NAG Library Function Document

nag_zero_nonlin_eqns_deriv_expert (c05rcc)

1 Purpose

nag_zero_nonlin_eqns_deriv_expert (c05rcc) is a comprehensive function that finds a solution of a system of nonlinear equations by a modification of the Powell hybrid method. You must provide the Jacobian.

2 Specification

```
#include <nag.h>
#include <nagc05.h>

void nag_zero_nonlin_eqns_deriv_expert (
     void  (*fcn)(Integer n, const double x[], double fvec[], double fjac[],
           Nag_Comm *comm, Integer *iflag),
     Integer n, double x[], double fvec[], double fjac[], double xtol,
     Integer maxfev, Nag_ScaleType scale_mode, double diag[], double factor,
     Integer nprint, Integer *nfev, Integer *njev, double r[], double qtf[],
     Nag_Comm *comm, NagError *fail)
```

3 Description

The system of equations is defined as:

$$f_i(x_1, x_2, \ldots, x_n) = 0, \qquad i = 1, 2, \ldots, n.$$

nag_zero_nonlin_eqns_deriv_expert (c05rcc) is based on the MINPACK routine HYBRJ (see Moré *et al.* (1980)). It chooses the correction at each step as a convex combination of the Newton and scaled gradient directions. The Jacobian is updated by the rank-1 method of Broyden. At the starting point, the Jacobian is requested, but it is not asked for again until the rank-1 method fails to produce satisfactory progress. For more details see Powell (1970).

4 References

Moré J J, Garbow B S and Hillstrom K E (1980) User guide for MINPACK-1 *Technical Report ANL-80-74* Argonne National Laboratory

Powell M J D (1970) A hybrid method for nonlinear algebraic equations *Numerical Methods for Nonlinear Algebraic Equations* (ed P Rabinowitz) Gordon and Breach

5 Arguments

1: **fcn** – function, supplied by the user *External Function*

Depending upon the value of **iflag**, **fcn** must either return the values of the functions f_i at a point x or return the Jacobian at x.

The specification of **fcn** is:

```
void  fcn (Integer n, const double x[], double fvec[], double fjac[],
      Nag_Comm *comm, Integer *iflag)
```

1: **n** – Integer *Input*

On entry: n, the number of equations.

2: **x**[**n**] – const double *Input*

On entry: the components of the point x at which the functions or the Jacobian must be evaluated.

3: **fvec**[**n**] – double *Input/Output*

On entry: if **iflag** $= 0$ or 2, **fvec** contains the function values $f_i(x)$ and must not be changed.

On exit: if **iflag** $= 1$ on entry, **fvec** must contain the function values $f_i(x)$ (unless **iflag** is set to a negative value by **fcn**).

4: **fjac**[**n** $\times$ **n**] – double *Input/Output*

Note: the (i, j)th element of the matrix is stored in **fjac**$[(j-1) \times \mathbf{n} + i - 1]$.

On entry: if **iflag** $= 0$, **fjac**$[(j-1) \times \mathbf{n} + i - 1]$ contains the value of $\dfrac{\partial f_i}{\partial x_j}$ at the point x, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$. When **iflag** $= 0$ or 1, **fjac** must not be changed.

On exit: if **iflag** $= 2$ on entry, **fjac**$[(j-1) \times \mathbf{n} + i - 1]$ must contain the value of $\dfrac{\partial f_i}{\partial x_j}$ at the point x, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$, (unless **iflag** is set to a negative value by **fcn**).

5: **comm** – Nag_Comm * *Communication Structure*

Pointer to structure of type Nag_Comm; the following members are relevant to **fcn**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_zero_nonlin_eqns_deriv_expert (c05rcc) you may allocate memory and initialize these pointers with various quantities for use by **fcn** when called from nag_zero_nonlin_eqns_deriv_expert (c05rcc) (see Section 3.2.1.1 in the Essential Introduction).

6: **iflag** – Integer * *Input/Output*

On entry: **iflag** $= 0$, 1 or 2.

iflag $= 0$
: **x**, **fvec** and **fjac** are available for printing (see **nprint**).

iflag $= 1$
: **fvec** is to be updated.

iflag $= 2$
: **fjac** is to be updated.

On exit: in general, **iflag** should not be reset by **fcn**. If, however, you wish to terminate execution (perhaps because some illegal point **x** has been reached), then **iflag** should be set to a negative integer value.

2: **n** – Integer *Input*

On entry: n, the number of equations.

Constraint: **n** > 0.

