
Multidimensional Root-Finding

This chapter describes functions for multidimensional root-finding (solving nonlinear systems with n equations in n unknowns). The library provides low level components for a variety of iterative solvers and convergence tests. These can be combined by the user to achieve the desired solution, with full access to the intermediate steps of the iteration. Each class of methods uses the same framework, so that you can switch between solvers at runtime without needing to recompile your program. Each instance of a solver keeps track of its own state, allowing the solvers to be used in multi-threaded programs. The solvers are based on the original Fortran library MINPACK.

The header file `gsl_multiroots.h' contains prototypes for the multidimensional root finding functions and related declarations.

Overview

The problem of multidimensional root finding requires the simultaneous solution of n equations, f_i, in n variables, x_i,

```
f_i (x_1, ..., x_n) = 0     for i = 1 ... n.
```

In general there are no bracketing methods available for n dimensional systems, and no way of knowing whether any solutions exist. All algorithms proceed from an initial guess using a variant of the Newton iteration,

```
x -> x' = x - J^{-1} f(x)
```

where x, f are vector quantities and J is the Jacobian matrix J_{ij} = d f_i / d x_j. Additional strategies can be used to enlarge the region of convergence. These include requiring a decrease in the norm |f| on each step proposed by Newton's method, or taking steepest-descent steps in the direction of the negative gradient of |f|.

Several root-finding algorithms are available within a single framework. The user provides a high-level driver for the algorithms, and the library provides the individual functions necessary for each of the steps. There are three main phases of the iteration. The steps are,

- initialize solver state, *s*, for algorithm *T*

- update *s* using the iteration *T*
- test *s* for convergence, and repeat iteration if necessary

The evaluation of the Jacobian matrix can be problematic, either because programming the derivatives is intractable or because computation of the n^2 terms of the matrix becomes too expensive. For these reasons the algorithms provided by the library are divided into two classes according to whether the derivatives are available or not.

The state for solvers with an analytic Jacobian matrix is held in a `gsl_multiroot_fdfsolver` struct. The updating procedure requires both the function and its derivatives to be supplied by the user.

The state for solvers which do not use an analytic Jacobian matrix is held in a `gsl_multiroot_fsolver` struct. The updating procedure uses only function evaluations (not derivatives). The algorithms estimate the matrix J or J^{-1} by approximate methods.

Initializing the Solver

The following functions initialize a multidimensional solver, either with or without derivatives. The solver itself depends only on the dimension of the problem and the algorithm and can be reused for different problems.

Function: gsl_multiroot_fsolver * **gsl_multiroot_fsolver_alloc** *(const gsl_multiroot_fsolver_type * T, size_t n)*

This function returns a pointer to a newly allocated instance of a solver of type *T* for a system of *n* dimensions. For example, the following code creates an instance of a hybrid solver, to solve a 3-dimensional system of equations.

```
const gsl_multiroot_fsolver_type * T
    = gsl_multiroot_fsolver_hybrid;
gsl_multiroot_fsolver * s
    = gsl_multiroot_fsolver_alloc (T, 3);
```

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

Function: gsl_multiroot_fdfsolver * **gsl_multiroot_fdfsolver_alloc** *(const gsl_multiroot_fdfsolver_type * T, size_t n)*

This function returns a pointer to a newly allocated instance of a derivative solver of type *T* for a system of *n* dimensions. For example, the following code creates an instance of a Newton-Raphson solver, for a 2-dimensional system of equations.

```
const gsl_multiroot_fdfsolver_type * T
    = gsl_multiroot_fdfsolver_newton;
gsl_multiroot_fdfsolver * s =
    gsl_multiroot_fdfsolver_alloc (T, 2);
```

If there is insufficient memory to create the solver then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

Function: int **gsl_multiroot_fsolver_set** *(gsl_multiroot_fsolver * s, gsl_multiroot_function * f, gsl_vector * x)*

This function sets, or resets, an existing solver *s* to use the function *f* and the initial guess *x*.

