NAG Library Function Document

nag_zero_sparse_nonlin_eqns_easy (c05qsc)

1 Purpose

nag_zero_sparse_nonlin_eqns_easy (c05qsc) is an easy-to-use function that finds a solution of a sparse system of nonlinear equations by a modification of the Powell hybrid method.

2 Specification

```
#include <nag.h>
#include <nagc05.h>
void nag_zero_sparse_nonlin_eqns_easy (
     void (*fcn)(Integer n, Integer lindf, const Integer indf[],
          const double x[], double fvec[], Nag_Comm *comm, Integer *iflag),
     Integer n, double x[], double fvec[], double xtol, Nag_Boolean init,
     double rcomm[], Integer lrcomm, Integer icomm[], Integer licomm,
     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_sparse_nonlin_eqns_easy (c05qsc) is based on the MINPACK routine HYBRD1 (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 sparse rank-1 method of Schubert (see Schubert (1970)). At the starting point, the sparsity pattern is determined and the Jacobian is approximated by forward differences, but these are not used again until the rank-1 method fails to produce satisfactory progress. Then, the sparsity structure is used to recompute an approximation to the Jacobian by forward differences with the least number of function evaluations. The function you supply must be able to compute only the requested subset of the function values. The sparse Jacobian linear system is solved at each iteration with nag_superlu_lu_factorize (f11mec) computing the Newton step. For more details see Powell (1970) and Broyden (1965).

4 References

Broyden C G (1965) A class of methods for solving nonlinear simultaneous equations *Mathematics of Computation* **19(92)** 577–593

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

Schubert L K (1970) Modification of a quasi-Newton method for nonlinear equations with a sparse Jacobian *Mathematics of Computation* **24(109)** 27–30

5 Arguments

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

fcn must return the values of the functions f_i at a point x.

The specification of **fcn** is:

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

1: **n** – Integer *Input*

On entry: n, the number of equations.

2: **lindf** – Integer *Input*

On entry: **lindf** specifies the number of indices i for which values of $f_i(x)$ must be computed.

3: **indf**[**lindf**] – const Integer *Input*

On entry: **indf** specifies the indices i for which values of $f_i(x)$ must be computed. The indices are specified in strictly ascending order.

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

On entry: the components of the point x at which the functions must be evaluated. **x**$[i-1]$ contains the coordinate x_i.

5: **fvec**[**n**] – double *Output*

On exit: **fvec**$[i-1]$ must contain the function values $f_i(x)$, for all indices i in **indf**.

6: **comm** – Nag_Comm *

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_sparse_nonlin_eqns_easy (c05qsc) you may allocate memory and initialize these pointers with various quantities for use by **fcn** when called from nag_zero_sparse_nonlin_eqns_easy (c05qsc) (see Section 3.2.1.1 in the Essential Introduction).

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

On entry: **iflag** > 0.

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.

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. **x**$[i-1]$ must contain the coordinate x_i.

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**. **fvec**$[i-1]$ contains the function values f_i.

5: **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.

6: **init** – Nag_Boolean *Input*

On entry: **init** must be set to Nag_TRUE to indicate that this is the first time nag_zero_sparse_nonlin_eqns_easy (c05qsc) is called for this specific problem. nag_zero_sparse_nonlin_eqns_easy (c05qsc) then computes the dense Jacobian and detects and stores its sparsity pattern (in **rcomm** and **icomm**) before proceeding with the iterations. This is noticeably time consuming when **n** is large. If not enough storage has been provided for **rcomm** or **icomm**, nag_zero_sparse_nonlin_eqns_easy (c05qsc) will fail. On exit with **fail**.**code** = NE_NOERROR, NE_NO_IMPROVEMENT, NE_TOO_MANY_FEVALS or NE_TOO_SMALL, **icomm**$[0]$ contains nnz, the number of nonzero entries found in the Jacobian. On subsequent calls, **init** can be set to Nag_FALSE if the problem has a Jacobian of the same sparsity pattern. In that case, the computation time required for the detection of the sparsity pattern will be smaller.

