NAG Library Function Document

nag_ode_ivp_bdf_gen (d02ejc)

1 Purpose

nag_ode_ivp_bdf_gen (d02ejc) integrates a stiff system of first-order ordinary differential equations over an interval with suitable initial conditions, using a variable-order, variable-step method implementing the Backward Differentiation Formulae (BDF), until a user-specified function, if supplied, of the solution is zero, and returns the solution at specified points, if desired.

2 Specification

```
#include <nag.h>
#include <nagd02.h>

void nag_ode_ivp_bdf_gen (Integer neq,
     void (*fcn)(Integer neq, double x, const double y[], double f[],
          Nag_User *comm),
     void (*pederv)(Integer neq, double x, const double y[], double pw[],
          Nag_User *comm),
     double *x, double y[], double xend, double tol, Nag_ErrorControl err_c,
     void (*output)(Integer neq, double *xsol, const double y[],
          Nag_User *comm),
     double (*g)(Integer neq, double x, const double y[], Nag_User *comm),
     Nag_User *comm, NagError *fail)
```

3 Description

nag_ode_ivp_bdf_gen (d02ejc) advances the solution of a system of ordinary differential equations

$$y_i' = f_i(x, y_1, y_2, \ldots, y_{\mathbf{neq}}), \quad i = 1, 2, \ldots, \mathbf{neq},$$

from $x = \mathbf{x}$ to $x = \mathbf{xend}$ using a variable-order, variable-step method implementing the BDF. The system is defined by **fcn**, which evaluates f_i in terms of x and $y_1, y_2, \ldots, y_{\mathbf{neq}}$ (see Section 5). The initial values of $y_1, y_2, \ldots, y_{\mathbf{neq}}$ must be given at $x = \mathbf{x}$.

The solution is returned via **output** at specified points, if desired: this solution is obtained by C^1 interpolation on solution values produced by the method. As the integration proceeds a check can be made on the user-specified function $g(x, y)$ to determine an interval where it changes sign. The position of this sign change is then determined accurately. It is assumed that $g(x, y)$ is a continuous function of the variables, so that a solution of $g(x, y) = 0.0$ can be determined by searching for a change in sign in $g(x, y)$. The accuracy of the integration, the interpolation and, indirectly, of the determination of the position where $g(x, y) = 0.0$, is controlled by the arguments **tol** and **err_c**. The Jacobian of the system $y' = f(x, y)$ may be supplied in function **pederv**, if it is available.

For a description of BDF and their practical implementation see Hall and Watt (1976).

4 References

Hall G and Watt J M (ed.) (1976) *Modern Numerical Methods for Ordinary Differential Equations* Clarendon Press, Oxford

5 Arguments

1: **neq** – Integer *Input*

On entry: the number of differential equations.

Constraint: **neq** ≥ 1.

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

fcn must evaluate the first derivatives y_i' (i.e., the functions f_i) for given values of their arguments $x, y_1, y_2, \ldots, y_{\mathbf{neq}}$.

The specification of **fcn** is:

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

1: **neq** – Integer *Input*

On entry: the number of differential equations.

2: **x** – double *Input*

On entry: the value of the independent variable x.

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

On entry: $y[i-1]$ holds the value of the variable y_i, for $i = 1, 2, \ldots, \mathbf{neq}$.

4: **f**[**neq**] – double *Output*

On exit: $f[i-1]$ must contain the value of f_i, for $i = 1, 2, \ldots, \mathbf{neq}$.

5: **comm** – Nag_User *

Pointer to a structure of type Nag_User with the following member:

p – Pointer

On entry/exit: the pointer **comm**→**p** should be cast to the required type, e.g., `struct user *s = (struct user *)comm → p`, to obtain the original object's address with appropriate type. (See the argument **comm** below.)

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

pederv must evaluate the Jacobian of the system (that is, the partial derivatives $\frac{\partial f_i}{\partial y_j}$) for given values of the variables $x, y_1, y_2, \ldots, y_{\mathbf{neq}}$.