Function: int **gsl_multiroot_fdfsolver_set** *(gsl_multiroot_fdfsolver * s, gsl_multiroot_function_fdf * fdf, gsl_vector * x)*

This function sets, or resets, an existing solver *s* to use the function and derivative *fdf* and the initial guess *x*.

Function: void **gsl_multiroot_fsolver_free** *(gsl_multiroot_fsolver * s)*

Function: void **gsl_multiroot_fdfsolver_free** *(gsl_multiroot_fdfsolver * s)*

These functions free all the memory associated with the solver *s*.

Function: const char * **gsl_multiroot_fsolver_name** *(const gsl_multiroot_fsolver * s)*

Function: const char * **gsl_multiroot_fdfsolver_name** *(const gsl_multiroot_fdfsolver * s)*

These functions return a pointer to the name of the solver. For example,

```
printf ("s is a '%s' solver\n",
        gsl_multiroot_fdfsolver_name (s));
```

would print something like `s is a 'newton' solver`.

Providing the function to solve

You must provide n functions of n variables for the root finders to operate on. In order to allow for general parameters the functions are defined by the following data types:

Data Type: **gsl_multiroot_function**

This data type defines a general system of functions with parameters.

```
int (* f) (const gsl_vector * x, void * params,
gsl_vector * f)
```

this function should store the vector result f(x,params) in *f* for argument *x* and parameters

params, returning an appropriate error code if the function cannot be computed.

`size_t n`
the dimension of the system, i.e. the number of components of the vectors *x* and *f*.

`void * params`
a pointer to the parameters of the function.

Here is an example using Powell's test function,

```
f_1(x) = A x_0 x_1 - 1,
f_2(x) = exp(-x_0) + exp(-x_1) - (1 + 1/A)
```

with A = 10^4. The following code defines a `gsl_multiroot_function` system `F` which you could pass to a solver:

```
struct powell_params { double A; };

int
powell (gsl_vector * x, void * p, gsl_vector * f) {
   struct powell_params * params
     = *(struct powell_params *)p;
   const double A = (params->A);
   const double x0 = gsl_vector_get(x,0);
   const double x1 = gsl_vector_get(x,1);

   gsl_vector_set (f, 0, A * x0 * x1 - 1);
   gsl_vector_set (f, 1, (exp(-x0) + exp(-x1)
                          - (1.0 + 1.0/A)));
   return GSL_SUCCESS
}

gsl_multiroot_function F;
struct powell_params params = { 10000.0 };

F.f = &powell;
F.n = 2;
F.params = &params;
```

Data Type: **gsl_multiroot_function_fdf**

This data type defines a general system of functions with parameters and the corresponding Jacobian matrix of derivatives,

`int (* f) (const gsl_vector * x, void * params, gsl_vector * f)`
this function should store the vector result f(x,params) in *f* for argument *x* and parameters *params*, returning an appropriate error code if the function cannot be computed.

`int (* df) (const gsl_vector * x, void * params, gsl_matrix * J)`
this function should store the *n*-by-*n* matrix result J_ij = d f_i(x,params) / d x_j in *J* for argument *x* and parameters *params*, returning an appropriate error code if the function cannot be computed.

`int (* fdf) (const gsl_vector * x, void *`

params, gsl_vector * *f*, gsl_matrix * *J*)
: This function should set the values of the *f* and *J* as above, for arguments *x* and parameters *params*. This function provides an optimization of the separate functions for f(x) and J(x) -- it is always faster to compute the function and its derivative at the same time.

size_t n
: the dimension of the system, i.e. the number of components of the vectors *x* and *f*.

void * params
: a pointer to the parameters of the function.

