NAG Library Function Document

nag_ode_bvp_fd_nonlin_gen (d02rac)

1 Purpose

nag_ode_bvp_fd_nonlin_gen (d02rac) solves a two-point boundary value problem with general boundary conditions for a system of ordinary differential equations, using a deferred correction technique and Newton iteration.

2 Specification

```
#include <nag.h>
#include <nagd02.h>
void nag_ode_bvp_fd_nonlin_gen (Integer neq, double *deleps,
     void (*fcn)(Integer neq, double x, double eps, const double y[],
          double f[], Nag_User *comm),
     Integer numbeg, Integer nummix,
     void (*g)(Integer neq, double eps, const double ya[], const double yb[],
          double bc[], Nag_User *comm),
     Nag_MeshSet init, Integer mnp, Integer *np, double x[], double y[],
     double tol, double abt[],
     void (*jacobf)(Integer neq, double x, double eps, const double y[],
          double f[], Nag_User *comm),
     void (*jacobg)(Integer neq, double eps, const double ya[],
          const double yb[], double aj[], double bj[], Nag_User *comm),
     void (*jaceps)(Integer neq, double x, double eps, const double y[],
          double f[], Nag_User *comm),
     void (*jacgep)(Integer neq, double eps, const double ya[],
          const double yb[], double bcep[], Nag_User *comm),
     Nag_User *comm, NagError *fail)
```

3 Description

nag_ode_bvp_fd_nonlin_gen (d02rac) solves a two-point boundary value problem for a system of n ordinary differential equations in the interval $[a,b]$ with $b>a$. The system is written in the form

$$y_i' = f_i(x, y_1, y_2, \ldots, y_n), \quad i = 1, 2, \ldots, n \tag{1}$$

and the derivatives f_i are evaluated by **fcn**. With the differential equations (1) must be given a system of n (nonlinear) boundary conditions

$$g_i(y(a), y(b)) = 0, \quad i = 1, 2, \ldots, n,$$

where

$$y(x) = [y_1(x), y_2(x), \ldots, y_n(x)]^{\mathrm{T}}. \tag{2}$$

The functions g_i are evaluated by **g**. The solution is computed using a finite difference technique with deferred correction allied to a Newton iteration to solve the finite difference equations. The technique used is described fully in Pereyra (1979).

You must supply an absolute error tolerance and may also supply an initial mesh for the finite difference equations and an initial approximate solution (alternatively a default mesh and approximation are used). The approximate solution is corrected using Newton iteration and deferred correction. Then, additional points are added to the mesh and the solution is recomputed with the aim of making the error everywhere less than your tolerance and of approximately equidistributing the error on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If, on the other hand, the solution is required at several specific points then you should use the interpolation functions provided in Chapter e01 if these points do not themselves form a convenient mesh.

The Newton iteration requires Jacobian matrices

$$\left(\frac{\partial f_i}{\partial y_j}\right), \left(\frac{\partial g_i}{\partial y_j(a)}\right) \quad \text{and} \quad \left(\frac{\partial g_i}{\partial y_j(b)}\right).$$

These may be supplied through **jacobf** for $\left(\frac{\partial f_i}{\partial y_j}\right)$ and **jacobg** for the others. Alternatively the Jacobians may be calculated by numerical differentiation using the algorithm described in Curtis *et al.* (1974).

For problems of the type (1) and (2) for which it is difficult to determine an initial approximation from which the Newton iteration will converge, a continuation facility is provided. You must set up a family of problems

$$y' = f(x, y, \epsilon), \quad g(y(a), y(b), \epsilon) = 0, \tag{3}$$

where $f = [f_1, f_2, \ldots, f_n]^{\mathrm{T}}$ etc., and where ϵ is a continuation parameter. The choice $\epsilon = 0$ must give a problem (3) which is easy to solve and $\epsilon = 1$ must define the problem whose solution is actually required. The function solves a sequence of problems with ϵ values

$$0 = \epsilon_1 < \epsilon_2 < \cdots < \epsilon_p = 1. \tag{4}$$

The number p and the values ϵ_i are chosen by the function so that each problem can be solved using the solution of its predecessor as a starting approximation. Jacobians $\frac{\partial f}{\partial \epsilon}$ and $\frac{\partial g}{\partial \epsilon}$ are required and they may be supplied by you via **jaceps** and **jacgep** respectively or may be computed by numerical differentiation.

4 References

Curtis A R, Powell M J D and Reid J K (1974) On the estimation of sparse Jacobian matrices *J. Inst. Maths. Applics.* **13** 117–119

Pereyra V (1979) PASVA3: An adaptive finite-difference Fortran program for first order nonlinear, ordinary boundary problems *Codes for Boundary Value Problems in Ordinary Differential Equations. Lecture Notes in Computer Science* (eds B Childs, M Scott, J W Daniel, E Denman and P Nelson) **76** Springer–Verlag

5 Arguments

1: **neq** – Integer *Input*

On entry: n, the number of differential equations.

