NAG Library Routine Document

D02HAF

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

D02HAF solves a two-point boundary value problem for a system of ordinary differential equations, using a Runge–Kutta–Merson method and a Newton iteration in a shooting and matching technique.

2 Specification

```
SUBROUTINE D02HAF (U, V, N, A, B, TOL, FCN, SOLN, M1, W, SDW, IFAIL)

INTEGER              N, M1, SDW, IFAIL
REAL (KIND=nag_wp) U(N,2), V(N,2), A, B, TOL, SOLN(N,M1), W(N,SDW)
EXTERNAL             FCN
```

3 Description

D02HAF solves a two-point boundary value problem for a system of n ordinary differential equations in the range a, b. 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. Initially, n boundary values of the variables y_i must be specified, some at a and some at b. You must supply estimates of the remaining n boundary values (called parameters below); the subroutine corrects these by a form of Newton iteration. It also calculates the complete solution on an equispaced mesh if required.

Starting from the known and estimated values of y_i at a, the subroutine integrates the equations from a to b (using a Runge–Kutta–Merson method). The differences between the values of y_i at b from integration and those specified initially should be zero for the true solution. (These differences are called residuals below.) The subroutine uses a generalized Newton method to reduce the residuals to zero, by calculating corrections to the estimated boundary values. This process is repeated iteratively until convergence is obtained, or until the routine can no longer reduce the residuals. See Hall and Watt (1976) for a simple discussion of shooting and matching techniques.

4 References

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

5 Arguments

1: U(N, 2) – REAL (KIND=nag_wp) array *Input/Output*

On entry: U(i, 1) must be set to the known or estimated value of y_i at a and U(i, 2) must be set to the known or estimated value of y_i at b, for $i = 1, 2, \ldots, n$.

On exit: the known values unaltered, and corrected values of the estimates, unless an error has occurred. If an error has occurred, U contains the known values and the latest values of the estimates.

2: V(N, 2) – REAL (KIND=nag_wp) array *Input*

On entry: $\mathrm{V}(i,j)$ must be set to 0.0 if $\mathrm{U}(i,j)$ is a known value and to 1.0 if $\mathrm{U}(i,j)$ is an estimated value, for $i = 1,2,\ldots,n$ and $j = 1,2$.

Constraint: precisely n of the $\mathrm{V}(i,j)$ must be set to 0.0, i.e., precisely n of the $\mathrm{U}(i,j)$ must be known values, and these must not be all at a or all at b.

3: N – INTEGER *Input*

On entry: n, the number of equations.

Constraint: $\mathrm{N} \geq 1$.

4: A – REAL (KIND=nag_wp) *Input*

On entry: a, the initial point of the interval of integration.

5: B – REAL (KIND=nag_wp) *Input*

On entry: b, the final point of the interval of integration.

6: TOL – REAL (KIND=nag_wp) *Input*

On entry: must be set to a small quantity suitable for:

(a) testing the local error in y_i during integration,

(b) testing for the convergence of y_i at b,

(c) calculating the perturbation in estimated boundary values for y_i, which are used to obtain the approximate derivatives of the residuals for use in the Newton iteration.

You are advised to check your results by varying TOL.

Constraint: $\mathrm{TOL} > 0.0$.

7: FCN – SUBROUTINE, supplied by the user. *External Procedure*

FCN must evaluate the functions f_i (i.e., the derivatives y_i'), for $i = 1,2,\ldots,n$, at a general point x.

The specification of FCN is:

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

In the description of the arguments of D02HAF below, n denotes the actual value of N in the call of D02HAF.

1: X – REAL (KIND=nag_wp) *Input*

On entry: x, the value of the argument.

2: Y($*$) – REAL (KIND=nag_wp) array *Input*

On entry: y_i, for $i = 1,2,\ldots,n$, the value of the argument.

3: F($*$) – REAL (KIND=nag_wp) array *Output*

On exit: the values of $f_i(x)$, for $i = 1,2,\ldots,n$.

FCN must either be a module subprogram USEd by, or declared as EXTERNAL in, the (sub) program from which D02HAF is called. Arguments denoted as *Input* must **not** be changed by this procedure.