The example of Powell's test function defined above can be extended to include analytic derivatives using the following code,

```
int
powell_df (gsl_vector * x, void * p, gsl_matrix * J)
{
   struct powell_params * params
     = *(struct powell_params *)p;
   const double A = (params->A);
   const double x0 = gsl_vector_get(x,0);
   const double x1 = gsl_vector_get(x,1);
   gsl_matrix_set (J, 0, 0, A * x1);
   gsl_matrix_set (J, 0, 1, A * x0);
   gsl_matrix_set (J, 1, 0, -exp(-x0));
   gsl_matrix_set (J, 1, 1, -exp(-x1));
   return GSL_SUCCESS
}

int
powell_fdf (gsl_vector * x, void * p,
            gsl_matrix * f, gsl_matrix * J) {
   struct powell_params * params
     = *(struct powell_params *)p;
   const double A = (params->A);
   const double x0 = gsl_vector_get(x,0);
   const double x1 = gsl_vector_get(x,1);

   const double u0 = exp(-x0);
   const double u1 = exp(-x1);

   gsl_vector_set (f, 0, A * x0 * x1 - 1);
   gsl_vector_set (f, 1, u0 + u1 - (1 + 1/A));

   gsl_matrix_set (J, 0, 0, A * x1);
   gsl_matrix_set (J, 0, 1, A * x0);
   gsl_matrix_set (J, 1, 0, -u0);
   gsl_matrix_set (J, 1, 1, -u1);
   return GSL_SUCCESS
}

gsl_multiroot_function_fdf FDF;

FDF.f = &powell_f;
FDF.df = &powell_df;
FDF.fdf = &powell_fdf;
FDF.n = 2;
```

```
FDF.params = 0;
```

Note that the function `powell_fdf` is able to reuse existing terms from the function when calculating the Jacobian, thus saving time.

Iteration

The following functions drive the iteration of each algorithm. Each function performs one iteration to update the state of any solver of the corresponding type. The same functions work for all solvers so that different methods can be substituted at runtime without modifications to the code.

Function: int **gsl_multiroot_fsolver_iterate** *(gsl_multiroot_fsolver * s)*
Function: int **gsl_multiroot_fdfsolver_iterate** *(gsl_multiroot_fdfsolver * s)*

These functions perform a single iteration of the solver *s*. If the iteration encounters an unexpected problem then an error code will be returned,

`GSL_EBADFUNC`
: the iteration encountered a singular point where the function or its derivative evaluated to `Inf` or `NaN`.

`GSL_ENOPROG`
: the iteration is not making any progress, preventing the algorithm from continuing.

The solver maintains a current best estimate of the root at all times. This information can be accessed with the following auxiliary functions,

Function: gsl_vector * **gsl_multiroot_fsolver_root** *(const gsl_multiroot_fsolver * s)*
Function: gsl_vector * **gsl_multiroot_fdfsolver_root** *(const gsl_multiroot_fdfsolver * s)*

These functions return the current estimate of the root for the solver *s*.

Function: gsl_vector * **gsl_multiroot_fsolver_f** *(const gsl_multiroot_fsolver * s)*
Function: gsl_vector * **gsl_multiroot_fdfsolver_f** *(const gsl_multiroot_fdfsolver * s)*

These functions return the function value f(x) at the current estimate of the root for the solver *s*.

Function: gsl_vector * **gsl_multiroot_fsolver_dx** *(const gsl_multiroot_fsolver * s)*
Function: gsl_vector * **gsl_multiroot_fdfsolver_dx** *(const gsl_multiroot_fdfsolver * s)*

These functions return the last step dx taken by the solver *s*.

Search Stopping Parameters

A root finding procedure should stop when one of the following conditions is true:

- A multidimensional root has been found to within the user-specified precision.
- A user-specified maximum number of iterations has been reached.
- An error has occurred.

The handling of these conditions is under user control. The functions below allow the user to test the precision of the current result in several standard ways.

Function: int **gsl_multiroot_test_delta** *(const gsl_vector * dx, const gsl_vector * x, double epsabs, double epsrel)*

This function tests for the convergence of the sequence by comparing the last step *dx* with the absolute error *epsabs* and relative error *epsrel* to the current position *x*. The test returns `GSL_SUCCESS` if the following condition is achieved,

```
|dx_i| < epsabs + epsrel |x_i|
```

for each component of *x* and returns `GSL_CONTINUE` otherwise.

