PHY 480 - Computational Physics
The Ising Model

Thomas Bolden

April 29, 2016

Github Repository at https://github.com/ThomasBolden/PHY-480-Spring-2016

Abstract

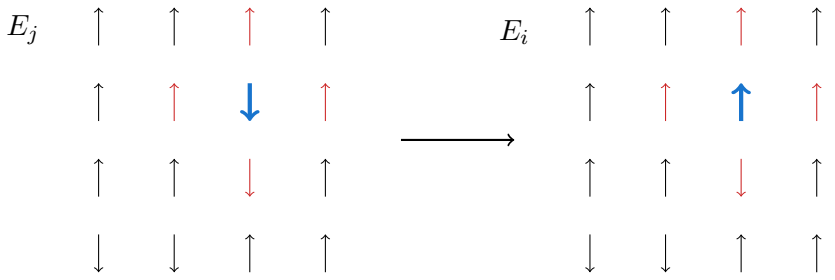
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

Introduction

Molecular thermodynamics is a field bridging both physics and chemistry. In molecular thermodynamics, statistics play a large role. One of the many important statistical measurements is the partition function. The partition function (*** what it do ***).

In this project, ...



Methods

.

Results

.

.

Conclusions

The first task was to.

Code

../Code/p4b.cpp

```
1  //   Project 4 b)
2  //   Program to compute the mean energy, magnetization,
3  //   specific heat capacity and susceptibility as functions of T.
4  //   We consider a periodic Ising system on a 2x2 grid.
5
6  #include <cmath>
7  #include <iostream>
8  #include <fstream>
9  #include <iomanip>
10 #include <cmath>
11 #include <time.h>
12 #include <random>
13 #include "lib.h"
14
```

```
15 using namespace  std;
16 ofstream ofile;
17
18 // inline function for periodic boundary conditions
19 inline int periodic(int i, int limit, int add) {
20   return (i+limit+add) % (limit);
21 }
22 // Function to read in data from screen
23 void read_input(int&, double&);
24 // Function to initialise energy and magnetization
25 void initialize(int, double, int **, double&, double&);
26 // The metropolis algorithm
27 void Metropolis(int, long&, int **, double&, double&, double *);
28 // prints to file the results of the calculations
29 void output(int, int, double, double *);
30
31 int main(int argc, char* argv[])
32 {
33   char *outfilename;
34   long idum;
35   int **spin_matrix, n_spins, mcs;
36   double w[17], average[5], temperature, E, M;
37
38   // Read in output file, abort if there are too few command-line arguments
39   if( argc <= 1 ){
40     cout << "Bad Usage: " << argv[0] <<
41       " read also output file on same line" << endl;
42     exit(1);
43   }
44   else{
45     outfilename=argv[1];
46   }
47   ofile.open(outfilename);
48   //    Read in initial values such as size of lattice, temp and cycles
49   read_input(n_spins, temperature);
50   idum = -1; // random starting point
51   int n[] = {1E4,1E5,1E6,1E7};
52   for (int j=0; j <= 3; j++) {
53     spin_matrix = (int**) matrix(n_spins, n_spins, sizeof(int));
54     mcs = n[j];
55     //    initialise energy and magnetization
56     E = M = 0.;
57     // setup array for possible energy changes
58     for( int de =-8; de <= 8; de++) w[de+8] = 0;
59     for( int de =-8; de <= 8; de+=4) w[de+8] = exp(-de/temperature);
60     // initialise array for expectation values
61     for( int i = 0; i < 5; i++) average[i] = 0.;
62     initialize(n_spins, temperature, spin_matrix, E, M);
63     // start Monte Carlo computation
64     for (int cycles = 1; cycles <= mcs; cycles++){
65     Metropolis(n_spins, idum, spin_matrix, E, M, w);
66     // update expectation values
67     average[0] += E;    average[1] += E*E;
68     average[2] += M;    average[3] += M*M;  average[4] += fabs(M);
```

```
69      }
70    // print results
71    output(n_spins, mcs, temperature, average);
72    free_matrix((void **) spin_matrix); // free memory
73    }
74    ofile.close();  // close output file
75    return 0;
76  }
77
78  // read in input data
79  void read_input(int& n_spins, double& temperature)
80  {
81    cout << "Lattice size or number of spins (x and y equal): ";
82    cin >> n_spins;
83    cout << "Temperature with dimension energy: ";
84    cin >> temperature;
85  } // end of function read_input
86
87
88  // function to initialise energy, spin matrix and magnetization
89  void initialize(int n_spins, double temperature, int **spin_matrix,
90                  double& E, double& M)
91  {
92    // setup spin matrix and intial magnetization; all spins up:
93    for(int y =0; y < n_spins; y++) {
94      for (int x= 0; x < n_spins; x++){
95        spin_matrix[y][x] = 1; // spin orientation for the ground state
96        M +=  (double) spin_matrix[y][x];
97      }
98    }
99    // setup spin matrix and intial magnetization; randomly drawn configuration:
100   // long idum_dum = -1;
101   // for(int y =0; y < n_spins; y++) {
102   //    for (int x= 0; x < n_spins; x++){
103   //      double a = (double) ran1(&idum_dum);
104   //      int r = 0;
105   //      if (a < 0.5) r = -1;
106   //      if (a <= 0.5) r = +1;
107   //      cout << "Randomly drawn number: " << r << endl;
108   //      spin_matrix[y][x] = r; // spin orientation for the ground state
109   //      M +=  (double) spin_matrix[y][x];
110   //    }
111   //  }
112   // setup initial energy:
113   for(int y =0; y < n_spins; y++) {
114     for (int x= 0; x < n_spins; x++){
115       E -=  (double) spin_matrix[y][x]*
116   (spin_matrix[periodic(y,n_spins,-1)][x] +
117    spin_matrix[y][periodic(x,n_spins,-1)]);
118     }
119   }
120 }// end function initialise
121
122 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++){
      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]);
      if ( ran1(&idum) <= w[deltaE+8] ) {
  spin_matrix[iy][ix] *= -1;  // flip one spin and accept new spin config
        M += (double) 2*spin_matrix[iy][ix];
        E += (double) deltaE;
      }
    }
  }
} // end of Metropolis sampling over spins


void output(int n_spins, int mcs, double temperature, double *average)
{
  double norm = 1/((double) (mcs));  // divided by total number of cycles
  double norm2 = 1.0/(n_spins*n_spins);   // divide by the total number of spins
  double T = temperature;
  double Eaverage = average[0]*norm;
  double E2average = average[1]*norm;
  double Maverage = average[2]*norm;
  double M2average = average[3]*norm;
  double Mabsaverage = average[4]*norm;
  // all expectation values are per spin, divide by 1/n_spins/n_spins
  double Evariance = (E2average - Eaverage*Eaverage)*norm2;
  double Mvariance = (M2average - Mabsaverage*Mabsaverage)*norm2;
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << "Number of Monte Carlo trials: " << mcs << endl;
  ofile << "Temperature:      Energy:      Heat capacity:  Magnetization:  Susceptibility:
abs(Magnetization):" << endl;
  ofile << setw(15) << setprecision(8) << temperature;
  ofile << setw(15) << setprecision(8) << Eaverage*norm2;
  ofile << setw(15) << setprecision(8) << Evariance/temperature/temperature;
  ofile << setw(15) << setprecision(8) << Maverage*norm2;
  ofile << setw(15) << setprecision(8) << Mvariance/temperature;
  ofile << setw(15) << setprecision(8) << Mabsaverage*norm2 << endl;
  // Exact results for T = 1.0 (J = 1):
  double cosh_fac = (3.0+cosh(8/T));
  double E_exact = -8.0*sinh(8/T)/cosh_fac*norm2;
  double Cv_exact = (8.0/T)*(8.0/T)*(1.0+3*cosh(8/T))/(cosh_fac*cosh_fac)*norm2;
  double Suscept_exact = 1/T*(12.0+8.0*exp(8.0/T)+8*cosh(8.0/T))/(cosh_fac*cosh_fac)*norm2;
  double absM_exact = (2.0*exp(8.0/T)+4.0)/cosh_fac*norm2;
  // Test print:
  ofile << setw(15) << setprecision(8) << T;
  ofile << setw(15) << setprecision(8) << E_exact;
  ofile << setw(15) << setprecision(8) << Cv_exact;
```

```
  ofile << setw(15) << setprecision(8) << 0.0;
  ofile << setw(15) << setprecision(8) << Suscept_exact;
  ofile << setw(15) << setprecision(8) << absM_exact << endl;
  // Print out relative errors:
  ofile << "Diff(E):          Diff(C_V):    Diff(Chi):     Diff(abs(M)):" << endl;
  ofile << setw(15) << setprecision(8) << abs(Eaverage*norm2-E_exact)/abs(E_exact);
  ofile << setw(15) << setprecision(8) << abs(Evariance/temperature/temperature-Cv_exact)/C
  ofile << setw(15) << setprecision(8) << abs(Mvariance/temperature-Suscept_exact)/Suscept_
  ofile << setw(15) << setprecision(8) << abs(Mabsaverage*norm2-absM_exact)/absM_exact << e

} // end output function
```

