```
//  file: eigen_basis.cpp
//
//  Program to find bound state eigenvalues for various potentials
//   by diagonalizing the Hamiltonian using the GSL eigenvalue/eigenvector
//   routines in a truncated harmonic oscillator basis.
//
//  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      01/24/04  original version, translated from eigen_basis.c
//      01/20/06  rearranged code to make it clearer
//
//  Notes:
//   * Based on the documentation for the GSL library under
//      "Eigensystems" and on Chap. 15 of "Computational Physics"
//      by Landau and Paez.
//   * Uses the GSL functions for computing eigenvalues
//      and eigenvectors of matrices.  The steps for this part are:
//       * define and allocate space for matrices and vectors we need
//       * load the matrix to be diagonalized (pointed to by Hmat_ptr)
//       * find the eigenvalues and eigenvectors with gsl_eigensymmv
//       * sort the results numerically
//       * print out the results
//   * As a convention (advocated in "Practical C++"), we'll append
//      "_ptr" to all pointers.
//   * When sorting eigenvalues,
//      GSL_EIGEN_SORT_VAL_ASC => ascending order in numerical value
//      GSL_EIGEN_SORT_VAL_DESC => descending order in numerical value
//      GSL_EIGEN_SORT_ABS_ASC => ascending order in magnitude
//      GSL_EIGEN_SORT_ABS_DESC => descending order in magnitude
//   * We use gls_integration_qagiu for the integrals from
//      0 to Infinity (calculating matrix elements of H).
//   * Start with l=0 (and generalize later)
//
//  To do:
//   * Add the Morse potential (function is given but not incorporated)
//   * Generalize to l>0.
//   * Make potential selection less kludgy.
//   * Improve efficiency (reduce run time)
//   * Split into more files (?) or convert to classes
//
///*********************************************************************

// include files
#include <iostream>             // note that .h is omitted
#include <iomanip>              // note that .h is omitted
#include <cmath>
using namespace std;

#include <gsl/gsl_eigen.h>              // gsl eigensystem routines
#include <gsl/gsl_integration.h>        // gsl integration routines

// structures and function prototypes
typedef struct                  // structure holding Hij parameters
{
  int i;                        // 1st matrix index
  int j;                        // 2nd matrix index
  double mass;                  // particle mass
  double b_ho;                  // harmonic oscillator parameter
  int potential_index;          // indicates which potential to use
}
hij_parameters;

typedef struct                  // structure holding potential parameters
{
  double param1;                // any three parameters
  double param2;
  double param3;
}
```



```
potential_parameters;

// potentials
double V_coulomb (double r, potential_parameters * potl_params_ptr);
double V_square_well (double r, potential_parameters * potl_params_ptr);
double V_morse (double r, potential_parameters * potl_params_ptr);

// i'th-j'th matrix element of Hamiltonian in ho basis
double Hij (hij_parameters ho_parameters);
double Hij_integrand (double x, void *params_ptr);

// harmonic oscillator routines from harmonic_oscillator.cpp
extern double ho_radial (int n, int l, double b_ho, double r);
extern double ho_eigenvalue (int n, int l, double b_ho, double mass);

// to square double precision numbers
inline double sqr (double x)  {return x*x;}

//************************* main program ***************************
int
main ()
{
  hij_parameters ho_parameters;  // parameters for the Hamiltonian

  // pick the potential based on the integer "answer"
  int answer = 0;
  while (answer != 1 && answer != 2)    // don't quit until 1 or 2!