Constraint: **neq** > 0.

2: **deleps** – double * *Input/Output*

On entry: must be given a value which specifies whether continuation is required. If **deleps** ≤ 0.0 or **deleps** ≥ 1.0 then it is assumed that continuation is not required. If $0.0 <$ **deleps** < 1.0 then it is assumed that continuation is required unless **deleps** $< \sqrt{\textit{machine precision}}$ when an error exit is taken. **deleps** is used as the increment $\epsilon_2 - \epsilon_1$ (see (4)) and the choice **deleps** $= 0.1$ is recommended.

On exit: an overestimate of the increment $\epsilon_p - \epsilon_{p-1}$ (in fact the value of the increment which would have been tried if the restriction $\epsilon_p = 1$ had not been imposed). If continuation was not requested then **deleps** $= 0.0$.

If continuation is not requested then **jaceps** and **jacgep** may each be replaced by the NAG defined null function pointer NULLFN.

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

fcn must evaluate the functions f_i (i.e., the derivatives y_i') at a general point x for a given value of ϵ, the continuation parameter (see Section 3).

The specification of **fcn** is:

```
void fcn (Integer neq, double x, double eps, const double y[],
     double f[], Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: n, the number of equations.

2: **x** – double *Input*

On entry: x, the value of the independent variable.

3: **eps** – double *Input*

On entry: ϵ, the value of the continuation parameter. This is 1 if continuation is not being used.

4: **y**[**neq**] – const double *Input*

On entry: y_i, for $i = 1, 2, \dots, n$, the values of the dependent variables at x.

5: **f**[**neq**] – double *Output*

On exit: the values of the derivatives f_i evaluated at x given ϵ, for $i = 1, 2, \dots, n$.

6: **comm** – Nag_User *

Pointer to structure of type Nag_User.

p – Pointer

The type Pointer will be `void *`. Before calling nag_ode_bvp_fd_nonlin_gen (d02rac) these pointers may be allocated memory and initialized with various quantities for use by **comm** when called from nag_ode_bvp_fd_nonlin_gen (d02rac).

4: **numbeg** – Integer *Input*

On entry: the number of left-hand boundary conditions (that is the number involving $y(a)$ only).

Constraint: $0 \leq$ **numbeg** $<$ **neq**.

5: **nummix** – Integer *Input*

On entry: the number of coupled boundary conditions (that is the number involving both $y(a)$ and $y(b)$).

Constraint: $0 \leq$ **nummix** $\leq$ **neq** $-$ **numbeg**.

6: **g** – function, supplied by the user *External Function*

g must evaluate the boundary conditions in equation (3) and place them in the array **bc**.

The specification of **g** is:

```
void g (Integer neq, double eps, const double ya[], const double yb[],
     double bc[], Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: n, the number of equations.

2: **eps** – double *Input*

On entry: ϵ, the value of the continuation parameter. This is 1 if continuation is not being used.

3: **ya**[**neq**] – const double *Input*

On entry: the value $y_i(a)$, for $i = 1, 2, \ldots, n$.

4: **yb**[**neq**] – const double *Input*

On entry: the value $y_i(b)$, for $i = 1, 2, \ldots, n$.

5: **bc**[**neq**] – double *Output*

On exit: the values $g_i(y(a), y(b), \epsilon)$, for $i = 1, 2, \ldots, n$. These must be ordered as follows:

(i) first, the conditions involving only $y(a)$ (see **numbeg**);

(ii) next, the **nummix** coupled conditions involving both $y(a)$ and $y(b)$ (see **nummix**); and,

(iii) finally, the conditions involving only $y(b)$ (**neq** − **numbeg** − **nummix**).

6: **comm** – Nag_User *

Pointer to structure of type Nag_User.

p – Pointer

The type Pointer will be `void *`. Before calling nag_ode_bvp_fd_nonlin_gen (d02rac) these pointers may be allocated memory and initialized with various quantities for use by **comm** when called from nag_ode_bvp_fd_nonlin_gen (d02rac).

7: **init** – Nag_MeshSet *Input*

On entry: indicates whether you wish to supply an initial mesh and approximate solution (**init** = Nag_UserInitMesh) or whether default values are to be used, (**init** = Nag_DefInitMesh).

Constraint: **init** = Nag_DefInitMesh or Nag_UserInitMesh.

8: **mnp** – Integer *Input*

On entry: **mnp** must be set to the maximum permitted number of points in the finite difference mesh.

Constraint: **mnp** ≥ 32.

9: **np** – Integer * *Input/Output*

On entry: must be set to the number of points to be used in the initial mesh.

Constraint: $4 \leq$ **np** $\leq$ **mnp**.

On exit: the number of points in the final mesh.

