```
//  file: variational_SHO.cpp
//
//  Program to demonstrate variational Monte Carlo applied
//   to a one-dimensional simple harmonic oscillator (SHO).
//   The ground-state energy (exact = 1/2) is calculated as a function of
//   a variational parameter a, using the Metropolis algorithm
//   to evaluate the expectation value of the Hamiltonian.
//
//  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      02/21/06  original version, based on Variational_SHO_metro.cpp
//      03/03/09  fixed Psi and E_local
//      03/07/09  added setting of max_step (thanks to Kyle Wendt)
//
//  Notes:
//   * To use the VariationalMC class, you must create the "Psi"
//      and "E_local" functions.  The wavefunction Psi is assumed
//      to be normalized.  (Why does this matter?  How could this
//      restriction be lifted?)
//   * The probability distribution function (PDF) for the Metropolis
//      algorithm is fixed to be |Psi(x)|^2.
//   * Arguments are declared as const "references" using a & in front.
//      These are like aliases for the variables.
//
//*****************************************************************
// include files
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
using namespace std;

#include "VariationalMC.h"          // Variational Monte Carlo class

// function prototypes
double Psi (const double &x, const var_params &my_params); // Trial wavefunction
double E_local (const double &x, const var_params &my_params); // Local energy
inline double sqr (double x) { return x*x; };        // square a double

// main program
int
main ()
{
  var_params my_params = {0., 0.};  // declare variational parameter structure

  // request range of parameters and # of MC steps from user
  double a_min, a_max, a_step;
  cout << "Input a_min, a_max, a_step: " << endl;
  cin >> a_min >> a_max >> a_step;
  if (a_min <= 0.) { a_min = a_step; }   // force a_min >= a_step
  int MC_steps;
  cout << "Number of Monte Carlo steps: ";
  cin >> MC_steps;

  // declare a VariationalMC object and set some properties
  double lower_limit = -10.;
  double upper_limit = +10.;
  VariationalMC my_VariationalMC (lower_limit, upper_limit, &Psi, &E_local);
  my_VariationalMC.set_MC_steps (MC_steps);

  ofstream VMC_out;
  VMC_out.open ("variational_SHO.dat");
  VMC_out << "# a    energy    error "  << endl;
  for (double a_loop = a_min; a_loop <= a_max; a_loop += a_step)
  {
    my_params.a = a_loop;   // set the current parameter a
    my_VariationalMC.set_var_params (my_params);  // pass parameters to object
    // set step size based on size of a_loop; goal is 50% success rate
```



```
    my_VariationalMC.set_max_step ( 6./sqrt(a_loop) );

    my_VariationalMC.take_steps ();   // do the Monte Carlo steps

    VMC_out << fixed << setprecision(7)
            << a_loop << " " << my_VariationalMC.get_energy() << " "
            << my_VariationalMC.get_error() << endl;
    cout << "Success rate for a = " << a_loop
         << " is " << my_VariationalMC.get_success_rate() << endl;
  }
  VMC_out.close();

  cout << "Output to variational_SHO.dat" << endl;

  return (0);
}

//*************************************************************************
// Evaluation of wavefunction at a point x
double
Psi (const double &x, const var_params &my_params)
{
  double a = my_params.a;
  return ( sqrt(a/2.) / cosh(a*x) );
}
//*************************************************************************

//*************************************************************************
// Local energy function at point x: E_local = (H*Psi)/Psi
double
E_local (const double &x, const var_params &my_params)
{
  double a = my_params.a;
  return ( sqr(a)*(1./sqr(cosh(a*x)) - sqr(tanh(a*x)))/2. + sqr(x)/2. );
}
//*************************************************************************
```

```
//  file: VariationalMC.h
//
//  Header file for the VariationalMC C++ class.
//
//  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      02/21/06  original version
//      03/07/09  fixed max_step bug (thanks to Kyle Wendt)
//
//*************************************************************
#ifndef VARIATIONALMC_H
#define VARIATIONALMC_H

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

// structure
typedef struct
{
  double a;
  double b;
}
var_params;


class VariationalMC
{
 public:
  VariationalMC ( const double lower, const double upper,
      double (*Psi_passed)(const double &x, const var_params &my_params),
      double (*E_local_passed)(const double &x, const var_params &my_params) );

      // constructor
  ~VariationalMC ( );  //destructor

  // accessor functions --- these set and get private variables
  void set_limits (const double lower, const double upper)
                 {lower_limit = lower; upper_limit = upper;};
  void set_MC_steps (const int MCS) {MC_steps = MCS;};
  void set_max_step (const double step) {max_step = step;};
  void set_var_params (const var_params new_params)
      {psi_params = new_params;  max_step = -1.; }  // reset max_step
  double get_energy () {return energy;};
  double get_error ()  {return error;};
  double get_success_rate () {return success_rate;}

  int take_steps ();  // do the Monte Carlo steps

 private:
   gsl_rng *rng_ptr;  // pointer to GSL random number generator (rng)

   int MC_steps;      // number of Monte Carlo steps
   int initial_skip;  // thermalization skip
   int num_walkers;   // number of independent random walkers
   int num_steps;     // number of steps taken by each walker
   double max_step;   // -max_step < delta_x < max_step

   double lower_limit;   // lower bound of x
   double upper_limit;   // upper bound of x

   var_params psi_params;  // parameters for wave function psi

   double energy;          // average energy
   double energy_squared;  // average of energy squared
   double error;           //
   double success_rate;    // calculated Metropolis success percentage
```



```
   inline double sqr (double x) {return (x*x);};

   int rng_init();         // initialize the random number generator
   double get_random();    // get a random number between 0 and 1

   // Supplied pointer to functions for wavefunction and local energy
   double (*Psi)(const double &x, const var_params &my_params);
   double (*E_local)(const double &x, const var_params &my_params);

   double rho_PDF (double x_now);  // wave function squared at current x
};   // don't forget the ; here!

