Module 10.3: nag_nlin_sys
Roots of a System of Nonlinear Equations

`nag_nlin_sys` produces a procedure for solving a set of n nonlinear equations in n unknowns

$$f_i(x) = 0, \quad i = 1, 2, \ldots, n, \quad x = (x_1, x_2, \ldots, x_n)^T.$$

Contents

Introduction

This module contains the procedure `nag_nlin_sys_sol` which is designed to solve a set of n nonlinear equations in n unknowns

$$f_i(x) = 0, \quad i = 1, 2, \ldots, n, \quad x = (x_1, x_2, \ldots, x_n)^T.$$

It is assumed that the functions are continuous and differentiable so that the matrix of first partial derivatives of the functions, the *Jacobian* matrix $J_{ij}(x) = \partial f_i / \partial x_j$ evaluated at the point x, exists, though it may not be possible to calculate it directly. Since the method is iterative, an initial guess at the solution has to be supplied, and the solution located will usually be the one closest to this initial guess.

Procedure: nag_nlin_sys_sol

1 Description

nag_nlin_sys_sol is a procedure designed to solve a set of n nonlinear equations in n unknowns

$$f_i(x) = 0, \quad i = 1, 2, \ldots, n, \quad x = (x_1, x_2, \ldots, x_n)^T.$$

2 Usage

```
USE nag_nlin_sys
CALL nag_nlin_sys_sol(fun, x  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n \geq 1$ — the number of equations

3.1 Mandatory Arguments

fun — subroutine

The procedure fun, supplied by the user, must calculate the vector of values $f_i(x)$ and, optionally, their first derivatives $\partial f_i/\partial x_j$ at any point x.

Its specification is:

```
subroutine fun(x, finish, f_vec, f_jac)

real(kind=wp), intent(in) ::  x(:)
```

Shape: x has shape (n).

Input: the point x at which the values of f_i and (optionally) $\partial f_i/\partial x_j$ are required, for $i, j = 1, 2, \ldots, n$.

```
logical, intent(inout) ::  finish
```

Input: finish will always be .false. on entry.

Output: if you wish to terminate the call to this procedure, you should set finish to .true.. If finish is .true. on exit from fun, then this procedure will terminate with error%code = 201.

