Computational Physics Lectures: Statistical physics and the Ising Model

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Oct 28, 2020

Ensembles

In statistical physics the concept of an ensemble is one of the cornerstones in the definition of thermodynamical quantities. An ensemble is a collection of microphysics systems from which we derive expectations values and thermodynamical properties related to experiment. As an example, the specific heat (which is a measurable quantity in the laboratory) of a system of infinitely many particles, can be derived from the basic interactions between the microscopic constituents. The latter can span from electrons to atoms and molecules or a system of classical spins. All these microscopic constituents interact via a well-defined interaction. We say therefore that statistical physics bridges the gap between the microscopic world and the macroscopic world. Thermodynamical quantities such as the specific heat or net magnetization of a system can all be derived from a microscopic theory.

Famous Ensembles

The table lists the most used ensembles in statistical physics together with frequently arising extensive (depend on the size of the systems such as the number of particles) and intensive variables (apply to all components of a system), in addition to associated potentials.

	Microcanonical	Canonical	Grand Canonical	Pressure canonical
Exchange of heat with the environment	no	yes	yes	yes
Exchange of particles with the environemt	no	no	yes	no
Thermodynamical parameters	$V, \mathcal{M}, \mathcal{D}$ E N	$V, \mathcal{M}, \mathcal{D}$ T N	$V, \mathcal{M}, \mathcal{D}$ T μ	$P, \mathcal{H}, \mathcal{E}$ T N
Potential	Entropy N	Helmholtz N	PV μ	Gibbs N
Energy	Internal N	Internal N	Internal μ	Enthalpy N

Canonical Ensemble

One of the most used ensembles is the canonical one, which is related to the microcanonical ensemble via a Legendre transformation. The temperature is an intensive variable in this ensemble whereas the energy follows as an expectation value. In order to calculate expectation values such as the mean energy $\langle E \rangle$ at a given temperature, we need a probability distribution. It is given by the Boltzmann distribution

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z}$$

with $\beta = 1/k_B T$ being the inverse temperature, k_B is the Boltzmann constant, E_i is the energy of a microstate i while Z is the partition function for the canonical ensemble defined as

The partition function is a normalization constant

In the canonical ensemble the partition function is

$$Z = \sum_{i=1}^{M} e^{-\beta E_i},$$

where the sum extends over all microstates M.

Helmoltz free energy, what does it mean?

The potential of interest in this case is Helmholtz' free energy. It relates the expectation value of the energy at a given temperatur T to the entropy at the same temperature via

$$F = -k_B T lnZ = \langle E \rangle - TS.$$

Helmholtz' free energy expresses the struggle between two important principles in physics, namely the strive towards an energy minimum and the drive towards higher entropy as the temperature increases. A higher entropy may be interpreted as a larger degree of disorder. When equilibrium is reached at a given temperature, we have a balance between these two principles. The numerical expression is Helmholtz' free energy.

Thermodynamical quantities

In the canonical ensemble the entropy is given by

$$S = k_B lnZ + k_B T \left(\frac{\partial lnZ}{\partial T} \right)_{N,V},$$

and the pressure by

$$p = k_B T \left(\frac{\partial lnZ}{\partial V} \right)_{N,T}.$$

Similarly we can compute the chemical potential as

$$\mu = -k_B T \left(\frac{\partial lnZ}{\partial N} \right)_{V,T}.$$

Thermodynamical quantities, the energy in the canonical ensemble

For a system described by the canonical ensemble, the energy is an expectation value since we allow energy to be exchanged with the surroundings (a heat bath with temperature T).

This expectation value, the mean energy, can be calculated using

$$\langle E \rangle = k_B T^2 \left(\frac{\partial lnZ}{\partial T} \right)_{V,N}$$

or using the probability distribution P_i as

$$\langle E \rangle = \sum_{i=1}^{M} E_i P_i(\beta) = \frac{1}{Z} \sum_{i=1}^{M} E_i e^{-\beta E_i}.$$

Energy and specific heat in the canonical ensemble

The energy is proportional to the first derivative of the potential, Helmholtz' free energy. The corresponding variance is defined as

$$\sigma_E^2 = \langle E^2 \rangle - \langle E \rangle^2 = \frac{1}{Z}\sum_{i=1}^{M} E_i^2 e^{-\beta E_i} - \left(\frac{1}{Z}\sum_{i=1}^{M} E_i e^{-\beta E_i} \right)^2 .$$

If we divide the latter quantity with kT^2 we obtain the specific heat at constant volume

$$C_V = \frac{1}{k_B T^2}\left(\langle E^2 \rangle - \langle E \rangle^2 \right),$$

which again can be related to the second derivative of Helmholtz' free energy.

Magnetic moments and susceptibility in the canonical ensemble

Using the same prescription, we can also evaluate the mean magnetization through

$$\langle \mathcal{M} \rangle = \sum_{i}^{M} \mathcal{M}_i P_i(\beta) = \frac{1}{Z}\sum_{i}^{M} \mathcal{M}_i e^{-\beta E_i},$$

and the corresponding variance

$$\sigma_{\mathcal{M}}^2 = \langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 = \frac{1}{Z}\sum_{i=1}^{M} \mathcal{M}_i^2 e^{-\beta E_i} - \left(\frac{1}{Z}\sum_{i=1}^{M} \mathcal{M}_i e^{-\beta E_i} \right)^2 .$$

This quantity defines also the susceptibility χ

$$\chi = \frac{1}{k_B T}\left(\langle \mathcal{M}^2 \rangle - \langle \mathcal{M} \rangle^2 \right).$$

Our model, the Ising model in one and two dimensions

The model we will employ in our studies of phase transitions at finite temperature for magnetic systems is the so-called Ising model. In its simplest form the energy is expressed as

$$E = -J \sum_{<kl>}^{N} s_k s_l - \mathcal{B}\sum_{k}^{N} s_k,$$

with $s_k = \pm 1$, N is the total number of spins, J is a coupling constant expressing the strength of the interaction between neighboring spins and $\mathcal{B}$ is an external magnetic field interacting with the magnetic moment set up by the spins.

