

# NAG Library Routine Document

## D02AGF

**Note:** before using this routine, please read the Users' Note for your implementation to check the interpretation of *bold italicised* terms and other implementation-dependent details.

### 1 Purpose

D02AGF solves a two-point boundary value problem for a system of ordinary differential equations, using initial value techniques and Newton iteration; it generalizes D02HAF to include the case where parameters other than boundary values are to be determined.

### 2 Specification

```

SUBROUTINE D02AGF (H, E, PARERR, PARAM, C, N, N1, M1, AUX, BCAUX, RAAUX,      &
                  PRSOL, MAT, COPY, WSPACE, WSPAC1, WSPAC2, IFAIL)
INTEGER           N, N1, M1, IFAIL
REAL (KIND=nag_wp) H, E(N), PARERR(N1), PARAM(N1), C(M1,N),          &
                  MAT(N1,N1), COPY(1,1), WSPACE(N,9), WSPAC1(N),      &
                  WSPAC2(N)
EXTERNAL          AUX, BCAUX, RAAUX, PRSOL

```

### 3 Description

D02AGF solves a two-point boundary value problem by determining the unknown parameters  $p_1, p_2, \dots, p_{n_1}$  of the problem. These parameters may be, but need not be, boundary values (as they are in D02HAF); they may include eigenvalue parameters in the coefficients of the differential equations, length of the range of integration, etc. The notation and methods used are similar to those of D02HAF and you are advised to study this first. (There the parameters  $p_1, p_2, \dots, p_{n_1}$  correspond to the unknown boundary conditions.) It is assumed that we have a system of  $n$  first-order ordinary differential equations of the form

$$\frac{dy_i}{dx} = f_i(x, y_1, y_2, \dots, y_n), \quad i = 1, 2, \dots, n,$$

and that derivatives  $f_i$  are evaluated by AUX. The system, including the boundary conditions given by BCAUX, and the range of integration and matching point,  $r$ , given by RAAUX, involves the  $n_1$  unknown parameters  $p_1, p_2, \dots, p_{n_1}$  which are to be determined, and for which initial estimates must be supplied. The number of unknown parameters  $n_1$  must not exceed the number of equations  $n$ . If  $n_1 < n$ , we assume that  $(n - n_1)$  equations of the system are not involved in the matching process. These are usually referred to as ‘driving equations’; they are independent of the parameters and of the solutions of the other  $n_1$  equations. In numbering the equations for AUX, the driving equations must be put last.

The estimated values of the parameters are corrected by a form of Newton iteration. The Newton correction on each iteration is calculated using a matrix whose  $(i, j)$ th element depends on the derivative of the  $i$ th component of the solution,  $y_i$ , with respect to the  $j$ th parameter,  $p_j$ . This matrix is calculated by a simple numerical differentiation technique which requires  $n_1$  evaluations of the differential system.

### 4 References

None.

### 5 Arguments

You are strongly recommended to read Sections 3 and 9 in conjunction with this section.

- 1: H – REAL (KIND=nag\_wp) Input/Output  
*On entry:* H must be set to an estimate of the step size,  $h$ , needed for integration.  
*On exit:* the last step length used.
- 2: E(N) – REAL (KIND=nag\_wp) array Input  
*On entry:*  $E(i)$  must be set to a small quantity to control the  $i$ th solution component. The element  $E(i)$  is used:  
 (i) in the bound on the local error in the  $i$ th component of the solution  $y_i$  during integration,  
 (ii) in the convergence test on the  $i$ th component of the solution  $y_i$  at the matching point in the Newton iteration.  
 The elements  $E(i)$  should not be chosen too small. They should usually be several orders of magnitude larger than *machine precision*.
- 3: PARERR(N1) – REAL (KIND=nag\_wp) array Input  
*On entry:*  $PARERR(i)$  must be set to a small quantity to control the  $i$ th parameter component. The element  $PARERR(i)$  is used:  
 (i) in the convergence test on the  $i$ th parameter in the Newton iteration,  
 (ii) in perturbing the  $i$ th parameter when approximating the derivatives of the components of the solution with respect to the  $i$ th parameter, for use in the Newton iteration.  
 The elements  $PARERR(i)$  should not be chosen too small. They should usually be several orders of magnitude larger than *machine precision*.
- 4: PARAM(N1) – REAL (KIND=nag\_wp) array Input/Output  
*On entry:*  $PARAM(i)$  must be set to an estimate for the  $i$ th parameter,  $p_i$ , for  $i = 1, 2, \dots, N1$ .  
*On exit:* the corrected value for the  $i$ th parameter, unless an error has occurred, when it contains the last calculated value of the parameter (possibly perturbed by  $PARERR(i) \times (1 + |PARAM(i)|)$  if the error occurred when calculating the approximate derivatives).
- 5: C(M1,N) – REAL (KIND=nag\_wp) array Output  
*On exit:* the solution when  $M1 > 1$  (see M1).  
 If  $M1 = 1$ , the elements of C are not used.
- 6: N – INTEGER Input  
*On entry:*  $n$ , the total number of differential equations.
- 7: N1 – INTEGER Input  
*On entry:*  $n_1$ , the number of parameters.  
 If  $N1 < N$ , the last  $N - N1$  differential equations (in AUX) are driving equations (see Section 3).  
*Constraint:*  $N1 \leq N$ .
- 8: M1 – INTEGER Input  
*On entry:* determines whether or not the final solution is computed as well as the parameter values.  
 $M1 = 1$   
 The final solution is not calculated;