10: **x**[**mnp**] – double *Input/Output*

On entry: you must set $\mathbf{x}[0] = a$ and $\mathbf{x}[\mathbf{np} - 1] = b$. If **init** = Nag_DefInitMesh on entry a default equispaced mesh will be used, otherwise you must specify a mesh by setting $\mathbf{x}[i - 1] = x_i$, for $i = 2, 3, \ldots, \mathbf{np} - 1$.

Constraints:

if **init** = Nag_DefInitMesh, $\mathbf{x}[0] < \mathbf{x}[\mathbf{np} - 1]$;
if **init** = Nag_UserInitMesh, $\mathbf{x}[0] < \mathbf{x}[1] < \cdots < \mathbf{x}[\mathbf{np} - 1]$.

On exit: $\mathbf{x}[0], \mathbf{x}[1], \ldots, \mathbf{x}[\mathbf{np} - 1]$ define the final mesh (with the returned value of **np**) and $\mathbf{x}[0] = a$ and $\mathbf{x}[\mathbf{np} - 1] = b$.

11: **y**[**neq** × **mnp**] – double *Input/Output*

On entry: if **init** = Nag_DefInitMesh, then **y** need not be set.

If **init** = Nag_UserInitMesh, then the array **y** must contain an initial approximation to the solution such that $\mathbf{y}[(j - 1) \times \mathbf{mnp} + i - 1]$ contains an approximation to

$$y_j(x_i), \quad i = 1, 2, \ldots, \mathbf{np} \text{ and } j = 1, 2, \ldots, n.$$

On exit: the approximate solution $z_j(x_i)$ satisfying (5) on the final mesh, that is

$$\mathbf{y}[(j - 1) \times \mathbf{mnp} + i - 1] = z_j(x_i), \quad i = 1, 2, \ldots, \mathbf{np} \text{ and } j = 1, 2, \ldots, n,$$

where **np** is the number of points in the final mesh. If an error has occurred then **y** contains the latest approximation to the solution. The remaining columns of **y** are not used.

12: **tol** – double *Input*

On entry: a positive absolute error tolerance. If

$$a = x_1 < x_2 < \cdots < x_{\mathbf{np}} = b$$

is the final mesh, $z_j(x_i)$ is the jth component of the approximate solution at x_i, and $y_j(x)$ is the jth component of the true solution of (1) and (2), then, except in extreme circumstances, it is expected that

$$\left|z_j(x_i) - y_j(x_i)\right| \leq \mathbf{tol}, \quad i = 1, 2, \ldots, \mathbf{np} \text{ and } j = 1, 2, \ldots, n. \qquad (5)$$

Constraint: **tol** > 0.0.

13: **abt**[**neq**] – double *Output*

On exit: $\mathbf{abt}[i - 1]$, for $i = 1, 2, \ldots, n$, holds the largest estimated error (in magnitude) of the ith component of the solution over all mesh points.

14: **jacobf** – function, supplied by the user *External Function*

jacobf evaluates the Jacobian $\left(\frac{\partial f_i}{\partial y_j}\right)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$, given x and y_j, for $j = 1, 2, \ldots, n$.

If all Jacobians are to be approximated internally by numerical differentiation then it must be replaced by the NAG defined null function pointer NULLFN.

The specification of **jacobf** is:

```
void jacobf (Integer neq, double x, double eps, const double y[],
     double f[], Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: n, the number of equations.

2: **x** – double *Input*

On entry: x, the value of the independent variable.

3: **eps** – double *Input*

On entry: ϵ, the value of the continuation parameter. This is 1 if continuation is not being used.

4: **y**[**neq**] – const double *Input*

On entry: y_i, for $i = 1, 2, \ldots, n$, the values of the dependent variables at x.

5: **f**[**neq** $\times$ **neq**] – double *Output*

On exit: $\mathbf{f}[(j-1) \times \mathbf{neq} + i - 1]$ must be set to the value of $\frac{\partial f_i}{\partial y_j}$, evaluated at the point (x, y), for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$.

6: **comm** – Nag_User *

Pointer to structure of type Nag_User.

p – Pointer

The type Pointer will be `void *`. Before calling nag_ode_bvp_fd_nonlin_gen (d02rac) these pointers may be allocated memory and initialized with various quantities for use by **comm** when called from nag_ode_bvp_fd_nonlin_gen (d02rac).

Note that if **jacobf** is supplied then **jacobg** must also be supplied. Note that if **jacobf** is supplied and continuation is requested then **jaceps** and **jacgep** must also be supplied.

15: **jacobg** – function, supplied by the user *External Function*

jacobg evaluates the Jacobians $\left(\frac{\partial g_i}{\partial y_j(a)}\right)$ and $\left(\frac{\partial g_i}{\partial y_j(b)}\right)$. The ordering of the rows of **aj** and **bj** must correspond to the ordering of the boundary conditions described in the specification of **g**.

If all Jacobians are to be approximated internally by numerical differentiation then it must be replaced by the NAG defined null function pointer NULLFN.

The specification of **jacobg** is:

```
void jacobg (Integer neq, double eps, const double ya[],
     const double yb[], double aj[], double bj[], Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: n, the number of equations.

