NAG Toolbox for MATLAB

d02sa

1 Purpose

d02sa solves a two-point boundary-value problem for a system of first-order ordinary differential equations with boundary conditions, combined with additional algebraic equations. It uses initial value techniques and a modified Newton iteration in a shooting and matching method.

2 Syntax

```
[p, pf, dp, swp, w, ifail] = d02sa(p, n1, pe, pf, e, dp, npoint, swp,
icount, range, bc, fcn, eqn, constr, ymax, monit, prsol, 'm', m, 'n', n)
```

3 Description

d02sa solves a two-point boundary-value problem for a system of n first-order ordinary differential equations with separated boundary conditions by determining certain unknown parameters $p_1, p_2, \ldots, p_m$. (There may also be additional algebraic equations to be solved in the determination of the parameters and, if so, these equations are defined by the (sub)program **eqn**.) The parameters may be, but need not be, boundary-values; they may include eigenvalues, parameters in the coefficients of the differential equations, coefficients in series expansions or asymptotic expansions for boundary-values, the length of the range of definition of the system of differential equations, etc.

It is assumed that we have a system of n differential equations of the form

$$y' = f(x, y, p), \tag{1}$$

where $p = (p_1, p_2, \ldots, p_m)^{\mathrm{T}}$ is the vector of parameters, and that the derivative f is evaluated by the user-supplied (sub)program **fcn**. Also, n_1 of the equations are assumed to depend on p. For $n_1 < n$ the $n - n_1$ equations of the system are not involved in the matching process. These are the driving equations; they should be independent of p and of the solution of the other n_1 equations. In numbering the equations in user-supplied (sub)programs **fcn** and **bc** the driving equations must be put **first** (as they naturally occur in most applications). The range of definition $[a, b]$ of the differential equations is defined by the **range** and may depend on the parameters $p_1, p_2, \ldots, p_m$ (that is, on p). (sub)program **range** must define the points $x_1, x_2, \ldots, x_{\mathbf{npoint}}$, **npoint** ≥ 2, which must satisfy

$$a = x_1 < x_2 < \cdots < x_{\mathbf{npoint}} = b \tag{2}$$

(or a similar relationship with all the inequalities reversed).

If **npoint** > 2 the points $x_1, x_2, \ldots, x_{\mathbf{npoint}}$ can be used to break up the range of definition. Integration is restarted at each of these points. This means that the differential equations (1) can be defined differently in each sub-interval $[x_i, x_{i+1}]$, for $i = 1, 2, \ldots,$ **npoint** $- 1$. Also, since initial and maximum integration step sizes can be supplied on each sub-interval (via the array **swp**), you can indicate parts of the range $[a, b]$ where the solution $y(x)$ may be difficult to obtain accurately and can take appropriate action.

The boundary conditions may also depend on the parameters and are applied at $a = x_1$ and $b = x_{\mathbf{npoint}}$. They are defined (in the user-supplied (sub)program **bc**) in the form

$$y(a) = g_1(p), > y(b) = g_2(p). \tag{3}$$

The boundary-value problem is solved by determining the unknown parameters p by a shooting and matching technique. The differential equations are always integrated from a to b with initial values $y(a) = g_1(p)$. The solution vector thus obtained at $x = b$ is subtracted from the vector $g_2(p)$ to give the n_1 residuals $r_1(p)$, ignoring the first $n - n_1$, driving equations. Because the direction of integration is always from a to b, it is unnecessary, in **bc**, to supply values for the first $n - n_1$ boundary-values at b, that is the first $n - n_1$ components of g_2 in (3). For $n_1 < m$ then $r_1(p)$. Together with the $m - n_1$ equations defined by the (sub)program **eqn**,

$$r_2(p) = 0, \tag{4}$$

these give a vector of residuals r, which at the solution, p, must satisfy

$$\begin{pmatrix} r_1(p) \\ r_2(p) \end{pmatrix} = 0. \tag{5}$$

These equations are solved by a pseudo-Newton iteration which uses a modified singular value decomposition of $J = \dfrac{\partial r}{\partial p}$ when solving the linear equations which arise. The Jacobian J used in Newton's method is obtained by numerical differentiation. The parameters at each Newton iteration are accepted only if the norm $\left\|D^{-1}\tilde{J}^{+}r\right\|_2$ is much reduced from its previous value. Here $\tilde{J}^{+}$ is the pseudo-inverse, calculated from the singular value decomposition, of a modified version of the Jacobian J (J^{+} is actually the inverse of the Jacobian in well-conditioned cases). D is a diagonal matrix with

$$d_{ii} = \max(|p_i|, \mathbf{pf}(i)) \tag{6}$$

where **pf** is an array of floor values.