The symbol $< kl >$ indicates that we sum over nearest neighbors only. Notice that for $J > 0$ it is energetically favorable for neighboring spins to be aligned. This feature leads to, at low enough temperatures, a cooperative phenomenon called spontaneous magnetization. That is, through interactions between nearest neighbors, a given magnetic moment can influence the alignment of spins that are separated from the given spin by a macroscopic distance. These long range correlations between spins are associated with a long-range order in which the lattice has a net magnetization in the absence of a magnetic field.

Boltzmann distribution

In order to calculate expectation values such as the mean energy $\langle E \rangle$ or magnetization $\langle \mathcal{M} \rangle$ in statistical physics at a given temperature, we need a probability distribution

$$P_i(\beta) = \frac{e^{-\beta E_i}}{Z}$$

with $\beta = 1/kT$ being the inverse temperature, k the Boltzmann constant, E_i is the energy of a state i while Z is the partition function for the canonical ensemble defined as

$$Z = \sum_{i=1}^{M} e^{-\beta E_i},$$

where the sum extends over all microstates M. P_i expresses the probability of finding the system in a given configuration i.

Energy for a specific configuration

The energy for a specific configuration i is given by

$$E_i = -J \sum_{<kl>}^{N} s_k s_l.$$

Configurations

To better understand what is meant with a configuration, consider first the case of the one-dimensional Ising model with $\mathcal{B} = 0$. In general, a given configuration of N spins in one dimension may look like

$\uparrow$	$\uparrow$	$\uparrow$	$\dots$	$\uparrow$	$\downarrow$	$\uparrow$	$\dots$	$\uparrow$	$\downarrow$
1	2	3	$\dots$	$i-1$	i	$i+1$	$\dots$	$N-1$	N

In order to illustrate these features let us further specialize to just two spins.

With two spins, since each spin takes two values only, we have $2^2 = 4$ possible arrangements of the two spins. These four possibilities are

$1 = \uparrow\uparrow \qquad 2 = \uparrow\downarrow \qquad 3 = \downarrow\uparrow \qquad 4 = \downarrow\downarrow$

Boundary conditions, free ends

What is the energy of each of these configurations?

For small systems, the way we treat the ends matters. Two cases are often used.

In the first case we employ what is called free ends. This means that there is no contribution from points to the right or left of the endpoints. For the one-dimensional case, the energy is then written as a sum over a single index

$$E_i = -J \sum_{j=1}^{N-1} s_j s_{j+1},$$

Free ends and the energy

If we label the first spin as s_1 and the second as s_2 we obtain the following expression for the energy

$$E = -J s_1 s_2.$$

The calculation of the energy for the one-dimensional lattice with free ends for one specific spin-configuration can easily be implemented in the following lines

```
for ( j=1; j < N; j++) {
     energy += spin[j]*spin[j+1];
}
```

where the vector *spin*[] contains the spin value $s_k = \pm 1$.

Free ends and energy

For the specific state E_1, we have chosen all spins up. The energy of this configuration becomes then

$$E_1 = E_{\uparrow\uparrow} = -J.$$

The other configurations give

$$E_2 = E_{\uparrow\downarrow} = +J,$$

$$E_3 = E_{\downarrow\uparrow} = +J,$$

and

$$E_4 = E_{\downarrow\downarrow} = -J.$$

Periodic boundary conditions

We can also choose so-called periodic boundary conditions. This means that the neighbour to the right of s_N is assumed to take the value of s_1. Similarly, the neighbour to the left of s_1 takes the value s_N. In this case the energy for the one-dimensional lattice reads

$$E_i = -J \sum_{j=1}^{N} s_j s_{j+1},$$

and we obtain the following expression for the two-spin case

$$E = -J(s_1 s_2 + s_2 s_1).$$

Energy with PBC

In this case the energy for E_1 is different, we obtain namely

$$E_1 = E_{\uparrow\uparrow} = -2J.$$

The other cases do also differ and we have

$$E_2 = E_{\uparrow\downarrow} = +2J,$$

$$E_3 = E_{\downarrow\uparrow} = +2J,$$

and

$$E_4 = E_{\downarrow\downarrow} = -2J.$$

Simple code for PBC

If we choose to use periodic boundary conditions we can code the above expression as

```
jm=N;
for ( j=1; j <=N ; j++) {
    energy += spin[j]*spin[jm];
    jm = j ;
}
```

The magnetization is however the same, defined as

$$\mathcal{M}_i = \sum_{j=1}^{N} s_j,$$

where we sum over all spins for a given configuration i.

Summing up

The table lists the energy and magnetization for both free ends and periodic boundary conditions.

State	Energy (FE)	Energy (PBC)	Magnetization
1 =↑↑	$-J$	$-2J$	2
2 =↑↓	J	$2J$	0
3 =↓↑	J	$2J$	0
4 =↓↓	$-J$	$-2J$	-2

Reorganizing

We can reorganize according to the number of spins pointing up, as shown in the table here

Number spins up	Degeneracy	Energy (FE)	Energy (PBC)	Magnetization
2	1	$-J$	$-2J$	2
1	2	J	$2J$	0
0	1	$-J$	$-2J$	-2

Our model, the Ising model in one and two dimensions

It is worth noting that for small dimensions of the lattice, the energy differs depending on whether we use periodic boundary conditions or free ends. This means also that the partition functions will be different, as discussed below. In the thermodynamic limit we have $N \to \infty$, and the final results do not depend on the kind of boundary conditions we choose.

For a one-dimensional lattice with periodic boundary conditions, each spin sees two neighbors. For a two-dimensional lattice each spin sees four neighboring spins. How many neighbors does a spin see in three dimensions?

Ising model in one and two dimensions

In a similar way, we could enumerate the number of states for a two-dimensional system consisting of two spins, i.e., a 2×2 Ising model on a square lattice with *periodic boundary conditions*. In this case we have a total of $2^4 = 16$ states. Some examples of configurations with their respective energies are listed here

$E = -8J$ $\begin{matrix}\uparrow & \uparrow \\ \uparrow & \uparrow\end{matrix}$ $E = 0$ $\begin{matrix}\uparrow & \uparrow \\ \uparrow & \downarrow\end{matrix}$ $E = 0$ $\begin{matrix}\downarrow & \downarrow \\ \uparrow & \downarrow\end{matrix}$ $E = -8J$ $\begin{matrix}\downarrow & \downarrow \\ \downarrow & \downarrow\end{matrix}$

List of configurations with energies and magnetic moment

In the table here we group these configurations according to their total energy and magnetization.