../Code/p4c.cpp

```
//   Project 4 c)
//   Program to study various expectation values as functions
//   of the number of Monte Carlo cycles. We use a 20x20 grid Ising model
//   with periodic boundary conditions. Values are plotted with python script.

#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <time.h>
#include <random>
#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&, double&, int&);
// Function to initialise energy and magnetization
void initialize(int, double, int **, double&, double&);
// The metropolis algorithm
void Metropolis(int, long&, int **, double&, double&, double *, int&);
// prints to file the results of the calculations
void output(int, int, double *, int);

int main(int argc, char* argv[])
{
  char *outfilename;
  long idum;
  int **spin_matrix, n_spins, mcs, mcs_i;
  double w[17], average[5], temperature, E, M;
  int no_accepted = 0;

  // Read in output file, abort if there are too few command-line arguments
  if( argc <= 1 ){
```

```
41      cout << "Bad Usage: " << argv[0] <<
42        " read also output file on same line" << endl;
43      exit(1);
44    }
45    else{
46      outfilename=argv[1];
47    }
48    read_input(n_spins, temperature, mcs);
49    // Write header in output file:
50    ofile.open(outfilename);
51    ofile << setiosflags(ios::showpoint | ios::uppercase);
52    ofile << "Final number of Monte Carlo trials: " << mcs << ", and temperature: " << tempe
53    ofile << "     Energy:      abs(Magnetization):    No of accepted moves: " << endl;
54    // Read in initial values such as size of lattice, temp and cycles
55    idum = -3; // random starting point
56    spin_matrix = (int**) matrix(n_spins, n_spins, sizeof(int));
57    //    initialise energy and magnetization
58    E = M = 0.;
59    // setup array for possible energy changes
60    for( int de =-8; de <= 8; de++) w[de+8] = 0;
61    for( int de =-8; de <= 8; de+=4) w[de+8] = exp(-de/temperature);
62    // initialise array for expectation values
63    for( int i = 0; i < 5; i++) average[i] = 0.;
64    initialize(n_spins, temperature, spin_matrix, E, M);
65    // Printing initial values of E and M to the result file:
66    double initial_Eaverage = E/n_spins/n_spins;
67    double initial_Mabsaverage = fabs(M)/n_spins/n_spins;
68    ofile << setw(15) << setprecision(8) << initial_Eaverage;
69    ofile << setw(15) << setprecision(8) << initial_Mabsaverage;
70    ofile << setw(15) << setprecision(8) << 0 << endl;
71    // start Monte Carlo computation:
72    for (int cycles = 1; cycles <= mcs; cycles++){
73      mcs_i = cycles; // Current number of cycles.
74      Metropolis(n_spins, idum, spin_matrix, E, M, w, no_accepted);
75      // update expectation values
76      average[0] += E;    average[1] += E*E;
77      average[2] += M;    average[3] += M*M;   average[4] += fabs(M);