8: SOLN(N, M1) – REAL (KIND=nag_wp) array *Output*

On exit: the solution when $\mathrm{M1} > 1$.

9: M1 – INTEGER *Input*

On entry: a value which controls output.

$\mathrm{M1} = 1$
The final solution is not evaluated.

$\mathrm{M1} > 1$
The final values of y_i at interval $(b - a)/(\mathrm{M1} - 1)$ are calculated and stored in the array SOLN by columns, starting with values y_i at a stored in $\mathrm{SOLN}(i, 1)$, for $i = 1, 2, \ldots, n$.

Constraint: $\mathrm{M1} \geq 1$.

10: W(N, SDW) – REAL (KIND=nag_wp) array *Output*

On exit: if IFAIL $= 2$, 3, 4 or 5, $\mathrm{W}(i, 1)$, for $i = 1, 2, \ldots, n$, contains the solution at the point where the integration fails and the point of failure is returned in $\mathrm{W}(1, 2)$.

11: SDW – INTEGER *Input*

On entry: the second dimension of the array W as declared in the (sub)program from which D02HAF is called.

Constraint: $\mathrm{SDW} \geq 3\mathrm{N} + 17 + \max(11, \mathrm{N})$.

12: IFAIL – INTEGER *Input/Output*

For this routine, the normal use of IFAIL is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see Section 3.4 in How to Use the NAG Library and its Documentation).

On entry: IFAIL must be set to a value with the decimal expansion cba, where each of the decimal digits c, b and a must have a value of 0 or 1.

$a = 0$ specifies hard failure, otherwise soft failure;

$b = 0$ suppresses error messages, otherwise error messages will be printed (see Section 6);

$c = 0$ suppresses warning messages, otherwise warning messages will be printed (see Section 6).

The recommended value for inexperienced users is 110 (i.e., hard failure with all messages printed).

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 $= 1$

One or more of the arguments V, N, M1, SDW, or TOL is incorrectly set.

IFAIL $= 2$

The step length for the integration is too short whilst calculating the residual (see Section 9).

IFAIL $= 3$

No initial step length could be chosen for the integration whilst calculating the residual.

Note: IFAIL $= 2$ or 3 can occur due to choosing too small a value for TOL or due to choosing the wrong direction of integration. Try varying TOL and interchanging a and b. These error exits can also occur for very poor initial estimates of the unknown initial values and, in extreme cases, because D02HAF cannot be used to solve the problem posed.

IFAIL $= 4$

As for IFAIL $= 2$ but the error occurred when calculating the Jacobian of the derivatives of the residuals with respect to the parameters.

IFAIL $= 5$

As for IFAIL $= 3$ but the error occurred when calculating the derivatives of the residuals with respect to the parameters.

IFAIL $= 6$

The calculated Jacobian has an insignificant column.

Note: IFAIL $= 4$, 5 or 6 usually indicate a badly scaled problem. You may vary the size of TOL or change to one of the more general routines D02HBF or D02SAF which afford more control over the calculations.

IFAIL $= 7$

The linear algebra routine (F08KBF (DGESVD)) used has failed. This error exit should not occur and can be avoided by changing the estimated initial values.

IFAIL $= 8$

The Newton iteration has failed to converge.

Note: IFAIL $= 8$ can indicate poor initial estimates or a very difficult problem. Consider varying TOL if the residuals are small in the monitoring output. If the residuals are large try varying the initial estimates.

IFAIL $= 9$
IFAIL $= 10$
IFAIL $= 11$
IFAIL $= 12$
IFAIL $= 13$

Indicates that a serious error has occurred in an internal call. Check all array subscripts and subroutine argument lists in calls to D02HAF. Seek expert help.

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.

7 Accuracy

If the process converges, the accuracy to which the unknown parameters are determined is usually close to that specified by you; the solution, if requested, may be determined to a required accuracy by varying TOL.

8 Parallelism and Performance

D02HAF is not thread safe and should not be called from a multithreaded user program. Please see Section 3.12.1 in How to Use the NAG Library and its Documentation for more information on thread safety.

D02HAF 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 D02HAF 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.