    {
      cout << "Enter 1 for Coulomb or 2 for square well potential: ";
      cin >> answer;
    }
  ho_parameters.potential_index = answer;

  // Set up the harmonic oscillator basis
  double b_ho;                    // ho length parameter
  cout << "Enter the oscillator parameter b: ";
  cin >> b_ho;

  double mass = 1;                // measure mass in convenient units
  ho_parameters.mass = mass;
  ho_parameters.b_ho = b_ho;

  // pick the dimension of the basis (matrix)
  int dimension;                  // dimension of the matrices and vectors
  cout << "Enter the dimension of the basis: ";
  cin >> dimension;

  // See the GSL documentation for matrix, vector structures
  //  Define and allocate space for the vectors, matrices, and workspace
                                  // original gsl matrix with Hamiltonian
  gsl_matrix *Hmat_ptr = gsl_matrix_alloc (dimension, dimension);
                                  // gsl vector with eigenvalues
  gsl_vector *Eigval_ptr = gsl_vector_alloc (dimension);
                                  // gsl matrix with eigenvectors
  gsl_matrix *Eigvec_ptr = gsl_matrix_alloc (dimension, dimension);
                                  // the workspace for gsl
  gsl_eigen_symmv_workspace *worksp= gsl_eigen_symmv_alloc (dimension);

  // Load the Hamiltonian matrix pointed to by Hmat_ptr
  for (int i = 0; i < dimension; i++)
    {
      for (int j = 0; j < dimension; j++)
        {
          ho_parameters.i = i;
          ho_parameters.j = j;
          gsl_matrix_set (Hmat_ptr, i, j, Hij (ho_parameters));
          // print statement for debugging
          cout << "i = " << i << ", j = " << j
            << ", Hij = " << Hij (ho_parameters) << endl;
```

```
        }
    }

  // Find the eigenvalues and eigenvectors of the real, symmetric
  //  matrix pointed to by Hmat_ptr.  It is partially destroyed
  //  in the process. The eigenvectors are pointed to by
  //  Eigvec_ptr and the eigenvalues by Eigval_ptr.
  gsl_eigen_symmv (Hmat_ptr, Eigval_ptr, Eigvec_ptr, worksp);

  // Sort the eigenvalues and eigenvectors in ascending order
  gsl_eigen_symmv_sort (Eigval_ptr, Eigvec_ptr, GSL_EIGEN_SORT_VAL_ASC);

  // Print out the results
  // Allocate a pointer to one of the eigenvectors of the matrix
  gsl_vector *eigenvector_ptr = gsl_vector_alloc (dimension);
  for (int i = 0; i < dimension; i++)
    {
      double eigenvalue = gsl_vector_get (Eigval_ptr, i);
      gsl_matrix_get_col (eigenvector_ptr, Eigvec_ptr, i);

      cout << "eigenvalue " << i+1 << " = "
           << scientific << eigenvalue << endl;

      // Don't print the eigenvectors yet . . .
      // cout << "eigenvector = " << endl;
      // for (j = 0; j < dimension; j++)
      // {
      //   cout << scientific << gsl_vector_get (eigenvector_ptr, j) << endl;
      // }

    }

  // free the space used by the vector and matrices  and workspace
  gsl_matrix_free (Eigvec_ptr);
  gsl_vector_free (Eigval_ptr);
  gsl_matrix_free (Hmat_ptr);
  gsl_vector_free (eigenvector_ptr);
  gsl_eigen_symmv_free (worksp);

  return (0);                   // successful completion
}

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

//************************* Hij ***************************
//
// Calculate the i'th-j'th matrix element of the Hamiltonian
//  in a Harmonic oscillator basis.  This routine just passes
//  the integrand Hij_integrand to a GSL integration routine
//  (gsl_integration_qagiu) that integrates it over r from 0
//  to infinity
//
// Take l=0 only for now
//
//************************************************************
double
Hij (hij_parameters ho_parameters)
{
  gsl_integration_workspace *work = gsl_integration_workspace_alloc (1000);
  gsl_function F_integrand;

  double lower_limit = 0.;      // start integral from 0 (to infinity)
  double abs_error = 1.0e-8;    // to avoid round-off problems
  double rel_error = 1.0e-8;    // the result will usually be much better
  double result = 0.;           // the result from the integration
  double error = 0.;            // the estimated error from the integration

  void *params_ptr;             // void pointer passed to function
```



```
  params_ptr = &ho_parameters;  // we'll pass i, j, mass, b_ho

  // set up the integrand
  F_integrand.function = &Hij_integrand;
  F_integrand.params = params_ptr;

  // carry out the integral over r from 0 to infinity
  gsl_integration_qagiu (&F_integrand, lower_limit,
                         abs_error, rel_error, 1000, work, &result, &error);
  // eventually we should do something with the error estimate

  return (result);              // send back the result of the integration
}

//************************* Hij_integrand **************************
//
// The integrand for the i'th-j'th matrix element of the
//  Hamiltonian matrix.