Function: int **gsl_multiroot_test_residual** *(const gsl_vector * f, double epsabs)*

This function tests the residual value *f* against the absolute error bound *epsabs*. The test returns `GSL_SUCCESS` if the following condition is achieved,

```
\sum_i |f_i| < epsabs
```

and returns `GSL_CONTINUE` otherwise. This criterion is suitable for situations where the precise location of the root, x, is unimportant provided a value can be found where the residual is small enough.

Algorithms using Derivatives

The root finding algorithms described in this section make use of both the function and its derivative. They require an initial guess for the location of the root, but there is no absolute guarantee of convergence -- the function must be suitable for this technique and the initial guess must be sufficiently close to the root for it to work. When the conditions are satisfied then convergence is quadratic.

Derivative Solver: **gsl_multiroot_fdfsolver_hybridsj**

This is a modified version of Powell's Hybrid method as implemented in the HYBRJ algorithm in MINPACK. Minpack was written by Jorge J. Moré, Burton S. Garbow and Kenneth E. Hillstrom. The Hybrid algorithm retains the fast convergence of Newton's method but will also reduce the residual when Newton's method is unreliable.

The algorithm uses a generalized trust region to keep each step under control. In order to be accepted a proposed new position x' must satisfy the condition |D (x' - x)| < \delta, where D is a diagonal scaling matrix and \delta is the size of the trust region. The components of D are computed internally, using the column norms of the Jacobian to estimate the sensitivity of the residual to each component of x. This improves the behavior of the algorithm for badly scaled functions.

On each iteration the algorithm first determines the standard Newton step by solving the system J dx = - f. If this step falls inside the trust region it is used as a trial step in the next stage. If not, the algorithm uses the linear combination of the Newton and gradient directions which is predicted to minimize the norm of the function while staying inside the trust region.

```
dx = - \alpha J^{-1} f(x) - \beta \nabla |f(x)|^2
```

This combination of Newton and gradient directions is referred to as a **dogleg step**.

The proposed step is now tested by evaluating the function at the resulting point, x'. If the step reduces the norm of the function sufficiently then it is accepted and size of the trust region is increased. If the proposed step fails to improve the solution then the size of the trust region is decreased and another trial step is computed.

The speed of the algorithm is increased by computing the changes to the Jacobian approximately, using a rank-1 update. If two successive attempts fail to reduce the residual then the full Jacobian is recomputed. The algorithm also monitors the progress of the solution and returns an error if several steps fail to make any improvement,

`GSL_ENOPROG`
: the iteration is not making any progress, preventing the algorithm from continuing.

`GSL_ENOPROGJ`
: re-evaluations of the Jacobian indicate that the iteration is not making any progress, preventing the algorithm from continuing.

Derivative Solver: **gsl_multiroot_fdfsolver_hybridj**

This algorithm is an unscaled version of `hybridsj`. The steps are controlled by a spherical trust region |x' - x| < \delta, instead of a generalized region. This can be useful if the generalized region estimated by `hybridsj` is inappropriate.

Derivative Solver: **gsl_multiroot_fdfsolver_newton**

Newton's Method is the standard root-polishing algorithm. The algorithm begins with an initial guess for the location of the solution. On each iteration a linear approximation to the function F is used to estimate the step which will zero all the components of the residual. The iteration is defined by the following sequence,

```
x -> x' = x - J^{-1} f(x)
```

where the Jacobian matrix J is computed from the derivative functions provided by *f*. The step dx is obtained by solving the linear system,

```
J dx = - f(x)
```

using LU decomposition.

Derivative Solver: **gsl_multiroot_fdfsolver_gnewton**

This is a modified version of Newton's method which attempts to improve global convergence by requiring every step to reduce the Euclidean norm of the residual, |f(x)|. If the Newton step leads to an increase in the norm then a reduced step of relative size,

```
t = (\sqrt(1 + 6 r) - 1) / (3 r)
```

is proposed, with r being the ratio of norms |f(x')|^2/|f(x)|^2. This procedure is repeated until a suitable step size is found.

