NAG Toolbox for MATLAB

d02ag

1 Purpose

d02ag solves the two-point boundary-value problem for a system of ordinary differential equations, using initial value techniques and Newton iteration; it generalizes d02ha to include the case where parameters other than boundary-values are to be determined.

2 Syntax

```
[h, param, c, ifail] = d02ag(h, e, parerr, param, m1, aux, bcaux, raaux, prsol, 'n', n, 'n1', n1)
```

3 Description

d02ag solves the two-point boundary-value problem by determining the unknown parameters $p_1, p_2, \ldots, p_{n_1}$ of the problem. These parameters may be, but need not be, boundary-values (as they are in d02ha); 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 d02ha and you are advised to study this first. (There the parameters $p_1, p_2, \ldots, p_{n_1}$ correspond to the unknown boundary conditions.) It is assumed that we have a system of n first-order ordinary differential equations of the form

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

and that derivatives f_i are evaluated by user-supplied (sub)program **aux**. The system, including the boundary conditions given by user-supplied (sub)program **bcaux**, and the range of integration and matching point, r, given by user-supplied (sub)program **raaux**, involves the n_1 unknown parameters $p_1, p_2, \ldots, p_{n_1}$ 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 the **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 ith component of the solution, y_i, with respect to the jth 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 Parameters

You are strongly recommended to read Sections 3 and 8 in conjunction with this section.

5.1 Compulsory Input Parameters

1: **h – double scalar**

h, **h** must be set to an estimate of the step size needed for integration.

2: **e(n) – double array**

e(i) must be set to a small quantity to control the ith solution component. The element **e**(i) is used:

(i) in the bound on the local error in the ith component of the solution y_i during integration,

(ii) in the convergence test on the ith component of the solution y_i at the matching point in the Newton iteration.

The elements $\mathbf{e}(i)$ should not be chosen too small. They should usually be several orders of magnitude larger than ***machine precision***.

3: **parerr(n1) – double array**

parerr(i) must be set to a small quantity to control the ith parameter component. The element **parerr**(i) is used:

(i) in the convergence test on the ith parameter in the Newton iteration,

(ii) in perturbing the ith parameter when approximating the derivatives of the components of the solution with respect to the ith 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) – double array**

param(i) must be set to an estimate for the ith parameter, p_i, for $i = 1, 2, \ldots,$ **n1**.

5: **m1 – int32 scalar**

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 user-supplied (sub)program **raaux**) stored in $\mathbf{c}(1, i)$.

6: **aux – string containing name of m-file**

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

Its specification is:

```
[f] = aux(y, x, param)
```

Input Parameters

1: **y(n) – double array**

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

2: **x – double scalar**

The value of the argument x.

3: **param(n1) – double array**

The value of the argument p_i, for $i = 1, 2, \ldots, n_1$.

Output Parameters

1: **f(n) – double array**

The value of f_i, for $i = 1, 2, \ldots, n$.

7: **bcaux – string containing name of m-file**

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

Its specification is:

```
[g0, g1] = bcaux(param)
```

Input Parameters

1: **param**$(n1)$ **– double array**

The value of the argument p_i, for $i = 1, 2, \ldots, n$.

Output Parameters

1: **g0**(n) **– double array**

The values y_i, for $i = 1, 2, \ldots, n$, at the boundary point x_0 (see user-supplied (sub)program **raaux**).

2: **g1**(n) **– double array**

The values y_i, for $i = 1, 2, \ldots, n$, at the boundary point x_1 (see user-supplied (sub)program **raaux**).

8: **raaux – string containing name of m-file**

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

Its specification is:

```
[x0, x1, r] = raaux(param)
```

Input Parameters

1: **param**$(n1)$ **– double array**

The value of the argument p_i, for $i = 1, 2, \ldots, n_1$.

Output Parameters

1: **x0 – double scalar**

Must contain the left-hand end of the range, x_0.

2: **x1 – double scalar**

Must contain the right-hand end of the range x_1.

3: **r – double scalar**

Must contain the matching point, r.

9: **prsol – string containing name of m-file**

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 $i = 1, 2, \ldots, n_1$, their errors, e_i, and the sum of squares of the errors at the matching point, r.

Its specification is:

```
[] = prsol(param, res, n1, err)
```

Input Parameters

1: **param(n1) – double array**

The current value of the parameters p_i, for $i = 1, 2, \ldots, n_1$.