The specification of **pederv** is:

```
void pederv (Integer neq, double x, const double y[], double pw[],
     Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: the number of differential equations.

2: **x** – double *Input*

On entry: the value of the independent variable x.

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

On entry: $y[i-1]$ holds the value of the variable y_i, for $i = 1, 2, \ldots, \mathbf{neq}$.

4: **pw**[**neq** × **neq**] – double *Output*

On exit: **pw**$[(i-1) \times \mathbf{neq} + j - 1]$ must contain the value of $\frac{\partial f_i}{\partial y_j}$, for $i, j = 1, 2, \ldots, \mathbf{neq}$.

5: **comm** – Nag_User *

Pointer to a structure of type Nag_User with the following member:

p – Pointer

On entry/exit: the pointer **comm**→**p** should be cast to the required type, e.g., `struct user *s = (struct user *)comm → p`, to obtain the original object's address with appropriate type. (See the argument **comm** below.)

If you do not wish to supply the Jacobian, the actual argument **pederv** must be the NAG defined null function pointer `NULLFN`.

4: **x** – double * *Input/Output*

On entry: the value of the independent variable x.

Constraint: $\mathbf{x} \neq \mathbf{xend}$.

On exit: if g is supplied, **x** contains the point where $g(x, y) = 0.0$, unless $g(x, y) \neq 0.0$ anywhere on the range **x** to **xend**, in which case, **x** will contain **xend**. If g is not supplied **x** contains **xend**, unless an error has occurred, when it contains the value of x at the error.

5: **y**[**neq**] – double *Input/Output*

On entry: $y[i-1]$ holds the value of the variable y_i, for $i = 1, 2, \ldots, \mathbf{neq}$.

On exit: the computed values of the solution at the final point $x = \mathbf{x}$.

6: **xend** – double *Input*

On entry: the final value of the independent variable.

xend < **x**
Iintegration proceeds in the negative direction.

Constraint: $\mathbf{xend} \neq \mathbf{x}$.

7: **tol** – double *Input*

On entry: a **positive** tolerance for controlling the error in the integration. Hence **tol** affects the determination of the position where $g(x, y) = 0.0$, if g is supplied.

nag_ode_ivp_bdf_gen (d02ejc) has been designed so that, for most problems, a reduction in **tol** leads to an approximately proportional reduction in the error in the solution. However, the actual relation between **tol** and the accuracy achieved cannot be guaranteed. You are strongly recommended to call nag_ode_ivp_bdf_gen (d02ejc) with more than one value for **tol** and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, you might compare the results obtained by calling nag_ode_ivp_bdf_gen (d02ejc) with $\mathbf{tol} = 10^{-p}$ and $\mathbf{tol} = 10^{-p-1}$ if p correct decimal digits are required in the solution.

Constraint: $\mathbf{tol} > 0.0$.

8: **err_c** – Nag_ErrorControl *Input*

On entry: the type of error control. At each step in the numerical solution an estimate of the local error, *est*, is made. For the current step to be accepted the following condition must be satisfied:

$$est = \sqrt{\frac{1}{\mathbf{neq}}\sum_{i=1}^{\mathbf{neq}}(e_i/(\tau_r \times |y_i| + \tau_a))^2} \leq 1.0$$

where τ_r and τ_a are defined by

err_c	τ_r	τ_a
Nag_Relative	**tol**	ϵ
Nag_Absolute	0.0	**tol**
Nag_Mixed	**tol**	**tol**

where ϵ is a small machine-dependent number and e_i is an estimate of the local error at y_i, computed internally. If the appropriate condition is not satisfied, the step size is reduced and the solution is recomputed on the current step. If you wish to measure the error in the computed solution in terms of the number of correct decimal places, then **err_c** should be set to Nag_Absolute on entry, whereas if the error requirement is in terms of the number of correct significant digits, then **err_c** should be set to Nag_Relative. If you prefer a mixed error test, then **err_c** should be set to Nag_Mixed. The recommended value for **err_c** is Nag_Relative.

Constraint: **err_c** = Nag_Absolute, Nag_Mixed or Nag_Relative.

9: **output** – function, supplied by the user *External Function*

output permits access to intermediate values of the computed solution (for example to print or plot them), at successive user-specified points. It is initially called by nag_ode_ivp_bdf_gen (d02ejc) with **xsol** = **x** (the initial value of x). You must reset **xsol** to the next point (between the current **xsol** and **xend**) where **output** is to be called, and so on at each call to **output**. If, after a call to **output**, the reset point **xsol** is beyond **xend**, nag_ode_ivp_bdf_gen (d02ejc) will integrate to **xend** with no further calls to **output**; if a call to **output** is required at the point **xsol** = **xend**, then **xsol** must be given precisely the value **xend**.

The specification of **output** is:

```
void output (Integer neq, double *xsol, const double y[],
     Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: the number of differential equations.