M1 > 1

The final values of the solution at interval (length of range)/(M1 – 1) are calculated and stored sequentially in the array C starting with the values of  $y_i$  evaluated at the first end point (see RAAUX) stored in C(1, i).

9: AUX – SUBROUTINE, supplied by the user. *External Procedure*

AUX must evaluate the functions  $f_i$  (i.e., the derivatives  $y_i'$ ) for given values of its arguments,  $x, y_1, \dots, y_n, p_1, \dots, p_{n_1}$ .

The specification of AUX is:

```
SUBROUTINE AUX ( F, Y, X, PARAM)
REAL (KIND=nag_wp) F(*), Y(*), X, PARAM(*)
```

In the description of the arguments of D02AGF below,  $n$  and  $n_1$  denote the numerical values of N and N1 in the call of D02AGF.

- |    |  |               |
|----|--|---------------|
| 1: | F(*) – REAL (KIND=nag_wp) array  | <i>Output</i> |
|    | <i>On exit:</i> the value of $f_i$ , for $i = 1, 2, \dots, n$ .                    |               |
| 2: | Y(*) – REAL (KIND=nag_wp) array  | <i>Input</i>  |
|    | <i>On entry:</i> $y_i$ , for $i = 1, 2, \dots, n$ , the value of the argument.     |               |
| 3: | X – REAL (KIND=nag_wp)   | <i>Input</i>  |
|    | <i>On entry:</i> $x$ , the value of the argument.                                  |               |
| 4: | PARAM(*) – REAL (KIND=nag_wp) array  | <i>Input</i>  |
|    | <i>On entry:</i> $p_i$ , for $i = 1, 2, \dots, n_1$ , the value of the parameters. |               |

AUX must either be a module subprogram USED by, or declared as EXTERNAL in, the (sub) program from which D02AGF is called. Arguments denoted as *Input* must **not** be changed by this procedure.

10: BCAUX – SUBROUTINE, supplied by the user. *External Procedure*

BCAUX must evaluate the values of  $y_i$  at the end points of the range given the values of  $p_1, \dots, p_{n_1}$ .

The specification of BCAUX is:

```
SUBROUTINE BCAUX (G0, G1, PARAM)
REAL (KIND=nag_wp) G0(*), G1(*), PARAM(*)
```

In the description of the arguments of D02AGF below,  $n$  and  $n_1$  denote the numerical values of N and N1 in the call of D02AGF.

- |    |  |               |
|----|--|---------------|
| 1: | G0(*) – REAL (KIND=nag_wp) array   | <i>Output</i> |
|    | <i>On exit:</i> the values $y_i$ , for $i = 1, 2, \dots, n$ , at the boundary point $x_0$ (see RAAUX). |               |
| 2: | G1(*) – REAL (KIND=nag_wp) array   | <i>Output</i> |
|    | <i>On exit:</i> the values $y_i$ , for $i = 1, 2, \dots, n$ , at the boundary point $x_1$ (see RAAUX). |               |
| 3: | PARAM(*) – REAL (KIND=nag_wp) array  | <i>Input</i>  |
|    | <i>On entry:</i> $p_i$ , for $i = 1, 2, \dots, n$ , the value of the parameters.                       |               |

BCAUX must either be a module subprogram USED by, or declared as EXTERNAL in, the (sub) program from which D02AGF is called. Arguments denoted as *Input* must **not** be changed by this procedure.