2: **res – double scalar**

The sum of squares of the errors in the parameters, $\sum_{i=1}^{n_1} e_i^2$.

3: **n1 – int32 scalar**

n_1, the number of parameters.

4: **err(n1) – double array**

The errors in the parameters, e_i, for $i = 1, 2, \ldots, n_1$.

Output Parameters

5.2 Optional Input Parameters

1: **n – int32 scalar**

Default: The dimension of the arrays **e**, **c**. (An error is raised if these dimensions are not equal.)

n, the total number of differential equations.

2: **n1 – int32 scalar**

Default: The dimension of the arrays **parerr**, **param**. (An error is raised if these dimensions are not equal.)

n_1, the number of parameters.

If **n1** $<$ **n**, the last **n** $-$ **n1** differential equations (in the user-supplied (sub)program **aux**) are driving equations (see Section 3).

Constraint: **n1** $\leq$ **n**.

5.3 Input Parameters Omitted from the MATLAB Interface

mat, copy, wspace, wspac1, wspac2

5.4 Output Parameters

1: **h – double scalar**

The last step length used.

2: **param(n1) – double array**

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).

3: **c(m1,n) – double array**

The solution when **m1** > 1 (see **m1**).

If **m1** $= 1$ then the elements of **c** are not used.

4: **ifail – int32 scalar**

0 unless the function detects an error (see Section 6).

6 Error Indicators and Warnings

Errors or warnings detected by the function:

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** $= 4$ (below) 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 user-supplied (sub)program **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 f03af).

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 user-supplied (sub)program **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.

A further discussion of these errors and the steps which might be taken to correct them is given in Section 8.

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 Further Comments

The time taken by d02ag 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 function (for example, see p_2 in example (i) of Section 9).

Wherever they occur in the procedure, the error parameters contained in the arrays **e** and **parerr** are used in 'mixed' form; that is $\mathbf{e}(i)$ always occurs in expressions of the form $\mathbf{e}(i) \times (1 + |y_i|)$, and $\mathbf{parerr}(i)$ always occurs in expressions of the form $\mathbf{parerr}(i) \times (1 + |p_i|)$. 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 user-supplied (sub)program **prsol**, as shown in the example (i) of Section 9, 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 $y_1, y_2, \ldots, y_n$ is sensitive to perturbations in the parameters p_i. Reduce the size of one or more values $\mathbf{parerr}(i)$ 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 d02ag. 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 $\mathbf{err}(i)$ (available on each call of the user-supplied (sub)program **prsol**) are much larger than the expected values of the solution at the matching point r, then the coding of the user-supplied (sub)programs **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 $\mathbf{param}(i)$.

The (sub)program 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 example (ii) in Section 9);

(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 (ii) in Section 9);

(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 example (i) in Section 9); and

(e) differential equations with certain terms defined by other independent (driving) differential equations.

9 Example

d02ag_aux.m

```
function [f] = d02ag_aux(y,x,param)
```

```
   f(1)=y(2);
   f(2)=(y(1)^3-y(2))/(2.0*x);
```

d02ag_bcaux.m

```
function [g0,g1] = d02ag_bcaux(param)

      z = 0.1;
      g0(1) = 0.1 + param(1)*sqrt(z)*0.1 + 0.01*z;
      g0(2) = param(1)*0.05/sqrt(z) + 0.01;
      g1(1) = 1.0/6.0;
      g1(2) = param(2);
;
```

d02ag_prsol.m

```
function [] = prsol(param, res, n1, err)
```

d02ag_raaux.m

```
function [x0, x1, r] = d02ag_raaux(param)

      x0 = 0.1;
      x1 = 16.0;
      r = 16.0;
```

```
h = 0.1;
e = [0.0001;
     0.0001];
parerr = [1e-05;
     0.001];
param = [0.2;
     0];
m1 = int32(6);
[hOut, paramOut, c, ifail] = ...
          d02ag(h,  e,  parerr,  param,  m1,  'd02ag_aux',  'd02ag_bcaux',
'd02ag_raaux', 'd02ag_prsol')
```

```
hOut =
   -0.1000
paramOut =
    0.0464
    0.0035
c =
    0.1025    0.0173
    0.1217    0.0042
    0.1338    0.0036
    0.1449    0.0034
    0.1557    0.0034
    0.1667    0.0035
ifail =
           0
```