78      output(n_spins, mcs_i, average, no_accepted);
79    }
80    // print results
81    free_matrix((void **) spin_matrix); // free memory
82
83    ofile.close();  // close output file
84    return 0;
85  }
86
87  // read in input data
88  void read_input(int& n_spins, double& temperature, int& mcs)
89  {
90    cout << "Final number of MC trials: ";
91    cin >> mcs;
92    cout << "Lattice size or number of spins (x and y equal): ";
93    cin >> n_spins;
94    cout << "Temperature with dimension energy: ";
```

```
  cin >> temperature;
} // end of function read_input


// function to initialise energy, spin matrix and magnetization
void initialize(int n_spins, double temperature, int **spin_matrix,
                double& E, double& M)
{

  // Setup spin matrix and intial magnetization; all spins up.
  // Commment/uncomment this section depending on use
  // Ground state configuration gives good convergence for low temperatures

  // for(int y =0; y < n_spins; y++) {
  //   for (int x= 0; x < n_spins; x++){
  //     spin_matrix[y][x] = 1; // spin orientation for the ground state
  //     M +=  (double) spin_matrix[y][x];
  //   }
  // }

  // Setup spin matrix and intial magnetization; random start configuration.
  // Commment/uncomment this section depending on use.
  // Random spin configuration gives good convergence for high temperatures

  long idum_dum = -2;
  for(int y =0; y < n_spins; y++) {
     for (int x= 0; x < n_spins; x++){
       double a = (double) ran1(&idum_dum);
       int r = 1;
       if (a < 0.5) {r = -1;}
       //cout << r;
       spin_matrix[y][x] = r; // spin orientation for the random state
       M += spin_matrix[y][x];
     }
     //cout << endl;
  }
  cout << "Initial magnetization: " << M << endl;

  // 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

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++){
      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]);
      if ( ran1(&idum) <= w[deltaE+8] ) {
  spin_matrix[iy][ix] *= -1;  // flip one spin and accept new spin config
        M += (double) 2*spin_matrix[iy][ix];
        E += (double) deltaE;
        acceptance += 1;
      }
    }
  }
} // end of Metropolis sampling over spins

void output(int n_spins, int mcs_i, double *average, int accepted)
{
  double norm = 1/((double) (mcs_i));  // divide by total number of cycles
  double norm2 = 1.0/(n_spins*n_spins);   // divide by the total number of spins
  double Eaverage = average[0]*norm;
  double Mabsaverage = average[4]*norm;
  // all expectation values are per spin, divide by 1/n_spins/n_spins
  ofile << setw(15) << setprecision(8) << Eaverage*norm2;
  ofile << setw(15) << setprecision(8) << Mabsaverage*norm2;
  ofile << setw(15) << setprecision(8) << accepted << endl;
} // end output function
```