See Deuflhard 1974 for further details of the variants of Newton's method used, Gay 1976 for the modification of the singular value decomposition and Gladwell 1979a for an overview of the method used.

Two facilities are provided to prevent the pseudo-Newton iteration running into difficulty. First, you are permitted to specify constraints on the values of the parameters p via a user-supplied logical function **constr**. These constraints are only used to prevent the Newton iteration using values for p which would violate them; that is, they are not used to determine the values of p. Secondly, you are permitted to specify a maximum value $y_{\max}$ for $\|y(x)\|_\infty$ at all points in the range $[a, b]$. It is intended that this facility be used to prevent machine 'overflow' in the integrations of equation (1) due to poor choices of the parameters p which might arise during the Newton iteration. When using this facility, it is presumed that you have an estimate of the likely size of $\|y(x)\|_\infty$ at all points $x \in [a, b]$. $y_{\max}$ should then be chosen rather larger (say by a factor of 10) than this estimate.

You are strongly advised to supply a (sub)program **monit** (or to call the 'default' function **d02hbx**, see **monit**) to monitor the progress of the pseudo-Newton iteration. You can output the solution of the problem $y(x)$ by supplying a suitable (sub)program **prsol** (an example is given in Section 9 of a function designed to output the solution at equally spaced points).

d02sa is designed to try all possible options before admitting failure and returning to you. Provided the function can start the Newton iteration from the initial point p it will exhaust all the options available to it (though you can override this by specifying a maximum number of iterations to be taken). The fact that all its options have been exhausted is the only error exit from the iteration. Other error exits are possible, however, whilst setting up the Newton iteration and when computing the final solution.

If you require more background information about the solution of boundary-value problems by shooting methods you are recommended to read the appropriate chapters of Hall and Watt 1976, and for a detailed description of d02sa Gladwell 1979a is recommended.

4 References

Deuflhard P 1974 A modified Newton method for the solution of ill-conditioned systems of nonlinear equations with application to multiple shooting *Numer. Math.* **22** 289–315

Gay D 1976 On modifying singular values to solve possibly singular systems of nonlinear equations *Working Paper 125* Computer Research Centre, National Bureau for Economics and Management Science, Cambridge, MA

Gladwell I 1979a The development of the boundary value codes in the ordinary differential equations chapter of the NAG Library *Codes for Boundary Value Problems in Ordinary Differential Equations. Lecture Notes in Computer Science* (ed B Childs, M Scott, J W Daniel, E Denman and P Nelson) **76** Springer–Verlag

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

5 Parameters

5.1 Compulsory Input Parameters

1: **p(m) – double array**

p(i) must be set to an estimate of the ith parameter, p_i, for $i = 1, 2, \ldots, m$.

2: **n1 – int32 scalar**

The number of differential equations active in the matching process, n_1. The active equations must be placed last in the numbering in the user-supplied (sub)programs **fcn** and **bc**. The **first n − n1** equations are used as the driving equations.

Constraint: **n1** $\leq$ **n**, **n1** $\leq$ **m** and **n1** > 0.

3: **pe(m) – double array**

pe(i), for $i = 1, 2, \ldots, m$, must be set to a positive value for use in the convergence test in the ith parameter p_i. See the description of **pf** for further details.

Constraint: **pe**$(i) > 0$, for $i = 1, 2, \ldots, m$.

4: **pf(m) – double array**

pf(i), for $i = 1, 2, \ldots, m$, should be set to a 'floor' value in the convergence test on the ith parameter p_i. If **pf**$(i) \leq 0.0$ on entry then it is set to the small positive value $\sqrt{\epsilon}$ (where ϵ may in most cases be considered to be ***machine precision***); otherwise it is used unchanged.