2: **xsol** – double * *Input/Output*

On entry: the value of the independent variable x.

On exit: you must set **xsol** to the next value of x at which **output** is to be called.

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

On entry: $y[i-1]$ holds the value of the variable y_i, for $i = 1, 2, \ldots, \mathbf{neq}$.

4: **comm** – Nag_User *

Pointer to a structure of type Nag_User with the following member:

p – Pointer

On entry/exit: the pointer **comm→p** should be cast to the required type, e.g., `struct user *s = (struct user *)comm → p`, to obtain the original object's address with appropriate type. (See the argument **comm** below.)

If you do not wish to access intermediate output, the actual argument **output** must be the NAG defined null function pointer `NULLFN`.

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

g must evaluate $g(x,y)$ for specified values x, y. It specifies the function g for which the first position x where $g(x,y) = 0$ is to be found.

The specification of **g** is:

```
double g (Integer neq, double x, const double y[], Nag_User *comm)
```

1: **neq** – Integer *Input*

On entry: the number of differential equations.

2: **x** – double *Input*

On entry: the value of the independent variable x.

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

On entry: $y[i-1]$ holds the value of the variable y_i, for $i = 1, 2, \ldots, \mathbf{neq}$.

4: **comm** – Nag_User *

Pointer to a structure of type Nag_User with the following member:

p – Pointer

On entry/exit: the pointer **comm→p** should be cast to the required type, e.g., `struct user *s = (struct user *)comm → p`, to obtain the original object's address with appropriate type. (See the argument **comm** below.)

If you do not require the root finding option, the actual argument **g** must be the NAG defined null double function pointer `NULLDFN`.

11: **comm** – Nag_User *

Pointer to a structure of type Nag_User with the following member:

p – Pointer

On entry/exit: the pointer **comm→p**, of type Pointer, allows you to communicate information to and from **fcn**, **pederv**, **output** and **g**. An object of the required type should be declared, e.g., a structure, and its address assigned to the pointer **comm→p** by means of a cast to Pointer in the calling program, e.g., `comm.p = (Pointer)&s`. The type pointer will be `void *` with a C compiler that defines `void *` and `char *` otherwise.

12: **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_REAL_ARG_EQ

On entry, $\mathbf{x} = \langle value \rangle$ while $\mathbf{xend} = \langle value \rangle$. These arguments must satisfy $\mathbf{x} \neq \mathbf{xend}$.

NE_ALLOC_FAIL

Dynamic memory allocation failed.

NE_BAD_PARAM

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

NE_INT_ARG_LT

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

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_SIGN_CHANGE

No change in sign of the function $g(x, y)$ was detected in the integration range.

NE_REAL_ARG_LE

On entry, **tol** must not be less than or equal to 0.0: **tol** $= \langle value \rangle$.

NE_TOL_PROGRESS

The value of **tol**, $\langle value \rangle$, is too small for the function to make any further progress across the integration range. Current value of **x** $= \langle value \rangle$.

NE_TOL_TOO_SMALL

The value of **tol**, $\langle value \rangle$, is too small for the function to take an initial step.

NE_XSOL_INCONSIST

On call $\langle value \rangle$ to the supplied print function **xsol** was set to a value behind the previous value of **xsol** in the direction of integration.
Previous **xsol** $= \langle value \rangle$, **xend** $= \langle value \rangle$, new **xsol** $= \langle value \rangle$.

NE_XSOL_NOT_RESET

On call $\langle value \rangle$ to the supplied print function **xsol** was not reset.

NE_XSOL_SET_WRONG

xsol was set to a value behind **x** in the direction of integration by the first call to the supplied print function.
The integration range is $(\langle value \rangle, \langle value \rangle)$, **xsol** $= \langle value \rangle$.

7 Accuracy

The accuracy of the computation of the solution vector **y** may be controlled by varying the local error tolerance **tol**. In general, a decrease in local error tolerance should lead to an increase in accuracy. You are advised to choose **err_c** = Nag_Relative unless you have a good reason for a different choice. It is particularly appropriate if the solution decays.

If the problem is a root-finding one, then the accuracy of the root determined will depend strongly on $\frac{\partial g}{\partial x}$ and $\frac{\partial g}{\partial y_i}$, for $i = 1, 2, \ldots,$ **neq**. Large values for these quantities may imply large errors in the root.