```
real(kind=wp), intent(out) ::  f_vec(:)
```

Shape: f_vec has shape (n).

Output: unless finish is set to .true., f_vec(i) must contain the value of f_i at the point x, for $i = 1, 2, \ldots, n$.

`real(kind=wp), intent(out), optional :: f_jac(:,:)`

Shape: `f_jac` has shape (n, n).

Output: if present, `f_jac(i,j)` must contain the value of the first derivative $\partial f_i/\partial x_j$ at the point x, for $i, j = 1, 2, \ldots, n$.

Note: if the optional argument `user_jac = .false.`(see Section 3.2), then `f_jac` is not present in any call to `fun`. If `user_jac = .true.`, then `f_jac` is present in some calls to `fun`, but not in others; therefore you *must* test for the presence of `f_jac` before assigning any values to it. In all cases, `f_jac` must appear as a dummy argument in the user-supplied procedure `fun`.

x(n) — real(kind=*wp*), intent(inout)

Input: an initial estimate of the solution.

Output: the final estimate of the solution.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

user_jac — logical, intent(in), optional

Input: specifies whether or not the Jacobian is to be evaluated by the user-supplied procedure `fun`.

If `user_jac=.true.`, the Jacobian will be evaluated by `fun`;

if `user_jac=.false.`, the Jacobian will be approximated by this procedure using finite differences, and `fun` is called to evaluate function values only.

Default: `user_jac=.true.`.

num_jac_eval — integer, intent(out), optional

Output: the number of calls made to `fun` to evaluate the Jacobian.

Note: `num_jac_eval` is set to 0 if `user_jac=.false.`.

jac_check(n) — real(kind=*wp*), intent(out), optional

Output: contains the measures of correctness of the respective gradients at the initial point x. If there is no loss of significance (see Further Details for more information), then if `jac_check(`i`)` is 1.0 the ith user-supplied gradient is correct, whilst if `jac_check(`i`)` is 0.0 the ith gradient is incorrect. For values of `jac_check(`i`)` between 0.0 and 1.0 the categorisation is less certain. In general, a value of `jac_check(`i`)` > 0.5 indicates that the ith gradient is probably correct.

Note: `jac_check` is set to 0 if `user_jac=.false.`.

jac_sub_diag — integer, intent(in), optional

jac_sup_diag — integer, intent(in), optional

Input: the number of subdiagonals and superdiagonals, respectively, within the band of the Jacobian matrix. (If the Jacobian is not banded, or you are unsure, the default value should be used.)

Default: `jac_sub_diag` $= n - 1$, `jac_sup_diag` $= n - 1$.

Constraints: `jac_sub_diag` ≥ 0, `jac_sup_diag` ≥ 0.

Note: both these arguments are ignored if `user_jac = .true.`.

f_err — real(kind=*wp*), intent(in), optional

Input: a rough estimate of the largest relative error in the functions. It is used in determining a suitable step for a forward difference approximation to the Jacobian.

Default: `f_err` = `EPSILON(1.0_wp)`.

Constraints: `f_err` $\geq$ `EPSILON(1.0_wp)`.

Note: this argument is ignored if `user_jac` = `.true.`.

x_tol — real(kind=*wp*), intent(in), optional

Input: the accuracy in `x` to which the solution is required.

Default: `x_tol` = `SQRT(EPSILON(1.0_wp))`.

Constraints: `x_tol` ≥ 0.0.

scale(n) — real(kind=*wp*), intent(inout), optional

Input: the multiplicative scale factors for the variables. If `scale(`i`)` $= 0.0$ for $i = 1, 2, \ldots, n$, then the variables will be scaled internally.

Output: the scale factors actually used.

Default: `scale(`i`)` $= 0.0$ for $i = 1, 2, \ldots, n$.

Constraints: `scale(`i`)` ≥ 0.0.

factor — real(kind=*wp*), intent(in), optional

Input: a quantity to be used in determining the initial step bound. In most cases, `factor` should lie between 0.1 and 100. (The step bound is `factor` $\times \|$ `scale` $\times$ `x` $\|_2$ if this is non-zero; otherwise the bound is `factor`.)

Default: `factor` = 100.0.

Constraints: `factor` ≥ 0.0.

max_fun_eval — integer, intent(in), optional

Input: the maximum number of calls to `fun`. If, at the end of an iteration, the number of calls to `fun` exceeds `max_fun_eval`, this procedure will exit with `error%code` = 202.

Default: `max_fun_eval` $= 200(n + 1)$.

Constraints: `max_fun_eval` > 0.