The Newton iteration is presumed to have converged if a full Newton step is taken (**istate** $= 1$ in the specification of the (sub)program **monit**), the singular values of the Jacobian are not being significantly perturbed (also see **monit**) and if the Newton correction C_i satisfies

$$|C_i| \leq \mathbf{pe}(i) \times \max(|p_i|, \mathbf{pf}(i)), \qquad i = 1, 2, \ldots, m,$$

where p_i is the current value of the ith parameter. The values **pf**(i) are also used in determining the Newton iterates as discussed in Section 3, see equation (6).

5: **e(n) – double array**

Values for use in controlling the local error in the integration of the differential equations. If err_i is an estimate of the local error in y_i, for $i = 1, 2, \ldots, n$, then

$$|err_i| \leq \mathbf{e}(i) \times \max\{\sqrt{\epsilon}, |y_i|\},$$

where ϵ may in most cases be considered to be ***machine precision***.

Suggested value: **e**$(i) = 10^{-5}$.

Default: **e**$(i) = 10^{-5}$

Constraint: **e**$(i) > 0.0$, for $i = 1, 2, \ldots,$ **n**.

6: **dp(m) – double array**

A value to be used in perturbing the parameter p_i in the numerical differentiation to estimate the Jacobian used in Newton's method. If **dp**$(i) = 0.0$ on entry, an estimate is made internally by setting

$$\mathbf{dp}(i) = \sqrt{\epsilon} \times \max(\mathbf{pf}(i), |p_i|), \tag{7}$$

where p_i is the initial value of the parameter supplied by you and ϵ may in most cases be considered to be ***machine precision***. The estimate of the Jacobian, J, is made using forward differences, that is for each i, for $i = 1, 2, \ldots, m$, p_i is perturbed to $p_i +$ **dp**(i) and the ith column of J is estimated as

$$(r(p_i + \mathbf{dp}(i)) - r(p_i))/\mathbf{dp}(i)$$

where the other components of p are unchanged (see (3) for the notation used). If this fails to

produce a Jacobian with significant columns, backward differences are tried by perturbing p_i to $p_i - \mathbf{dp}(i)$ and if this also fails then central differences are used with p_i perturbed to $p_i + 10.0 \times \mathbf{dp}(i)$. If this also fails then the calculation of the Jacobian is abandoned. If the Jacobian has not previously been calculated then an error exit is taken. If an earlier estimate of the Jacobian is available then the current parameter set, p_i, for $i = 1, 2, \ldots, M$, is abandoned in favour of the last parameter set from which useful progress was made and the singular values of the Jacobian used at the point are modified before proceeding with the Newton iteration. You are recommended to use the default value $\mathbf{dp}(i) = 0.0$ unless you have prior knowledge of a better choice. If any of the perturbations described are likely to lead to an unfortunate set of parameter values then you should use user-supplied logical function **constr** to prevent such perturbations (all changes of parameters are checked by a call to **constr**).

7: **npoint – int32 scalar**

2 plus the number of break points in the range of definition of the system of differential equations (1).

Constraint: **npoint** ≥ 2.

8: **swp(ldswp,6) – double array**

ldswp, the first dimension of the array, must be at least **npoint**.

$\mathbf{swp}(i, 1)$ must contain an estimate for an initial step size for integration across the ith sub-interval $[\mathbf{x}(i), \mathbf{x}(i+1)]$, for $i = 1, 2, \ldots, \mathbf{npoint} - 1$ (see the user-supplied (sub)program **range**). $\mathbf{swp}(i, 1)$ should have the same sign as $\mathbf{x}(i+1) - \mathbf{x}(i)$ if it is nonzero. If $\mathbf{swp}(i, 1) = 0.0$, on entry, a default value for the initial step size is calculated internally. This is the recommended mode of entry.

$\mathbf{swp}(i, 3)$ must contain a lower bound for the modulus of the step size on the ith sub-interval $[\mathbf{x}(i), \mathbf{x}(i+1)]$, for $i = 1, 2, \ldots, \mathbf{npoint} - 1$. If $\mathbf{swp}(i, 3) = 0.0$ on entry, a very small default value is used. By setting $\mathbf{swp}(i, 3) > 0.0$ but smaller than the expected step sizes (assuming you have some insight into the likely step sizes) expensive integrations with parameters p far from the solution can be avoided.