- 11: RAAUX – SUBROUTINE, supplied by the user. *External Procedure*

RAAUX must evaluate the end points,  $x_0$  and  $x_1$ , of the range and the matching point,  $r$ , given the values  $p_1, p_2, \dots, p_{n_1}$ .

The specification of RAAUX is:

```
SUBROUTINE RAAUX (X0, X1, R, PARAM)
REAL (KIND=nag_wp) X0, X1, R, PARAM(*)
```

In the description of the arguments of D02AGF below,  $n_1$  denotes the numerical value of N1 in the call of D02AGF.

- |    |  |               |
|----|--|---------------|
| 1: | X0 – REAL (KIND=nag_wp)  | <i>Output</i> |
|    | <i>On exit:</i> must contain the left-hand end of the range, $x_0$ .               |               |
| 2: | X1 – REAL (KIND=nag_wp)  | <i>Output</i> |
|    | <i>On exit:</i> must contain the right-hand end of the range $x_1$ .               |               |
| 3: | R – REAL (KIND=nag_wp)   | <i>Output</i> |
|    | <i>On exit:</i> must contain the matching point, $r$ .                             |               |
| 4: | PARAM(*) – REAL (KIND=nag_wp) array  | <i>Input</i>  |
|    | <i>On entry:</i> $p_i$ , for $i = 1, 2, \dots, n_1$ , the value of the parameters. |               |

RAAUX must either be a module subprogram USED by, or declared as EXTERNAL in, the (sub) program from which D02AGF is called. Arguments denoted as *Input* must **not** be changed by this procedure.

- 12: PRSOL – SUBROUTINE, supplied by the user. *External Procedure*

PRSOL is called at each iteration of the Newton method and can be used to print the current values of the parameters  $p_i$ , for  $i = 1, 2, \dots, n_1$ , their errors,  $e_i$ , and the sum of squares of the errors at the matching point,  $r$ .

The specification of PRSOL is:

```
SUBROUTINE PRSOL (PARAM, RES, N1, ERR)
INTEGER N1
REAL (KIND=nag_wp) PARAM(N1), RES, ERR(N1)
```

- |    |  |              |
|----|--|--------------|
| 1: | PARAM(N1) – REAL (KIND=nag_wp) array   | <i>Input</i> |
|    | <i>On entry:</i> $p_i$ , for $i = 1, 2, \dots, n_1$ , the current value of the parameters.     |              |
| 2: | RES – REAL (KIND=nag_wp)   | <i>Input</i> |
|    | <i>On entry:</i> the sum of squares of the errors in the arguments, $\sum_{i=1}^{n_1} e_i^2$ . |              |
| 3: | N1 – INTEGER   | <i>Input</i> |
|    | <i>On entry:</i> $n_1$ , the number of parameters.   |              |

4:    ERR(N1) – REAL (KIND=nag_wp) array <i>On entry:</i> the errors in the parameters, $e_i$ , for $i = 1, 2, \dots, n_1$ .	<i>Input</i>
---	--------------

PRSOL must either be a module subprogram USED by, or declared as EXTERNAL in, the (sub) program from which D02AGF is called. Arguments denoted as *Input* must **not** be changed by this procedure.

13: MAT(N1,N1) – REAL (KIND=nag_wp) array	<i>Workspace</i>
14: COPY(1,1) – REAL (KIND=nag_wp) array	<i>Input</i>
15: WSPACE(N,9) – REAL (KIND=nag_wp) array	<i>Workspace</i>
16: WSPAC1(N) – REAL (KIND=nag_wp) array	<i>Workspace</i>
17: WSPAC2(N) – REAL (KIND=nag_wp) array	<i>Workspace</i>
18: IFAIL – INTEGER	<i>Input/Output</i>

*On entry:* IFAIL must be set to 0,  $-1$  or 1. If you are unfamiliar with this argument you should refer to Section 3.4 in How to Use the NAG Library and its Documentation for details.

For environments where it might be inappropriate to halt program execution when an error is detected, the value  $-1$  or 1 is recommended. If the output of error messages is undesirable, then the value 1 is recommended. Otherwise, if you are not familiar with this argument, the recommended value is 0. **When the value  $-1$  or 1 is used it is essential to test the value of IFAIL on exit.**

*On exit:* IFAIL = 0 unless the routine detects an error or a warning has been flagged (see Section 6).