Algorithms without Derivatives

The algorithms described in this section do not require any derivative information to be supplied by the user. Any derivatives needed are approximated from by finite difference.

Solver: **gsl_multiroot_fsolver_hybrids**

This is a version of the Hybrid algorithm which replaces calls to the Jacobian function by its finite difference approximation. The finite difference approximation is computed using `gsl_multiroots_fdjac` with a relative step size of `GSL_SQRT_DBL_EPSILON`.

Solver: **gsl_multiroot_fsolver_hybrid**

This is a finite difference version of the Hybrid algorithm without internal scaling.

Solver: **gsl_multiroot_fsolver_dnewton**

The **discrete Newton algorithm** is the simplest method of solving a multidimensional system. It uses the Newton iteration

```
x -> x - J^{-1} f(x)
```

where the Jacobian matrix J is approximated by taking finite differences of the function *f*. The approximation scheme used by this implementation is,

```
J_{ij} = (f_i(x + \delta_j) - f_i(x)) /  \delta_j
```

where \delta_j is a step of size \sqrt\epsilon |x_j| with \epsilon being the machine precision (\epsilon \approx 2.22 \times 10^-16). The order of convergence of Newton's algorithm is quadratic, but the finite differences require n^2 function evaluations on each iteration. The algorithm may become unstable if the finite differences are not a good approximation to the true derivatives.

Solver: **gsl_multiroot_fsolver_broyden**

The **Broyden algorithm** is a version of the discrete Newton algorithm which attempts to avoids the expensive update of the Jacobian matrix on each iteration. The changes to the Jacobian are also approximated, using a rank-1 update,

```
J^{-1} \to J^{-1} - (J^{-1} df - dx) dx^T J^{-1} / dx^T J^{-1} df
```

where the vectors dx and df are the changes in x and f. On the first iteration the inverse Jacobian is estimated using finite differences, as in the discrete Newton algorithm. This approximation gives a fast update but is unreliable if the changes are not small, and the estimate of the inverse Jacobian becomes worse as time passes. The algorithm has a tendency to become unstable unless it starts close to the root. The Jacobian is refreshed if this instability is detected (consult the source for details).

This algorithm is not recommended and is included only for demonstration purposes.

Examples

The multidimensional solvers are used in a similar way to the one-dimensional root finding algorithms. This first example demonstrates the `hybrids` scaled-hybrid algorithm, which does

not require derivatives. The program solves the Rosenbrock system of equations,

```
f_1 (x, y) = a (1 - x)
f_2 (x, y) = b (y - x^2)
```

with a = 1, b = 10. The solution of this system lies at (x,y) = (1,1) in a narrow valley.

The first stage of the program is to define the system of equations,

```
#include <stdlib.h>
#include <stdio.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_multiroots.h>

struct rparams
  {
    double a;
    double b;
  };

int
rosenbrock_f (const gsl_vector * x, void *params,
              gsl_vector * f)
{
  double a = ((struct rparams *) params)->a;
  double b = ((struct rparams *) params)->b;

  const double x0 = gsl_vector_get (x, 0);
  const double x1 = gsl_vector_get (x, 1);

  const double y0 = a * (1 - x0);
  const double y1 = b * (x1 - x0 * x0);

  gsl_vector_set (f, 0, y0);
  gsl_vector_set (f, 1, y1);

  return GSL_SUCCESS;
}
```