2: **eps** – double *Input*

On entry: ϵ, the value of the continuation parameter. This is 1 if continuation is not being used.

3: **ya**[**neq**] – const double *Input*

On entry: the value $y_i(a)$, for $i = 1, 2, \ldots, n$.

4: **yb**[**neq**] – const double *Input*

On entry: the value $y_i(b)$, for $i = 1, 2, \ldots, n$.

5: **aj**[**neq** $\times$ **neq**] – double *Output*

On exit: **aj**[$(i-1) \times$ **neq** $+ j - 1$] must be set to the value $\dfrac{\partial g_i}{\partial y_j(a)}$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$.

6: **bj**[**neq** $\times$ **neq**] – double *Output*

On exit: **bj**[$(i-1) \times$ **neq** $+ j - 1$] must be set to the value $\dfrac{\partial g_i}{\partial y_j(b)}$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$.

7: **comm** – Nag_User *

Pointer to structure of type Nag_User.

p – Pointer

The type Pointer will be `void *`. Before calling nag_ode_bvp_fd_nonlin_gen (d02rac) these pointers may be allocated memory and initialized with various quantities for use by **comm** when called from nag_ode_bvp_fd_nonlin_gen (d02rac).

Note that if **jacobg** is supplied then **jacobf** must also be supplied.

16: **jaceps** – function, supplied by the user *External Function*

jaceps evaluates the derivative $\dfrac{\partial f_i}{\partial \epsilon}$ given x and y if continuation is being used.

If all Jacobians (derivatives) are to be approximated internally by numerical differentiation, or continuation is not being used, then it must be replaced by the NAG defined null function pointer NULLFN.

The specification of **jaceps** is:

```
void jaceps (Integer neq, double x, double eps, const double y[],
     double f[], Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: n, the number of equations.

2: **x** – double *Input*

On entry: x, the value of the independent variable.

3: **eps** – double *Input*

On entry: ϵ, the value of the continuation parameter.

4: **y**[**neq**] – const double *Input*

On entry: the solution values y_i, for $i = 1, 2, \ldots, n$, at the point x.

5: **f**[**neq**] – double *Output*

On exit: **f**[$i - 1$] must contain the value $\dfrac{\partial f_i}{\partial \epsilon}$ at the point (x, y), for $i = 1, 2, \ldots, n$.

6: **comm** – Nag_User *

Pointer to structure of type Nag_User.

p – Pointer

The type Pointer will be `void *`. Before calling nag_ode_bvp_fd_nonlin_gen (d02rac) these pointers may be allocated memory and initialized with various quantities for use by **comm** when called from nag_ode_bvp_fd_nonlin_gen (d02rac).

Note that if **jaceps** is defined then **jacgep** must also be defined.

17: **jacgep** – function, supplied by the user *External Function*

jacgep evaluates the derivatives $\frac{\partial g_i}{\partial \epsilon}$ if continuation is being used.

If all Jacobians (derivatives) are to be approximated internally by numerical differentiation, or continuation is not being used, then it must be replaced by the NAG defined null function pointer NULLFN.

The specification of **jacgep** is:

```
void jacgep (Integer neq, double eps, const double ya[],
     const double yb[], double bcep[], Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: n, the number of equations.

2: **eps** – double *Input*

On entry: ϵ, the value of the continuation parameter.

3: **ya**[**neq**] – const double *Input*

On entry: the value of $y_i(a)$, for $i = 1, 2, \dots, n$.

4: **yb**[**neq**] – const double *Input*

On entry: the value of $y_i(b)$, for $i = 1, 2, \dots, n$.

5: **bcep**[**neq**] – double *Output*

On exit: **bcep**$[i-1]$ must contain the value of $\frac{\partial g_i}{\partial \epsilon}$, for $i = 1, 2, \dots, n$.

6: **comm** – Nag_User *

Pointer to structure of type Nag_User.

p – Pointer

The type Pointer will be `void *`. Before calling nag_ode_bvp_fd_nonlin_gen (d02rac) these pointers may be allocated memory and initialized with various quantities for use by **comm** when called from nag_ode_bvp_fd_nonlin_gen (d02rac).

Note that if **jacgep** is defined then **jaceps** must also be defined.

18: **comm** – Nag_User *

Pointer to structure of type Nag_User.

p – Pointer

The type Pointer will be `void *`. Before calling nag_ode_bvp_fd_nonlin_gen (d02rac) these pointers may be allocated memory and initialized with various quantities for use by **comm** when called from nag_ode_bvp_fd_nonlin_gen (d02rac).

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

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_2_INT_ARG_ZERO

On entry, **numbeg** $= 0$ and **nummix** $= 0$. These arguments must not both be zero.