## 6 Error Indicators and Warnings

If on entry IFAIL = 0 or  $-1$ , explanatory error messages are output on the current error message unit (as defined by X04AAF).

Errors or warnings detected by the routine:

IFAIL = 1

This indicates that  $N1 > N$  on entry, that is the number of parameters is greater than the number of differential equations.

IFAIL = 2

As for IFAIL = 4 except that the integration failed while calculating the matrix for use in the Newton iteration.

IFAIL = 3

The current matching point  $r$  does not lie between the current end points  $x_0$  and  $x_1$ . If the values  $x_0$ ,  $x_1$  and  $r$  depend on the parameters  $p_i$ , this may occur at any time in the Newton iteration if care is not taken to avoid it when coding RAAUX.

IFAIL = 4

The step length for integration H has halved more than 13 times (or too many steps were needed to reach the end of the range of integration) in attempting to control the local truncation error whilst integrating to obtain the solution corresponding to the current values  $p_i$ . If, on failure, H has the sign of  $r - x_0$  then failure has occurred whilst integrating from  $x_0$  to  $r$ , otherwise it has occurred whilst integrating from  $x_1$  to  $r$ .

IFAIL = 5

The matrix of the equations to be solved for corrections to the variable parameters in the Newton method is singular (as determined by F07ADF (DGETRF)).

IFAIL = 6

A satisfactory correction to the parameters was not obtained on the last Newton iteration employed. A Newton iteration is deemed to be unsatisfactory if the sum of the squares of the residuals (which can be printed using PRSOL) has not been reduced after three iterations using a new Newton correction.

IFAIL = 7

Convergence has not been obtained after 12 satisfactory iterations of the Newton method.

IFAIL = -99

An unexpected error has been triggered by this routine. Please contact NAG.

See Section 3.9 in How to Use the NAG Library and its Documentation for further information.

IFAIL = -399

Your licence key may have expired or may not have been installed correctly.

See Section 3.8 in How to Use the NAG Library and its Documentation for further information.

IFAIL = -999

Dynamic memory allocation failed.

See Section 3.7 in How to Use the NAG Library and its Documentation for further information.

A further discussion of these errors and the steps which might be taken to correct them is given in Section 9.

## 7 Accuracy

If the process converges, the accuracy to which the unknown parameters are determined is usually close to that specified by you; and the solution, if requested, is usually determined to the accuracy specified.

## 8 Parallelism and Performance

D02AGF is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

D02AGF 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 routine. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

## 9 Further Comments

The time taken by D02AGF depends on the complexity of the system, and on the number of iterations required. In practice, integration of the differential equations is by far the most costly process involved.

There may be particular difficulty in integrating the differential equations in one direction (indicated by IFAIL = 2 or 4). The value of  $r$  should be adjusted to avoid such difficulties.

If the matching point  $r$  is at one of the end points  $x_0$  or  $x_1$  and some of the parameters are used **only** to determine the boundary values at this point, then good initial estimates for these parameters are not required, since they are completely determined by the routine (for example, see  $p_2$  in EX1 of Section 10).

Wherever they occur in the procedure, the error parameters contained in the arrays E and PARERR are used in 'mixed' form; that is  $E(i)$  always occurs in expressions of the form  $E(i) \times (1 + |y_i|)$ , and

PARERR( $i$ ) always occurs in expressions of the form PARERR( $i$ )  $\times$  (1 +  $|p_i|$ ). Though not ideal for every application, it is expected that this mixture of absolute and relative error testing will be adequate for most purposes.

Note that **convergence is not guaranteed**. You are strongly advised to provide an output PRSOL, as shown in EX1 of Section 10, in order to monitor the progress of the iteration. Failure of the Newton iteration to converge (see IFAIL = 6 or 7) usually results from poor starting approximations to the parameters, though occasionally such failures occur because the elements of one or both of the arrays PARERR or E are too small. (It should be possible to distinguish these cases by studying the output from PRSOL.) Poor starting approximations can also result in the failure described under IFAIL = 4 and 5 in Section 6 (especially if these errors occur after some Newton iterations have been completed, that is, after two or more calls of PRSOL). More frequently, a singular matrix in the Newton method (monitored as IFAIL = 5) occurs because the mathematical problem has been posed incorrectly. The case IFAIL = 4 usually occurs because  $h$  or  $r$  has been poorly estimated, so these values should be checked first. If IFAIL = 2 is monitored, the solution  $y_1, y_2, \dots, y_n$  is sensitive to perturbations in the parameters  $p_i$ . Reduce the size of one or more values PARERR( $i$ ) to reduce the perturbations. Since only one value  $p_i$  is perturbed at any time when forming the matrix, the perturbation which is too large can be located by studying the final output from PRSOL and the values of the parameters returned by D02AGF. If this change leads to other types of failure improve the initial values of  $p_i$  by other means.