Wherever it occurs in the routine, the error argument TOL is used in 'mixed' form; that is TOL always occurs in expressions of the form $\text{TOL} \times (1 + |y_i|)$. Though not ideal for every application, it is expected that this mixture of absolute and relative error testing will be adequate for most purposes.

You are strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation. You may select the unit numbers on which this output is to appear by calls of X04AAF (for error messages) or X04ABF (for monitoring information) – see Section 10 for an example. Otherwise the default unit numbers will be used, as specified in the Users' Note. The monitoring information produced at each iteration includes the current parameter values, the residuals and 2-norms: a basic norm and a current norm. At each iteration the aim is to find parameter values which make the current norm less than the basic norm. Both these norms should tend to zero as should the residuals. (They would all be zero if the exact parameters were used as input.) For more details, you may consult the specification of D02SAF, and especially the description of the argument MONIT there.

The computing time for integrating the differential equations can sometimes depend critically on the quality of the initial estimates. If it seems that too much computing time is required and, in particular, if the values of the residuals printed by the monitoring routine are much larger than the expected values of the solution at b, then the coding of FCN should be checked for errors. If no errors can be found, an independent attempt should be made to improve the initial estimates. In practical problems it is not uncommon for the differential equation to have a singular point at one or both ends of the range. Suppose a is a singular point; then the derivatives y_i' in (1) (in Section 3) cannot be evaluated at a, usually because one or more of the expressions for f_i give overflow. In such a case it is necessary for you to take a a short distance away from the singularity, and to find values for y_i at the new value of a (e.g., use the first one or two terms of an analytical (power series) solution). You should experiment with the new position of a; if it is taken too close to the singular point, the derivatives f_i will be inaccurate, and the routine may sometimes fail with $\text{IFAIL} = 2$ or 3 or, in extreme cases, with an overflow condition. A more general treatment of singular solutions is provided by the subroutine D02HBF.

Another difficulty which often arises in practice is the case when one end of the range, b say, is at infinity. You must approximate the end point by taking a finite value for b, which is obtained by estimating where the solution will reach its asymptotic state. The estimate can be checked by repeating the calculation with a larger value of b. If b is very large, and if the matching point is also at b, the numerical solution may suffer a considerable loss of accuracy in integrating across the range, and the program may fail with $\text{IFAIL} = 6$ or 8. (In the former case, solutions from all initial values at a are tending to the same curve at infinity.) The simplest remedy is to try to solve the equations with a smaller value of b, and then to increase b in stages, using each solution to give boundary value estimates for the next calculation. For problems where some terms in the asymptotic form of the solution are known, D02HBF will be more successful.

If the unknown quantities are not boundary values, but are eigenvalues or the length of the range or some other parameters occurring in the differential equations, D02HBF may be used.

10 Example

This example finds the angle at which a projectile must be fired for a given range.

The differential equations are:

$$\begin{aligned} y' &= \tan\phi \\ v' &= \frac{-0.032\tan\phi}{v} - \frac{0.02v}{\cos\phi} \\ \phi\prime &= \frac{-0.032}{v^2}, \end{aligned}$$

with the following boundary conditions:

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

$$y = 0 \qquad\qquad \text{at} \quad x = 5.$$

The remaining boundary conditions are estimated as:

$$\phi = 1.15 \qquad\qquad \text{at} \quad x = 0,$$

$$\phi = 1.2, \quad v = 0.46 \quad \text{at} \quad x = 5.$$

We write $y = \mathrm{Z}(1)$, $v = \mathrm{Z}(2)$, $\phi = \mathrm{Z}(3)$. To check the accuracy of the results the problem is solved twice with $\mathrm{TOL} = 5.0\mathrm{E}{-3}$ and $5.0\mathrm{E}{-4}$ respectively. Note the call to X04ABF before the call to D02HAF.