NE_2_REAL_ARG_LE

On entry, $\mathbf{x}[0] = \langle value \rangle$ and $\mathbf{x}[\mathbf{np} - 1] = \langle value \rangle$.
Constraint: $\mathbf{x}[0] < \mathbf{x}[\mathbf{np} - 1]$.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

On entry, argument **init** had an illegal value.

NE_CONV_CONT

Convergence failure. There are a number of possible causes.

(a) Faulty coding of the Jacobian calculation functions.

(b) If Jacobians have not been supplied then inaccurate Jacobians have been calculated internally (not very likely).

(c) A poor choice of initial mesh or initial starting conditions either by the user or by default. Try using the continuation facility.

The Newton iteration has failed to converge.
This could be due to there being too few points in the initial mesh or to the initial approximate solution being too inaccurate.
If this latter reason is suspected or you cannot make changes to prevent this error, you should use the function with a continuation facility instead.

NE_CONV_CONT_DELEPS

deleps is required to be less than ***machine precision*** for continuation to proceed. It is likely that either the problem has no solution for some value near the current value of ϵ or that the problem is so difficult that even with continuation it is unlikely to be solved using this function. Using more mesh points may help.

The continuation step is required to be less than ***machine precision*** for continuation to proceed. It is likely that either the problem has no solution for some value of the continuation parameter near the current value or that the problem is so difficult that even with continuation it is unlikely to be solved using this function. In the latter case using more mesh points initially may help.

NE_CONV_CONT_DEP

There is no dependence on the continuation parameter when continuation is being used. This can be due to faulty coding of derivatives with respect to the continuation parameter or to a zero initial choice of approximate solution.

There is no dependence on ϵ when continuation is being used. This may be due to faulty coding of **jaceps** or **jacgep**, or in some circumstances, to a zero initial choice of approximate solution (such as is chosen when **init** $=$ Nag_DefInitMesh).

NE_CONV_JACOBG

The Jacobian calculated by **jacobg** (or the equivalent matrix calculated by numerical differentiation) is singular. This may be due to faulty coding of **jacobg** or in some circumstances, to a zero initial choice of approximate solution (such as is chosen when **init** $=$ Nag_DefInitMesh).

The Jacobian for the boundary conditions is singular.
This may occur due to faulty coding of the Jacobian or, in some circumstances, to a zero initial choice of approximate solution.

NE_CONV_MESH

A finer mesh is required for the accuracy requested; that is, **mnp** $= \langle value \rangle$ is not large enough.

NE_CONV_ROUNDOFF

Newton iteration has reached round-off level.
If desired accuracy has not been reached, then **tol** is too small for this problem and this ***machine precision***.

Solution cannot be improved due to roundoff error. Too much accuracy might have been requested.

NE_INT_ARG_LT

On entry, **mnp** $= \langle value \rangle$.
Constraint: **mnp** ≥ 32.

On entry, **neq** $= \langle value \rangle$.
Constraint: **neq** ≥ 1.

On entry, **np** $= \langle value \rangle$.
Constraint: **np** ≥ 4.

On entry, **numbeg** $= \langle value \rangle$.
Constraint: **numbeg** ≥ 0.

On entry, **nummix** $= \langle value \rangle$.
Constraint: **nummix** ≥ 0.

NE_INT_RANGE_CONS

On entry, **np** $= \langle value \rangle$ and **mnp** $= \langle value \rangle$.
Constraint: **np** $\leq$ **mnp**.

On entry, **numbeg** $= \langle value \rangle$ and **neq** $= \langle value \rangle$.
Constraint: **numbeg** $<$ **neq**.

On entry, **numbeg** $= \langle value \rangle$, **nummix** $= \langle value \rangle$ and **neq** $= \langle value \rangle$.
Constraint: **numbeg** $+$ **nummix** $\leq$ **neq**.

NE_INTERNAL_ERROR

A continuation error occurred, but continuation is not being used.
Please contact NAG.

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.

A serious error occurred in a call to the internal integrator.
The error code internally was $\langle$*value*$\rangle$.
Please contact NAG.

NE_INVALID_FUN_JAC

Only one of **jacobf** or **jacobg** has been set to non-null possibly implying user-defined Jacobians. Both must be non-null.

NE_INVALID_FUN_JAC_CONT

deleps has been set to $\langle$*value*$\rangle$ implying continuation and both **jacobf** and **jacobg** have been set to non-null implying user-defined Jacobians. Hence the functions **jaceps** and **jacgep** must also be non-null.

NE_INVALID_FUN_JAC_NO_CONT

deleps has been set to $\langle$*value*$\rangle$ implying no continuation and both **jacobf** and **jacobg** have been set to non-null implying user-defined Jacobians. Hence the functions **jaceps** and **jacgep** must be **NULL**.