8 Parallelism and Performance

nag_ode_ivp_bdf_gen (d02ejc) is not threaded in any implementation.

9 Further Comments

If more than one root is required, then to determine the second and later roots nag_ode_ivp_bdf_gen (d02ejc) may be called again starting a short distance past the previously determined roots.

If it is easy to code, you should supply the function **pederv**. However, it is important to be aware that if **pederv** is coded incorrectly, a very inefficient integration may result and possibly even a failure to complete the integration (**fail.code** = NE_TOL_PROGRESS).

10 Example

We illustrate the solution of five different problems. In each case the differential system is the well-known stiff Robertson problem.

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= 0.04y_1 - 10^4 y_2 y_3 - 3 \times 10^7 y_2^2 \\ y_3' &= 3 \times 10^7 y_2^2 \end{aligned}$$

with initial conditions $y_1 = 1.0$, $y_2 = y_3 = 0.0$ at $x = 0.0$. We solve each of the following problems with local error tolerances 1.0e−3 and 1.0e−4.

(i) To integrate to $x = 10.0$ producing output at intervals of 2.0 until a point is encountered where $y_1 = 0.9$. The Jacobian is calculated numerically.

(ii) As (i) but with the Jacobian calculated analytically.

(iii) As (i) but with no intermediate output.

(iv) As (i) but with no root-finding termination condition.

(v) Integrating the equations as in (i) but with no intermediate output and no root-finding termination condition.