$\mathbf{swp}(i, 2)$ must contain an upper bound on the modulus of the step size to be used in the integration on $[\mathbf{x}(i), \mathbf{x}(i+1)]$, for $i = 1, 2, \ldots, \mathbf{npoint} - 1$. If $\mathbf{swp}(i, 2) = 0.0$ on entry no bound is assumed. This is the recommended mode of entry unless the solution is expected to have important features which might be 'missed' in the integration if the step size were permitted to be chosen freely.

9: **icount – int32 scalar**

An upper bound on the number of Newton iterations. If $\mathbf{icount} = 0$ on entry, no check on the number of iterations is made (this is the recommended mode of entry).

Constraint: **icount** ≥ 0.

10: **range – string containing name of m-file**

range must specify the break points x_i, for $i = 1, 2, \ldots, \mathbf{npoint}$, which may depend on the parameters p_j, for $j = 1, 2, \ldots, m$.

Its specification is:

```
[x] = range(npoint, p, m)
```

Input Parameters

1: **npoint – int32 scalar**

2 plus the number of break points in (a, b).

2: **p(m) – double array**

The current estimate of the ith parameter, for $i = 1, 2, \ldots, m$.

3: **m – int32 scalar**

m, the number of parameters.

Output Parameters

1: **x(npoint) – double array**

The ith break point, for $i = 1, 2, \ldots,$ **npoint**. The sequence $(\mathbf{x}(i))$ must be strictly monotonic, that is either

$$a = \mathbf{x}(1) < \mathbf{x}(2) < \cdots < \mathbf{x}(\mathbf{npoint}) = b$$

or

$$a = \mathbf{x}(1) > \mathbf{x}(2) > \cdots > \mathbf{x}(\mathbf{npoint}) = b.$$

11: **bc – string containing name of m-file**

bc must place in **g1** and **g2** the boundary conditions at a and b respectively.

Its specification is:

```
[g1, g2] = bc(p, m, n)
```

Input Parameters

1: **p(m) – double array**

An estimate of the ith parameter, p_i, for $i = 1, 2, \ldots, m$.

2: **m – int32 scalar**

m, the number of parameters.

3: **n – int32 scalar**

n, the number of differential equations.

Output Parameters

1: **g1(n) – double array**

The value of $y_i(a)$, (where this may be a known value or a function of the parameters p_j, for $j = 1, 2, \ldots, m$), for $i = 1, 2, \ldots, n$.

2: **g2(n) – double array**

The value of $y_i(b)$, for $i = 1, 2, \ldots, n$, (where these may be known values or functions of the parameters p_j, for $j = 1, 2, \ldots, m$). If $n > n_1$, so that there are some driving equations, then the first $n - n_1$ values of **g2** need not be set since they are never used.

12: **fcn – string containing name of m-file**

fcn must evaluate the functions f_i (i.e., the derivatives y_i'), for $i = 1, 2, \ldots, n$.

Its specification is:

```
[f] = fcn(x, y, n, p, m, ii)
```

Input Parameters

1: **x – double scalar**

The value of the argument x.

2: **y(n) – double array**

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

3: **n – int32 scalar**

n, the number of equations.

4: **p(m) – double array**

The current estimate of the ith parameter, p_i, for $i = 1, 2, \ldots, m$.

5: **m – int32 scalar**

m, the number of parameters.

6: **ii – int32 scalar**

Specifies the sub-interval $[x_i, x_{i+1}]$ on which the derivatives are to be evaluated.

Output Parameters

1: **f(n) – double array**

The derivative of y_i evaluated at x, for $i = 1, 2, \ldots, n$. **f**(i) may depend upon the parameters p_j, for $j = 1, 2, \ldots, m$. If there are any driving equations (see Section 3) then these must be numbered first in the ordering of the components of **f**.

13: **eqn – string containing name of m-file**

eqn is used to describe the additional algebraic equations to be solved in the determination of the parameters, p_i, for $i = 1, 2, \ldots, m$. If there are no additional algebraic equations (i.e., $m = n_1$) then **eqn** is never called and the string `'d02hbz'` should be used as the actual argument.