../Code/p4d.cpp

```
//   Project 4 d)
//   Program to compute probability distribution (of energy) of steady state in
//   a 20x20 grid periodic Ising model. Histogram is plotted and computed
//   with periodic boundary conditions. Values are plotted with python script.

#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <time.h>
#include <random>
#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&, double&, int&);
// Function to initialise energy and magnetization
```

```
25 void initialize(int, double, int **, double&);
26 // The metropolis algorithm
27 void Metropolis(int, long&, int **, double&, double *);
28 // prints to file the results of the calculations
29 void output(int, double);
30
31 int main(int argc, char* argv[])
32 {
33   char *outfilename;
34   long idum;
35   int **spin_matrix, n_spins, mcs;
36   double w[17], temperature, E;
37   int steady_state_tolerance_cycles = 5E3;
38   double meanE2, meanE, Evariance, norm;
39
40   // Read in output file, abort if there are too few command-line arguments
41   if( argc <= 1 ){
42     cout << "Bad Usage: " << argv[0] <<
43       " read also output file on same line" << endl;
44     exit(1);
45   }
46   else { outfilename=argv[1]; }
47   read_input(n_spins, temperature, mcs);
48   // Normalize with the number of spins in the grid:
49   double norm2 = 1.0/((double) (n_spins*n_spins));
50   // Write header in output file:
51   ofile.open(outfilename);
52   ofile << setiosflags(ios::showpoint | ios::uppercase);
53   ofile << "Final number of Monte Carlo trials: " << mcs << ", and temperature: " << tempe
54   ofile << "Energy for each MC cycle after reached steady state. Computation started after
55   ofile << steady_state_tolerance_cycles << " cycles." << endl;
56   //    Read in initial values such as size of lattice, temp and cycles
57   idum = -1; // random starting point
58   spin_matrix = (int**) matrix(n_spins, n_spins, sizeof(int));
59   //    initialise energy and magnetization
60   E = 0.;
61   meanE2 = meanE = 0.0;
62   // setup array for possible energy changes
63   for( int de =-8; de <= 8; de++) w[de+8] = 0;
64   for( int de =-8; de <= 8; de+=4) w[de+8] = exp(-de/temperature);
65   // initialise array for expectation values
66   initialize(n_spins, temperature, spin_matrix, E);
67   // start Monte Carlo computation:
68   for (int cycles = 1; cycles <= mcs; cycles++){
69     Metropolis(n_spins, idum, spin_matrix, E, w);
70     // update expectation values if tolerance cycle is passed:
71     if (cycles >= steady_state_tolerance_cycles) {
72       output(n_spins, E);
73       // We calculate the averages per spin this time:
74       meanE += E*norm2; meanE2 += E*E*norm2*norm2;
75       }
76
77   }
78   // Calculate variance in energy for the cycles that occured after
```

```
  // reaching the steady state:
  norm = 1/((double) (mcs - steady_state_tolerance_cycles+1));
  meanE *= norm;
  meanE2 *= norm;

  // Print some results for verification:
  cout << "mean E per spin:   " << meanE << endl;
  cout << "mean E^2 per spin: " << meanE2 << endl;

  Evariance = meanE2 - meanE*meanE;
  ofile << " Final variance in energy per spin: ";
  ofile << setw(15) << setprecision(8) << Evariance << endl;
  // print results
  free_matrix((void **) spin_matrix); // free memory

  ofile.close();  // close output file
  return 0;
}

// read in input data
void read_input(int& n_spins, double& temperature, int& mcs)
{
  cout << "Final number of MC trials: ";
  cin >> mcs;
  cout << "Lattice size or number of spins (x and y equal): ";
  cin >> n_spins;
  cout << "Temperature with dimension energy: ";
  cin >> temperature;
} // end of function read_input


// function to initialise energy, spin matrix and magnetization
void initialize(int n_spins, double temperature, int **spin_matrix,
                double& E)
{
  // Setup spin matrix and intial magnetization; all spins up.
  // Commment/uncomment this section depending on use
  // Ground state configuration gives good convergence for low temperatures

  // for(int y =0; y < n_spins; y++) {
  //   for (int x= 0; x < n_spins; x++){
  //     spin_matrix[y][x] = 1; // spin orientation for the ground state
  //   }
  // }

  // Setup spin matrix and intial magnetization; random start configuration.
  // Commment/uncomment this section depending on use.
  // Random spin configuration gives good convergence for high temperatures

  long idum_dum = 2;
  for(int y =0; y < n_spins; y++) {
     for (int x= 0; x < n_spins; x++){
       double a = (double) ran1(&idum_dum);
       int r = 1;
```

```
       if (a < 0.5) {r = -1;}
       spin_matrix[y][x] = r; // spin orientation for the random state
     }
   }

  // 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

void Metropolis(int n_spins, long& idum, int **spin_matrix, double &E, double *w)
{
  // loop over all spins
  for(int y =0; y < n_spins; y++) {
    for (int x= 0; x < n_spins; x++){
      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]);
      if ( ran1(&idum) <= w[deltaE+8] ) {
  spin_matrix[iy][ix] *= -1;  // flip one spin and accept new spin config
        E += (double) deltaE;
      }
    }
  }
} // end of Metropolis sampling over spins

void output(int n_spins, double E)
{
  double norm2 = 1.0/(n_spins*n_spins);   // divide by the total number of spins
  double energy = E*norm2;
  // all expectation values are per spin, divide by 1/n_spins/n_spins
  ofile << setw(15) << setprecision(8) << energy << endl;
} // end output function
```