10.1 Program Text

```
/* nag_ode_ivp_bdf_gen (d02ejc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 *
 */

#include <nag.h>
#include <math.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, const double y[],
                           double f[], Nag_User *comm);
  static void NAG_CALL pederv(Integer neq, double x, const double y[],
                              double pw[], Nag_User *comm);
  static double NAG_CALL g(Integer neq, double x, const double y[],
                           Nag_User *comm);
  static void NAG_CALL out(Integer neq, double *tsol, const double y[],
                           Nag_User *comm);
#ifdef __cplusplus
}
#endif

struct user
```

```
{
  double xend, h;
  Integer k;
  Integer *use_comm;
};

#define NEQ 3
int main(void)
{
  static Integer use_comm[4] = { 1, 1, 1, 1 };
  Integer exit_status = 0, j, neq;
  NagError fail;
  Nag_User comm;
  double tol, x, *y = 0;
  struct user s;

  INIT_FAIL(fail);

  printf("nag_ode_ivp_bdf_gen (d02ejc) Example Program Results\n");

  /* For communication with user-supplied functions
   * assign address of user defined structure
   * to comm.p.
   */
  s.use_comm = use_comm;
  comm.p = (Pointer) &s;

  neq = NEQ;
  if (neq >= 1) {
    if (!(y = NAG_ALLOC(neq, double)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
  }
  else {
    exit_status = 1;
    return exit_status;
  }
  s.xend = 10.0;
  printf("\nCase 1: calculating Jacobian internally\n");
  printf(" intermediate output, root-finding\n\n");

  for (j = 3; j <= 4; ++j) {
    tol = pow(10.0, -(double) j);
    printf("\n Calculation with tol = %10.1e\n", tol);
    x = 0.0;
    y[0] = 1.0;
    y[1] = 0.0;
    y[2] = 0.0;
    s.k = 4;
    s.h = (s.xend - x) / (double) (s.k + 1);
    printf("     X         Y(1)         Y(2)         Y(3)\n");
    /* nag_ode_ivp_bdf_gen (d02ejc).
     * Ordinary differential equations solver, stiff, initial
     * value problems using the Backward Differentiation
     * Formulae
     */
    nag_ode_ivp_bdf_gen(neq, fcn, NULLFN, &x, y, s.xend, tol, Nag_Relative,
                        out, g, &comm, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_ode_ivp_bdf_gen (d02ejc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
    printf("  Root of Y(1)-0.9 at %5.3f\n", x);
    printf("  Solution is ");
    printf("%7.4f %8.5f %7.4f\n", y[0], y[1], y[2]);
  }
  printf("\nCase 2: calculating Jacobian by pederv\n");
```

```
  printf(" intermediate output, root-finding\n\n");

  for (j = 3; j <= 4; ++j) {
    tol = pow(10.0, -(double) j);
    printf("\n Calculation with tol = %10.1e\n", tol);
    x = 0.0;
    y[0] = 1.0;
    y[1] = 0.0;
    y[2] = 0.0;
    s.k = 4;
    s.h = (s.xend - x) / (double) (s.k + 1);
    printf("     X         Y(1)         Y(2)         Y(3)\n");
    /* nag_ode_ivp_bdf_gen (d02ejc), see above. */
    nag_ode_ivp_bdf_gen(neq, fcn, pederv, &x, y, s.xend, tol, Nag_Relative,
                        out, g, &comm, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_ode_ivp_bdf_gen (d02ejc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
    printf("  Root of Y(1)-0.9 at %5.3f\n", x);
    printf("  Solution is ");
    printf("%7.4f %8.5f %7.4f\n", y[0], y[1], y[2]);
  }
  printf("\nCase 3: calculating Jacobian internally\n");
  printf(" no intermediate output, root-finding\n\n");
  for (j = 3; j <= 4; ++j) {
    tol = pow(10.0, -(double) j);
    printf("\n Calculation with tol = %10.1e\n", tol);
    x = 0.0;
    y[0] = 1.0;
    y[1] = 0.0;
    y[2] = 0.0;

    /* nag_ode_ivp_bdf_gen (d02ejc), see above. */
    nag_ode_ivp_bdf_gen(neq, fcn, NULLFN, &x, y, s.xend, tol, Nag_Relative,
                        NULLFN, g, &comm, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_ode_ivp_bdf_gen (d02ejc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

    printf("  Root of Y(1)-0.9 at %5.3f\n", x);
    printf("  Solution is ");
    printf("%7.4f %8.5f %7.4f\n", y[0], y[1], y[2]);
  }
  printf("\nCase 4: calculating Jacobian internally\n");
  printf(" intermediate output, no root-finding\n\n");

  for (j = 3; j <= 4; ++j) {
    tol = pow(10.0, -(double) j);
    printf("\n Calculation with tol = %10.1e\n", tol);
    x = 0.0;
    y[0] = 1.0;
    y[1] = 0.0;
    y[2] = 0.0;
    s.k = 4;
    s.h = (s.xend - x) / (double) (s.k + 1);
    printf("     X         Y(1)         Y(2)         Y(3)\n");
    /* nag_ode_ivp_bdf_gen (d02ejc), see above. */
    nag_ode_ivp_bdf_gen(neq, fcn, NULLFN, &x, y, s.xend, tol, Nag_Relative,
                        out, NULLDFN, &comm, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_ode_ivp_bdf_gen (d02ejc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
    printf("%8.2f", x);
    printf("%13.4f%13.5f%13.4f\n", y[0], y[1], y[2]);
  }
```

```
  printf("\nCase 5: calculating Jacobian internally\n");
  printf(" no intermediate output, no root-finding (integrate to xend)\n\n");

  for (j = 3; j <= 4; ++j) {
    tol = pow(10.0, -(double) j);
    printf("\n Calculation with tol = %10.1e\n", tol);
    x = 0.0;
    y[0] = 1.0;
    y[1] = 0.0;
    y[2] = 0.0;
    printf("     X         Y(1)         Y(2)         Y(3)\n");
    printf("%8.2f", x);
    printf("%13.4f%13.5f%13.4f\n", y[0], y[1], y[2]);
    /* nag_ode_ivp_bdf_gen (d02ejc), see above. */
    nag_ode_ivp_bdf_gen(neq, fcn, NULLFN, &x, y, s.xend, tol, Nag_Relative,
                        NULLFN, NULLDFN, &comm, &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_ode_ivp_bdf_gen (d02ejc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
    printf("%8.2f", x);
    printf("%13.4f%13.5f%13.4f\n", y[0], y[1], y[2]);
  }
END:
  NAG_FREE(y);
  return exit_status;
}

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

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

  f[0] = y[0] * -0.04 + y[1] * 1e4 * y[2];
  f[1] = y[0] * 0.04 - y[1] * 1e4 * y[2] - y[1] * 3e7 * y[1];
  f[2] = y[1] * 3e7 * y[1];
}

static void NAG_CALL pederv(Integer neq, double x, const double y[],
                            double pw[], Nag_User *comm)
{
#define PW(I, J) pw[((I) -1)*neq + (J) -1]
  struct user *s = (struct user *) comm->p;

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

  PW(1, 1) = -0.04;
  PW(1, 2) = y[2] * 1e4;
  PW(1, 3) = y[1] * 1e4;
  PW(2, 1) = 0.04;
  PW(2, 2) = y[2] * -1e4 - y[1] * 6e7;
  PW(2, 3) = y[1] * -1e4;
  PW(3, 1) = 0.0;
  PW(3, 2) = y[1] * 6e7;
  PW(3, 3) = 0.0;
}

static double NAG_CALL g(Integer neq, double x, const double y[],
                         Nag_User *comm)
{
  struct user *s = (struct user *) comm->p;
```