7: **rcomm**[**lrcomm**] – double *Communication Array*

rcomm MUST NOT be altered between successive calls to nag_zero_sparse_nonlin_eqns_easy (c05qsc).

8: **lrcomm** – Integer *Input*

On entry: the dimension of the array **rcomm**.

Constraint: **lrcomm** $\geq 12 + nnz$ where nnz is the number of nonzero entries in the Jacobian, as computed by nag_zero_sparse_nonlin_eqns_easy (c05qsc).

9: **icomm**[**licomm**] – Integer *Communication Array*

If **fail**.**code** = NE_NOERROR, NE_NO_IMPROVEMENT, NE_TOO_MANY_FEVALS or NE_TOO_SMALL on exit, **icomm**$[0]$ contains nnz where nnz is the number of nonzero entries in the Jacobian.

icomm MUST NOT be altered between successive calls to nag_zero_sparse_nonlin_eqns_easy (c05qsc).

10: **licomm** – Integer *Input*

On entry: the dimension of the array **icomm**.

Constraint: **licomm** $\geq 8 \times \mathbf{n} + 19 + nnz$ where nnz is the number of nonzero entries in the Jacobian, as computed by nag_zero_sparse_nonlin_eqns_easy (c05qsc).

11: **comm** – Nag_Comm *

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

12: **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.
See Section 3.2.1.2 in the Essential Introduction for further information.

NE_BAD_PARAM

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

NE_INT

On entry, **licomm** = ⟨*value*⟩.
Constraint: **licomm** ≥ ⟨*value*⟩.

On entry, **lrcomm** = ⟨*value*⟩.
Constraint: **lrcomm** ≥ ⟨*value*⟩.

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.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in the Essential Introduction for further information.

NE_NO_IMPROVEMENT

The iteration is not making good progress. 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_sparse_nonlin_eqns_easy (c05qsc) from a different starting point may avoid the region of difficulty. The condition number of the Jacobian is ⟨*value*⟩.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

NE_REAL

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

NE_TOO_MANY_FEVALS

There have been at least 200 × (**n** + 1) calls to **fcn**. Consider setting **init** = Nag_FALSE and restarting the calculation from the 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, nag_zero_sparse_nonlin_eqns_easy (c05qsc) tries to ensure that

$$\|x - \hat{x}\|_2 \leq \textbf{xtol} \times \|\hat{x}\|_2.$$

If this condition is satisfied with **xtol** $= 10^{-k}$, then the larger components of x have k significant decimal digits. There is a danger that the smaller components of x may have large relative errors, but the fast rate of convergence of nag_zero_sparse_nonlin_eqns_easy (c05qsc) 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 are reasonably well behaved. If this condition is not satisfied, then nag_zero_sparse_nonlin_eqns_easy (c05qsc) may incorrectly indicate convergence. The validity of the answer can be checked, for example, by rerunning nag_zero_sparse_nonlin_eqns_easy (c05qsc) with a lower value for **xtol**.

8 Parallelism and Performance

nag_zero_sparse_nonlin_eqns_easy (c05qsc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_zero_sparse_nonlin_eqns_easy (c05qsc) 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 X06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also 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_sparse_nonlin_eqns_easy (c05qsc). The total size of these arrays amounts to $8 \times n + 2 \times q$ double elements and $10 \times n + 2 \times q + 5$ integer elements where the integer q is bounded by $8 \times nnz$ and n^2 and depends on the sparsity pattern of the Jacobian.

The time required by nag_zero_sparse_nonlin_eqns_easy (c05qsc) 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_sparse_nonlin_eqns_easy (c05qsc) to process each evaluation of the functions depends on the number of nonzero entries in the Jacobian. The timing of nag_zero_sparse_nonlin_eqns_easy (c05qsc) is strongly influenced by the time spent evaluating the functions.

When **init** is Nag_TRUE, the dense Jacobian is first evaluated and that will take time proportional to n^2.

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:

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

It then perturbs the equations by a small amount and solves the new system.

10.1 Program Text

```
/* nag_zero_sparse_nonlin_eqns_easy (c05qsc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 23, 2011.
 */