3: **x**[**n**] – double *Input/Output*

On entry: an initial guess at the solution vector.

On exit: the final estimate of the solution vector.

4: **fvec**[**n**] – double *Output*

On exit: the function values at the final point returned in **x**.

5: **fjac**[**n** × **n**] – double *Output*

Note: the (i, j)th element of the matrix is stored in **fjac**$[(j-1) \times$ **n** $+ i - 1]$.

On exit: the orthogonal matrix Q produced by the QR factorization of the final approximate Jacobian, stored by columns.

6: **xtol** – double *Input*

On entry: the accuracy in **x** to which the solution is required.

Suggested value: $\sqrt{\epsilon}$, where ϵ is the ***machine precision*** returned by nag_machine_precision (X02AJC).

Constraint: **xtol** ≥ 0.0.

7: **maxfev** – Integer *Input*

On entry: the maximum number of calls to **fcn** with **iflag** $\neq 0$. nag_zero_nonlin_eqns_deriv_expert (c05rcc) will exit with **fail.code** = NE_TOO_MANY_FEVALS, if, at the end of an iteration, the number of calls to **fcn** exceeds **maxfev**.

Suggested value: **maxfev** $= 100 \times ($**n** $+ 1)$.

Constraint: **maxfev** > 0.

8: **scale_mode** – Nag_ScaleType *Input*

On entry: indicates whether or not you have provided scaling factors in **diag**.

If **scale_mode** = Nag_ScaleProvided the scaling must have been specified in **diag**.

Otherwise, if **scale_mode** = Nag_NoScaleProvided, the variables will be scaled internally.

Constraint: **scale_mode** = Nag_NoScaleProvided or Nag_ScaleProvided.

9: **diag**[**n**] – double *Input/Output*

On entry: if **scale_mode** = Nag_ScaleProvided, **diag** must contain multiplicative scale factors for the variables.

If **scale_mode** = Nag_NoScaleProvided, **diag** need not be set.

Constraint: if **scale_mode** = Nag_ScaleProvided, **diag**$[i-1] > 0.0$, for $i = 1, 2, \ldots, n$.

On exit: the scale factors actually used (computed internally if **scale_mode** = Nag_NoScaleProvided).

10: **factor** – double *Input*

On entry: a quantity to be used in determining the initial step bound. In most cases, **factor** should lie between 0.1 and 100.0. (The step bound is **factor** $\times$ $\|$**diag** $\times$ **x**$\|_2$ if this is nonzero; otherwise the bound is **factor**.)

Suggested value: **factor** $= 100.0$.

Constraint: **factor** > 0.0.

11: **nprint** – Integer *Input*

On entry: indicates whether (and how often) special calls to **fcn**, with **iflag** set to 0, are to be made for printing purposes.

nprint ≤ 0
: No calls are made.

nprint > 0
: **fcn** is called at the beginning of the first iteration, every **nprint** iterations thereafter and immediately before the return from nag_zero_nonlin_eqns_deriv_expert (c05rcc).

12: **nfev** – Integer * *Output*

On exit: the number of calls made to **fcn** to evaluate the functions.

13: **njev** – Integer * *Output*

On exit: the number of calls made to **fcn** to evaluate the Jacobian.

14: $\mathbf{r}[\mathbf{n} \times (\mathbf{n}+\mathbf{1})/\mathbf{2}]$ – double *Output*

On exit: the upper triangular matrix R produced by the QR factorization of the final approximate Jacobian, stored row-wise.

15: **qtf**[**n**] – double *Output*

On exit: the vector $Q^{\mathrm{T}} f$.

16: **comm** – Nag_Comm * *Communication Structure*

The NAG communication argument (see Section 3.2.1.1 in the Essential Introduction).

17: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

NE_DIAG_ELEMENTS

On entry, **scale_mode** = Nag_ScaleProvided and **diag** contained a non-positive element.

NE_INT

On entry, **maxfev** = ⟨*value*⟩.
Constraint: **maxfev** > 0.

On entry, **n** = ⟨*value*⟩.
Constraint: **n** > 0.

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

NE_NO_IMPROVEMENT

The iteration is not making good progress, as measured by the improvement from the last ⟨*value*⟩ iterations. This failure exit may indicate that the system does not have a zero, or that the solution is very close to the origin (see Section 7). Otherwise, rerunning nag_zero_nonlin_eqns_deriv_expert (c05rcc) from a different starting point may avoid the region of difficulty.

