large,
           double T_start, double T_end, double T_start_fine,
           double T_end_fine){
  /*
    Creates an array `T_degC` with the temperatures to loop over in the main
    function, given the fine temperature step range and the sizes of the
    temperature steps.
  */
  *nT = (int) ((T_end_fine - T_start_fine)/dT_small
           +(T_start_fine-T_start + T_end-T_end_fine)/dT_large +1);
  double *T_degC = malloc(sizeof(double[*nT]));
  T_degC[0] = T_start;
  for (int iT=1; iT<*nT; iT++){ // loop over all temps
    if (T_degC[iT-1]>=T_start_fine && T_degC[iT-1]<T_end_fine){
      T_degC[iT] = T_degC[iT-1] + dT_small;
    }else{
      T_degC[iT] = T_degC[iT-1] + dT_large;
    }
  }
  return T_degC;
}


void init_ordered_lattice(int N_atoms, int N_Cu, int *lattice){
  /*
     Initialize lattice with Cu atoms (1) in Cu lattice (i=0:N_Cu-1)
     and Zn (0) in Zn lattice (i=N_cu:N_atoms-1):
  */
  for( int i=0; i<N_Cu; i++){
    lattice[i] = 1;
  }
  for( int i=N_Cu; i<N_atoms; i++){
    lattice[i] = 0;
  }
}

void init_random_lattice(int N_atoms, int N_Cu, int *lattice, gsl_rng *q){
  /*
    Initialize lattice with Cu and Zn atoms randomly distributed:
  */
  for( int i=0; i<N_Cu; i++){
    lattice[i] = (int)(gsl_rng_uniform(q)+0.5);
    lattice[i+N_Cu] = 1-lattice[i];
  }
}


void init_nearestneighbor(int Nc, int (*nearest)[N_neigh]){
  /*
    Create a matrix `nearest[i][j]` with the index of the `j`th neares
    neighbors to site `i`.
    N.B. Each site has `N_neigh` (8) nearest neighbors.
  */
  int i_atom;
  int N_Cu = Nc*Nc*Nc;
  for( int i=0; i<Nc; i++){
    for( int j=0; j<Nc; j++){
      for( int k=0; k<Nc; k++){
        i_atom = k + Nc*j + Nc*Nc*i;
        // k i j in one lattice <=> "k-0.5" "i-0.5" "j-0.5" in the other lattice
        // use mod to handle periodic boundary conditions
        nearest[i_atom][0] = k        + Nc*j          + Nc*Nc*i           +N_Cu;
        nearest[i_atom][1] = k        + Nc*j          + Nc*Nc*((i+1)%Nc)  +N_Cu;
        nearest[i_atom][2] = k        + Nc*((j+1)%Nc) + Nc*Nc*i           +N_Cu;
        nearest[i_atom][3] = k        + Nc*((j+1)%Nc) + Nc*Nc*((i+1)%Nc)  +N_Cu;
        nearest[i_atom][4] = (k+1)%Nc + Nc*j          + Nc*Nc*i           +N_Cu;
        nearest[i_atom][5] = (k+1)%Nc + Nc*j          + Nc*Nc*((i+1)%Nc)  +N_Cu;
        nearest[i_atom][6] = (k+1)%Nc + Nc*((j+1)%Nc) + Nc*Nc*i           +N_Cu;
        nearest[i_atom][7] = (k+1)%Nc + Nc*((j+1)%Nc) + Nc*Nc*((i+1)%Nc)  +N_Cu;

        // k i j in one lattice <=> "k+0.5" "i+0.5" "j+0.5" in the other lattice
        // use mod to handle periodic boundary conditions
        // note that mod([negative])<0 :
        i_atom += N_Cu;
        nearest[i_atom][0] =k           + Nc*j             + Nc*Nc*i;
        nearest[i_atom][1] =k           + Nc*j             + Nc*Nc*((i-1+Nc)%Nc);
        nearest[i_atom][2] =k           + Nc*((j-1+Nc)%Nc) + Nc*Nc*i;
        nearest[i_atom][3] =k           + Nc*((j-1+Nc)%Nc) + Nc*Nc*((i-1+Nc)%Nc);
        nearest[i_atom][4] =(k-1+Nc)%Nc + Nc*j             + Nc*Nc*i;
        nearest[i_atom][5] =(k-1+Nc)%Nc + Nc*j             + Nc*Nc*((i-1+Nc)%Nc);
        nearest[i_atom][6] =(k-1+Nc)%Nc + Nc*((j-1+Nc)%Nc) + Nc*Nc*i;
        nearest[i_atom][7] =(k-1+Nc)%Nc + Nc*((j-1+Nc)%Nc) + Nc*Nc*((i-1+Nc)%Nc);
      }
    }
  }
}