Number spins up	Degeneracy	Energy	Magnetization
4	1	$-8J$	4
3	4	0	2
2	4	0	0
2	2	$8J$	0
1	4	0	-2
0	1	$-8J$	-4

Phase Transitions and Critical Phenomena

A phase transition is marked by abrupt macroscopic changes as external parameters are changed, such as an increase of temperature. The point where a phase transition takes place is called a critical point.

We distinguish normally between two types of phase transitions; first-order transitions and second-order transitions. An important quantity in studies of phase transitions is the so-called correlation length ξ and various correlations functions like spin-spin correlations. For the Ising model we shall show below that the correlation length is related to the spin-correlation function, which again defines the magnetic susceptibility. The spin-correlation function is nothing but the covariance and expresses the degree of correlation between spins.

Phase Transitions and Critical Phenomena, correlation length

The correlation length defines the length scale at which the overall properties of a material start to differ from its bulk properties. It is the distance over which the fluctuations of the microscopic degrees of freedom (for example the position of atoms) are significantly correlated with each other. Usually it is of the order of few interatomic spacings for a solid. The correlation length ξ depends however on external conditions such as pressure and temperature.

Classification of phase transitions

First order/discontinuous phase transitions are characterized by two or more states on either side of the critical point that can coexist at the critical point. As we pass through the critical point we observe a discontinuous behavior of thermodynamical functions. The correlation length is normally finite at the critical point. Phenomena such as hysteris occur, viz. there is a continuation of state below the critical point into one above the critical point. This continuation is metastable so that the system may take a macroscopically long time to readjust. A classical example is the melting of ice. It takes a specific amount of time before all the ice has melted. The temperature remains constant and water and

ice can coexist for a macroscopic time. The energy shows a discontinuity at the critical point, reflecting the fact that a certain amount of heat is needed in order to melt all the ice

Second-order phase Transitions

Second order or continuous transitions are different and in general much difficult to understand and model. The correlation length diverges at the critical point, fluctuations are correlated over all distance scales, which forces the system to be in a unique critical phase. The two phases on either side of the critical point become identical. The disappearance of a spontaneous magnetization is a classical example of a second-order phase transitions. Structural transitions in solids are other types of second-order phase transitions.

Phase Transitions and Critical Phenomena

System	Transition	Order Parameter
Liquid-gas	Condensation/evaporation	Density difference $\Delta\rho = \rho_{liquid} - \rho_{gas}$
Binary liquid	mixture/Unmixing	Composition difference
Quantum liquid	Normal fluid/superfluid	$< \phi >$, ψ = wavefunction
Liquid-solid	Melting/crystallisation	Reciprocal lattice vector
Magnetic solid	Ferromagnetic	Spontaneous magnetisation M
	Antiferromagnetic	Sublattice magnetisation M
Dielectric solid	Ferroelectric	Polarization P
	Antiferroelectric	Sublattice polarisation P

Eherenfest definition of phase Transitions

Using Ehrenfest's definition of the order of a phase transition we can relate the behavior around the critical point to various derivatives of the thermodynamical potential. In the canonical ensemble we are using, the thermodynamical potential is Helmholtz' free energy

$$F = \langle E \rangle - TS = -kTlnZ$$

meaning $lnZ = -F/kT = -F\beta$. The energy is given as the first derivative of F

$$\langle E \rangle = -\frac{\partial lnZ}{\partial \beta} = \frac{\partial(\beta F)}{\partial \beta}.$$

and the specific heat is defined via the second derivative of F

$$C_V = -\frac{1}{kT^2}\frac{\partial^2(\beta F)}{\partial \beta^2}.$$

Phase Transitions and Critical Phenomena

We can relate observables to various derivatives of the partition function and the free energy. When a given derivative of the free energy or the partition function is discontinuous or diverges (logarithmic divergence for the heat capacity from the Ising model) we talk of a phase transition of order of the derivative. A first-order phase transition is recognized in a discontinuity of the energy, or the first derivative of F. The Ising model exhibits a second-order phase transition since the heat capacity diverges. The susceptibility is given by the second derivative of F with respect to external magnetic field. Both these quantities diverge.

The Ising Model and Phase Transitions

The Ising model in two dimensions with $\mathcal{B} = 0$ undergoes a phase transition of second order. What it actually means is that below a given critical temperature T_C, the Ising model exhibits a spontaneous magnetization with $\langle \mathcal{M} \rangle \neq 0$. Above T_C the average magnetization is zero. The mean magnetization approaches zero at T_C with an infinite slope. Such a behavior is an example of what are called critical phenomena. A critical phenomenon is normally marked by one or more thermodynamical variables which vanish above a critical point. In our case this is the mean magnetization $\langle \mathcal{M} \rangle \neq 0$. Such a parameter is normally called the order parameter.

The Ising Model and Phase Transitions, mean magnetization

It is possible to show that the mean magnetization is given by (for temperature below T_C)

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta} \, ,$$

where $\beta = 1/8$ is a so-called critical exponent. A similar relation applies to the heat capacity

$$C_V(T) \sim |T_C - T|^{-\alpha} \, ,$$

and the susceptibility

$$\chi(T) \sim |T_C - T|^{-\gamma} \, ,$$

with $\alpha = 0$ and $\gamma = -7/4$.

The Ising Model and Phase Transitions, correlation length

Another important quantity is the correlation length, which is expected to be of the order of the lattice spacing for T is close to T_C. Because the spins become more and more correlated as T approaches T_C, the correlation length increases

as we get closer to the critical temperature. The discontinuous behavior of the correlation ξ near T_C is

$$\xi(T) \sim |T_C - T|^{-\nu} . \tag{1}$$

The Ising Model and Phase Transitions, critical behavior

A second-order phase transition is characterized by a correlation length which spans the whole system. The correlation length is typically of the order of some few interatomic distances. The fact that a system like the Ising model, whose energy is described by the interaction between neighboring spins only, can yield correlation lengths of macroscopic size at a critical point is still a feature which is not properly understood.