Its specification is:

```
[e] = eqn(q, p, m)
```

Input Parameters

1: **q – int32 scalar**

The number of algebraic equations, $m - n_1$.

2: **p(m) – double array**

The current estimate of the ith parameter p_i, for $i = 1, 2, \ldots, m$.

3: **m – int32 scalar**

m, the number of parameters.

Output Parameters

1: **e(q) – double array**

The vector of residuals, $r_2(p)$, that is the amount by which the current estimates of the parameters fail to satisfy the algebraic equations.

14: **constr – string containing name of m-file**

constr is used to prevent the pseudo-Newton iteration running into difficulty. **constr** should return the value **true** if the constraints are satisfied by the parameters $p_1, p_2, \ldots, p_m$. Otherwise **constr** should return the value **false**. Usually the dummy function **d02hby**, which returns the value **true** at all times, will suffice and in the first instance this is recommended as the actual parameter.

Its specification is:

```
[result] = constr(p, m)
```

Input Parameters

1: **p(m) – double array**

An estimate of the ith parameter, p_i, for $i = 1, 2, \ldots, m$.

2: **m – int32 scalar**

m, the number of parameters.

Output Parameters

1: **result – logical scalar**

The result of the function.

15: **ymax – double scalar**

A nonnegative value which is used as a bound on all values $\|y(x)\|_\infty$ where $y(x)$ is the solution at any point x between $\mathbf{x}(1)$ and $\mathbf{x}(\mathbf{npoint})$ for the current parameters $p_1, p_2, \ldots, p_m$. If this bound is exceeded the integration is terminated and the current parameters are rejected. Such a rejection will result in an error exit if it prevents the initial residual or Jacobian, or the final solution, being calculated. If $\mathbf{ymax} = 0$ on entry, no bound on the solution y is used; that is the integrations proceed without any checking on the size of $\|y\|_\infty$.

16: **monit – string containing name of m-file**

monit enables you to monitor the values of various quantities during the calculation. It is called by d02sa after every calculation of the norm $\left\|\mathbf{d}^{-1}\tilde{J}_{+r}\right\|_2$ which determines the strategy of the Newton method, every time there is an internal error exit leading to a change of strategy, and before an error exit when calculating the initial Jacobian. Usually the function **d02hbx** will be adequate and you are advised to use this as the actual parameter for **monit** in the first instance. (In this case a call to x04ab must be made prior to the call of d02sa.) If no monitoring is required, the string `'d02sas'` may be used.

Its specification is:

```
[] = monit(istate, iflag, ifail1, p, m, f, pnorm, pnorm1, eps, d)
```

Input Parameters

1: **istate – int32 scalar**

The state of the Newton iteration.

istate $= 0$

The calculation of the residual, Jacobian and $\left\|\mathbf{d}^{-1}\tilde{J}^{+}r\right\|_2$ are taking place.

istate $= 1$ to 5

During the Newton iteration a factor of $2^{(-\mathbf{istate}+1)}$ of the Newton step is being used to try to reduce the norm.

istate $= 6$

The current Newton step has been rejected and the Jacobian is being re-calculated.

istate $= -6$ to -1

An internal error exit has caused the rejection of the current set of parameter values, p. $-$**istate** is the value which **istate** would have taken if the error had not occurred.

istate $= -7$

An internal error exit has occurred when calculating the initial Jacobian.

2: **iflag – int32 scalar**

Whether or not the Jacobian being used has been calculated at the beginning of the current iteration. If the Jacobian has been updated then **iflag** $= 1$; otherwise **iflag** $= 2$. The Jacobian is only calculated when convergence to the current parameter values has been slow.

3: **ifail1 – int32 scalar**

If $-6 \leq$ **istate** ≤ -1, **ifail1** specifies the **ifail** error number that would be produced were control returned to you. **ifail1** is unspecified for values of **istate** outside this range.

4: **p(m) – double array**

The current estimate of the ith parameter, p_i, for $i = 1, 2, \ldots, m$.

5: **m – int32 scalar**

m, the number of parameters.