../Code/p4e.cpp

```
// Project 4 e)
// Plotting various quantities as functions of temperature to see indications
// of a phase transition. We study a periodic Ising model for sizes 20x20,
// 40x40, 60x60 and 80x80 in the temperature range T = [2.0,2.4].
// Exact transition occurs at T_c = 2.269 in the thermodynamic limit.

#include <cmath>
#include <iostream>
#include <fstream>
```

```
#include <iomanip>
#include "lib.h"
#include <time.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 *, double);

int main(int argc, char* argv[])
{
  char *outfilename;
  long idum;
  int **spin_matrix, n_spins, mcs, mcs_i;
  double w[17], average[5], initial_temp, final_temp, E, M, temp_step;
  // This should really be adjusted as temperature is changed:
  int steady_state_tolerance_cycles = 5E3;
  double calculation_time;

  // 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);
  int effective_mcs = mcs - steady_state_tolerance_cycles;
  int initial_temp_bol = 1; // Boolean variable: 1 if initial configuration.
  spin_matrix = (int**) matrix(n_spins, n_spins, sizeof(int));
  // Initialization of spin matrix (ordered initial state for low temps):
  E = M = 0.;
  double temperature = initial_temp;
  initialize(n_spins, temperature, spin_matrix, E, M);

  idum = -1; // random starting point
  for (double temperature = initial_temp; temperature <= final_temp; temperature+=temp_step
    // setup array for possible energy changes
    for( int de =-8; de <= 8; de++) w[de+8] = 0;
```

```
64      for( int de =-8; de <= 8; de+=4) w[de+8] = exp(-de/temperature);
65      // initialise array for expectation values
66      for( int i = 0; i < 5; i++) average[i] = 0.;
67
68      // Manual initialization here not needed
69      // - use previous spin matrix as initial for the next computation.
70
71      //initialize(n_spins, temperature, spin_matrix, E, M);
72      // start Monte Carlo computation
73      clock_t start, finish;
74      start = clock();
75      for (int cycles = 1; cycles <= mcs; cycles++){
76        Metropolis(n_spins, idum, spin_matrix, E, M, w);
77        // update expectation values
78        // Initialize time:
79        if (cycles >= steady_state_tolerance_cycles) {
80          //cout << "Hit! Average contributions counted. cycles = " << cycles << endl;
81          average[0] += E;    average[1] += E*E;
82          average[2] += M;    average[3] += M*M; average[4] += fabs(M);
83        }
84      }
85      finish = clock();
86      calculation_time = (finish - start)/(double)CLOCKS_PER_SEC;
87      // write final results to file:
88      //cout << "Final effective number of cycles: " << effective_mcs << endl;
89      output(n_spins, effective_mcs, temperature, average, calculation_time);
90    }
91    free_matrix((void **) spin_matrix); // free memory
92    ofile.close();  // close output file
93    return 0;
94  }
95
96  // read in input data
97  void read_input(int& n_spins, int& mcs, double& initial_temp,
98          double& final_temp, double& temp_step)
99  {
100   cout << "Number of Monte Carlo trials: ";
101   cin >> mcs;
102   cout << "Lattice size or number of spins (x and y equal): ";
103   cin >> n_spins;
104   cout << "Initial temperature with dimension energy: ";
105   cin >> initial_temp;
106   cout << "Final temperature with dimension energy: ";
107   cin >> final_temp;
108   cout << "Temperature step with dimension energy: ";