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

  return y[0] - 0.9;
}

static void NAG_CALL out(Integer neq, double *xsol, const double y[],
                         Nag_User *comm)
{
  struct user *s = (struct user *) comm->p;

  printf("%8.2f", *xsol);
  printf("%13.4f%13.5f%13.4f\n", y[0], y[1], y[2]);

  *xsol = s->xend - (double) s->k * s->h;
  s->k--;
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_ode_ivp_bdf_gen (d02ejc) Example Program Results

Case 1: calculating Jacobian internally
 intermediate output, root-finding


 Calculation with tol =    1.0e-03
     X         Y(1)         Y(2)         Y(3)
    0.00       1.0000      0.00000      0.0000
(User-supplied callback g, first invocation.)
(User-supplied callback fcn, first invocation.)
    2.00       0.9416      0.00003      0.0583
    4.00       0.9055      0.00002      0.0945
  Root of Y(1)-0.9 at 4.377
  Solution is  0.9000  0.00002  0.1000

 Calculation with tol =    1.0e-04
     X         Y(1)         Y(2)         Y(3)
    0.00       1.0000      0.00000      0.0000
    2.00       0.9416      0.00003      0.0584
    4.00       0.9055      0.00002      0.0945
  Root of Y(1)-0.9 at 4.377
  Solution is  0.9000  0.00002  0.1000

Case 2: calculating Jacobian by pederv
 intermediate output, root-finding


 Calculation with tol =    1.0e-03
     X         Y(1)         Y(2)         Y(3)
(User-supplied callback pederv, first invocation.)
    0.00       1.0000      0.00000      0.0000
    2.00       0.9416      0.00003      0.0583
    4.00       0.9055      0.00002      0.0945
  Root of Y(1)-0.9 at 4.377
  Solution is  0.9000  0.00002  0.1000

 Calculation with tol =    1.0e-04
     X         Y(1)         Y(2)         Y(3)
    0.00       1.0000      0.00000      0.0000
    2.00       0.9416      0.00003      0.0584
    4.00       0.9055      0.00002      0.0945
  Root of Y(1)-0.9 at 4.377
```

```
  Solution is  0.9000  0.00002  0.1000

Case 3: calculating Jacobian internally
 no intermediate output, root-finding


 Calculation with tol =    1.0e-03
  Root of Y(1)-0.9 at 4.377
  Solution is  0.9000  0.00002  0.1000

 Calculation with tol =    1.0e-04
  Root of Y(1)-0.9 at 4.377
  Solution is  0.9000  0.00002  0.1000

Case 4: calculating Jacobian internally
 intermediate output, no root-finding


 Calculation with tol =    1.0e-03
     X         Y(1)         Y(2)         Y(3)
    0.00       1.0000      0.00000      0.0000
    2.00       0.9416      0.00003      0.0583
    4.00       0.9055      0.00002      0.0945
    6.00       0.8793      0.00002      0.1207
    8.00       0.8586      0.00002      0.1414
   10.00       0.8414      0.00002      0.1586
   10.00       0.8414      0.00002      0.1586

 Calculation with tol =    1.0e-04
     X         Y(1)         Y(2)         Y(3)
    0.00       1.0000      0.00000      0.0000
    2.00       0.9416      0.00003      0.0584
    4.00       0.9055      0.00002      0.0945
    6.00       0.8793      0.00002      0.1207
    8.00       0.8585      0.00002      0.1414
   10.00       0.8414      0.00002      0.1586
   10.00       0.8414      0.00002      0.1586

Case 5: calculating Jacobian internally
 no intermediate output, no root-finding (integrate to xend)


 Calculation with tol =    1.0e-03
     X         Y(1)         Y(2)         Y(3)
    0.00       1.0000      0.00000      0.0000
   10.00       0.8414      0.00002      0.1586

 Calculation with tol =    1.0e-04
     X         Y(1)         Y(2)         Y(3)
    0.00       1.0000      0.00000      0.0000
   10.00       0.8414      0.00002      0.1586
```