The main program begins by creating the function object `f`, with the arguments `(x,y)` and parameters `(a,b)`. The solver `s` is initialized to use this function, with the `hybrids` method.

```
int
main (void)
{
  const gsl_multiroot_fsolver_type *T;
  gsl_multiroot_fsolver *s;

  int status;
  size_t i, iter = 0;

  const size_t n = 2;
  struct rparams p = {1.0, 10.0};
  gsl_multiroot_function f = {&rosenbrock_f, n, &p};

  double x_init[2] = {-10.0, -5.0};
```

```
  gsl_vector *x = gsl_vector_alloc (n);

  gsl_vector_set (x, 0, x_init[0]);
  gsl_vector_set (x, 1, x_init[1]);

  T = gsl_multiroot_fsolver_hybrids;
  s = gsl_multiroot_fsolver_alloc (T, 2);
  gsl_multiroot_fsolver_set (s, &f, x);

  print_state (iter, s);

  do
    {
      iter++;
      status = gsl_multiroot_fsolver_iterate (s);

      print_state (iter, s);

      if (status)   /* check if solver is stuck */
        break;

      status =
        gsl_multiroot_test_residual (s->f, 1e-7);
    }
  while (status == GSL_CONTINUE && iter < 1000);

  printf ("status = %s\n", gsl_strerror (status));

  gsl_multiroot_fsolver_free (s);
  gsl_vector_free (x);
  return 0;
}
```

Note that it is important to check the return status of each solver step, in case the algorithm becomes stuck. If an error condition is detected, indicating that the algorithm cannot proceed, then the error can be reported to the user, a new starting point chosen or a different algorithm used.

The intermediate state of the solution is displayed by the following function. The solver state contains the vector `s->x` which is the current position, and the vector `s->f` with corresponding function values.

```
int
print_state (size_t iter, gsl_multiroot_fsolver * s)
{
  printf ("iter = %3u x = % .3f % .3f "
          "f(x) = % .3e % .3e\n",
          iter,
          gsl_vector_get (s->x, 0),
          gsl_vector_get (s->x, 1),
          gsl_vector_get (s->f, 0),
          gsl_vector_get (s->f, 1));
}
```

Here are the results of running the program. The algorithm is started at (-10,-5) far from the solution. Since the solution is hidden in a narrow valley the earliest steps follow the gradient of the function downhill, in an attempt to reduce the large value

of the residual. Once the root has been approximately located, on iteration 8, the Newton behavior takes over and convergence is very rapid.

```
iter =  0 x = -10.000  -5.000  f(x) = 1.100e+01 -1.050e+03
iter =  1 x = -10.000  -5.000  f(x) = 1.100e+01 -1.050e+03
iter =  2 x =  -3.976  24.827  f(x) = 4.976e+00  9.020e+01
iter =  3 x =  -3.976  24.827  f(x) = 4.976e+00  9.020e+01
iter =  4 x =  -3.976  24.827  f(x) = 4.976e+00  9.020e+01
iter =  5 x =  -1.274  -5.680  f(x) = 2.274e+00 -7.302e+01
iter =  6 x =  -1.274  -5.680  f(x) = 2.274e+00 -7.302e+01
iter =  7 x =   0.249   0.298  f(x) = 7.511e-01  2.359e+00
iter =  8 x =   0.249   0.298  f(x) = 7.511e-01  2.359e+00
iter =  9 x =   1.000   0.878  f(x) = 1.268e-10 -1.218e+00
iter = 10 x =   1.000   0.989  f(x) = 1.124e-11 -1.080e-01
iter = 11 x =   1.000   1.000  f(x) = 0.000e+00  0.000e+00
status = success
```

Note that the algorithm does not update the location on every iteration. Some iterations are used to adjust the trust-region parameter, after trying a step which was found to be divergent, or to recompute the Jacobian, when poor convergence behavior is detected.

The next example program adds derivative information, in order to accelerate the solution. There are two derivative functions `rosenbrock_df` and `rosenbrock_fdf`. The latter computes both the function and its derivative simultaneously. This allows the optimization of any common terms. For simplicity we substitute calls to the separate `f` and `df` functions at this point in the code below.

```
int
rosenbrock_df (const gsl_vector * x, void *params,
               gsl_matrix * J)
{
  const double a = ((struct rparams *) params)->a;
  const double b = ((struct rparams *) params)->b;

  const double x0 = gsl_vector_get (x, 0);

  const double df00 = -a;
  const double df01 = 0;
  const double df10 = -2 * b  * x0;
  const double df11 = b;

  gsl_matrix_set (J, 0, 0, df00);
  gsl_matrix_set (J, 0, 1, df01);
  gsl_matrix_set (J, 1, 0, df10);
  gsl_matrix_set (J, 1, 1, df11);

  return GSL_SUCCESS;
}

int
rosenbrock_fdf (const gsl_vector * x, void *params,
                gsl_vector * f, gsl_matrix * J)
{
  rosenbrock_f (x, params, f);
```

```
  rosenbrock_df (x, params, J);

  return GSL_SUCCESS;
}
```