The Ising Model and Phase Transitions, critical temperature

In our actual calculations of the two-dimensional Ising model, we are however always limited to a finite lattice and ξ will be proportional with the size of the lattice at the critical point. Through finite size scaling relations it is possible to relate the behavior at finite lattices with the results for an infinitely large lattice. The critical temperature scales then as

$$T_C(L) - T_C(L = \infty) \propto aL^{-1/\nu}, \tag{2}$$

with a a constant and ν defined in Eq. (1).

The Ising Model and Phase Transitions, correlation length

The correlation length for a finite lattice size can then be shown to be proportional to

$$\xi(T) \propto L \sim |T_C - T|^{-\nu} .$$

and if we set $T = T_C$ one can obtain the following relations for the magnetization, energy and susceptibility for $T \leq T_C$

$$\langle \mathcal{M}(T) \rangle \sim (T - T_C)^{\beta} \propto L^{-\beta/\nu},$$

$$C_V(T) \sim |T_C - T|^{-\gamma} \propto L^{\alpha/\nu},$$

and

$$\chi(T) \sim |T_C - T|^{-\alpha} \propto L^{\gamma/\nu}.$$

The Metropolis Algorithm and the Two-dimensional Ising Model

In our case we have as the Monte Carlo sampling function the probability for finding the system in a state s given by

$$P_s = \frac{e^{-(\beta E_s)}}{Z},$$

with energy E_s, $\beta = 1/kT$ and Z is a normalization constant which defines the partition function in the canonical ensemble. As discussed above

$$Z(\beta) = \sum_s e^{-(\beta E_s)}$$

is difficult to compute since we need all states.

The Metropolis Algorithm and the Two-dimensional Ising Model

In a calculation of the Ising model in two dimensions, the number of configurations is given by 2^N with $N = L \times L$ the number of spins for a lattice of length L. Fortunately, the Metropolis algorithm considers only ratios between probabilities and we do not need to compute the partition function at all. The algorithm goes as follows

1. Establish an initial state with energy E_b by positioning yourself at a random configuration in the lattice
2. Change the initial configuration by flipping e.g., one spin only. Compute the energy of this trial state E_t.
3. Calculate $\Delta E = E_t - E_b$. The number of values ΔE is limited to five for the Ising model in two dimensions, see the discussion below.
4. If $\Delta E \leq 0$ we accept the new configuration, meaning that the energy is lowered and we are hopefully moving towards the energy minimum at a given temperature. Go to step 7.
5. If $\Delta E > 0$, calculate $w = e^{-(\beta \Delta E)}$.
6. Compare w with a random number r. If $r \leq w$, then accept the new configuration, else we keep the old configuration.
7. The next step is to update various expectations values.
8. The steps (2)-(7) are then repeated in order to obtain a sufficently good representation of states.

9. Each time you sweep through the lattice, i.e., when you have summed over all spins, constitutes what is called a Monte Carlo cycle. You could think of one such cycle as a measurement. At the end, you should divide the various expectation values with the total number of cycles. You can choose whether you wish to divide by the number of spins or not. If you divide with the number of spins as well, your result for e.g., the energy is now the energy per spin.

The Metropolis Algorithm and the Two-dimensional Ising Model, practical issues

The crucial step is the calculation of the energy difference and the change in magnetization. This part needs to be coded in an as efficient as possible way since the change in energy is computed many times. In the calculation of the energy difference from one spin configuration to the other, we will limit the change to the flipping of one spin only. For the Ising model in two dimensions it means that there will only be a limited set of values for ΔE. Actually, there are only five possible values.

Five possible energy differences

To see this, select first a random spin position x, y and assume that this spin and its nearest neighbors are all pointing up. The energy for this configuration is $E = -4J$. Now we flip this spin as shown below. The energy of the new configuration is $E = 4J$, yielding $\Delta E = 8J$.

$$E = -4J \qquad \begin{array}{ccc} & \uparrow & \\ \uparrow & \uparrow & \uparrow \\ & \uparrow & \end{array} \qquad \Longrightarrow \qquad E = 4J \qquad \begin{array}{ccc} & \uparrow & \\ \uparrow & \downarrow & \uparrow \\ & \uparrow & \end{array}$$

The four other possibilities are as follows

$$E = -2J \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \uparrow & \uparrow \\ & \uparrow & \end{array} \qquad \Longrightarrow \qquad E = 2J \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \downarrow & \uparrow \\ & \uparrow & \end{array}$$

with $\Delta E = 4J$,

$$E = 0 \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \uparrow & \uparrow \\ & \downarrow & \end{array} \qquad \Longrightarrow \qquad E = 0 \qquad \begin{array}{ccc} & \uparrow & \\ \downarrow & \downarrow & \uparrow \\ & \downarrow & \end{array}$$

with $\Delta E = 0$,

$$E = 2J \qquad \downarrow \begin{matrix} \downarrow \\ \uparrow \\ \downarrow \end{matrix} \uparrow \qquad \Longrightarrow \qquad E = -2J \qquad \downarrow \begin{matrix} \downarrow \\ \downarrow \\ \downarrow \end{matrix} \uparrow$$

with $\Delta E = -4J$ and finally

$$E = 4J \qquad \downarrow \begin{matrix} \downarrow \\ \uparrow \\ \downarrow \end{matrix} \downarrow \qquad \Longrightarrow \qquad E = -4J \qquad \downarrow \begin{matrix} \downarrow \\ \downarrow \\ \downarrow \end{matrix} \downarrow$$

with $\Delta E = -8J$.