gsl_rng* init_random(){
  /*
    Initializes a GSL random nuber generator, and returns the pointer.
  */
  gsl_rng *q;
  const  gsl_rng_type *rng_T;    // static  info  about  rngs
  gsl_rng_env_setup ();          // setup  the  rngs
  rng_T = gsl_rng_default;       // specify  default  rng
  q = gsl_rng_alloc(rng_T);      // allocate  default  rng
  gsl_rng_set(q,time(NULL));     // Initialize  rng
  return q;
}


/************************* file I/O functions *********************************/
```

```c
void write_equil_to_file(double T_degC, double *E_equilibration, int N_bonds,
             double *P, int N_eq){
  /*
    Writes the energy per bond `E_equilibration`/`N_bonds` and order
    parameter `P`, at each Monte Carlo step during the equlibration runs.
  */
  FILE *file_pointer;
  char file_name[256];
  sprintf(file_name,"../data/E_equilibration-T%d.tsv", (int) T_degC);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_eq; i++){
    fprintf(file_pointer, "%.8f\t%.8f \n", E_equilibration[i]/N_bonds,P[i]);
  }
  fclose(file_pointer);
}

void write_production(double *T_degC, int nT, double *E_mean_approx,
              double *E_mean, double *E_sq_mean,
              double *P_mean, double *P_sq_mean,
              double *r_mean, double *r_sq_mean){
  /*
    Writes the macro parameters `E_mean_approx`, `E_mean`, `E_sq_mean`,
    `P_mean`, `P_sq_mean`, `r_mean`, and `r_sq_mean` for each temperature
    to file.
  */
  FILE *file_pointer;
  char file_name[256];
  sprintf(file_name,"../data/E_production.tsv");
  file_pointer = fopen(file_name, "w");
  fprintf(file_pointer, "%% T[degC]\t E_approx\t<E-E_approx>\t<(E-E_approx)^2>\↩
      tP\tr\n");
  for (int iT=0; iT<nT; iT++){
    fprintf(file_pointer, "%.2f\t%.8e\t%.8e\t%.8e\t%.8f\t%.8f\t %.8f\t%.8f \n",
        T_degC[iT], E_mean_approx[iT], E_mean[iT], E_sq_mean[iT], P_mean[iT],
        P_sq_mean[iT], r_mean[iT], r_sq_mean[iT]);
  }
  fclose(file_pointer);
}

void write_stat_inefficiency_to_file(double T_degC, double *phi, double *var_F,
                    int N_k, int N_skip){
  /*
    Writes the auto-correlation `phi` and block varaiances `var_F` for each
    tested temperature to file.
  */
  FILE *file_pointer;
  char file_name[256];
  sprintf(file_name,"../data/stat_inefficiency-T%d.tsv", (int) T_degC);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_k; i++){
    fprintf(file_pointer, "%d\t%.8f\t%.8f \n", i*N_skip, phi[i],var_F[i]);
  }
  fclose(file_pointer);
}
```

B Auxiliary

B.1 Makefile

```makefile

CC = gcc
CFLAGS = -O3 -Wall

LIBS = -lm -lgsl -lgslcblas

HEADERS = funcs.h
OBJECTS = funcs.o


%.o: %.c $(HEADERS)
    $(CC) -c -o $@ $< $(CFLAGS)

all: Task2



Task2: $(OBJECTS) main_T2.c
    $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

# $(PROGRAMS): $(OBJECTS) main_T1.c
#   $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

clean:
    rm -f *.o
    touch *.c
```

C MATLAB scripts

C.1 Task 1 and analysis scripts for Task 2