num_fun_eval — integer, intent(out), optional

Output: the number of calls made to `fun` to evaluate the functions.

f_vec(n) — real(kind=*wp*), intent(out), optional

Output: the function values at the final point, `x`.

q_jac(n, n) — real(kind=*wp*), intent(out), optional

Output: the orthogonal matrix Q produced by the QR factorization of the final approximate Jacobian.

r_jac($n(n + 1)/2$) — real(kind=*wp*), intent(out), optional

Output: the upper triangular matrix R produced by the QR factorization of the final approximate Jacobian, stored row-wise, i.e., R_{ij} is stored in `r_jac(`$j + (2(n + 1) - i)(i - 1)/2$`)` for $i \geq j$ where $i, j = 1, \ldots, n$.

qt_f(n) — real(kind=*wp*), intent(out), optional

Output: the vector $Q^T f$.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
304	Invalid presence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	Execution terminated by the user.
	finish has been set to .true. in the user-supplied procedure fun.
202	Maximum number of function evaluations reached.
	Consider restarting the calculation from the final point held in x.
203	Tolerance required is too small.
	No further improvement in the approximate solution, x, is possible.
204	The iteration is not making good progress.
	The last 5 Jacobian evaluations indicate that further improvement cannot be made.
205	The iteration is not making good progress.
	The last 10 iterations indicate that further improvement cannot be made.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 and 2 of this module document.

6 Further Comments

6.1 Algorithmic Detail

This procedure is based upon the MINPACK procedures HYBRJ and HYBRD (See Moré *et al.* [1]). It chooses the correction at each step as a convex combination of the Newton and scaled gradient directions. Under reasonable conditions this guarantees global convergence for starting points far from the solution and a fast rate of convergence. The Jacobian is updated by the rank-1 method of Broyden. At the starting point the Jacobian is calculated (or approximated by finite differences), but it is not recalculated or approximated again until the rank-1 method fails to produce satisfactory progress. For more details see Powell [3].

6.2 Accuracy

If $\hat{x}$ is the true solution and D denotes the diagonal matrix whose entries are defined by the array `scale`, then this procedure tries to ensure that $\|D(x-\hat{x})\|_2 \leq$ `x_tol` $\times\|D\hat{x}\|_2$. If this condition is satisfied with `x_tol` $= 10^{-k}$, 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 this procedure usually avoids this possibility.

If `x_tol` is less than `EPSILON(1.0_`*wp*`)` and the above test is satisfied with `EPSILON(1.0_`*wp*`)` in place of `x_tol`, then the procedure exits with `error%code` $= 203$.

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

The test assumes that the functions and the Jacobian (if provided) are reasonably well behaved and consistently coded. If this condition is not satisfied, then this procedure may incorrectly indicate convergence. The validity of the answer can be checked, for example, by rerunning this procedure with a tighter tolerance.

6.3 Timing

The time required by this procedure 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 this procedure to process each call of `fun` is about $11.5n^2$ and $1.3n^3$ for the evaluation of the Jacobian. Unless `fun` can be evaluated quickly, the timing of this procedure will be strongly influenced by the time spent in `fun`. Ideally the problem should be scaled so that, at the solution, the function values are of comparable magnitude.

Example 1: Solution of a tridiagonal system of equations (finite difference Jacobian)

To determine the values $x_1, \ldots, x_9$ which satisfy the tridiagonal equations:

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

The Jacobian is approximated by the procedure.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE nlin_sys_ex01_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  SUBROUTINE fun(x,finish,f_vec,f_jac)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIZE
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: finish
    ! .. Array Arguments ..
    REAL (wp), OPTIONAL, INTENT (OUT) :: f_jac(:,:)