6: **f(m) – double array**

The residual r corresponding to the current parameter values, provided $1 \leq$ **istate** ≤ 5 or **istate** $= -7$. **f** is unspecified for other values of **istate**.

7: **pnorm – double scalar**

A quantity against which all reductions in norm are currently measured.

8: **pnorm1 – double scalar**

p, the norm of the current parameters. It is set for $1 \leq$ **istate** ≤ 5 and is undefined for other values of **istate**.

9: **eps – double scalar**

Gives some indication of the convergence rate. It is the current singular value modification factor (see Gay 1976). It is zero initially and whenever convergence is

proceeding steadily. **eps** is $\epsilon^{3/8}$ or greater (where ϵ may in most cases be considered ***machine precision***) when the singular values of J are approximately zero or when convergence is not being achieved. The larger the value of **eps** the worse the convergence rate. When **eps** becomes too large the Newton iteration is terminated.

10: **d(m) – double array**

J, the singular values of the current modified Jacobian matrix. If $\mathbf{d}(m)$ is small relative to $\mathbf{d}(1)$ for a number of Jacobians corresponding to different parameter values then the computed results should be viewed with suspicion. It could be that the matching equations do not depend significantly on some parameter (which could be due to a programming error in user-supplied (sub)program **fcn**, user-supplied (sub)program **bc**, user-supplied (sub)program **range** or (sub)program **eqn**). Alternatively, the system of differential equations may be very ill-conditioned when viewed as an initial value problem, in which case d02sa is unsuitable. This may also be indicated by some singular values being very large. These values of $\mathbf{d}(i)$, for $i = 1, 2, \ldots, m$, should not be changed.

Output Parameters

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

prsol can be used to obtain values of the solution y at a selected point z by integration across the final range $[\mathbf{x}(1), \mathbf{x}(\mathbf{npoint})]$. If no output is required **d02hbw** can be used as the actual parameter.

Its specification is:

```
[z] = prsol(z, y, n)
```

Input Parameters

1: **z – double scalar**

Contains x_1 on the first call. On subsequent calls **z** contains its previous output value.

The next point at which output is required. The new point must be nearer $\mathbf{x}(\mathbf{npoint})$ than the old.

If **z** is set to a point outside $[\mathbf{x}(1), \mathbf{x}(\mathbf{npoint})]$ the process stops and control returns from d02sa to the (sub)program from which d02sa is called. Otherwise the next call to **prsol** is made by d02sa at the point **z**, with solution values $y_1, y_2, \ldots, y_n$ at **z** contained in **y**. If **z** is set to $\mathbf{x}(\mathbf{npoint})$ exactly, the final call to **prsol** is made with $y_1, y_2, \ldots, y_n$ as values of the solution at $\mathbf{x}(\mathbf{npoint})$ produced by the integration. In general the solution values obtained at $\mathbf{x}(\mathbf{npoint})$ from **prsol** will differ from the values obtained at this point by a call to user-supplied (sub)program **bc**. The difference between the two solutions is the residual r. You are reminded that the points $\mathbf{x}(1), \mathbf{x}(2), \ldots, \mathbf{x}(\mathbf{npoint})$ are available in the locations $\mathbf{swp}(1,4), \mathbf{swp}(2,4), \ldots, \mathbf{swp}(\mathbf{npoint}, 4)$ at all times.

2: **y(n) – double array**

The solution value y_i at z, for $i = 1, 2, \ldots, n$.

3: **n – int32 scalar**

n, the total number of differential equations.

Output Parameters

1: **z – double scalar**

Contains x_1 on the first call. On subsequent calls **z** contains its previous output value.

The next point at which output is required. The new point must be nearer $\mathbf{x}(\mathbf{npoint})$ than the old.

