```
//  file: multifit_test.cpp
//
//  C++ Program to test a fitting routine from
//   the gsl numerical library.
//
//  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      12/27/03  original C++ version, modified from C version
//      05/01/04  added some comments and changed some variable names
//      02/15/05  added many more comments
//      02/12/06  minor improvements (inline sqr, cmath, declarations)
//
//  Notes:
//   * Example taken from the GNU Scientific Library Reference Manual
//      Edition 1.4, for GSL Version 1.4, August 2003
//   * Compile and link with:
//       g++ -Wall -o multifit_test multifit_test.cpp -lgsl -lgslcblas
//   * gsl routines have built-in
//       extern "C" {
//          <header stuff>
//       }
//      so they can be called from C++ programs without modification
//
//*************************************************************************//

// The following description is taken from the GSL documentation
//
//  The problem of multidimensional nonlinear least-squares fitting
//   requires the minimization of the squared residuals of n functions,
//   f_i, in p parameters, x_i,
//
//       \Phi(x) = (1/2) \sum_{i=1}^{n} f_i(x_1, ..., x_p)^2
//               = (1/2) || F(x) ||^2
//
//  All algorithms proceed from an initial guess using the linearization,
//
//       \psi(p) = || F(x+p) || ~=~ || F(x) + J p ||
//
//  where x is the initial point, p is the proposed step and J is the
//   Jacobian matrix J_{ij} = d f_i / d x_j. Additional strategies are
//   used to enlarge the region of convergence. These include requiring
//   a decrease in the norm ||F|| on each step or using a trust region
//   to avoid steps which fall outside the linear regime.
//
//
//
// The following example program fits a weighted exponential model with
// background to experimental data, Y = A \exp(-\lambda t) + b. The
// first part of the program sets up the functions expb_f and expb_df to
// calculate the model and its Jacobian. The appropriate fitting
// function is given by,
//
// f_i = ((A \exp(-\lambda t_i) + b) - y_i)/\sigma_i
//
// where we have chosen t_i = i. The Jacobian matrix J is the derivative
// of these functions with respect to the three parameters (A, \lambda,
// b). It is given by,
//
// J_{ij} = d f_i / d x_j
//
// where x_0 = A, x_1 = \lambda and x_2 = b.
//
//
// The iteration terminates when the change in x is smaller than 0.0001,
// as both an absolute and relative change. Here are the results of
// running the program,
//
// iter: 0 x = 1.00000000 0.00000000 0.00000000 |f(x)| = 118.574
```



```
// iter: 1 x = 1.64919392 0.01780040 0.64919392 |f(x)| = 77.2068
// iter: 2 x = 2.86269020 0.08032198 1.45913464 |f(x)| = 38.0579
// iter: 3 x = 4.97908864 0.11510525 1.06649948 |f(x)| = 10.1548
// iter: 4 x = 5.03295496 0.09912462 1.00939075 |f(x)| = 6.4982
// iter: 5 x = 5.05811477 0.10055914 0.99819876 |f(x)| = 6.33121
// iter: 6 x = 5.05827645 0.10051697 0.99756444 |f(x)| = 6.33119
// iter: 7 x = 5.05828006 0.10051819 0.99757710 |f(x)| = 6.33119
//
// A      = 5.05828 +/- 0.05983
// lambda = 0.10052 +/- 0.00309
// b      = 0.99758 +/- 0.03944
// status = success
//
// The approximate values of the parameters are found correctly. The
// errors on the parameters are given by the square roots of the
// diagonal elements of the covariance matrix.
//
//*************************************************************************

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

#include <gsl/gsl_errno.h>
#include <gsl/gsl_rng.h>            // gsl random number generators
#include <gsl/gsl_randist.h>        // gsl random number distributions
#include <gsl/gsl_vector.h>         // gsl vector and matrix definitions
#include <gsl/gsl_blas.h>           // gsl linear algebra stuff
#include <gsl/gsl_multifit_nlin.h>  // gsl multidimensional fitting

// function prototypes
int print_state (size_t iter, gsl_multifit_fdfsolver * s);
int expb_f (const gsl_vector * xvec_ptr, void *params, gsl_vector * f_ptr);
int expb_df (const gsl_vector * xvec_ptr, void *params,
             gsl_matrix * Jacobian_ptr);
int expb_fdf (const gsl_vector * xvec_ptr, void *params,
              gsl_vector * f_ptr, gsl_matrix * Jacobian_ptr);

double fit (int i, gsl_multifit_fdfsolver * solver_ptr);
double err (int i, gsl_matrix * covariance_ptr);

// global structure to hold data and errors
struct data
{
  size_t n;
  double *y;
  double *sigma;
};

const int N = 40;   // # of points in the data set

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

int
main (void)
{
  // The main part of the program sets up a Levenberg-Marquardt solver and
  // some simulated random data. The data uses the known parameters
  // (1.0,5.0,0.1) combined with gaussian noise (standard deviation = 0.1)
  // over a range of N = 40 timesteps. The initial guess for the parameters
  // is chosen (arbitrarily) as (0.0, 1.0, 0.0).