The Metropolis Algorithm and the Two-dimensional Ising Model, elements of program

This means in turn that we could construct an array which contains all values of $e^{\beta\Delta E}$ before doing the Metropolis sampling. Else, we would have to evaluate the exponential at each Monte Carlo sampling. For the two-dimensional Ising model there are only five possible values. It is rather easy to convice oneself that for the one-dimensional Ising model we have only three possible values. The main part of the Ising model program is shown here

```
/*
   Program to solve the two-dimensional Ising model
   The coupling constant J = 1
   Boltzmann's constant = 1, temperature has thus dimension energy
   Metropolis sampling is used. Periodic boundary conditions.
*/
#include <iostream>
#include <fstream>
#include <iomanip>
#include "lib.h"
using namespace  std;
ofstream ofile;
// inline function for periodic boundary conditions
inline int periodic(int i, int limit, int add) {
  return (i+limit+add) % (limit);
}
// Function to read in data from screen
void read_input(int&, int&, double&, double&, double&);
// Function to initialise energy and magnetization
void initialize(int, double, int **, double&, double&);
// The metropolis algorithm
void Metropolis(int, long&, int **, double&, double&, double *);
// prints to file the results of the calculations
void output(int, int, double, double *);

//  main program
int main(int argc, char* argv[])
{
  char *outfilename;
  long idum;
```

```
  int **spin_matrix, n_spins, mcs;
  double w[17], average[5], initial_temp, final_temp, E, M, temp_step;

  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
    exit(1);
  }
  else{
    outfilename=argv[1];
  }
  ofile.open(outfilename);
  //    Read in initial values such as size of lattice, temp and cycles
  read_input(n_spins, mcs, initial_temp, final_temp, temp_step);
  spin_matrix = (int**) matrix(n_spins, n_spins, sizeof(int));
  idum = -1; // random starting point
  for ( double temp = initial_temp; temp <= final_temp; temp+=temp_step){
    //    initialise energy and magnetization
    E = M = 0.;
    // setup array for possible energy changes
    for( int de =-8; de <= 8; de++) w[de+8] = 0;
    for( int de =-8; de <= 8; de+=4) w[de+8] = exp(-de/temp);
    // initialise array for expectation values
    for( int i = 0; i < 5; i++) average[i] = 0.;
    initialize(n_spins, double temp, spin_matrix, E, M);
    // start Monte Carlo computation
    for (int cycles = 1; cycles <= mcs; cycles++){
      Metropolis(n_spins, idum, spin_matrix, E, M, w);
      // update expectation values
      average[0] += E;    average[1] += E*E;
      average[2] += M;    average[3] += M*M; average[4] += fabs(M);
    }
    // print results
    output(n_spins, mcs, temp, average);
  }
  free_matrix((void **) spin_matrix); // free memory
  ofile.close();  // close output file
  return 0;
}
```

Coding energy differences

The array $w[17]$ contains values of ΔE spanning from $-8J$ to $8J$ and it is precalculated in the main part for every new temperature. The program takes as input the initial temperature, final temperature, a temperature step, the number of spins in one direction (we force the lattice to be a square lattice, meaning that we have the same number of spins in the x and the y directions) and the number of Monte Carlo cycles.

Efficient expression for energy differences

For every Monte Carlo cycle we run through all spins in the lattice in the function metropolis and flip one spin at the time and perform the Metropolis test. However, every time we flip a spin we need to compute the actual energy difference ΔE in order to access the right element of the array which stores

$e^{\beta\Delta E}$. This is easily done in the Ising model since we can exploit the fact that only one spin is flipped, meaning in turn that all the remaining spins keep their values fixed. The energy difference between a state E_1 and a state E_2 with zero external magnetic field is

$$\Delta E = E_2 - E_1 = J\sum_{<kl>}^{N} s_k^1 s_l^1 - J\sum_{<kl>}^{N} s_k^2 s_l^2,$$

which we can rewrite as

$$\Delta E = -J\sum_{<kl>}^{N} s_k^2(s_l^2 - s_l^1),$$

where the sum now runs only over the nearest neighbors k.

Final energy difference

Since the spin to be flipped takes only two values, $s_l^1 = \pm 1$ and $s_l^2 = \pm 1$, it means that if $s_l^1 = 1$, then $s_l^2 = -1$ and if $s_l^1 = -1$, then $s_l^2 = 1$. The other spins keep their values, meaning that $s_k^1 = s_k^2$. If $s_l^1 = 1$ we must have $s_l^1 - s_l^2 = 2$, and if $s_l^1 = -1$ we must have $s_l^1 - s_l^2 = -2$. From these results we see that the energy difference can be coded efficiently as

$$\Delta E = 2Js_l^1\sum_{<k>}^{N} s_k, \tag{3}$$

where the sum runs only over the nearest neighbors k of spin l. We can compute the change in magnetisation by flipping one spin as well. Since only spin l is flipped, all the surrounding spins remain unchanged.

Change in magnetization

The difference in magnetisation is therefore only given by the difference $s_l^1 - s_l^2 = \pm 2$, or in a more compact way as

$$M_2 = M_1 + 2s_l^2, \tag{4}$$

where M_1 and M_2 are the magnetizations before and after the spin flip, respectively. Eqs. (3) and (4) are implemented in the function **metropolis** shown here

```
void Metropolis(int n_spins, long& idum, int **spin_matrix, double& E, double&M, double *w)
{
  // loop over all spins
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      // Find random position
      int ix = (int) (ran1(&idum)*(double)n_spins);
      int iy = (int) (ran1(&idum)*(double)n_spins);
```

```
      int deltaE =  2*spin_matrix[iy][ix]*
        (spin_matrix[iy][periodic(ix,n_spins,-1)]+
         spin_matrix[periodic(iy,n_spins,-1)][ix] +
         spin_matrix[iy][periodic(ix,n_spins,1)] +
         spin_matrix[periodic(iy,n_spins,1)][ix]);
      // Here we perform the Metropolis test
      if ( ran1(&idum) <= w[deltaE+8] ) {
        spin_matrix[iy][ix] *= -1;  // flip one spin and accept new spin config
        // update energy and magnetization
        M += (double) 2*spin_matrix[iy][ix];
        E += (double) deltaE;
      }
    }
  }
} // end of Metropolis sampling over spins
```

A small note

Note that we loop over all spins but that we choose the lattice positions x and y randomly. If the move is accepted after performing the Metropolis test, we update the energy and the magnetisation. The new values are used to update the averages computed in the main function.