If **z** is set to a point outside $[\mathbf{x}(1), \mathbf{x}(\mathbf{npoint})]$ the process stops and control returns from d02sa to the (sub)program from which d02sa is called. Otherwise the next call to **prsol** is made by d02sa at the point **z**, with solution values $y_1, y_2, \ldots, y_n$ at **z** contained in **y**. If **z** is set to $\mathbf{x}(\mathbf{npoint})$ exactly, the final call to **prsol** is made with $y_1, y_2, \ldots, y_n$ as values of the solution at $\mathbf{x}(\mathbf{npoint})$ produced by the integration. In general the solution values obtained at $\mathbf{x}(\mathbf{npoint})$ from **prsol** will differ from the values obtained at this point by a call to user-supplied (sub)program **bc**. The difference between the two solutions is the residual r. You are reminded that the points $\mathbf{x}(1), \mathbf{x}(2), \ldots, \mathbf{x}(\mathbf{npoint})$ are available in the locations $\mathbf{swp}(1,4), \mathbf{swp}(2,4), \ldots, \mathbf{swp}(\mathbf{npoint},4)$ at all times.

5.2 Optional Input Parameters

1: **m – int32 scalar**

Default: The dimension of the arrays **p**, **pe**, **pf**, **dp**. (An error is raised if these dimensions are not equal.)

m, the number of parameters.

Constraint: $\mathbf{m} > 0$.

2: **n – int32 scalar**

Default: The dimension of the array **e**.

n, the total number of differential equations.

Constraint: $\mathbf{n} > 0$.

5.3 Input Parameters Omitted from the MATLAB Interface

ldswp, ldw, sdw

5.4 Output Parameters

1: **p(m) – double array**

The corrected value for the ith parameter, unless an error has occurred, when it contains the last calculated value of the parameter.

2: **pf(m) – double array**

The values actually used.

3: **dp(m) – double array**

The values actually used.

4: **swp(ldswp,6) – double array**

$\mathbf{swp}(i,1)$ contains the initial step size used on the last integration on $[\mathbf{x}(i), \mathbf{x}(i+1)]$, for $i = 1, 2, \ldots, \mathbf{npoint} - 1$, (excluding integrations during the calculation of the Jacobian).

$\mathbf{swp}(i,2)$, for $i = 1, 2, \ldots, \mathbf{npoint} - 1$, is usually unchanged. If the maximum step size $\mathbf{swp}(i,2)$ is so small or the length of the range $[\mathbf{x}(i), \mathbf{x}(i+1)]$ is so short that on the last integration the step size was not controlled in the main by the size of the error tolerances $\mathbf{e}(i)$ but by these other factors, then $\mathbf{swp}(\mathbf{npoint},2)$ is set to the floating-point value of i if the problem last occurred in $[\mathbf{x}(i), \mathbf{x}(i+1)]$. Any results obtained when this value is returned as nonzero should be viewed with caution.

$\mathbf{swp}(i,3)$, for $i = 1, 2, \ldots, \mathbf{npoint} - 1$, are unchanged.

If an error exit with **ifail** = 4, 5 or 6 (see Section 6) occurs on the integration made from $\mathbf{x}(i)$ to $\mathbf{x}(i+1)$ the floating-point value of i is returned in $\mathbf{swp}(\mathbf{npoint}, 1)$. The actual point $x \in [\mathbf{x}(i), \mathbf{x}(i+1)]$ where the error occurred is returned in $\mathbf{swp}(1, 5)$ (see also the specification of **w**). The floating-point value of **npoint** is returned in $\mathbf{swp}(\mathbf{npoint}, 1)$ if the error exit is caused by a call to user-supplied (sub)program **bc**.

If an error exit occurs when estimating the Jacobian matrix (**ifail** = 7, 8, 9, 10, 11 or 12, see Section 6) and if parameter p_i was the cause of the failure then on exit $\mathbf{swp}(\mathbf{npoint}, 1)$ contains the floating-point value of i.

$\mathbf{swp}(i, 4)$ contains the point $\mathbf{x}(i)$, for $i = 1, 2, \ldots,$ **npoint**, used at the solution p or at the final values of p if an error occurred.

swp is also partly used as workspace.

5: **w(ldw,sdw) – double array**

In the case of an error exit of the type where the point of failure is returned in $\mathbf{swp}(1, 5)$, the solution at this point of failure is returned in $\mathbf{w}(i, 1)$, for $i = 1, 2, \ldots, n$.

Otherwise **w** is used for workspace.

