```
//  file: autocorrelation_test.cpp
//
//  C++ Program to demonstrate the use of an autocorrelation function.
//
//  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      05/15/04  original version, based on discussion in Pang
//      02/25/05  modified comments
//      02/20/06  switched sqr to inline function
//
//  Notes:
//   * Compile and link with:
//       g++ -Wall -o autocorrelation_test autocorrelation_test.cpp
//           -lgsl -lgslcblas
//   * gsl routines have built-in
//       extern "C" {
//          <header stuff>
//       }
//      so they can be called from C++ programs without modification
//
//*************************************************************************//

// include files
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
using namespace std;

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

// function prototypes
double full_integrand (double x);
double weight_function (double x);
double remaining_integrand (double x);

inline double sqr (double x) { return x*x; };        // square a double
extern unsigned long int random_seed ();   // routine to generate a seed

// global variables
double lower_limit = 0.;
double upper_limit = 10.;
double norm = 1./0.886227;  // norm for these integration limits

//*************************************************************************
int
main (void)
{
  // width of the integral
  double volume = upper_limit - lower_limit;

  //  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
  double random = 0.;   // this will be our random variable from 0 to 1

  int iterations = 10000;  // Monte Carlo iterations

  // first estimate the integral based on random samples
  double sum_random = 0.;     // accumulate the sum for the integral
  double sum_sq_random = 0.;  // accumulate the squares for the variance
  for (int i = 0; i < iterations; i++)
  {
    random = gsl_ran_flat (rng_ptr, 0., 1.);
    double x = lower_limit + random * (upper_limit - lower_limit);
    double result = full_integrand (x);
    sum_random += result;
    sum_sq_random += sqr (result);
  }

  double estimate_random = volume * sum_random  / double(iterations);
  double error_random = volume * sqrt( (sum_sq_random/double(iterations)
                                        - sqr(sum_random/double(iterations)))
                                        /double(iterations) );
  cout << "estimate from random sampling = " << estimate_random
       << " +/- " << error_random;

  // ---------------------------------------------------------------
  //
  // now estimate the integral based on Metropolis
  //
  double max_step = 0.1;    // maximum absolute change in x
  int initial_skip = 0;     // how many initial configurations to throw away
  int skip = 1;             // how often to accumulate a configuration
  double sum_metropolis = 0.;       // sum for the integral
  double sum_sq_metropolis = 0.;    // sum for the variance
  int successes = 0;

  // Initial configuration
  random = gsl_ran_flat (rng_ptr, 0., 1.);
  double x = lower_limit + random * (upper_limit - lower_limit);
  double weight_x = weight_function (x);

  // Loop through many Monte Carlo steps (MCS); each is just a (possible) new x
  for (int i = 0; i < initial_skip + iterations*skip; i++)
  {
    // Choose dx randomly distributed in -max_step < dx < max_step
    random = gsl_ran_flat (rng_ptr, 0., 1.);
    double dx = (2. * random - 1.) * max_step;

    double x_new = x + dx;
    if ((x_new < lower_limit) || (x_new > upper_limit))
    {
      // reject because outside of limits --> leave x unchanged
    }
    else
    {
      double weight_x_new = weight_function (x_new);
      random = gsl_ran_flat (rng_ptr, 0., 1.);
      if ( weight_x_new > (weight_x * random) )
      {
        // accept the move
        weight_x = weight_x_new;
        x = x_new;
        successes++;
      }
      else
      {
        // reject from Metropolis
      }
    }

    // accumulate the result every skip MCS
    if ((i >= initial_skip) && (i % skip == 0))
    {
      double result = remaining_integrand (x);
      sum_metropolis += result;
      sum_sq_metropolis += sqr (result);
    }

  }

  double estimate_metropolis = sum_metropolis / double(iterations);
  double error_metropolis = sqrt(fabs(sum_sq_metropolis/double(iterations)
     - sqr(estimate_metropolis))/double(iterations));
```

```
  cout << endl;
  cout << "    estimate from metropolis = " << estimate_metropolis
       << " +/- " << error_metropolis << endl;
  cout << "Metropolis acceptance rate: "
       << double(successes) / double(iterations*skip)
       << endl;

  return 0;
}

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

double
full_integrand (double x)
{
  return ( norm * sqr(x) * exp(-sqr(x)) );
}

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

double
weight_function (double x)
{
  return ( exp(-sqr(x)) );
}

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

double
remaining_integrand (double x)
{
  return ( sqr(x) );
}

//************************************************************************
```