Initialization

We need also to initialize various variables. This is done in the function here.

```
// function to initialise energy, spin matrix and magnetization
void initialize(int n_spins, double temp, int **spin_matrix,
                double& E, double& M)
{
  // setup spin matrix and intial magnetization
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      if (temp < 1.5) spin_matrix[y][x] = 1; // spin orientation for the ground state
      M +=  (double) spin_matrix[y][x];
    }
  }
  // setup initial energy
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      E -=  (double) spin_matrix[y][x]*
        (spin_matrix[periodic(y,n_spins,-1)][x] +
         spin_matrix[y][periodic(x,n_spins,-1)]);
    }
  }
}// end function initialise
```

The Metropolis Algorithm and the Two-dimensional Ising Model, elements of program

Here follows an alternative Ising model code using the Mersenne twister engine as described in the c++ "random class":" ".

```
/*
   Program to solve the two-dimensional Ising model
```

```
   with zero external field and no parallelization using the Mersenne twister engine for generati
   numbers.
   The coupling constant J = 1
   Boltzmann's constant = 1, temperature has thus dimension energy
   Metropolis sampling is used. Periodic boundary conditions.
   The code needs an output file on the command line and the variables mcs, nspins,
   initial temp, final temp and temp step.
   Run as
   ./executable Outputfile numberof spins number of MC cycles initial temp final temp tempstep
   ./test.x Lattice 100 10000000 2.1 2.4 0.01
   Compile and link as
   c++ -O3 -std=c++11 -Rpass=loop-vectorize -o Ising.x IsingModel.cpp -larmadillo
*/

#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <random>
#include <armadillo>
#include <string>
using namespace  std;
using namespace arma;
// output file
ofstream ofile;

// inline function for periodic boundary conditions
inline int periodic(int i, int limit, int add) {
  return (i+limit+add) % (limit);
}
// Function to initialise energy and magnetization
void InitializeLattice(int, mat &, double&, double&);
// The metropolis algorithm including the loop over Monte Carlo cycles
void MetropolisSampling(int, int, double, vec &);
// prints to file the results of the calculations
void output(int, int, double, vec);

// Main program begins here

int main(int argc, char* argv[])
{
  string filename;
  int NSpins, MCcycles;
  double InitialTemp, FinalTemp, TempStep;
  if (argc <= 5) {
    cout << "Bad Usage: " << argv[0] <<
      " read output file, Number of spins, MC cycles, initial and final temperature and tempurate
    exit(1);
  }
  if (argc > 1) {
    filename=argv[1];
    NSpins = atoi(argv[2]);
    MCcycles = atoi(argv[3]);
    InitialTemp = atof(argv[4]);
    FinalTemp = atof(argv[5]);
    TempStep = atof(argv[6]);
  }
  // Declare new file name and add lattice size to file name
  string fileout = filename;
  string argument = to_string(NSpins);
```

```
  fileout.append(argument);
  ofile.open(fileout);
  // Start Monte Carlo sampling by looping over T first
  for (double Temperature = InitialTemp; Temperature <= FinalTemp; Temperature+=TempStep){
    vec ExpectationValues = zeros<mat>(5);
    // start Monte Carlo computation
    MetropolisSampling(NSpins, MCcycles, Temperature, ExpectationValues);
    output(NSpins, MCcycles, Temperature, ExpectationValues);
  }
  ofile.close();  // close output file
  return 0;
}

// function to initialise energy, spin matrix and magnetization
void InitializeLattice(int NSpins, mat &SpinMatrix,  double& Energy, double& MagneticMoment)
{
  // setup spin matrix and initial magnetization
  for(int x =0; x < NSpins; x++) {
    for (int y= 0; y < NSpins; y++){
      SpinMatrix(x,y) = 1.0; // spin orientation for the ground state
      MagneticMoment +=  (double) SpinMatrix(x,y);
    }
  }
  // setup initial energy
  for(int x =0; x < NSpins; x++) {
    for (int y= 0; y < NSpins; y++){
      Energy -=  (double) SpinMatrix(x,y)*
        (SpinMatrix(periodic(x,NSpins,-1),y) +
         SpinMatrix(x,periodic(y,NSpins,-1)));
    }
  }
}// end function initialise


// The Monte Carlo part with the Metropolis algo with sweeps over the lattice
void MetropolisSampling(int NSpins, int MCcycles, double Temperature, vec &ExpectationValues)
{
  // Initialize the seed and call the Mersenne algo
  std::random_device rd;
  std::mt19937_64 gen(rd());
  // Then set up the uniform distribution for x \in [[0, 1]
  std::uniform_real_distribution<double> distribution(0.0,1.0);
  //  Allocate memory for spin matrix
  mat SpinMatrix = zeros<mat>(NSpins,NSpins);
  //    initialise energy and magnetization
  double Energy = 0.;     double MagneticMoment = 0.;
  // initialize array for expectation values
  InitializeLattice(NSpins, SpinMatrix, Energy, MagneticMoment);
  // setup array for possible energy changes
  vec EnergyDifference = zeros<mat>(17);
  for( int de =-8; de <= 8; de+=4) EnergyDifference(de+8) = exp(-de/Temperature);
  for (int cycles = 1; cycles <= MCcycles; cycles++){
    // The sweep over the lattice, looping over all spin sites
    for(int x =0; x < NSpins; x++) {
      for (int y= 0; y < NSpins; y++){
        int ix = (int) (distribution(gen)*(double)NSpins);
        int iy = (int) (distribution(gen)*(double)NSpins);
        int deltaE =  2*SpinMatrix(ix,iy)*
          (SpinMatrix(ix,periodic(iy,NSpins,-1))+
           SpinMatrix(periodic(ix,NSpins,-1),iy) +
           SpinMatrix(ix,periodic(iy,NSpins,1)) +
```

```
           SpinMatrix(periodic(ix,NSpins,1),iy));
         if ( distribution(gen) <= EnergyDifference(deltaE+8) ) {
           SpinMatrix(ix,iy) *= -1.0;  // flip one spin and accept new spin config
           MagneticMoment += (double) 2*SpinMatrix(ix,iy);
           Energy += (double) deltaE;
         }
       }
     }
     // update expectation values  for local node
     ExpectationValues(0) += Energy;    ExpectationValues(1) += Energy*Energy;
     ExpectationValues(2) += MagneticMoment;
     ExpectationValues(3) += MagneticMoment*MagneticMoment;
     ExpectationValues(4) += fabs(MagneticMoment);
   }

} // end of Metropolis sampling over spins

void output(int NSpins, int MCcycles, double temperature, vec ExpectationValues)
{
  double norm = 1.0/((double) (MCcycles));  // divided by  number of cycles
  double E_ExpectationValues = ExpectationValues(0)*norm;
  double E2_ExpectationValues = ExpectationValues(1)*norm;
  double M_ExpectationValues = ExpectationValues(2)*norm;
  double M2_ExpectationValues = ExpectationValues(3)*norm;
  double Mabs_ExpectationValues = ExpectationValues(4)*norm;
  // all expectation values are per spin, divide by 1/NSpins/NSpins
  double Evariance = (E2_ExpectationValues- E_ExpectationValues*E_ExpectationValues)/NSpins/NSpins
  double Mvariance = (M2_ExpectationValues - Mabs_ExpectationValues*Mabs_ExpectationValues)/NSpins
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << setw(15) << setprecision(8) << temperature;
  ofile << setw(15) << setprecision(8) << E_ExpectationValues/NSpins/NSpins;
  ofile << setw(15) << setprecision(8) << Evariance/temperature/temperature;
  ofile << setw(15) << setprecision(8) << M_ExpectationValues/NSpins/NSpins;
  ofile << setw(15) << setprecision(8) << Mvariance/temperature;
  ofile << setw(15) << setprecision(8) << Mabs_ExpectationValues/NSpins/NSpins << endl;
} // end output function
```

