```
//  file: sampling_test.cpp
//
//  Program to test random sampling vs. importance sampling against
//   the exact probability distribution of energies for a one-dimensional
//   Ising system at a given temperature.
//
//  Programmers:  Sung Yong Park  parksy@pacific.mps.ohio-state.edu
//                Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      29-Apr-2004  original version (exact.cpp, random_sampling.cpp,
//                    and mc.cpp) by S.Y. Park
//      08-Mar-2004  added seeded GSL random number generators
//                    (see random_seed.cpp) and combined into one code
//      19-Feb-2006  minor upgrades
//      22-Feb-2009  added energy_i (after various intermediate upgrades)
//
//  Notes:
//   * the units of energies are such that energies are always integers
//   * 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;

#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
int next_configuration (int configuration[], int k);  // find next configuration

// global constants
const double J_ising = 1.;      // The "J" in the Ising model (+1 or -1 ONLY)
const int num_sites = 20;       // number of lattice sites with spins
const int num_energies = 2*num_sites + 1;  // number of different energies
const int num_samples = 100000; // # of random samples or Monte Carlo steps
const double kT = 20.;          // temperature (in energy units)

// i'th possible energy (from -num_sites to +num_sites)
inline double energy_i (int i) {return double(i - num_sites);};

//*************************************************************************
int
main (void)
{
  int energy_count[num_energies];       // exact count of energies at kT
  double dist_exact[num_energies];      // exact energy distribution at kT
  double dist_random[num_energies];     // energy distribution from random
                                        //  sampling of configurations
  double dist_metropolis[num_energies]; // energy distribution at kT from
                                        //  importance sampling (Metropolis)

  // initialize all energy distribution histograms to zero
  for (int i = 0; i < num_energies; i++)
  {
    energy_count[i] = 0;
    dist_exact[i] = 0.;
    dist_random[i] = 0.;
    dist_metropolis[i] = 0.;
  }
```



```
  //*****************************************************************
  // Find the exact canonical ensemble energy probability distribution at kT

  // initialize the current configuration as all spin down
  int config_exact[num_sites];
  for (int i = 0; i < num_sites; i++)
  {
    config_exact[i] = -1;
  }

  // generate each successive configuration and find its energy
  int num_configs = 0;   // keep track of how many configurations are found
  int end = 0;
  do   // Note the "while" statement below --> check "end"
  {
     int k = 0;
     end = next_configuration (config_exact, k); // generate the next config.
     num_configs++;
     double energy = calculate_energy (config_exact);
     energy_count[num_sites + int(energy)]++;  // histogram the energies
     // Note that energy_count[i] is non-zero only for some i's
  }
  while (end != 1);  // end when all configs. are considered (so end=1)
  cout << num_configs << " total configurations" << endl << endl;

  // find the partition function at temperature kT
  double Z_exact = 0.;
  for (int i = 0; i < num_energies; i++)
  {   // this assumes the allowed energies are integers
    Z_exact += double(energy_count[i]) * exp(-energy_i(i)/kT);
  }

  // find the exact canonical energy distribution P(E)
  for (int i = 0; i < num_energies; i++)
  {
    dist_exact[i] = double(energy_count[i]) * exp(-energy_i(i)/kT) / Z_exact;
  }

  //*****************************************************************
  // Find the energy distribution from randomly selected configurations

  //  Use 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

  int config_random[num_sites];   // current configuration
  for (int config = 0; config < num_samples; config++)
  {
    for (int i = 0; i < num_sites; i++)
    {
      // generate a random number uniformly between 0. and 1.
      double random = gsl_ran_flat (rng_ptr, 0., 1.);
      if (random > 0.5)
      {
        config_random[i] = -1;  // spin down
      }
      else
      {
        config_random[i] = +1;  // spin up
      }
    }
    double energy = calculate_energy( config_random );
    dist_random[num_sites + int(energy)]++;
  }
  // normalize the energy distribution
  for (int i = 0; i < num_energies; i++)
  {
    dist_random[i] /= double(num_samples);
  }
```

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

// 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_samples Monte Carlo steps (mcs)
for (int config = 0; config < num_samples; config++)
{
  for (int i = 0; i < num_sites; i++)  // Entire loop is one mcs
  {
    // pick a random lattice site
    double random = gsl_ran_flat (rng_ptr, 0., 1.);
    int id = int(random * num_sites);   // compute the site number

    // flip that spin (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
    }
  }
  dist_metropolis[num_sites + int(energy0)] += 1.;  // add to distribution
}

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

//*****************************************************************
// output the distributions of energies P(E)
cout << "# energy   exact    random    metropolis " << endl;
for (int i = 0; i < num_energies; i += 4)  // Note the += 4 here!
{
  cout << fixed << " " << setw(5) << i - num_sites << " "
       << fixed << setprecision(8)
       << setw(11) << dist_exact[i] << " "
       << setw(11) << dist_random[i] << " "
       << setw(11) << dist_metropolis[i] << " "
       << endl;
}
```



```
  cout << 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 boundary condition for the endpoints is specified here.
//
//*********************************************************************
double
calculate_energy (int configuration[])
{
  int nearest = 0;
  double energy = 0.;

  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 the boundary condition
  nearest = num_sites - 1;
  energy += - J_ising * double(configuration[nearest] * configuration[0]);

  return (energy);
}

//************************ next_configuration ***************************
//
//  Given an input array of integers configuration[], which specifies the
//   spin at each lattice point and the current site, find the next
//   configuration.  The ordering corresponds to counting up in base 2,
//   with a down spin (-1) being 0 and an up spin (+1) being 1.
//  Calling next_configuration repeatedly steps through all of the possible
//   spin configurations.  This may be too clever to be reliable!
//
//*********************************************************************
int
next_configuration (int configuration[], int k)
{
  int end = 0;   // flag for when to quit

  if (configuration[k] == -1)
  {
    configuration[k] = +1;  // if the current spin is down, flip it
  }
  else
  {
    configuration[k] = -1;  // flip the current spin from up to down
    k++;
    if (k < num_sites)
    {
      end = next_configuration (configuration, k);  // recursion!!
    }
    else
    {
      end = 1;   // quit at this point
    }
  }

  return (end);
}
```