```matlab
%% initial

tmp = matlab.desktop.editor.getActive; %% cd to current path
cd(fileparts(tmp.Filename));
set(0,'DefaultFigureWindowStyle','docked');
warning('off','MATLAB:handle_graphics:exceptions:SceneNode'); % interpreter
GRAY = 0.7*[0.9 0.9 1];
kB = 8.61733e-5;
%% task 1: MFT
doSave = 1;
clc

Pmin = 0;
Pmax = 1;

E_CuCU = -.436;
E_ZnZn = -.113;
E_CuZn = -.294;

E0=2*(E_CuCU+E_ZnZn+2*E_CuZn);
Delta_E=(E_CuCU+E_ZnZn-2*E_CuZn);
E_max = (E_CuCU + E_ZnZn)/2;

E0_bar=E0/Delta_E;
E_MFT=@(P) E0 - 2*P.^2*Delta_E;
E_MFT_bar=@(P) E0_bar - 2*P.^2;
dE_MFTdP =@(P) - 4*P*Delta_E;

F_MFT = @(P,Tbar) E_MFT_bar(P) + Tbar*(-2*log(2) + (1+P).*log(1+P)+(1-P).*log(1-↩
    P));
P_eq=@(Tbar)  fminbnd(@(P)F_MFT(P, Tbar), Pmin, Pmax, optimset('TolX',1e-9));

Tbar = linspace(0,3,1000)';
T_MFT=Tbar*Delta_E/kB;
T_MFT_degC = T_MFT - 273.15;
Peq = zeros(size(Tbar));
for iT = 1:numel(Tbar)
    Peq(iT) = P_eq(Tbar(iT));
end

% plot P(T) and make a fit
figure(1);clf
plot(Tbar, Peq);hold on

dT=2-Tbar(Tbar<2);
Peq_nonzero = Peq(Tbar<2);

I_good = (dT<0.1);
log_dT = log(dT(I_good));
log_P  = log(Peq_nonzero(I_good));
A=[ones(size(log_dT)), log_dT]\log_P;
b     = exp(A(1));
alpha = A(2);
fprintf('alpha = %.3f\n', alpha)

P_approx = @(alpha,b,Tbar) b*(2-Tbar).^alpha;
plot(Tbar(Tbar<2),P_approx(alpha,b,Tbar(Tbar<2)),'k:')
xlabel('$\bar T$')
ylabel('$P$')
legend('$P$', 'fit $P \propto (2-\bar T)^\beta$')
ylim([0 1.3]);
if doSave; setFigureSize(gcf, 300, 600); end

% plot E_MFT and the fit
figure(2);clf
plot(Tbar,E_MFT(Peq)); hold on
plot(Tbar,E_MFT(P_approx(alpha,b,Tbar)),'k:')
xlabel('$\bar T$')
ylabel('$u$ [eV/cell]')
legend('$u_{\rm MFT}$', 'fit $P \propto (2-\bar T)^\beta$', 'location', '↩
    NorthWest');
ylim([-2.36 -2.3]);
if doSave; setFigureSize(gcf, 300, 600); end

figure(3);clf
C_MFT=diff(E_MFT(Peq))./diff(T_MFT);
plot(Tbar(1:end-1), C_MFT*1e3); hold on
C_approx=4*b^2*kB*alpha*(2-Tbar).^(2*alpha-1);
```

```
plot(Tbar(Tbar<2),1e3*C_approx(Tbar<2),'k:')
xlabel('$\bar T$')
ylabel('$C$ [meV K$^{-1}$/cell]')
legend('$C_{\rm MFT}$', 'fit $P \propto (2-\bar T)^\beta$', 'location', '↩
    NorthWest');
ylim([0 0.3])
if doSave; setFigureSize(gcf, 300, 600); end

ImproveFigureCompPhys()
if doSave
    saveas(1, '../figures/P_MFT.eps', 'epsc');
    saveas(2, '../figures/E_MFT.eps', 'epsc');
    saveas(3, '../figures/C_MFT.eps', 'epsc');
end