The computing time for integrating the differential equations can sometimes depend critically on the quality of the initial estimates for the parameters  $p_i$ . If it seems that too much computing time is required and, in particular, if the values ERR( $i$ ) (available on each call of PRSOL) are much larger than the expected values of the solution at the matching point  $r$ , then the coding of AUX, BCAUX and RAAUX should be checked for errors. If no errors can be found, an independent attempt should be made to improve the initial estimates for PARAM( $i$ ).

The subroutine can be used to solve a very wide range of problems, for example:

- (a) eigenvalue problems, including problems where the eigenvalue occurs in the boundary conditions;
- (b) problems where the differential equations depend on some parameters which are to be determined so as to satisfy certain boundary conditions (see EX1 in Section 10);
- (c) problems where one of the end points of the range of integration is to be determined as the point where a variable  $y_i$  takes a particular value (see EX2 in Section 10);
- (d) singular problems and problems on infinite ranges of integration where the values of the solution at  $x_0$  or  $x_1$  or both are determined by a power series or an asymptotic expansion (or a more complicated expression) and where some of the coefficients in the expression are to be determined (see EX1 in Section 10); and
- (e) differential equations with certain terms defined by other independent (driving) differential equations.

## 10 Example

For this routine two examples are presented. There is a single example program for D02AGF, with a main program and the code to solve the two example problems given in Example 1 (EX1) and Example 2 (EX2).

### Example 1 (EX1)

This example finds the solution of the differential equation

$$y'' = \frac{y^3 - y'}{2x}$$

on the range  $0 \leq x \leq 16$ , with boundary conditions  $y(0) = 0.1$  and  $y(16) = 1/6$ .

We cannot use the differential equation at  $x = 0$  because it is singular, so we take the truncated series expansion

$$y(x) = \frac{1}{10} + p_1 \frac{\sqrt{x}}{10} + \frac{x}{100}$$

near the origin (which is correct to the number of terms given in this case). Here  $p_1$  is one of the parameters to be determined. We choose the range as  $[0.1, 16]$  and setting  $p_2 = y'(16)$ , we can determine all the boundary conditions. We take the matching point to be 16, the end of the range, and so a good initial guess for  $p_2$  is not necessary. We write  $y = Y(1)$ ,  $y' = Y(2)$ , and estimate  $p_1 = \text{PARAM}(1) = 0.2$ ,  $p_2 = \text{PARAM}(2) = 0.0$ .

### Example 2 (EX2)

This example finds the gravitational constant  $p_1$  and the range  $p_2$  over which a projectile must be fired to hit the target with a given velocity. The differential equations are

$$y' = \tan \phi$$

$$v' = \frac{-(p_1 \sin \phi + 0.00002v^2)}{v \cos \phi}$$

$$\phi' = \frac{-p_1 k}{v^2}$$

on the range  $0 < x < p_2$  with boundary conditions

$$y = 0, \quad v = 500, \quad \phi = 0.5 \quad \text{at} \quad x = 0$$

$$y = 0, \quad v = 450, \quad \phi = p_3 \quad \text{at} \quad x = p_2.$$

We write  $y = Y(1)$ ,  $v = Y(2)$ ,  $\phi = Y(3)$ , and we take the matching point  $r = p_2$ . We estimate  $p_1 = \text{PARAM}(1) = 32$ ,  $p_2 = \text{PARAM}(2) = 6000$  and  $p_3 = \text{PARAM}(3) = 0.54$  (though this estimate is not important).