NE_INVALID_FUN_NO_JAC_CONT

deleps has been set to $\langle$*value*$\rangle$ implying continuation and both **jacobf** and **jacobg** have been set to **NULL** implying no user-defined Jacobians. Hence the functions **jaceps** and **jacgep** must also be **NULL**.

NE_NOT_STRICTLY_INCREASING

On entry the mesh points are not in strictly ascending order.
For $i = \langle$*value*$\rangle$, mesh point $i = \langle$*value*$\rangle$, but mesh point $i + 1 = \langle$*value*$\rangle$.

NE_REAL_ARG_LE

On entry, **tol** $= \langle$*value*$\rangle$.
Constraint: **tol** > 0.0.

7 Accuracy

The solution returned by the function will be accurate to your tolerance as defined by the relation (5) except in extreme circumstances. The final error estimate over the whole mesh for each component is given in the array **abt**. If too many points are specified in the initial mesh, the solution may be more accurate than requested and the error may not be approximately equidistributed.

8 Parallelism and Performance

nag_ode_bvp_fd_nonlin_gen (d02rac) is not threaded in any implementation.

9 Further Comments

There are too many factors present to quantify the timing. The time taken by nag_ode_bvp_fd_nonlin_gen (d02rac) is negligible only on very simple problems.

In the case where you wish to solve a sequence of similar problems, the use of the final mesh and solution from one case as the initial mesh is strongly recommended for the next.

10 Example

This example solves the differential equation

$$y''' = -yy'' - 2\epsilon\left(1 - y'^2\right)$$

with $\epsilon = 1$ and boundary conditions

$$y(0) = y'(0) = 0, \quad y'(10) = 1$$

to an accuracy specified by **tol** $= 1.0\text{e}{-4}$. The continuation facility is used with the continuation parameter ϵ introduced as in the differential equation above and with **deleps** $= 0.1$ initially. (The continuation facility is not needed for this problem and is used here for illustration.)