The iteration is not making good progress, as measured by the improvement from the last ⟨*value*⟩ Jacobian evaluations. This failure exit may indicate that the system does not have a zero, or that the solution is very close to the origin (see Section 7). Otherwise, rerunning nag_zero_nonlin_eqns_deriv_expert (c05rcc) from a different starting point may avoid the region of difficulty.

NE_REAL

On entry, **factor** $=$ ⟨*value*⟩.
Constraint: **factor** > 0.0.

On entry, **xtol** $=$ ⟨*value*⟩.
Constraint: **xtol** ≥ 0.0.

NE_TOO_MANY_FEVALS

There have been at least **maxfev** calls to **fcn**: **maxfev** $=$ ⟨*value*⟩. Consider restarting the calculation from the final point held in **x**.

NE_TOO_SMALL

No further improvement in the solution is possible. **xtol** is too small: **xtol** $=$ ⟨*value*⟩.

NE_USER_STOP

iflag was set negative in **fcn**. **iflag** $=$ ⟨*value*⟩.

7 Accuracy

If $\hat{x}$ is the true solution and D denotes the diagonal matrix whose entries are defined by the array **diag**, then nag_zero_nonlin_eqns_deriv_expert (c05rcc) tries to ensure that

$$\|D(x-\hat{x})\|_2 \leq \mathbf{xtol} \times \|D\hat{x}\|_2.$$

If this condition is satisfied with **xtol** $= 10^{-k}$, then the larger components of Dx have k significant decimal digits. There is a danger that the smaller components of Dx may have large relative errors, but the fast rate of convergence of nag_zero_nonlin_eqns_deriv_expert (c05rcc) usually obviates this possibility.

If **xtol** is less than ***machine precision*** and the above test is satisfied with the ***machine precision*** in place of **xtol**, then the function exits with **fail**.**code** $=$ NE_TOO_SMALL.

Note: this convergence test is based purely on relative error, and may not indicate convergence if the solution is very close to the origin.

The convergence test assumes that the functions and the Jacobian are coded consistently and that the functions are reasonably well behaved. If these conditions are not satisfied, then nag_zero_nonlin_eqns_deriv_expert (c05rcc) may incorrectly indicate convergence. The coding of the Jacobian can be checked using nag_check_derivs (c05zdc). If the Jacobian is coded correctly, then the validity of the answer can be checked by rerunning nag_zero_nonlin_eqns_deriv_expert (c05rcc) with a lower value for **xtol**.

8 Parallelism and Performance

nag_zero_nonlin_eqns_deriv_expert (c05rcc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_zero_nonlin_eqns_deriv_expert (c05rcc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

Local workspace arrays of fixed lengths are allocated internally by nag_zero_nonlin_eqns_deriv_expert (c05rcc). The total size of these arrays amounts to $4 \times n$ double elements.

The time required by nag_zero_nonlin_eqns_deriv_expert (c05rcc) to solve a given problem depends on n, the behaviour of the functions, the accuracy requested and the starting point. The number of arithmetic operations executed by nag_zero_nonlin_eqns_deriv_expert (c05rcc) is approximately $11.5 \times n^2$ to process each evaluation of the functions and approximately $1.3 \times n^3$ to process each evaluation of the Jacobian. The timing of nag_zero_nonlin_eqns_deriv_expert (c05rcc) is strongly influenced by the time spent evaluating the functions.

Ideally the problem should be scaled so that, at the solution, the function values are of comparable magnitude.

10 Example

This example determines the values $x_1, \ldots, x_9$ which satisfy the tridiagonal equations:

$$\begin{array}{rcll} (3-2x_1)x_1 - 2x_2 & = & -1, & \\ -x_{i-1} + (3-2x_i)x_i - 2x_{i+1} & = & -1, & i = 2,3,\ldots,8 \\ -x_8 + (3-2x_9)x_9 & = & -1. & \end{array}$$