    REAL (wp), INTENT (OUT) :: f_vec(:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Local Scalars ..
    INTEGER :: n
    ! .. Executable Statements ..

    n = SIZE(x)

    f_vec = (3.0_wp-2.0_wp*x)*x + 1.0_wp
    f_vec(2:n) = f_vec(2:n) - x(1:n-1)
    f_vec(1:n-1) = f_vec(1:n-1) - 2.0_wp*x(2:n)

  END SUBROUTINE fun

END MODULE nlin_sys_ex01_mod

PROGRAM nag_nlin_sys_ex01

  ! Example Program Text for nag_nlin_sys
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_nlin_sys, ONLY : nag_nlin_sys_sol
  USE nlin_sys_ex01_mod, ONLY : fun, wp
```

```
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Parameters ..
    INTEGER, PARAMETER :: n = 9
    ! .. Local Arrays ..
    REAL (wp) :: x(n)
    ! .. Executable Statements ..

    WRITE (nag_std_out,*) 'Example Program Results for nag_nlin_sys_ex01'

    ! Starting values for the initial approximate solution
    x = -0.1_wp

    ! Solve the system of non-linear equations

    CALL nag_nlin_sys_sol(fun,x,user_jac=.FALSE.)

    WRITE (nag_std_out,'(/1X,A/3(/3F12.4))') 'Final approximate solution', x

  END PROGRAM nag_nlin_sys_ex01
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_nlin_sys_ex01

 Final approximate solution

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

Example 2: Solution of a tridiagonal system of equations (user-supplied Jacobian)

To determine the values $x_1, \ldots, x_9$ which satisfy the tridiagonal equations:

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

This example illustrates the use of the optional argument `jac_check` in order to validate the user-supplied Jacobian for consistency with the function values specified.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE nlin_sys_ex02_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  SUBROUTINE fun(x,finish,f_vec,f_jac)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC PRESENT, SIZE
    ! .. Scalar Arguments ..
    LOGICAL, INTENT (INOUT) :: finish
    ! .. Array Arguments ..
    REAL (wp), OPTIONAL, INTENT (OUT) :: f_jac(:,:)
    REAL (wp), INTENT (OUT) :: f_vec(:)
    REAL (wp), INTENT (IN) :: x(:)
    ! .. Local Scalars ..
    INTEGER :: i, n
    ! .. Executable Statements ..

    n = SIZE(x)

    f_vec = (3.0_wp-2.0_wp*x)*x + 1.0_wp
    f_vec(2:n) = f_vec(2:n) - x(1:n-1)
    f_vec(1:n-1) = f_vec(1:n-1) - 2.0_wp*x(2:n)

    IF (PRESENT(f_jac)) THEN
      f_jac = 0.0_wp
      DO i = 1, n
        f_jac(i,i) = 3.0_wp - 4.0_wp*x(i)
        IF (i>1) f_jac(i,i-1) = -1.0_wp
        IF (i<n) f_jac(i,i+1) = -2.0_wp
      END DO
    END IF

  END SUBROUTINE fun

END MODULE nlin_sys_ex02_mod
```

```
PROGRAM nag_nlin_sys_ex02

  ! Example Program Text for nag_nlin_sys
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_nlin_sys, ONLY : nag_nlin_sys_sol
  USE nlin_sys_ex02_mod, ONLY : fun, wp
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Parameters ..
  INTEGER, PARAMETER :: n = 9
  CHARACTER (*), PARAMETER :: fmt1 = '(/,1X,A,/,3(/,3F12.4))'
  CHARACTER (*), PARAMETER :: fmt2 = '(/,1X,A,I4)'
  ! .. Local Scalars ..
  INTEGER :: num_fun_eval, num_jac_eval
  ! .. Local Arrays ..
  REAL (wp) :: jac_check(n), x(n)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_nlin_sys_ex02'

  ! Starting values for the initial approximate solution
  x = -0.1_wp

  ! Solve the system of non-linear equations: Include the optional
  ! argument jac_check to measure the consistency of the function
  ! and the Jacobian

  CALL nag_nlin_sys_sol(fun,x,num_fun_eval=num_fun_eval, &
   num_jac_eval=num_jac_eval,jac_check=jac_check)

  WRITE (nag_std_out,fmt1) 'Final approximate solution', x
  WRITE (nag_std_out,fmt1) 'Jacobian consistency errors', jac_check
  WRITE (nag_std_out,fmt2) 'Number of function evaluations =', &
   num_fun_eval
  WRITE (nag_std_out,fmt2) 'Number of Jacobian evaluations =', &
   num_jac_eval

END PROGRAM nag_nlin_sys_ex02
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_nlin_sys_ex02

 Final approximate solution

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

 Jacobian consistency errors

      1.0000      1.0000      1.0000
      1.0000      1.0000      1.0000
```

```
     1.0000      1.0000      1.0000

 Number of function evaluations =  17

 Number of Jacobian evaluations =   1
```