%% task 2: equilibration and statistical inefficiency
clc;
doSave = 1;
Ts=[-200:20:600]';
TsToPlot = [300 440 600]';
t_eq=0;

figure(1);clf;

for i=1:numel(TsToPlot)
    data = load(sprintf('../data/E_equilibration-T%d.tsv',TsToPlot(i)));
    E = data(:,1);
    steps = 1:length(E);
    plot(steps, E*1000); hold on
end
legstr = strcat({'$T='}, num2str(TsToPlot), '^\circ$ C');
legend(legstr, 'location', 'NorthWest');
ylabel('$E$ [meV/$N_{\rm bonds}$]')
xlabel('$N_{\rm steps}$')
ax = gca;
ax.XTickLabel = {'0', '$10^5$', '$2\cdot 10^5$','$3\cdot 10^5$','$4\cdot 10^5$',↩
    '$5\cdot 10^5$'}';

ImproveFigureCompPhys(1)

figure(3); clf;figure(2); clf;
[ns_Phi,ns_block] = deal(nan(size(Ts)));
Nskip = 10; % did not use all k's when calculating block averages
N_avg = 100; % moving average
for i=1:numel(Ts)
    data = load(sprintf('../data/stat_inefficiency-T%d.tsv',Ts(i)));
    k = data(:,1);
    block_size = k+Nskip;
    phi = data(:,2);
    VarF_norm = data(:,3);
    kstar = k(find(log(phi)<-2, 1, 'first'));
    if ~isempty(kstar)
        ns_Phi(i) = kstar;
    end

    filtereddata = movmean(VarF_norm,N_avg);
    ns_block(i) = filtereddata(end);

    if any(Ts(i) == TsToPlot)
        figure(2)

        semilogx(k, log(phi));hold on;
        plot([0.1 kstar kstar], [-2 -2 -6],':k')

        figure(3)
        semilogy(block_size, VarF_norm, '.'); hold on;
        plot(block_size(N_avg:end), filtereddata(N_avg:end));
        plot(block_size, filtereddata(end)*ones(size(block_size)), ':k');

    end
end

figure(4); clf;
plot(Ts, ns_Phi, 'k',Ts, ns_block, '--r')
ax = gca;
ax.YTickLabel = {'0', '$10^5$', '$2\cdot 10^5$','$3\cdot 10^5$','$4\cdot 10^5$',↩
    '$5\cdot 10^5$'}';
ylabel('$n_s$');
legend('correlation function $\Phi$', 'block average');
xlabel('$T$ [$^\circ$C]');
ImproveFigureCompPhys(gcf)

legs_Phi = cell(6,1);
legs_block = cell(9,1);
for i = 1:numel(TsToPlot)
    tt = ['$T=' num2str(TsToPlot(i)) '$ K: '];
    legs_Phi{1 + 2*(i-1)} = [tt 'data'];
    legs_Phi{2 + 2*(i-1)} = 'estimated $n_s$';
    legs_block{1 + 3*(i-1)} = [tt 'data'];
    legs_block{2 + 3*(i-1)} = 'moving average';
```

```matlab
    legs_block{3 + 3*(i-1)} = 'estimated $n_s$';
end

figure(2);

legend(legs_Phi, 'location', 'northeastoutside');
xlabel('$k$'); ylabel('ln $\Phi_k$');
ylim([-3.5 0]);
xlim([2e3 3e5])
figure(3);
ax = gca;
[ax.Children(:).MarkerSize] = deal(12);
legend(legs_block, 'location', 'northeastOutSide');
xlabel('block size $B$');
ylabel('$B$ Var[$F$]/Var[$f$]');
ylim([2e3 2e5])
ax = gca;
ax.XTickLabel = {'0', '$10^5$', '$2\cdot 10^5$','$3\cdot 10^5$','$4\cdot 10^5$',↩
    '$5\cdot 10^5$'};