  const size_t n = N;  // # of points
  const size_t p = 3;  // # of parameters
  const double error = 0.1;  // standard deviation for all data oints
```

```
  // allocate space for a covariance matrix of size p by p
  gsl_matrix *covariance_ptr = gsl_matrix_alloc (p, p);

  double y[N], sigma[N];   // the data is in y and the error bar in sigma
  struct data my_data = { n, y, sigma };   // combine into a structure

  double x_init[3] = { 1.0, 0.0, 0.0 };  // initial guesse of parameters

  gsl_vector_view xvec_ptr = gsl_vector_view_array (x_init, p);

  // allocate and setup for generating gaussian distibuted random numbers
  gsl_rng_env_setup ();
  const gsl_rng_type *type = gsl_rng_default;
  gsl_rng *rng_ptr = gsl_rng_alloc (type);

  // set up the function to be fit
  gsl_multifit_function_fdf my_func;
  my_func.f = &expb_f;         // the function of residuals
  my_func.df = &expb_df;       // the gradient of this function
  my_func.fdf = &expb_fdf;     // combined function and gradient
  my_func.n = n;               // number of points in the data set
  my_func.p = p;               // number of parameters in the fit function
  my_func.params = &my_data;  // structure with the data and error bars

  // This sets up the data to be fitted, with gaussian noise added
  cout << endl << endl << "Initial data: " << endl;
  cout <<   " t    y(t)    sigma(t) " << endl;
  for (size_t i = 0; i < n; i++)
    {
      double t = (double) i;
      y[i] = 1.0 + 5 * exp (-0.1 * t) + gsl_ran_gaussian (rng_ptr, error);
      sigma[i] = error;

      // print out the data to make sure it looks ok
      cout << "  " << t << "  " << y[i] << "  " << sigma[i] << endl;
    };
  cout << endl;

  const gsl_multifit_fdfsolver_type *type_ptr = gsl_multifit_fdfsolver_lmsder;
  gsl_multifit_fdfsolver *solver_ptr
       = gsl_multifit_fdfsolver_alloc (type_ptr, n, p);
  gsl_multifit_fdfsolver_set (solver_ptr, &my_func, &xvec_ptr.vector);

  size_t max_iterations = 500;  // stop at this point if not converged
  size_t iteration = 0;         // initialize iteration counter
  print_state (iteration, solver_ptr);
  int status;           // return value from gsl function calls (e.g., error)
  do
    {
      iteration++;

      // perform a single iteration of the fitting routine
      status = gsl_multifit_fdfsolver_iterate (solver_ptr);

      // print out the status of the fit
      cout << "status = " << gsl_strerror (status) << endl;

      // customized routine to print out current parameters
      print_state (iteration, solver_ptr);

      if (status)    // check for a nonzero status code
        {
          break;  // this should only happen if an error code is returned
        }

      // test for convergence with an absolute and relative error (see manual)
      status = gsl_multifit_test_delta (solver_ptr->dx, solver_ptr->x,
                                        1e-4, 1e-4);
    }
```



```
  while (status == GSL_CONTINUE && iteration < max_iterations);

  // calculate the covariance matrix of the best-fit parameters
  gsl_multifit_covar (solver_ptr->J, 0.0, covariance_ptr);

  // print out the covariance matrix using the gsl function (not elegant!)