10.1 Program Text

```
/* nag_ode_bvp_fd_nonlin_gen (d02rac) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 *
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nagd02.h>

#ifdef __cplusplus
extern "C"
{
#endif
  static void NAG_CALL fcn(Integer neq, double x, double eps,
                           const double y[], double f[], Nag_User *comm);
  static void NAG_CALL g(Integer neq, double eps, const double ya[],
                         const double yb[], double bc[], Nag_User *comm);
  static void NAG_CALL jaceps(Integer neq, double x, double eps,
                              const double y[], double f[], Nag_User *comm);
  static void NAG_CALL jacgep(Integer neq, double eps, const double ya[],
                              const double yb[], double bcep[],
                              Nag_User *comm);
  static void NAG_CALL jacobf(Integer neq, double x, double eps,
                              const double y[], double f[], Nag_User *comm);
  static void NAG_CALL jacobg(Integer neq, double eps, const double ya[],
                              const double yb[], double aj[], double bj[],
                              Nag_User *comm);
#ifdef __cplusplus
}
#endif

#define NEQ 3
#define MNP 40

#define Y(I, J) y[(I) *tdy + J]
int main(void)
{

  static Integer use_comm[6] = { 1, 1, 1, 1, 1, 1 };
  double *abt = 0;
  double deleps;
  double tol;
  double *x = 0, *y = 0;
  Integer exit_status = 0;
  Integer i, j;
  Integer np;
  Integer numbeg, nummix;
  Integer neq, mnp, tdy;
  Nag_User comm;
  NagError fail;

  INIT_FAIL(fail);

  printf("nag_ode_bvp_fd_nonlin_gen (d02rac) Example Program Results\n");

  /* For communication with user-supplied functions: */
  comm.p = (Pointer) &use_comm;

  printf("\nCalculation using analytic Jacobians\n\n");
  neq = NEQ;
  mnp = MNP;
  if (neq >= 1) {
    if (!(abt = NAG_ALLOC(neq, double)) ||
        !(x = NAG_ALLOC(mnp, double)) || !(y = NAG_ALLOC(neq * mnp, double)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
    tdy = mnp;
  }
  else {
    exit_status = 1;
    return exit_status;
  }
  tol = 1.0e-4;
  np = 17;
  numbeg = 2;
  nummix = 0;
  x[0] = 0.0;
  x[np - 1] = 10.0;
  deleps = 0.1;

  /* nag_ode_bvp_fd_nonlin_gen (d02rac).
   * Ordinary differential equations solver, for general
   * nonlinear two-point boundary value problems, using a
   * finite difference technique with deferred correction
   */
  nag_ode_bvp_fd_nonlin_gen(neq, &deleps, fcn, numbeg, nummix, g,
                            Nag_DefInitMesh, mnp, &np, x, y, tol, abt, jacobf,
                            jacobg, jaceps, jacgep, &comm, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ode_bvp_fd_nonlin_gen (d02rac).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }

  printf("Solution on final mesh of %" NAG_IFMT " points \n", np);
  printf("%7s%15s%13s%13s\n","x","y_1","y_2","y_3");

  for (j = 0; j < np; ++j) {
    printf("%10.6f", x[j]);
    for (i = 0; i < neq; ++i)
      printf("%13.4f", Y(i, j));
    printf("\n");
  }

  printf("\n\nMaximum estimated error by components \n");

  for (i = 0; i < 2; ++i)
    printf(" %11.2e", abt[i]);
  printf(" \n");

END:
  NAG_FREE(abt);
  NAG_FREE(x);
  NAG_FREE(y);
  return exit_status;
}

#undef Y

static void NAG_CALL fcn(Integer neq, double x, double eps, const double y[],
                         double f[], Nag_User *comm)
{
  Integer *use_comm = (Integer *) comm->p;

  if (use_comm[0]) {
    printf("(User-supplied callback fcn, first invocation.)\n");
    use_comm[0] = 0;
  }

  f[0] = y[1];
  f[1] = y[2];
  f[2] = -y[0] * y[2] - (1.0 - y[1] * y[1]) * 2.0 * eps;
}

static void NAG_CALL g(Integer neq, double eps, const double ya[],
                       const double yb[], double bc[], Nag_User *comm)
{
  Integer *use_comm = (Integer *) comm->p;

  if (use_comm[1]) {
    printf("(User-supplied callback g, first invocation.)\n");
    use_comm[1] = 0;
  }

  bc[0] = ya[0];
  bc[1] = ya[1];
  bc[2] = yb[1] - 1.0;
} /* g */

static void NAG_CALL jaceps(Integer neq, double x, double eps,
                            const double y[], double f[], Nag_User *comm)
{
  Integer *use_comm = (Integer *) comm->p;

  if (use_comm[2]) {
    printf("(User-supplied callback jaceps, first invocation.)\n");
    use_comm[2] = 0;
  }

  f[0] = 0.0;
  f[1] = 0.0;
  f[2] = (1.0 - y[1] * y[1]) * -2.0;
}

static void NAG_CALL jacgep(Integer neq, double eps, const double ya[],
                            const double yb[], double bcep[], Nag_User *comm)
{
  Integer i;
  Integer *use_comm = (Integer *) comm->p;

  if (use_comm[3]) {
    printf("(User-supplied callback jacgep, first invocation.)\n");
    use_comm[3] = 0;
  }

  for (i = 0; i < neq; ++i)
    bcep[i] = 0.0;
}

static void NAG_CALL jacobf(Integer neq, double x, double eps,
                            const double y[], double f[], Nag_User *comm)
{
  Integer i, j;
  Integer *use_comm = (Integer *) comm->p;
```

```
#define Y(I)    y[(I) -1]
#define F(I, J) f[((I) -1)*neq+(J) -1]

  if (use_comm[4]) {
    printf("(User-supplied callback jacobf, first invocation.)\n");
    use_comm[4] = 0;
  }

  for (i = 1; i <= neq; ++i) {
    for (j = 1; j <= neq; ++j)
      F(i, j) = 0.0;
  }
  F(1, 2) = 1.0;
  F(2, 3) = 1.0;
  F(3, 1) = -Y(3);
  F(3, 2) = Y(2) * 4.0 * eps;
  F(3, 3) = -Y(1);
}

static void NAG_CALL jacobg(Integer neq, double eps, const double ya[],
                            const double yb[], double aj[], double bj[],
                            Nag_User *comm)
{
  Integer i, j;
  Integer *use_comm = (Integer *) comm->p;

#define AJ(I, J) aj[((I) -1)*neq+(J) -1]
#define BJ(I, J) bj[((I) -1)*neq+(J) -1]

  if (use_comm[5]) {
    printf("(User-supplied callback jacobg, first invocation.)\n");
    use_comm[5] = 0;
  }

  for (i = 1; i <= neq; ++i)
    for (j = 1; j <= neq; ++j) {
      AJ(i, j) = 0.0;
      BJ(i, j) = 0.0;
    }
  AJ(1, 1) = 1.0;
  AJ(2, 2) = 1.0;
  BJ(3, 2) = 1.0;
}
```

10.2 Program Data

None.