ImproveFigureCompPhys(2, 'LineColor', {'LINNEAGREEN','LINNEAGREEN','GERIBLUE','↩
    GERIBLUE', 'k', 'k'},...
    'LineStyle', {':','-.',':','-', ':', '--'})
ImproveFigureCompPhys(3, 'LineColor', {'LINNEAGREEN','LINNEAGREEN','LINNEAGREEN'↩
    ,...
    'GERIBLUE','GERIBLUE','GERIBLUE', 'k', 'k', 'k'},...
    'LineStyle', {':','-.','none',':','-','none', ':', '--','none'});
if doSave
    figure(1);
    setFigureSize(gcf, 300, 600);
    saveas(gcf, '../figures/equilibration.eps', 'epsc');
    figure(2);
    setFigureSize(gcf, 350, 900);
    saveas(gcf, '../figures/stat_inefficiency_Phi.eps', 'epsc');
    figure(3);
    setFigureSize(gcf, 350, 900);
    saveas(gcf, '../figures/stat_inefficiency_block.eps', 'epsc');
    figure(4);
    setFigureSize(gcf, 300, 600);
    saveas(gcf, '../figures/stat_inefficiency_both.eps', 'epsc');
end

%% task 2: U, C, P and r

doSave = 1;

data = load('../data/E_production.tsv');
T_degC = data(:,1);
N_Cu = 1e3;
N_timeSteps = 1e7;

Emean_approx = data(:,2)/N_Cu; % divide by N_Cu to get energy and Cv per cell
Emean_shifted = data(:,3)/N_Cu;
E_sq_mean_shifted = data(:,4)/N_Cu^2;

E_Var = (E_sq_mean_shifted - Emean_shifted.^2);

Cv = 1./(kB * (T_degC+273.15).^2).*E_Var*N_Cu;
U = (Emean_shifted + Emean_approx);
U_std = sqrt(E_Var/N_timeSteps);
P = data(:,5);
P_std = sqrt((data(:,6)-P.^2)/N_timeSteps); % without ns so far
r = data(:,7);
r_std = sqrt((data(:,8)- r.^2)/N_timeSteps);

ind = zeros(size(Ts));
for i = 1:numel(Ts)
    ind(i) = find(Ts(i) == T_degC);
end

figure(1);clf;
plot(T_degC, U); hold on;
errorbar(Ts, U(ind), 2*U_std(ind).*sqrt(ns_Phi), '.k','linewidth', 2.5); hold on↩
    ;
plot(T_MFT_degC, E_MFT(Peq), '-.'); hold on
ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'r'});
legend('$u$', '$u\pm 2 \sigma$ (with $n_{s, \rm \Phi})$', '$E_{\rm MFT}$', '↩
    Location', 'NorthWest');
ylabel('$u$ [eV/cell]')
axis tight

figure(2); clf;
plot(T_degC(2:end), 1e3*diff(U)./diff(T_degC)); hold on;
plot(T_degC, 1e3*Cv);
plot(T_MFT_degC(1:end-1), 1e3*C_MFT, '-.');
ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'k',GRAY});
legend('$C, {\partial u/ \partial T}$','$C, {\rm Var}(E)$', '$C_{\rm MFT}$', '↩
    Location', 'NorthWest');
ylabel('$C$ [meV/cell]')
ylim([0 0.6])

```

```
figure(3);clf;
plot(T_degC, P, 'r'); hold on;
errorbar(Ts, P(ind), 2*P_std(ind).*sqrt(ns_Phi), '.k', 'linewidth', 2.5); hold ↩
    on;
plot(T_MFT_degC, Peq, '-.k');
ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'r'});
legend('$P$', '$P\pm 2 \sigma$ (with $n_{s, \rm \Phi})$', '$P_{\rm MFT}$', '↩
    Location', 'SouthWest');
ylabel('$P$ ')
axis tight

figure(4);clf;
plot(T_degC, r, 'r');hold on;
errorbar(Ts, r(ind), 2*r_std(ind).*sqrt(ns_Phi), '.k','linewidth', 1.5);hold on;
plot(T_degC, P.^2, '--',T_MFT_degC, Peq.^2, '-.');
ImproveFigureCompPhys(gcf, 'LineColor', {'GERIBLUE', 'LINNEAGREEN','r'});
legend('$r$', '$r\pm 2 \sigma$ (with $n_{s, \rm \Phi})$', '$P^2$','$r_{\rm MFT}$↩
    ', 'Location', 'SouthWest');
ylabel('$r$ ')
axis tight
ImproveFigureCompPhys((2:4), 'linewidth', 2)