Two-dimensional Ising Model, energy per spin and specific heat

The following Python program, based on the above C++ codes, plots the expectation value of the energy and its fluctuation, that is the specific heat. Both quantities are plotted per spin and genererated for a 20×20 lattice.

```
#  Code for the two-dimensional Ising model with periodic boundary conditions
import numpy, sys, math
from  matplotlib import pyplot as plt
import numpy as np


def periodic (i, limit, add):
    """
    Choose correct matrix index with periodic
    boundary conditions

    Input:
    - i:     Base index
    - limit: Highest \"legal\" index
```

```
    - add:   Number to add or subtract from i
    """
    return (i+limit+add) % limit

def monteCarlo(temp, size, trials):
    """
    Calculate the energy and magnetization
    (\"straight\" and squared) for a given temperature

    Input:
    - temp:   Temperature to calculate for
    - size:   dimension of square matrix
    - trials: Monte-carlo trials (how many times do we
                                  flip the matrix?)

    Output:
    - E_av:       Energy of matrix averaged over trials, normalized to spins**2
    - E_variance: Variance of energy, same normalization * temp**2
    """

    #Setup spin matrix, initialize to ground state
    spin_matrix = numpy.zeros( (size,size), numpy.int8) + 1

    #Create and initialize variables
    E = 0
    E_av = E2_av = 0

    #Setup array for possible energy changes
    w = numpy.zeros(17,numpy.float64)
    for de in xrange(-8,9,4): #include +8
        w[de+8] = math.exp(-de/temp)

    #Calculate initial energy
    for j in xrange(size):
        for i in xrange(size):
            E -= spin_matrix.item(i,j)*\
                 (spin_matrix.item(periodic(i,size,-1),j) + spin_matrix.item(i,periodic(j,size,1))

    #Start metropolis MonteCarlo computation
    for i in xrange(trials):
        #Metropolis
        #Loop over all spins, pick a random spin each time
        for s in xrange(size**2):
            x = int(numpy.random.random()*size)
            y = int(numpy.random.random()*size)
            deltaE = 2*spin_matrix.item(x,y)*\
                     (spin_matrix.item(periodic(x,size,-1), y) +\
                      spin_matrix.item(periodic(x,size,1),  y) +\
                      spin_matrix.item(x, periodic(y,size,-1)) +\
                      spin_matrix.item(x, periodic(y,size,1)))
            if numpy.random.random() <= w[deltaE+8]:
                #Accept!
                spin_matrix[x,y] *= -1
                E += deltaE

        #Update expectation values
        E_av    += E
        E2_av   += E**2

    E_av       /= float(trials);
    E2_av      /= float(trials);
```

```
    #Calculate variance and normalize to per-point and temp
    E_variance  = (E2_av-E_av*E_av)/float(size*size*temp*temp);
    #Normalize returned averages to per-point
    E_av       /= float(size*size);

    return (E_av, E_variance)


# Main program

# values of the lattice, number of Monte Carlo cycles and temperature domain
size        =   20
trials      =   100000
temp_init   = 1.8
temp_end    = 2.6
temp_step   = 0.1


temps = numpy.arange(temp_init,temp_end+temp_step/2,temp_step,float)
Dim = np.size(temps)
energy = np.zeros(Dim)
heatcapacity = np.zeros(Dim)
temperature = np.zeros(Dim)
for temp in temps:
    (E_av, E_variance) = monteCarlo(temp,size,trials)
    temperature[temp] = temp
    energy[temp] = E_av
    heatcapacity[temp] = E_variance
plt.figure(1)
plt.subplot(211)
plt.axis([1.8,2.6,-2.0, -1.0])
plt.xlabel(r'Temperature $J/(k_B)$')
plt.ylabel(r'Average energy per spin  $E/N$')
plt.plot(temperature, energy, 'b-')
plt.subplot(212)
plt.axis([1.8,2.6, 0.0, 2.0])
plt.plot(temperature, heatcapacity, 'r-')
plt.xlabel(r'Temperature $J/(k_B)$')
plt.ylabel(r'Heat capacity per spin  $C_V/N$')
plt.savefig('energycv.pdf')
plt.show()
```