//   * uses a harmonic oscillator basis
//   * the harmonic oscillator S-eqn was used to eliminate the
//      2nd derivative from the Hamiltonian in favor of the
//      HO energy and potential.  This was checked against an
//      explicit (but crude) 2nd derivative (now commented).
//
//************************************************************
double
Hij_integrand (double x, void *params_ptr)
{
  potential_parameters potl_params;     // parameters to pass to potential
  double Zesq;                  // Ze^2 for Coulomb potential
  double R, V0;                 // radius and depth of square well
  int potential_index;          // index 1,2,... for potental

  int l = 0;                    // orbital angular momentum
  int n_i, n_j;                 // principal quantum number (1,2,...)
  double mass, b_ho;            // local ho parameters
  double hbar = 1.;             // units with hbar = 1
  double omega;                 // harmonic oscillator frequency
  double ho_pot;                // value of ho potential at current x

  // define variables for debugging 2nd derivative
  double h = 0.01;
  double fp, f, fm, deriv2;


  n_i = ((hij_parameters *) params_ptr)->i + 1; // n starts at 1
  n_j = ((hij_parameters *) params_ptr)->j + 1;
  mass = ((hij_parameters *) params_ptr)->mass;
  b_ho = ((hij_parameters *) params_ptr)->b_ho;
  omega = hbar / (mass * b_ho * b_ho);  // definition of omega
  ho_pot = (1. / 2.) * mass * (omega * omega) * (x * x);        // ho pot'l
  potential_index = ((hij_parameters *) params_ptr)->potential_index;

  // debugging code to calculate 2nd derivative by hand
  f = ho_radial (n_j, l, b_ho, x);
  fp = ho_radial (n_j, l, b_ho, x + h);
  fm = ho_radial (n_j, l, b_ho, x - h);
  deriv2 = -((fp - f) - (f - fm)) / (h * h) / (2. * mass);


  // set up the potential according to potential index
  switch (potential_index)
    {
    case 1:                     // coulomb
      Zesq = 1.;
      potl_params.param1 = Zesq;
      return (ho_radial (n_i, l, b_ho, x)
              * (ho_eigenvalue (n_j, l, b_ho, mass) - ho_pot
                 + V_coulomb (x, &potl_params))
```

```
              * ho_radial (n_j, l, b_ho, x));
      break;
    case 2:                       // square well
      R = 1.;
      V0 = 50.;
      potl_params.param1 = V0;
      potl_params.param2 = R;
      return (ho_radial (n_i, l, b_ho, x)
              * (ho_eigenvalue (n_j, l, b_ho, mass) - ho_pot
                 + V_square_well (x, &potl_params))
              * ho_radial (n_j, l, b_ho, x));
      break;
    default:
      cout << "Shouldn't get here!\n";
      return (1);
      break;
    }


  // debugging code to use crude 2nd derivative
  // return (ho_radial (n_i, l, b_ho, x)
  //       * (deriv2 + V_coulomb (x, &potl_params)
  //          * ho_radial (n_j, l, b_ho, x)));

}

//************************* Potentials ************************

//************************* V_coulomb **************************
//
// Coulomb potential with charge Z:  Ze^2/r
//  --> hydrogen-like atom
//
//   Zesq stands for Ze^2
//
//**************************************************************
double
V_coulomb (double r, potential_parameters * potl_params_ptr)
{
  double Zesq = potl_params_ptr->param1;

  return (-Zesq / r);
}

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

//************************ V_square_well ***********************
//
// Square well potential of radius R and depth V0
//
//**************************************************************
double
V_square_well (double r, potential_parameters * potl_params_ptr)
{
  double V0 = potl_params_ptr->param1;
  double R = potl_params_ptr->param2;

  if (r < R)
    {
      return (-V0);               // inside the well of depth V0
    }
  else
    {
      return (0.);                // outside the well
    }
}

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



```
//************************ V_morse ***************************
//
// Morse potential with equilibrium bond length r_eq and potential
//  energy for bond formation D_eq
//
//**************************************************************
double
V_morse (double r, potential_parameters * potl_params_ptr)
{
  double D_eq = potl_params_ptr->param1;
  double r_eq = potl_params_ptr->param2;

  return ( D_eq * sqr(1. - exp(-(r-r_eq))) );
}

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