if doSave
    for ifig = 1:4;
        figure(ifig)
        setFigureSize(gcf, 300, 600);
        xlabel('$T$ [$^\circ$C]');
        xlim([-200 Inf])
    end
    ImproveFigureCompPhys(1:4);
    saveas(1, '../figures/U.eps', 'epsc');
    saveas(2, '../figures/C.eps', 'epsc');
    saveas(3, '../figures/P.eps', 'epsc');
    saveas(4, '../figures/r.eps', 'epsc');
end

%% test with critical exponents
Tcrit = 430;
dT=Tcrit-T_degC(T_degC<Tcrit);
P_nonzero = abs(P(T_degC<Tcrit));

I_good = (dT<30 & P_nonzero>0.4);
log_dT = log(dT(I_good));
log_P  = log(P_nonzero(I_good));
A=[ones(size(log_dT)), log_dT]\log_P;
b      = exp(A(1));
alpha = A(2);
fprintf('P: alpha = %.3f\n', alpha)
P_approx = @(alpha,b,T) b*(Tcrit-T).^alpha;

%figure(5);clf;
%loglog(dT,P_nonzero) ; hold on;
%plot(dT, P_approx(alpha, b, Tcrit-dT), 'g')

figure(3)
Tvec = linspace(300,Tcrit);
plot(Tvec, P_approx(alpha, b, Tvec), ':k')
ImproveFigureCompPhys(gcf)



Cv_good = abs(Cv(T_degC<Tcrit));
I_good = (dT<150);
log_dT = log(dT(I_good));
log_C  = log(Cv_good(I_good));
A=[ones(size(log_dT)), log_dT]\log_C;
b      = exp(A(1));
alpha = A(2);
fprintf('Cv: alpha = %.3f\n', alpha)
C_approx = @(alpha,b,T) b*(Tcrit-T).^alpha;

%figure(6);clf;
%loglog(dT,Cv_good) ; hold on;
%plot(dT, C_approx(alpha, b, Tcrit-dT), 'g')

figure(2);
plot(Tvec, 1e3*C_approx(alpha, b, Tvec), ':r')
ImproveFigureCompPhys(gcf)
%%
clf;
Ufunc = @(r) 4*(r+1)*E_CuZn + 1.93 * (E_ZnZn+ E_CuCU) * (1-r);
plot(r, U, 'k',r, Ufunc(r))
```

C.2 Improve figure appearance: ImproveFigureCompPhys.m

```
function ImproveFigureCompPhys(varargin)
```

```matlab
%ImproveFigureCompPhys Improves the figures of supplied handles
%  Input:
% - none (improve all figures) or handles to figures to improve
% - optional:
%       LineWidth  int
%       LineStyle  column vector cell, e.g. {'-','--'}',
%       LineColor  column vector cell, e.g. {'k',[0 1 1], 'MYBLUE'}'
%                            colors: MYBLUE,MYORANGE,MYGREEN,MYPURPLE, MYYELLOW,
%                            MYLIGHTBLUE, MYRED
%       Marker column vector cell, e.g. {'.', 'o', 'x'}'

% ImproveFigure was originally written by Adam Stahl, but has been heavily
% modified by Linnea Hesslow


%%% Handle inputs
% If no inputs or if the first argument is a string (a property rather than
% a handle), use all open figures
if nargin == 0 || ischar(varargin{1})
    %Get all open figures
    figHs = findobj('Type','figure');
    nFigs = length(figHs);
else
    % Check the supplied figure handles
    figHs = varargin{1};
    figHs = figHs(ishandle(figHs) == 1); %Keep only those handles that are ↩
         proper graphics handles
    nFigs = length(figHs);
end

% Define desired properties
titleSize = 24;
interpreter = 'latex';
lineWidth = 4;
axesWidth = 1.5;
labelSize = 22;
textSize = 20;
legTextSize = 18;
tickLabelSize = 18;
LineColor = {};
LineStyle = {};
Marker = {};