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

static void NAG_CALL fcn(Integer  n, Integer lindf, const Integer indf[],
                const double x[], double fvec[], Nag_Comm *comm, Integer *iflag)
{
  double    theta;
  Integer   i, ind;

  *iflag = 0;
  theta = (double) (comm->iuser[0]) * pow(0.5, 7);
  for (ind = 0; ind < lindf; ind++) {
    i = indf[ind] - 1;
    fvec[i] = (3.0 - (2.0 + theta) * x[i]) * x[i] + 1.0;
    if (i > 0) fvec[i] = fvec[i] - x[i - 1];
    if (i < n-1) fvec[i] = fvec[i] - 2.0 * x[i + 1];
  }
}

int main(void)
{
  Integer     exit_status = 0, n = 9, i, j, licomm, lrcomm;
  double      fnorm, xtol;
  Nag_Boolean init;
  Nag_Comm    comm;
  Integer     iuser[1], *icomm = 0;
  double      ruser[1], *rcomm = 0, *fvec = 0, *x = 0;
  NagError    fail;

  printf("nag_zero_sparse_nonlin_eqns_easy (c05qsc) Example Program Results\n");
  lrcomm = 12 + 3 * n;
  licomm = 8 * n + 19 + 3 * n;
  if (
      !(fvec = NAG_ALLOC(n, double)) ||
      !(x = NAG_ALLOC(n, double)) ||
      !(rcomm = NAG_ALLOC(lrcomm, double)) ||
      !(icomm = NAG_ALLOC(licomm, Integer))
      ) {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }
  comm.iuser = iuser;
  comm.user = ruser;
  xtol = sqrt(nag_machine_precision);
  /* The following starting values provide a rough solution. */
  for (j = 0; j < n; j++) x[j] = -1.0E0;
  for (i = 0; i <= 1; i++) {
    INIT_FAIL(fail);
    /* Perturb the system? */
    comm.iuser[0] = i;
    init = (i == 0 ? Nag_TRUE : Nag_FALSE);
    /* nag_zero_sparse_nonlin_eqns_easy (c05qsc).
     * Solution of a sparse system of nonlinear equations using function
     * values only (easy-to-use).
     */
    nag_zero_sparse_nonlin_eqns_easy(fcn, n, x, fvec, xtol, init, rcomm, lrcomm,
                                     icomm, licomm, &comm, &fail);
```

```
    if (fail.code == NE_NOERROR) {
      /* Compute Euclidean norm. */
      fnorm = 0.0E0;
      for (j = 0; j <n; j++) fnorm = pow(fvec[j], 2);
      fnorm = sqrt(fnorm);
      printf("\nFinal 2-norm of the residuals = %12.4e\n", fnorm);
      printf("\nFinal approximate solution\n\n");
      for (j = 0; j < n; j++)
        printf("%12.4f%s", x[j], (j + 1) % 3 ? " " : "\n");
      printf("\n");
    } else {
      printf("Error from nag_zero_sparse_nonlin_eqns_easy (c05qsc).\n%s\n",
             fail.message);
      if (fail.code == NE_TOO_MANY_FEVALS ||
          fail.code == NE_TOO_SMALL ||
          fail.code == NE_NO_IMPROVEMENT) {
        printf("\nApproximate solution\n");
        for (j = 0; j < n; j++)
          printf("%12.4f%s", x[j], (j + 1) % 3 ? " " : "\n");
        printf("\n");
      }
    }
  }
 END:
  NAG_FREE(fvec);
  NAG_FREE(x);
  NAG_FREE(rcomm);
  NAG_FREE(icomm);
  return exit_status;
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_zero_sparse_nonlin_eqns_easy (c05qsc) Example Program Results

Final 2-norm of the residuals =   1.7592e-09

Final approximate solution

     -0.5707      -0.6816      -0.7017
     -0.7042      -0.7014      -0.6919
     -0.6658      -0.5960      -0.4164


Final 2-norm of the residuals =   2.6329e-13

Final approximate solution

     -0.5697      -0.6804      -0.7004
     -0.7029      -0.7000      -0.6906
     -0.6646      -0.5951      -0.4159
```