## 10.1 Program Text

```
! D02AGF Example Program Text
! Mark 26 Release. NAG Copyright 2016.

Module d02agfe_mod

! D02AGF Example Program Module:
! Parameters and User-defined Routines

! iprint: set iprint = 1 for output at each Newton iteration.
! nin: the input channel number
! nout: the output channel number

! For Example 1:
! n_ex1 : number of differential equations,
! nl_ex1: number of parameters.

! For Example 2:
! n_ex2 : number of differential equations,
! nl_ex2: number of parameters.

! .. Use Statements ..
Use nag_library, Only: nag_wp
! .. Implicit None Statement ..
Implicit None
! .. Accessibility Statements ..
Private
Public :: aux1, aux2, bcaux1, bcaux2, prsol1, &
prsol2, raaux1, raaux2

! .. Parameters ..
Integer, Parameter :: iprint = 0
Integer, Parameter, Public :: nl_ex1 = 2, nl_ex2 = 3, nin = 5, &
nout = 6, n_ex1 = 2, n_ex2 = 3

Contains
Subroutine aux1(f,y,x,param)

! .. Scalar Arguments ..
Real (Kind=nag_wp), Intent (In) :: x
! .. Array Arguments ..
Real (Kind=nag_wp), Intent (Out) :: f(*)
```



```

      Real (Kind=nag_wp), Intent (In) :: param(*), y(*)
!      .. Executable Statements ..
      f(1) = y(2)
      f(2) = (y(1)**3-y(2))/(2.0_nag_wp*x)
      Return
End Subroutine aux1
Subroutine raaux1(x0,x1,r,param)

!      .. Scalar Arguments ..
      Real (Kind=nag_wp), Intent (Out) :: r, x0, x1
!      .. Array Arguments ..
      Real (Kind=nag_wp), Intent (In) :: param(*)
!      .. Executable Statements ..
      x0 = 0.1_nag_wp
      x1 = 16.0_nag_wp
      r = 16.0_nag_wp
      Return
End Subroutine raaux1
Subroutine bcaux1(g0,g1,param)

!      .. Array Arguments ..
      Real (Kind=nag_wp), Intent (Out) :: g0(*), g1(*)
      Real (Kind=nag_wp), Intent (In) :: param(*)
!      .. Local Scalars ..
      Real (Kind=nag_wp)          :: z
!      .. Intrinsic Procedures ..
      Intrinsic                   :: sqrt
!      .. Executable Statements ..
      z = 0.1_nag_wp
      g0(1) = 0.1_nag_wp + param(1)*sqrt(z)*0.1_nag_wp + 0.01_nag_wp*z
      g0(2) = param(1)*0.05_nag_wp/sqrt(z) + 0.01_nag_wp
      g1(1) = 1.0_nag_wp/6.0_nag_wp
      g1(2) = param(2)
      Return
End Subroutine bcaux1
Subroutine prsoll(param,res,nl,err)

!      .. Scalar Arguments ..
      Real (Kind=nag_wp), Intent (In) :: res
      Integer, Intent (In)          :: nl
!      .. Array Arguments ..
      Real (Kind=nag_wp), Intent (In) :: err(nl), param(nl)
!      .. Local Scalars ..
      Integer                          :: i
!      .. Executable Statements ..
      If (iprint/=0) Then
        Write (nout,99999) 'Current parameters ', (param(i),i=1,nl)
        Write (nout,99998) 'Residuals ', (err(i),i=1,nl)
        Write (nout,99998) 'Sum of residuals squared ', res
        Write (nout,*)
      End If
      Return

99999  Format (1X,A,6(E14.6,2X))
99998  Format (1X,A,6(E12.4,1X))
End Subroutine prsoll
Subroutine aux2(f,y,x,param)

!      .. Parameters ..
      Real (Kind=nag_wp), Parameter :: eps = 2.0E-5_nag_wp
!      .. Scalar Arguments ..
      Real (Kind=nag_wp), Intent (In) :: x
!      .. Array Arguments ..
      Real (Kind=nag_wp), Intent (Out) :: f(*)
      Real (Kind=nag_wp), Intent (In) :: param(*), y(*)
!      .. Local Scalars ..
      Real (Kind=nag_wp)          :: c, s
!      .. Intrinsic Procedures ..
      Intrinsic                   :: cos, sin
!      .. Executable Statements ..
      c = cos(y(3))