10.1 Program Text

```
!   D02HAF Example Program Text
!   Mark 26 Release. NAG Copyright 2016.

    Module d02hafe_mod

!     D02HAF Example Program Module:
!            Parameters and User-defined Routines

!     .. Use Statements ..
      Use nag_library, Only: nag_wp
!     .. Implicit None Statement ..
      Implicit None
!     .. Accessibility Statements ..
      Private
      Public                           :: fcn
!     .. Parameters ..
      Real (Kind=nag_wp), Parameter, Public :: one = 1.0_nag_wp
      Real (Kind=nag_wp), Parameter, Public :: zero = 0.0_nag_wp
      Integer, Parameter, Public       :: iset = 1, n = 3, nin = 5, nout = 6
      Integer, Parameter, Public       :: sdw = 3*n + 17 + max(11,n)
!     .. Intrinsic Procedures ..
      Intrinsic                        :: max
    Contains
      Subroutine fcn(x,y,f)

!       .. Scalar Arguments ..
        Real (Kind=nag_wp), Intent (In) :: x
!       .. Array Arguments ..
        Real (Kind=nag_wp), Intent (Out) :: f(*)
        Real (Kind=nag_wp), Intent (In) :: y(*)
!       .. Intrinsic Procedures ..
        Intrinsic                        :: cos, tan
!       .. Executable Statements ..
        f(1) = tan(y(3))
        f(2) = -0.032_nag_wp*tan(y(3))/y(2) - 0.02_nag_wp*y(2)/cos(y(3))
        f(3) = -0.032_nag_wp/y(2)**2
        Return
      End Subroutine fcn
    End Module d02hafe_mod

    Program d02hafe

!     D02HAF Example Main Program

!     .. Use Statements ..
      Use nag_library, Only: d02haf, nag_wp, x04abf
      Use d02hafe_mod, Only: fcn, iset, n, nin, nout, one, sdw, zero
!     .. Implicit None Statement ..
      Implicit None
!     .. Local Scalars ..
      Real (Kind=nag_wp)               :: a, b, dx, tol
      Integer                          :: i, ifail, l, m1, outchn
!     .. Local Arrays ..
      Real (Kind=nag_wp), Allocatable  :: soln(:,:), x(:)
      Real (Kind=nag_wp)               :: u(n,2), v(n,2), w(n,sdw)
!     .. Intrinsic Procedures ..
      Intrinsic                        :: real
!     .. Executable Statements ..
      Write (nout,*) 'D02HAF Example Program Results'
!     Skip heading in data file
      Read (nin,*)
!     m1: solution is returned and printed for m1-1 grid points on [a, b].
      Read (nin,*) m1
      Allocate (soln(n,m1),x(m1))
!     a: left-hand boundary point, b: right-hand boundary point.
      Read (nin,*) a, b

!     Evaluate solution points x.
      x(1) = a
      dx = (b-a)/real(m1-1,kind=nag_wp)
      Do i = 2, m1 - 1
        x(i) = x(i-1) + dx
      End Do
      x(m1) = b

!     Set output channel for monitoring information.
      outchn = nout
      Call x04abf(iset,outchn)

!     Flag known (zero) and estimated (one) values in u
      v(1:2,1:2) = zero
      v(2,2) = one
      v(3,1:2) = one
!     Set known values of u
      u(1,1:2) = zero
      u(2,1) = 0.5_nag_wp

loop: Do l = 4, 5
        tol = 5.0_nag_wp*10.0_nag_wp**(-l)
        Write (nout,*)
!       Set estimates of u
        u(2,2) = 0.46_nag_wp
        u(3,1) = 1.15_nag_wp
        u(3,2) = -1.2_nag_wp

!       ifail: behaviour on error exit
!              =1 for quiet-soft exit
!       * Set ifail to 111 to obtain monitoring information *
        ifail = 1
        Call d02haf(u,v,n,a,b,tol,fcn,soln,m1,w,sdw,ifail)
        If (ifail>=0) Then
          Write (nout,99999) 'Results with TOL = ', tol
          Write (nout,*)
          If (ifail==0) Then
            Write (nout,*) ' X-value and final solution'
            Do i = 1, m1
              If (l==4) Then
                Write (nout,99998) x(i), soln(1:n,i)
              Else
                Write (nout,99997) x(i), soln(1:n,i)
              End If
            End Do
          Else
            Write (nout,99996) ' IFAIL =', ifail
          End If
        Else
          Write (nout,99995) ifail
          Exit loop
        End If
      End Do loop

99999 Format (1X,A,E10.3)
99998 Format (1X,F4.1,3(1X,F9.3))
99997 Format (1X,F4.1,1X,3F10.4)
99996 Format (1X,A,I4)
99995 Format (1X,/,1X,' ** D02HAF returned with IFAIL = ',I5)
    End Program d02hafe
```