  cout << endl << "Covariance matrix: " << endl;
  gsl_matrix_fprintf (stdout, covariance_ptr, "%g");

  cout.setf (ios::fixed, ios::floatfield);      // output in fixed format
  cout.precision (5);                           // # of digits in doubles

  int width = 7;                // setw width for output
  cout << endl << "Best fit results:" << endl;
  cout << "A   = " << setw (width) << fit (0, solver_ptr)
    << " +/- " << setw (width) << err (0, covariance_ptr) << endl;

  cout << "B   = " << setw (width) << fit (1, solver_ptr)
    << " +/- " << setw (width) << err (1, covariance_ptr) << endl;

  cout << "C   = " << setw (width) << fit (2, solver_ptr)
    << " +/- " << setw (width) << err (2, covariance_ptr) << endl;

  cout << "status = " << gsl_strerror (status) << endl;

  gsl_multifit_fdfsolver_free (solver_ptr);  // free up the solver

  return 0;
}

//***************************************************************************
//  Simple function to print results of each iteration in nice format
//
int
print_state (size_t iteration, gsl_multifit_fdfsolver * solver_ptr)
{
  cout.setf (ios::fixed, ios::floatfield);      // output in fixed format
  cout.precision (4);           // digits in doubles

  int width = 8;                // setw width for output
  cout << "iteration " << iteration << ": "
    << " x = " << setw (width) << gsl_vector_get (solver_ptr->x, 0)
    << setw (width) << gsl_vector_get (solver_ptr->x, 1)
    << setw (width) << gsl_vector_get (solver_ptr->x, 2)
    << " |f(x)| = " << scientific << gsl_blas_dnrm2 (solver_ptr->f)
    << endl << endl;

  return 0;
}


//***************************************************************************
//  Function returning the residuals for each point; that is, the
//   difference of the fit function using the current parameters
//   and the data to be fit.
//
int
expb_f (const gsl_vector * xvec_ptr, void *params_ptr, gsl_vector * f_ptr)
{
  size_t n = ((struct data *) params_ptr)->n;
  double *y = ((struct data *) params_ptr)->y;
  double *sigma = ((struct data *) params_ptr)->sigma;

  double A = gsl_vector_get (xvec_ptr, 0);
  double lambda = gsl_vector_get (xvec_ptr, 1);
  double b = gsl_vector_get (xvec_ptr, 2);

  size_t i;
```

```
  for (i = 0; i < n; i++)
    {
      // Model Yi = A * exp(-lambda * i) + b
      double t = (double) i;
      double Yi = A * exp (-lambda * t) + b;
      gsl_vector_set (f_ptr, i, (Yi - y[i]) / sigma[i]);
    }

  return GSL_SUCCESS;
}

//***************************************************************************
//  Function returning the Jacobian of the residual function
//
int
expb_df (const gsl_vector * xvec_ptr, void *params_ptr,
         gsl_matrix * Jacobian_ptr)
{
  size_t n = ((struct data *) params_ptr)->n;
  double *sigma = ((struct data *) params_ptr)->sigma;

  double A = gsl_vector_get (xvec_ptr, 0);
  double lambda = gsl_vector_get (xvec_ptr, 1);

  size_t i;

  for (i = 0; i < n; i++)
    {
      // Jacobian matrix J(i,j) = dfi / dxj,
      // where fi = (Yi - yi)/sigma[i],
      //       Yi = A * exp(-lambda * i) + b
      // and the xj are the parameters (A,lambda,b)
      double t = (double) i;
      double sig = sigma[i];
      double elamt = exp (-lambda * t);
      gsl_matrix_set (Jacobian_ptr, i, 0, elamt / sig);
      gsl_matrix_set (Jacobian_ptr, i, 1, -t * A * elamt / sig);
      gsl_matrix_set (Jacobian_ptr, i, 2, 1 / sig);
    }
  return GSL_SUCCESS;
}

//***************************************************************************
//  Function combining the residual function and its Jacobian
//
int
expb_fdf (const gsl_vector * xvec_ptr, void *params_ptr,
          gsl_vector * f_ptr, gsl_matrix * Jacobian_ptr)
{
  expb_f (xvec_ptr, params_ptr, f_ptr);
  expb_df (xvec_ptr, params_ptr, Jacobian_ptr);

  return GSL_SUCCESS;
}

//***************************************************************************
//  Function to return the i'th best-fit parameter
//
inline double
fit (int i, gsl_multifit_fdfsolver * solver_ptr)
{
  return gsl_vector_get (solver_ptr->x, i);
}

//***************************************************************************
//  Function to retrieve the square root of the diagonal elements of
//   the covariance matrix.
//
inline double
```



```
err (int i, gsl_matrix * covariance_ptr)
{
  return sqrt (gsl_matrix_get (covariance_ptr, i, i));
}
```