```
//  file: ising_model.cpp
//
//  Program to explore aspects of Monte Carlo simulation with the
//   Metropolis algorithm using the one- and two-dimensional Ising model.
//
//  Programmers:  Sung Yong Park  parksy@pacific.mps.ohio-state.edu
//                Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      29-Apr-2004  original version (mc2d.cpp, mc2d.cooling.cpp
//                    by S.Y. Park
//      08-Mar-2004  added seeded GSL random number generators
//                    (see random_seed.cpp) and made compatible with
//                    sampling_test.cpp
//      20-Feb-2005  minor changes to comments
//      20-Feb-2007  eliminated compile stuff,
//
//  Notes:
//   * uses the GSL random number functions and random_seed() to seed them.
//   * uses the GSL random number functions and random_seed(),
//     and both the gsl_rng.h and gsl_randist.h header files are needed.
//
//*********************************************************************

// include files
#include <iostream>             // cout and cin
#include <iomanip>              // manipulators like setprecision
#include <fstream>              // file input and output
#include <cmath>
using namespace std;            // we need this when .h is omitted

#include <gsl/gsl_rng.h>        // GSL random number generators
#include <gsl/gsl_randist.h>    // GSL random distributions

// function prototypes
extern unsigned long int random_seed ();        // routine to generate a seed
double calculate_energy (int configuration[]);  // calculate the energy given
                                                //  a spin configuration
// global constants
const double J_ising = 1.;      // The "J" in the Ising model (+1 or -1 ONLY)
const int linear_sites = 20;    // number of lattice sites in one direction

// For one dimensional Ising model, uncomment the next two lines:
/*
const int num_sites = linear_sites;
const int dimension = 1;
*/
// For two dimensional Ising model, uncomment the next two lines
const int num_sites = linear_sites * linear_sites;
const int dimension = 2;

                                // number of different energies
const int num_energies = 2 * dimension * num_sites + 1;
const int num_mcs = 1000;       // # of Monte Carlo steps (mcs)


//*************************************************************************
int
main (void)
{
  double kT = 2.;           // temperature (in energy units)
  cout << "What temperature? (kT) ";
  cin >> kT;

  double dist_metropolis[num_energies]; // energy distribution at kT from
                                        //  importance sampling (Metropolis)
  // initialize energy distribution histogram to zero
  for (int i = 0; i < num_energies; i++)
  {
```



```
    dist_metropolis[i] = 0.;
  }

  //  Set up the GSL random number generators (rng's)
  gsl_rng *rng_ptr = gsl_rng_alloc (gsl_rng_taus);   // allocate an rng
  gsl_rng_set (rng_ptr, random_seed());              // seed the rng

  // Open up an output file
  ofstream out;
  out.open ("ising_model.dat");
  out << "# Ising model in " << dimension << "dimensions at kT = "
      << kT << endl;
  out << "# time   energy " << endl;


  // Find the energy distribution from a Markov chain of configurations
  double energy, energy0;
  int config_metropolis[num_sites];   // current configuration

  // generate a random configuration to start and find its energy
  for (int i = 0; i < num_sites; i++)
  {
    double random = gsl_ran_flat (rng_ptr, 0., 1.);
    if (random > 0.5)
    {
      config_metropolis[i] = -1;  // spin down
    }
    else
    {
      config_metropolis[i] = +1;  // spin up
    }
    energy0 = calculate_energy ( config_metropolis );
  }

  // Take num_mcs Monte Carlo steps (mcs)
  for (int step = 0; step < num_mcs; step++)
  {
    for (int i = 0; i < num_sites; i++)  // Entire loop is only one mcs
    {
      // pick a random lattice site
      double random = gsl_ran_flat (rng_ptr, 0., 1.);
      int id = int(random * num_sites);  // from 0 to num_sites

      // flip that spin (i.e., if +/- 1, change to -/+ 1)
      config_metropolis[id] *= -1;

      energy = calculate_energy( config_metropolis );  // new energy
      double delta_energy = energy - energy0;

      // decide whether to accept or reject the new configuration
      random = gsl_ran_flat (rng_ptr, 0., 1.);
      if ( (delta_energy > 0.) && (random > exp(-delta_energy/kT)) )
      {
        // reject the new configuration: flip the spin back
        config_metropolis[id] *= -1;
      }
      else
      {
        energy0 = energy;  // accept the new configuration
      }
    }
    // add to distribution
    dist_metropolis[dimension * num_sites + (int)energy0] += 1.;

    // print out every once in a while the current energy
    if ( (step < 100) || (step % 10 == 0) )
    {
      out << fixed << " " << setw(5) << step << "  "
          << fixed << setprecision(3)
```

```
          << setw(10) << energy0 << endl;
    }
  }
  out.close ();
  cout << "Time evolution of energy output to ising_model.dat" << endl;

  // normalize the distribution
  for (int i = 0; i < num_energies; i++)
  {
    dist_metropolis[i] /= double(num_mcs);
  }


  //*********************************************************************
  // output the distributions of energies P(E)
  ofstream histogram;
  histogram.open ("ising_model_histogram.dat");
  histogram << "# energy   metropolis " << endl;
  for (int i = 0; i < num_energies; i += 2)
  {
    histogram << fixed << " " << setw(5) << i - dimension*num_sites << " "
              << fixed << setprecision(8)
              << setw(11) << dist_metropolis[i] << " "
              << endl;
  }
  histogram << endl;
  histogram.close();
  cout << "Energy distribution P(E) output to ising_model_histogram.dat"
       << endl;

  return (0);
}

//************************ calculate_energy ******************************
//
//  Given the array of integers configuration[0...num_sites-1], which
//   specifies the spin at each lattice point, find the energy of that
//   configuration [eq.(12.6) in Session 12 notes].
//  Note that a freeboundary condition is specified here.
//
//*********************************************************************
double
calculate_energy (int configuration[])
{
  int nearest = 0;
  double energy = 0.;

  if (dimension == 1)
  {
    for (int i = 0; i < num_sites - 1; i++)  // step through all but one site
    {
      nearest = i + 1;   // nearest neighbor on a line
      energy += - J_ising * double(configuration[i] * configuration[nearest]);
    }

    // now for the last site using periodic boundary conditions
    nearest = num_sites - 1;
    energy += - J_ising * double(configuration[nearest] * configuration[0]);
  }
  else if (dimension == 2)
  {
  // go through the 2d lattice with free boundary conditions
    for (int i = 0; i < linear_sites - 1; i++)
    {
      for (int j = 0; j < linear_sites - 1; j++)
      {
        int id = i + j * linear_sites;

        // x direction
```



```
        nearest = id + 1;
        energy += - J_ising * double(configuration[id]*configuration[nearest]);

        // y direction;
        nearest = id + linear_sites;
        energy += - J_ising * double(configuration[id]*configuration[nearest]);
      }
    }
  }

  return (energy);
}
```