#endif
```

```
//  file: VariationalMC.cpp
//
//  Definitions for the VariationalMC C++ class.
//
//  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      02/22/06  original version, based on Variational_SHO_metro.cpp
//      03/03/09  fixed step size and initial skip bugs
//      03/07/09  fixed max_step bug (thanks to Kyle Wendt)
//
//  Notes:
//    * Many upgrades are needed.  max_step needs to be more general.
//
//*************************************************************
// include files
#include <cmath>
#include <iostream>

#include "VariationalMC.h"       // header file for VariationalMC class

extern unsigned long int random_seed ();   // routine to generate a seed

//*************************************************************************

// Constructor for VariationalMC
VariationalMC::VariationalMC ( const double lower, const double upper,
      double (*Psi_passed)(const double &x, const var_params &my_params),
      double (*E_local_passed)(const double &x, const var_params &my_params)  )
{
  lower_limit = lower;
  upper_limit = upper;
  Psi = Psi_passed;
  E_local = E_local_passed;

  // Set defaults
  initial_skip = 2000;
  num_walkers = 50;
  max_step = -1.;

  // initialize random number generator
  rng_init();
}
//*************************************************************************

// Copy constructor (use default copy constructor for now)
//*************************************************************************

// Destructor for VariationalMC
VariationalMC::~VariationalMC ()
{
   // put an appropriate destructor here
}
//*************************************************************************

// Initialize the random number generator (rng)
int
VariationalMC::rng_init ( )
{
  //  Use the GSL random number generators (rng's)
  rng_ptr = gsl_rng_alloc (gsl_rng_taus);       // allocate an rng
  gsl_rng_set (rng_ptr, random_seed ());        // seed the rng

  return (0);
}
//*************************************************************************

// Return a random number between 0 and 1
double
VariationalMC::get_random ( )
{
  return ( gsl_ran_flat(rng_ptr, 0., 1.) );
}
//*************************************************************************

// Return the Probability Density Function rho evaluated at x_now
double
VariationalMC::rho_PDF (double x_now)
{
  return ( sqr( Psi(x_now,psi_params) ) );
}
//*************************************************************************

// Take the Monte Carlo steps and calculate energy and error
int
VariationalMC::take_steps ()
{
  if (max_step < 0.) { max_step = 4./sqrt(psi_params.a); }  // default step size

  // find out how many steps each walker takes and allocate an array
  num_steps = int( double(MC_steps)/double(num_walkers) );
  double *x_walker = new double [num_walkers];

  // set initial point for each walker (within max_step of origin)
  for (int j = 0; j < num_walkers; j++)
  {
    x_walker[j] = max_step * (get_random() - 1./2.);
  }

  // do Metropolis walk
  int successes = 0;
  double energy_sum = 0;
  double energy_sq_sum = 0;
  for (int i = 0; i < num_steps + initial_skip; i++)  // allow for initial_skip
  {
    for (int j = 0; j < num_walkers; j++)
    {
      // take a trial step
      double x_trial = max_step * (get_random() - 1./2.);
      double ratio = rho_PDF(x_trial) / rho_PDF(x_walker[j]);
      double random = get_random();
      if (ratio >= random)
      {   // accept the step
        x_walker[j] = x_trial;
        if (i >= initial_skip) {successes++;}
      }

      // compute energy
      if (i >= initial_skip)
      {
        double E_L = E_local (x_walker[j], psi_params);
        energy_sum += E_L;
        energy_sq_sum += sqr(E_L);
      }
    }
  }

  double total_points = double(num_walkers * num_steps);
  energy = energy_sum / total_points;
  energy_squared = energy_sq_sum / total_points;
  error = sqrt( (energy_squared - sqr(energy)) / total_points );
  success_rate = double(successes) / total_points;

  delete [] x_walker;

  return (0);
}
//*************************************************************************
```