109   cin >> temp_step;
110 } // end of function read_input
111
112
113 // function to initialise energy, spin matrix and magnetization
114 void initialize(int n_spins, double temperature, int **spin_matrix,
115         double& E, double& M)
116 {
117   // setup spin matrix and intial magnetization
```

```
118   for(int y =0; y < n_spins; y++) {
119     for (int x= 0; x < n_spins; x++){
120       spin_matrix[y][x] = 1; // spin orientation for the ground state
121       M +=  (double) spin_matrix[y][x];
122     }
123   }
124 
125   // long idum_dum = -2;
126   // for(int y =0; y < n_spins; y++) {
127   //    for (int x= 0; x < n_spins; x++){
128   //      double a = (double) ran1(&idum_dum);
129   //      int r = 1;
130   //      if (a < 0.5) {r = -1;}
131   //      //cout << r;
132   //      spin_matrix[y][x] = r; // spin orientation for the random state
133   //      M += spin_matrix[y][x];
134   //    }
135   //    //cout << endl;
136   // }
137   // setup initial energy
138   for(int y =0; y < n_spins; y++) {
139     for (int x= 0; x < n_spins; x++){
140       E -=  (double) spin_matrix[y][x]*
141     (spin_matrix[periodic(y,n_spins,-1)][x] +
142      spin_matrix[y][periodic(x,n_spins,-1)]);
143     }
144   }
145 }// end function initialise
146 
147 void Metropolis(int n_spins, long& idum, int **spin_matrix, double& E, double&M, double *w
148 {
149   // loop over all spins
150   for(int y =0; y < n_spins; y++) {
151     for (int x= 0; x < n_spins; x++){
152       int ix = (int) (ran1(&idum)*(double)n_spins);
153       int iy = (int) (ran1(&idum)*(double)n_spins);
154       int deltaE =  2*spin_matrix[iy][ix]*
155     (spin_matrix[iy][periodic(ix,n_spins,-1)]+
156      spin_matrix[periodic(iy,n_spins,-1)][ix] +
157      spin_matrix[iy][periodic(ix,n_spins,1)] +
158      spin_matrix[periodic(iy,n_spins,1)][ix]);
159       if ( ran1(&idum) <= w[deltaE+8] ) {
160     spin_matrix[iy][ix] *= -1;  // flip one spin and accept new spin config
161         M += (double) 2*spin_matrix[iy][ix];
162         E += (double) deltaE;
163       }
164     }
165   }
166 } // end of Metropolis sampling over spins
167 
168 
169 void output(int n_spins, int mcs, double temperature, double *average, double calc_time)
170 {
171   double norm = 1/((double) (mcs));  // divided by total number of cycles
```

```
  double Eaverage = average[0]*norm;
  double E2average = average[1]*norm;
  double Maverage = average[2]*norm;
  double M2average = average[3]*norm;
  double Mabsaverage = average[4]*norm;
  // all expectation values are per spin, divide by 1/n_spins/n_spins
  double Evariance = (E2average - Eaverage*Eaverage)/n_spins/n_spins;
  double Mvariance = (M2average - Mabsaverage*Mabsaverage)/n_spins/n_spins;
  ofile << setiosflags(ios::showpoint | ios::uppercase);
  ofile << setw(15) << setprecision(8) << temperature;
  ofile << setw(15) << setprecision(8) << Eaverage/n_spins/n_spins;
  ofile << setw(15) << setprecision(8) << Evariance/temperature/temperature;
  ofile << setw(15) << setprecision(8) << Maverage/n_spins/n_spins;
  ofile << setw(15) << setprecision(8) << Mvariance/temperature;
  ofile << setw(15) << setprecision(8) << Mabsaverage/n_spins/n_spins;
  ofile << setw(15) << setprecision(8) << calc_time << endl;;
} // end output function
```

References

[1] M. Hjorth-Jensen, *Computational Physics*, University of Oslo (2015).