The main program now makes calls to the corresponding `fdfsolver` versions of the functions,

```
int
main (void)
{
  const gsl_multiroot_fdfsolver_type *T;
  gsl_multiroot_fdfsolver *s;

  int status;
  size_t i, iter = 0;

  const size_t n = 2;
  struct rparams p = {1.0, 10.0};
  gsl_multiroot_function_fdf f = {&rosenbrock_f,
                                  &rosenbrock_df,
                                  &rosenbrock_fdf,
                                  n, &p};

  double x_init[2] = {-10.0, -5.0};
  gsl_vector *x = gsl_vector_alloc (n);

  gsl_vector_set (x, 0, x_init[0]);
  gsl_vector_set (x, 1, x_init[1]);

  T = gsl_multiroot_fdfsolver_gnewton;
  s = gsl_multiroot_fdfsolver_alloc (T, n);
  gsl_multiroot_fdfsolver_set (s, &f, x);

  print_state (iter, s);

  do
    {
      iter++;

      status = gsl_multiroot_fdfsolver_iterate (s);

      print_state (iter, s);

      if (status)
        break;

      status = gsl_multiroot_test_residual (s->f, 1e-7);
    }
  while (status == GSL_CONTINUE && iter < 1000);

  printf ("status = %s\n", gsl_strerror (status));

  gsl_multiroot_fdfsolver_free (s);
  gsl_vector_free (x);
  return 0;
}
```

The addition of derivative information to the `hybrids` solver does not make any significant difference to its behavior, since it able to approximate the Jacobian numerically with sufficient

accuracy. To illustrate the behavior of a different derivative solver we switch to `gnewton`. This is a traditional Newton solver with the constraint that it scales back its step if the full step would lead "uphill". Here is the output for the `gnewton` algorithm,

```
iter = 0 x = -10.000  -5.000 f(x) =  1.100e+01 -1.050e+03
iter = 1 x =  -4.231 -65.317 f(x) =  5.231e+00 -8.321e+02
iter = 2 x =   1.000 -26.358 f(x) = -8.882e-16 -2.736e+02
iter = 3 x =   1.000   1.000 f(x) = -2.220e-16 -4.441e-15
status = success
```

The convergence is much more rapid, but takes a wide excursion out to the point (-4.23,-65.3). This could cause the algorithm to go astray in a realistic application. The hybrid algorithm follows the downhill path to the solution more reliably.

References and Further Reading

The original version of the Hybrid method is described in the following articles by Powell,

- M.J.D. Powell, "A Hybrid Method for Nonlinear Equations" (Chap 6, p 87-114) and "A Fortran Subroutine for Solving systems of Nonlinear Algebraic Equations" (Chap 7, p 115-161), in *Numerical Methods for Nonlinear Algebraic Equations*, P. Rabinowitz, editor. Gordon and Breach, 1970.

The following papers are also relevant to the algorithms described in this section,

- J.J. Moré, M.Y. Cosnard, "Numerical Solution of Nonlinear Equations", *ACM Transactions on Mathematical Software*, Vol 5, No 1, (1979), p 64-85
- C.G. Broyden, "A Class of Methods for Solving Nonlinear Simultaneous Equations", *Mathematics of Computation*, Vol 19 (1965), p 577-593
- J.J. Moré, B.S. Garbow, K.E. Hillstrom, "Testing Unconstrained Optimization Software", ACM Transactions on Mathematical Software, Vol 7, No 1 (1981), p 17-41

Go to the first, previous, next, last section, table of contents.