Mathematical Background

1 Introduction

`nag_nlin_sys_sol` is designed to solve a set of nonlinear equations in n unknowns

$$f_i(x) = 0, \quad i = 1, 2, \ldots, n, \quad x = (x_1, x_2, \ldots, x_n)^T. \tag{1}$$

It is assumed that the functions are continuous and differentiable so that the matrix of first partial derivatives of the functions, the *Jacobian* matrix $J_{ij}(x) = \partial f_i / \partial x_j$ evaluated at the point x, exists, though it may not be possible to calculate it directly.
The functions f_i must be independent, otherwise there will be an infinity of solutions and the methods will fail. However, even when the functions are independent the solutions may not be unique. Since the methods are iterative, an initial guess at the solution has to be supplied, and the solution located will usually be the one closest to this initial guess.

The solution of a set of nonlinear equations

$$f_i(x_1, x_2, \ldots, x_n) = 0, \quad i = 1, 2, \ldots, n \tag{2}$$

can be regarded as a special case of the problem of finding a minimum of a sum of squares

$$s(x) = \sum_{i=1}^{m} [f_i(x_1, x_2, \ldots, x_n)]^2 \quad (m \geq n). \tag{3}$$

So the procedures in Chapter 9 (Optimization) are relevant as well as the special nonlinear equations procedures.

The procedure `nag_nlin_sys_sol` is provided for solving a set of nonlinear equations. This procedure requires the f_i (and possibly their derivatives) to be calculated in user-supplied functions. These should be set up carefully so the procedure can work as efficiently as possible.

The main decision which has to be made by the user is whether to supply the derivatives $\partial f_i / \partial x_j$. It is advisable to do so if possible, since the results obtained by algorithms which use derivatives are generally more reliable than those obtained by algorithms which do not use derivatives.

Firstly, the calculation of the functions and their derivatives should be ordered so that *cancellation errors* are avoided. This is particularly important in a procedure that uses these quantities to build up estimates of higher derivatives.

Secondly, *scaling* of the variables has a considerable effect on the efficiency of the procedure. The problem should be designed so that the elements of x are of similar magnitude. The same comment applies to the functions, all the f_i should be of comparable size.

The accuracy is usually determined by the accuracy parameters of the procedure, but the following points may be useful.

- Greater accuracy in the solution may be requested by choosing smaller input values for the accuracy parameters. However, if unreasonable accuracy is demanded, rounding errors may become important and cause a failure.
- An approximation to the error in the solution is given by e, where e is the solution to the set of linear equations $J(x)e = -f(x)$ where $f(x) = (f_1(x), f_2(x), \ldots, f_n(x))^T$.
- If the functions $f_i(x)$ are changed by small amounts ε_i, for $i = 1, 2, \ldots, n$, then the corresponding change in the solution x is given approximately by σ, where σ is the solution of the set of linear equations $J(x)\sigma = -\varepsilon$. Thus one can estimate the sensitivity of x to any uncertainties in the specification of $f_i(x)$, for $i = 1, 2, \ldots, n$.

Further Details

1 Derivative Checking

In order to check the user-supplied derivatives for consistency with the functions themselves, the optional argument `jac_check` can be specified in the argument list of `nag_nlin_sys_sol`. You are strongly advised to make use of this functionality whenever the Jacobian is provided.

The checking procedure is based upon the MINPACK procedure CHKDER (see Moré *et al.* [1]). It checks the ith gradient for consistency with the ith function by computing a forward-difference approximation along a suitably chosen direction and comparing this approximation with the user-supplied gradient along the same direction. The principal characteristic of the checking procedure is its invariance under changes in scale of the variables or functions.

This procedure does not perform reliably if the cancellation or rounding errors cause a severe loss of significance in the evaluation of a function. Therefore, none of the components of x should be unusually small (in particular zero) or any other value which may cause loss of significance.

References

[1] Moré J J, Garbow B S, and Hillstrom K E (1974) User guide for MINPACK-1 *Technical Report ANL-80-74* Argonne National Laboratory

[2] Ortega J M and Rheinboldt W C (1970) *Iterative Solution of Nonlinear Equations in Several Variables* Academic Press

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

[4] Rabinowitz P (1970) *Numerical Methods for Nonlinear Algebraic Equations* Gordon and Breach