```

```

    s = sin(y(3))
    f(1) = s/c
    f(2) = -(param(1)*s+eps*y(2)*y(2))/(y(2)*c)
    f(3) = -param(1)/(y(2)*y(2))
    Return
End Subroutine aux2
Subroutine raaux2(x0,x1,r,param)

!     .. Scalar Arguments ..
!     Real (Kind=nag_wp), Intent (Out) :: r, x0, x1
!     .. Array Arguments ..
!     Real (Kind=nag_wp), Intent (In) :: param(*)
!     .. Executable Statements ..
    x0 = 0.0_nag_wp
    x1 = param(2)
    r = param(2)
    Return
End Subroutine raaux2
Subroutine bcaux2(g0,g1,param)

!     .. Array Arguments ..
!     Real (Kind=nag_wp), Intent (Out) :: g0(*), g1(*)
!     Real (Kind=nag_wp), Intent (In) :: param(*)
!     .. Executable Statements ..
    g0(1) = 0.0E0_nag_wp
    g0(2) = 500.0E0_nag_wp
    g0(3) = 0.5E0_nag_wp
    g1(1) = 0.0E0_nag_wp
    g1(2) = 450.0E0_nag_wp
    g1(3) = param(3)
    Return
End Subroutine bcaux2
Subroutine prsol2(param,res,n1,err)

!     .. Scalar Arguments ..
!     Real (Kind=nag_wp), Intent (In) :: res
!     Integer, Intent (In)          :: n1
!     .. Array Arguments ..
!     Real (Kind=nag_wp), Intent (In) :: err(n1), param(n1)
!     .. Local Scalars ..
!     Integer                          :: i
!     .. Executable Statements ..
    If (iprint/=0) Then
        Write (nout,99999) 'Current parameters ', (param(i),i=1,n1)
        Write (nout,99998) 'Residuals ', (err(i),i=1,n1)
        Write (nout,99998) 'Sum of residuals squared ', res
        Write (nout,*)
    End If
    Return

99999  Format (1X,A,6(E14.6,2X))
99998  Format (1X,A,6(E12.4,1X))
End Subroutine prsol2
End Module d02agfe_mod
Program d02agfe

!     D02AGF Example Main Program

!     .. Use Statements ..
    Use d02agfe_mod, Only: nout
!     .. Implicit None Statement ..
    Implicit None
!     .. Executable Statements ..
    Write (nout,*) 'D02AGF Example Program Results'

    Call ex1

    Call ex2

Contains
    Subroutine ex1

```

```

! .. Use Statements ..
Use nag_library, Only: d02agf, nag_wp
Use d02agfe_mod, Only: aux1, bcaux1, n1_ex1, nin, n_ex1, prsol1,      &
    raux1
! .. Local Scalars ..
Real (Kind=nag_wp)          :: h, r, soler, x, x1, xx
Integer                    :: i, ifail, j, m1
! .. Local Arrays ..
Real (Kind=nag_wp), Allocatable :: c(:,,:), e(:), mat(:,,:), param(:),      &
    parerr(:), wspac1(:), wspac2(:),      &
    wspace(:,:)
Real (Kind=nag_wp)          :: dummy(1,1)
! .. Intrinsic Procedures ..
Intrinsic                  :: real
! .. Executable Statements ..
! Skip heading in data file
! Read (nin,*)
! m1: final solution calculated at m1 points in range including
! end points.
! Read (nin,*) m1
! Allocate (c(m1,n_ex1),e(n_ex1),mat(n_ex1,n_ex1),param(n_ex1),      &
!   parerr(n_ex1),wspac1(n_ex1),wspac2(n_ex1),wspace(n_ex1,9))
! h: step size estimate,
! param: initial parameter estimates,
! parerr: Newton iteration tolerances,
! soler: bound on the local error.
! Read (nin,*) h
! Read (nin,*) param(1:n1_ex1)
! Read (nin,*) parerr(1:n1_ex1)
! Read (nin,*) soler
! e(1:n_ex1) = soler
! Write (nout,*)
! Write (nout,*)
! Write (nout,*) 'Case 1'
! Write (nout,*)
!
! ifail: behaviour on error exit
! =0 for hard exit, =1 for quiet-soft, =-1 for noisy-soft
! ifail = 0
! Call d02agf(h,e,parerr,param,c,n_ex1,n1_ex1,m1,aux1,bcaux1,raux1,      &
!   prsol1,mat,dummy,wspace,wspac1,wspac2,ifail)
!
! Write (nout,*) 'Final parameters'
! Write (nout,99999)(param(i),i=1,n1_ex1)
! Write (nout,*)
! Write (nout,*) 'Final solution'
! Write (nout,*) 'X-value      Components of solution'
! Call raux1(x,x1,r,param)
! h = (x1-x)/real(m1-1,kind=nag_wp)
! xx = x
! Do i = 1, m1
!   Write (nout,99998) xx, (c(i,j),j=1,n_ex1)
!   xx = xx + h
! End Do
!
! Return
!
99999  Format (1X,3E16.6)
99998  Format (1X,F7.2,3E13.4)
End Subroutine ex1
Subroutine ex2
! .. Use Statements ..
Use nag_library, Only: d02agf, nag_wp
Use d02agfe_mod, Only: aux2, bcaux2, n1_ex2, nin, n_ex2, prsol2,      &
    raux2
! .. Local Scalars ..
Real (Kind=nag_wp)          :: h, r, soler, x, x1, xx
Integer                    :: i, ifail, j, m1
! .. Local Arrays ..