% define colors
co =  [ 0    0.4470    0.7410
    0.8500    0.3250    0.0980
    0.9290    0.6940    0.1250
    0.4940    0.1840    0.5560
    0.4660    0.6740    0.1880
    0.3010    0.7450    0.9330
    0.6350    0.0780    0.1840 ];
colors = struct('MYBLUE', co(1,:),...
    'MYORANGE', co(2,:),...
    'MYYELLOW', co(3,:),...
    'MYPURPLE', co(4,:),...
    'MYGREEN', co(5,:),...
    'MYLIGHTBLUE', co(6,:),...
    'MYRED',co(7,:),...
    'GERIBLUE', [0.3000    0.1500    0.7500],...
    'GERIRED', [1.0000    0.2500    0.1500],...
    'GERIYELLOW', [0.9000    0.7500    0.1000],...
    'LIGHTGREEN', [0.4    0.85    0.4],...
    'LINNEAGREEN', [7 184 4]/255);

% Loop through the supplied arguments and check for properties to set.
for i = 1:nargin
    if ischar(varargin{i})
        switch lower(varargin{i})   %Compare lower case strings
            case 'linewidth'
                lineWidth = varargin{i+1};
            case 'linestyle'
                LineStyle = varargin{i+1};
            case 'linecolor'
                LineColor = varargin{i+1};
                for iLineColor = 1:numel(LineColor)
                    if isfield(colors, LineColor{iLineColor})
                        LineColor{iLineColor} = colors.(LineColor{iLineColor});
                    end
                end
            case 'marker'
                Marker = varargin{i+1};
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% Improve the figure(s)

for iFig = 1:nFigs

    fig = figHs(iFig);

    lineObjects = findall(fig, 'Type', 'line');
    textObjects = findall(fig, 'Type', 'text');
    axesObjects = findall(fig, 'Type', 'axes');
    legObjects =  findall(fig, 'Type', 'legend');
    contourObjects = findall(fig,'Type','contour'); % not counted as lines

    %%% TEXT APPEARANCE: first set all to textSize and then change the ones
    %%% that need to be changed again

    %Change size of any text objects in the plot
    set(textObjects,'FontSize',textSize);
    set(legObjects,'FontSize',legTextSize);

    %%% FIX LINESTYLE, COLOR ETC. FOR EACH PLOT SEPARATELY
    for iAx =  1:numel(axesObjects)
        lineObjInAx = findall(axesObjects(iAx), 'Type', 'line');

        %set line style and color style (only works if all figs have some
        %number of line plots..)
        if ~isempty(LineStyle)
            set(lineObjInAx, {'LineStyle'}, LineStyle)
            set(contourObjects, {'LineStyle'}, LineStyle); %%%%%%
        end
        if ~isempty(LineColor)
            set(lineObjInAx, {'Color'}, LineColor)
            set(contourObjects, {'LineColor'}, LineColor); %%%%%%
        end
        if ~isempty(Marker)
            set(lineObjInAx, {'Marker'}, Marker)
            set(lineObjInAx, {'Markersize'}, num2cell(10+22*strcmp(Marker, '.'))↩
                )
        end

        %%% change font sizes.
        % Tick label size
        xLim = axesObjects(iAx).XLim;
        axesObjects(iAx).FontSize = tickLabelSize;
        axesObjects(iAx).XLim = xLim;
        %Change label size
        axesObjects(iAx).XLabel.FontSize = labelSize;
        axesObjects(iAx).YLabel.FontSize = labelSize;

        %Change title size
        axesObjects(iAx).Title.FontSize = titleSize;
    end

    %%% LINE APPEARANCE
    %Change line thicknesses
    set(lineObjects,'LineWidth',lineWidth);
    set(contourObjects, 'LineWidth', lineWidth);
    set(axesObjects, 'LineWidth',axesWidth)

    % set interpreter: latex or tex
    set(textObjects, 'interpreter', interpreter)
    set(legObjects, 'Interpreter', interpreter)
    set(axesObjects,'TickLabelInterpreter', interpreter);
end
end
```

C.3 Change size of figures: `setFigureSize.m`

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