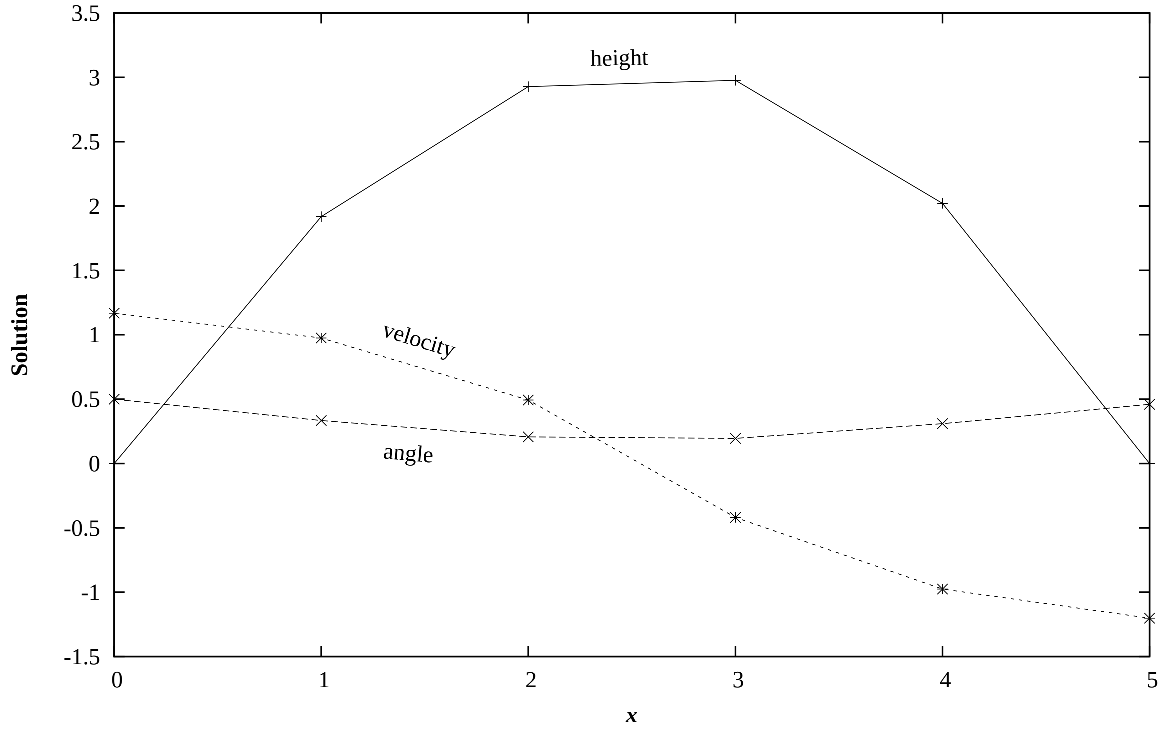
10.2 Program Data

```
D02HAF Example Program Data
   6                      : m1
   0.0  5.0               : a, b
```

10.3 Program Results

```
 D02HAF Example Program Results

 Results with TOL =  0.500E-03

  X-value and final solution
  0.0     0.000     0.500     1.168
  1.0     1.918     0.334     0.975
  2.0     2.928     0.207     0.493
  3.0     2.977     0.196    -0.419
  4.0     2.021     0.310    -0.975
  5.0    -0.000     0.460    -1.201

 Results with TOL =  0.500E-04

  X-value and final solution
  0.0     0.0000    0.5000    1.1681
  1.0     1.9176    0.3343    0.9749
  2.0     2.9281    0.2070    0.4929
  3.0     2.9771    0.1955   -0.4195
  4.0     2.0210    0.3095   -0.9752
  5.0    -0.0000    0.4597   -1.2014
```

Example Program
Solution of Two-point Boundary-value Problem
using Runge-Kutta-Merson and Newton Correction in a Shooting Method


height
velocity
angle
Solution
x