```

```

      Real (Kind=nag_wp), Allocatable :: c(:,,:), e(:), mat(:,,:), param(:), &
                                         parerr(:), wspace1(:), wspace2(:), &
                                         wspace(:,:)
      Real (Kind=nag_wp)                :: dummy(1,1)
!
! .. Executable Statements ..
      Read (nin,*)
!
! Read in problem parameters
!
! m1: final solution calculated at m1 points in range including
!
! end points.
      Read (nin,*) m1
      Allocate (c(m1,n_ex2),e(n_ex2),mat(n_ex2,n_ex2),param(n_ex2), &
               parerr(n_ex2),wspace1(n_ex2),wspace2(n_ex2),wspace(n_ex2,9))
      Write (nout,*)
      Write (nout,*)
      Write (nout,*) 'Case 2'
      Write (nout,*)
!
! h: step size estimate,
!
! param: initial parameter estimates,
!
! parerr: Newton iteration tolerances,
!
! soler: bound on the local error.
      Read (nin,*) h
      Read (nin,*) param(1:n1_ex2)
      Read (nin,*) parerr(1:n1_ex2)
      Read (nin,*) soler
      e(1:n_ex2) = soler

!
! ifail: behaviour on error exit
!
!       =0 for hard exit, =1 for quiet-soft, =-1 for noisy-soft
      ifail = 0
      Call d02agf(h,e,parerr,param,c,n_ex2,n1_ex2,m1,aux2,bcaux2,raaux2, &
                 prsol2,mat,dummy,wspace,wspace1,wspace2,ifail)

      Write (nout,*) 'Final parameters'
      Write (nout,99999)(param(i),i=1,n_ex2)
      Write (nout,*)
      Write (nout,*) 'Final solution'
      Write (nout,*) 'X-value      Components of solution'
      Call raaux2(x,x1,r,param)
      h = (x1-x)/5.0E0_nag_wp
      xx = x
      Do i = 1, 6
         Write (nout,99998) xx, (c(i,j),j=1,n_ex2)
         xx = xx + h
      End Do

      Return

99999  Format (1X,3E16.6)
99998  Format (1X,F7.0,3E13.4)
      End Subroutine ex2
      End Program d02agfe

```

## 10.2 Program Data

D02AGF Example Program Data

```

6           : m1
0.1        : h
0.2 0.0    : param
1.0E-5  1.0E-3 : parer
1.0E-4    : soler

6           : m1, n, n1
1.0E1     : h
3.2E1 6.0E3 5.4E-1 : param
1.0E-5 1.0E-4 1.0E-4 : parer
1.0E-2   : soler

```

### 10.3 Program Results

D02AGF Example Program Results

Case 1

Final parameters

0.464269E-01 0.349429E-02

Final solution

X-value	Components of solution	
0.10	0.1025E+00	0.1734E-01
3.28	0.1217E+00	0.4180E-02
6.46	0.1338E+00	0.3576E-02
9.64	0.1449E+00	0.3418E-02
12.82	0.1557E+00	0.3414E-02
16.00	0.1667E+00	0.3494E-02

Case 2

Final parameters

0.323729E+02 0.596317E+04 -0.535231E+00

Final solution

X-value	Components of solution		
0.	0.0000E+00	0.5000E+03	0.5000E+00
1193.	0.5298E+03	0.4516E+03	0.3281E+00
2385.	0.8076E+03	0.4203E+03	0.1231E+00
3578.	0.8208E+03	0.4094E+03	-0.1032E+00
4771.	0.5563E+03	0.4200E+03	-0.3296E+00
5963.	0.0000E+00	0.4500E+03	-0.5352E+00