10.1 Program Text

```
/* nag_zero_nonlin_eqns_deriv_expert (c05rcc) Example Program.
 *
 * Copyright 2013 Numerical Algorithms Group.
 *
 * Mark 24, 2013.
 */

#include <nag.h>
#include <nagx04.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <math.h>
#include <nagc05.h>
#include <nagx02.h>

#ifdef __cplusplus
extern "C" {
#endif
static void NAG_CALL fcn(Integer n, const double x[], double fvec[],
                         double fjac[], Nag_Comm *comm, Integer *iflag);
#ifdef __cplusplus
}
#endif

static Integer nprint = 0;

int main(void)
{
  static double ruser[1] = {-1.0};
  Integer  exit_status = 0, i, n = 9, maxfev, nfev, njev;
  double   *diag = 0, *fjac = 0, *fvec = 0, *qtf = 0, *r = 0, *x = 0;
  double   factor, xtol;
  /* Nag Types */
  NagError fail;
  Nag_Comm comm;
  Nag_ScaleType scale_mode;

  INIT_FAIL(fail);

  printf("nag_zero_nonlin_eqns_deriv_expert (c05rcc) "
         "Example Program Results\n");

  /* For communication with user-supplied functions: */
  comm.user = ruser;

  if (n > 0)
    {
      if (!(diag = NAG_ALLOC(n, double)) ||
          !(fjac = NAG_ALLOC(n*n, double)) ||
          !(fvec = NAG_ALLOC(n, double)) ||
          !(qtf = NAG_ALLOC(n, double)) ||
          !(r = NAG_ALLOC(n*(n+1)/2, double)) ||
          !(x = NAG_ALLOC(n, double)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
    }
  else
    {
      printf("Invalid n.\n");
      exit_status = 1;
      goto END;
    }

  /* The following starting values provide a rough solution. */
  for (i = 0; i < n; i++)
    x[i] = -1.0;

  /* nag_machine_precision (x02ajc).
   * The machine precision
   */
  xtol = sqrt(nag_machine_precision);

  for (i = 0; i < n; i++)
    diag[i] = 1.0;

  maxfev = 2000;
  scale_mode = Nag_ScaleProvided;
  factor = 100.0;

  /* nag_zero_nonlin_eqns_deriv_expert (c05rcc).
   * Solution of a system of nonlinear equations (function
   * values only)
   */
  nag_zero_nonlin_eqns_deriv_expert(fcn, n, x, fvec, fjac, xtol, maxfev,
                                    scale_mode, diag, factor, nprint, &nfev,
                                    &njev, r, qtf, &comm, &fail);

  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_zero_nonlin_eqns_deriv_expert (c05rcc).\n%s\n",
             fail.message);
      exit_status = 1;
      if (fail.code != NE_TOO_MANY_FEVALS &&
          fail.code != NE_TOO_SMALL &&
          fail.code != NE_NO_IMPROVEMENT)
        goto END;
    }

  printf(fail.code == NE_NOERROR ? "Final approximate" : "Approximate");
  printf(" solution\n\n");
  for (i = 0; i < n;  i++)
    printf("%12.4f%s", x[i], (i%3 == 2 || i == n-1)?"\n":" ");
  if (fail.code != NE_NOERROR)
    exit_status = 2;

 END:
  NAG_FREE(diag);
  NAG_FREE(fjac);
  NAG_FREE(fvec);
  NAG_FREE(qtf);
  NAG_FREE(r);
  NAG_FREE(x);
  return exit_status;
}
static void NAG_CALL fcn(Integer n, const double x[], double fvec[],
                         double fjac[], Nag_Comm *comm, Integer *iflag)
{
  Integer j, k;

  if (comm->user[0] == -1.0)
    {
      printf("(User-supplied callback fcn, first invocation.)\n");
      comm->user[0] = 0.0;
    }
  if (*iflag==0)
    {
      if (nprint>0)
        {
          /* Insert print statements here if desired. */
        }
    }
  else if (*iflag != 2)
    {
      for (k = 0; k < n; k++)
        {
          fvec[k] = (3.0-x[k]*2.0) * x[k] + 1.0;
          if (k > 0) fvec[k] -= x[k-1];
          if (k < n-1) fvec[k] -= x[k+1] * 2.0;
        }
    }
  else
    {
      for (k = 0; k < n; k++)
        {
          for (j = 0; j < n; j++)
            fjac[j*n + k] = 0.0;
          fjac[k*n + k] = 3.0 - x[k] * 4.0;
          if (k > 0)
            fjac[(k-1)*n + k] = -1.0;
          if (k < n-1)
            fjac[(k+1)*n + k] = -2.0;
        }
    }
  /* Set iflag negative to terminate execution for any reason. */
  *iflag = 0;
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_zero_nonlin_eqns_deriv_expert (c05rcc) Example Program Results
(User-supplied callback fcn, first invocation.)
Final approximate solution

     -0.5707      -0.6816      -0.7017
     -0.7042      -0.7014      -0.6919
```

```
    -0.6658      -0.5960      -0.4164
```