Two-dimensional Ising Model and analysis of spin values

The following python code displays the values of the spins as function of temperature. The blue color corresponds to spin up states while red represents spin down states. Increasing the temperature as input parameter, see the parameters below, results in a a net magnetization which becomes zero. At low temperatures, the system is highly ordered with essentially only one specific spin value.

```
# coding=utf-8
#2-dimensional ising model with visualization

import numpy, sys, math
import pygame

#Needed for visualize when using SDL
screen = None;
```

```
font   = None;
BLOCKSIZE = 10

def periodic (i, limit, add):
    """
    Choose correct matrix index with periodic
    boundary conditions

    Input:
    - i:     Base index
    - limit: Highest \"legal\" index
    - add:   Number to add or subtract from i
    """
    return (i+limit+add) % limit

def visualize(spin_matrix, temp, E, M, method):
    """
    Visualize the spin matrix

    Methods:
    method = -1:No visualization (testing)
    method = 0: Just print it to the terminal
    method = 1: Pretty-print to terminal
    method = 2: SDL/pygame single-pixel
    method = 3: SDL/pygame rectangle
    """

    #Simple terminal dump
    if method == 0:
        print "temp:", temp, "E:", E, "M:", M
        print spin_matrix
    #Pretty-print to terminal
    elif method == 1:
        out = ""
        size = len(spin_matrix)
        for y in xrange(size):
            for x in xrange(size):
                if spin_matrix.item(x,y) == 1:
                    out += "X"
                else:
                    out += " "
            out += "\n"
        print "temp:", temp, "E:", E, "M:", M
        print out + "\n"
    #SDL single-pixel (useful for large arrays)
    elif method == 2:
        size = len(spin_matrix)
        screen.lock()
        for y in xrange(size):
            for x in xrange(size):
                if spin_matrix.item(x,y) == 1:
                    screen.set_at((x,y),(0,0,255))
                else:
                    screen.set_at((x,y),(255,0,0))
        screen.unlock()
        pygame.display.flip()
    #SDL block (usefull for smaller arrays)
    elif method == 3:
        size = len(spin_matrix)
        screen.lock()
        for y in xrange(size):
```

```
                for x in xrange(size):
                    if spin_matrix.item(x,y) == 1:
                        rect = pygame.Rect(x*BLOCKSIZE,y*BLOCKSIZE,BLOCKSIZE,BLOCKSIZE)
                        pygame.draw.rect(screen,(0,0,255),rect)
                    else:
                        rect = pygame.Rect(x*BLOCKSIZE,y*BLOCKSIZE,BLOCKSIZE,BLOCKSIZE)
                        pygame.draw.rect(screen,(255,0,0),rect)
            screen.unlock()
            pygame.display.flip()
        #SDL block w/ data-display
        elif method == 4:
            size = len(spin_matrix)
            screen.lock()
            for y in xrange(size):
                for x in xrange(size):
                    if spin_matrix.item(x,y) == 1:
                        rect = pygame.Rect(x*BLOCKSIZE,y*BLOCKSIZE,BLOCKSIZE,BLOCKSIZE)
                        pygame.draw.rect(screen,(255,255,255),rect)
                    else:
                        rect = pygame.Rect(x*BLOCKSIZE,y*BLOCKSIZE,BLOCKSIZE,BLOCKSIZE)
                        pygame.draw.rect(screen,(0,0,0),rect)
            s = font.render("<E> = %5.3E; <M> = %5.3E" % E,M,False,(255,0,0))
            screen.blit(s,(0,0))

            screen.unlock()
            pygame.display.flip()


def monteCarlo(temp, size, trials, visual_method):
    """
    Calculate the energy and magnetization
    (\"straight\" and squared) for a given temperature

    Input:
    - temp:   Temperature to calculate for
    - size:   dimension of square matrix
    - trials: Monte-carlo trials (how many times do we
                                  flip the matrix?)
    - visual_method: What method should we use to visualize?

    Output:
    - E_av:       Energy of matrix averaged over trials, normalized to spins**2
    - E_variance: Variance of energy, same normalization * temp**2
    - M_av:       Magnetic field of matrix, averaged over trials, normalized to spins**2
    - M_variance: Variance of magnetic field, same normalization * temp
    - Mabs:       Absolute value of magnetic field, averaged over trials
    """

    #Setup spin matrix, initialize to ground state
    spin_matrix = numpy.zeros( (size,size), numpy.int8) + 1

    #Create and initialize variables
    E    = M     = 0
    E_av = E2_av = M_av = M2_av = Mabs_av = 0

    #Setup array for possible energy changes
    w = numpy.zeros(17,numpy.float64)
    for de in xrange(-8,9,4): #include +8
        w[de+8] = math.exp(-de/temp)
```

```
    #Calculate initial magnetization:
    M = spin_matrix.sum()
    #Calculate initial energy
    for j in xrange(size):
        for i in xrange(size):
            E -= spin_matrix.item(i,j)*\
                 (spin_matrix.item(periodic(i,size,-1),j) + spin_matrix.item(i,periodic(j,size,1)))

    #Start metropolis MonteCarlo computation
    for i in xrange(trials):
        #Metropolis
        #Loop over all spins, pick a random spin each time
        for s in xrange(size**2):
            x = int(numpy.random.random()*size)
            y = int(numpy.random.random()*size)
            deltaE = 2*spin_matrix.item(x,y)*\
                     (spin_matrix.item(periodic(x,size,-1), y) +\
                      spin_matrix.item(periodic(x,size,1),  y) +\
                      spin_matrix.item(x, periodic(y,size,-1)) +\
                      spin_matrix.item(x, periodic(y,size,1)))
            if numpy.random.random() <= w[deltaE+8]:
                #Accept!
                spin_matrix[x,y] *= -1
                M += 2*spin_matrix[x,y]
                E += deltaE

        #Update expectation values
        E_av    += E
        E2_av   += E**2
        M_av    += M
        M2_av   += M**2
        Mabs_av += int(math.fabs(M))

        visualize(spin_matrix, temp,E/float(size**2),M/float(size**2), method);

    #Normalize average values
    E_av       /= float(trials);
    E2_av      /= float(trials);
    M_av       /= float(trials);
    M2_av      /= float(trials);
    Mabs_av    /= float(trials);
    #Calculate variance and normalize to per-point and temp
    E_variance  = (E2_av-E_av*E_av)/float(size*size*temp*temp);
    M_variance  = (M2_av-M_av*M_av)/float(size*size*temp);
    #Normalize returned averages to per-point
    E_av       /= float(size*size);
    M_av       /= float(size*size);
    Mabs_av    /= float(size*size);

    return (E_av, E_variance, M_av, M_variance, Mabs_av)


# Main program

size        =   100
trials      =   100000
temp        = 2.1
method      =   3

#Initialize pygame
if method == 2 or method == 3 or method == 4:
```

```
    pygame.init()
    if method == 2:
        screen = pygame.display.set_mode((size,size))
    elif method == 3:
        screen = pygame.display.set_mode((size*10,size*10))
    elif method == 4:
        screen = pygame.display.set_mode((size*10,size*10))
        font   = pygame.font.Font(None,12)


(E_av, E_variance, M_av, M_variance, Mabs_av) = monteCarlo(temp,size,trials, method)
print "%15.8E %15.8E %15.8E %15.8E %15.8E %15.8E\n" % (temp, E_av, E_variance, M_av, M_variance, M

pygame.quit();
```