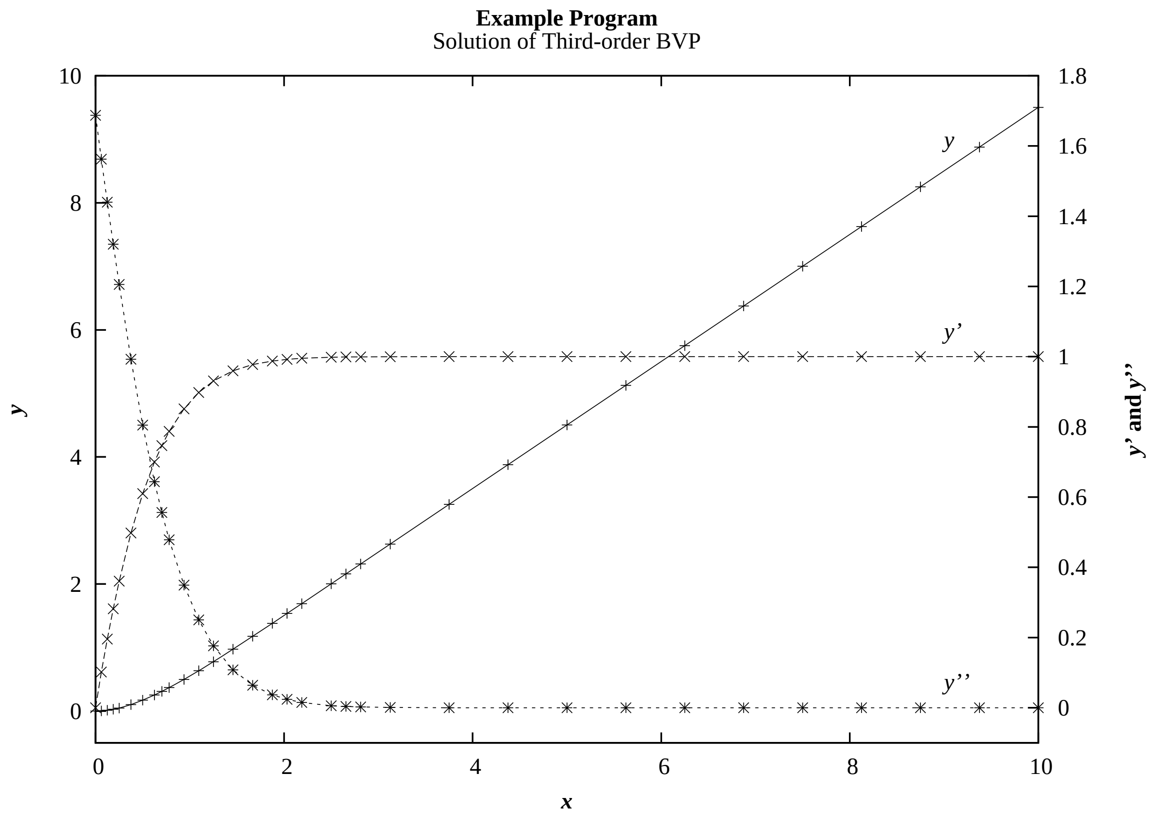
10.3 Program Results

```
nag_ode_bvp_fd_nonlin_gen (d02rac) Example Program Results

Calculation using analytic Jacobians

(User-supplied callback fcn, first invocation.)
(User-supplied callback g, first invocation.)
(User-supplied callback jacobf, first invocation.)
(User-supplied callback jacobg, first invocation.)
(User-supplied callback jaceps, first invocation.)
(User-supplied callback jacgep, first invocation.)
Solution on final mesh of 33 points
      x           y_1          y_2          y_3
  0.000000      0.0000       0.0000       1.6872
  0.062500      0.0032       0.1016       1.5626
  0.125000      0.0125       0.1954       1.4398
  0.187500      0.0275       0.2816       1.3203
  0.250000      0.0476       0.3605       1.2054
  0.375000      0.1015       0.4976       0.9924
  0.500000      0.1709       0.6097       0.8048
  0.625000      0.2530       0.6999       0.6438
  0.703125        0.3095        0.7467        0.5563
  0.781250        0.3695        0.7871        0.4784
  0.937500        0.4978        0.8513        0.3490
  1.093750        0.6346        0.8977        0.2502
  1.250000        0.7776        0.9308        0.1763
  1.458333        0.9748        0.9598        0.1077
  1.666667        1.1768        0.9773        0.0639
  1.875000        1.3815        0.9876        0.0367
  2.031250        1.5362        0.9922        0.0238
  2.187500        1.6915        0.9952        0.0151
  2.500000        2.0031        0.9983        0.0058
  2.656250        2.1591        0.9990        0.0035
  2.812500        2.3153        0.9994        0.0021
  3.125000        2.6277        0.9998        0.0007
  3.750000        3.2526        1.0000        0.0001
  4.375000        3.8776        1.0000        0.0000
  5.000000        4.5026        1.0000        0.0000
  5.625000        5.1276        1.0000       -0.0000
  6.250000        5.7526        1.0000        0.0000
  6.875000        6.3776        1.0000       -0.0000
  7.500000        7.0026        1.0000        0.0000
  8.125000        7.6276        1.0000       -0.0000
  8.750000        8.2526        1.0000        0.0000
  9.375000        8.8776        1.0000       -0.0000
 10.000000        9.5026        1.0000        0.0000

Maximum estimated error by components
    6.92e-05    1.81e-05
```


Example Program
Solution of Third-order BVP
y
y'
y''
x
y' and y''