6: **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

One or more of the parameters **n**, **n1**, **m**, **ldswp**, **npoint**, **icount**, **ldw**, **sdw**, **e**, **pe** or **ymax** has been incorrectly set.

ifail = 2

The constraints have been violated by the initial parameters.

ifail = 3

The condition $\mathbf{x}(1) < \mathbf{x}(2) < \cdots < \mathbf{x}(\mathbf{npoint})$ (or $\mathbf{x}(1) > \mathbf{x}(2) > \cdots > \mathbf{x}(\mathbf{npoint})$) has been violated on a call to user-supplied (sub)program **range** with the initial parameters.

ifail = 4

In the integration from $\mathbf{x}(1)$ to $\mathbf{x}(\mathbf{npoint})$ with the initial or the final parameters, the step size was reduced too far for the integration to proceed. Consider reversing the order of the points $\mathbf{x}(1), \mathbf{x}(2), \ldots, \mathbf{x}(\mathbf{npoint})$. If this error exit still results, it is likely that d02sa is not a suitable method for solving the problem, or the initial choice of parameters is very poor, or the accuracy requirement specified by $\mathbf{e}(i)$, for $i = 1, 2, \ldots, n$, is too stringent.

ifail = 5

In the integration from $\mathbf{x}(1)$ to $\mathbf{x}(\mathbf{npoint})$ with the initial or final parameters, an initial step could not be found to start the integration on one of the intervals $\mathbf{x}(i)$ to $\mathbf{x}(i+1)$. Consider reversing the order of the points. If this error exit still results it is likely that d02sa is not a suitable function for solving the problem, or the initial choice of parameters is very poor, or the accuracy requirement specified by $\mathbf{e}(i)$, for $i = 1, 2, \ldots, n$, is much too stringent.

ifail $= 6$

In the integration from $\mathbf{x}(1)$ to $\mathbf{x}(\mathbf{npoint})$ with the initial or final parameters, the solution exceeded **ymax** in magnitude (when $\mathbf{ymax} > 0$). It is likely that the initial choice of parameters was very poor or **ymax** was incorrectly set.

Note: on an error with **ifail** $= 4$, 5 or 6 with the initial parameters, the interval in which failure occurs is contained in $\mathbf{swp}(\mathbf{npoint}, 1)$. If a (sub)program **monit** similar to the one in Section 9 is being used then it is a simple matter to distinguish between errors using the initial and final parameters. None of the error exits **ifail** $= 4$, 5 or 6 should occur on the **final** integration (when computing the solution) as this integration has already been performed previously with exactly the same parameters p_i, for $i = 1, 2, \ldots, m$. Seek expert help if this error occurs.

ifail $= 7$

On calculating the initial approximation to the Jacobian, the constraints were violated.

ifail $= 8$

On perturbing the parameters when calculating the initial approximation to the Jacobian, the condition $\mathbf{x}(1) < \mathbf{x}(2) < \cdots < \mathbf{x}(\mathbf{npoint})$ (or $\mathbf{x}(1) > \mathbf{x}(2) > \cdots > \mathbf{x}(\mathbf{npoint})$) is violated.

ifail $= 9$

On calculating the initial approximation to the Jacobian, the integration step size was reduced too far to make further progress (see **ifail** $= 4$).

ifail $= 10$

On calculating the initial approximation to the Jacobian, the initial integration step size on some interval was too small (see **ifail** $= 5$).

ifail $= 11$

On calculating the initial approximation to the Jacobian, the solution of the system of differential equations exceeded **ymax** in magnitude (when $\mathbf{ymax} > 0$).

Note: all the error exits **ifail** $= 7$, 8, 9, 10 and 11 can be treated by reducing the size of some or all the elements of **dp**.

ifail $= 12$

On calculating the initial approximation to the Jacobian, a column of the Jacobian is found to be insignificant. This could be due to an element $\mathbf{dp}(i)$ being too small (but nonzero) or the solution having no dependence on one of the parameters (a programming error).

Note: on an error exit with **ifail** $= 7$, 8, 9, 10, 11 or 12, if a perturbation of the parameter p_i is the cause of the error then $\mathbf{swp}(\mathbf{npoint}, 1)$ will contain the floating-point value of i.

ifail $= 13$

After calculating the initial approximation to the Jacobian, the calculation of its singular value decomposition failed. It is likely that the error will never occur as it is usually associated with the Jacobian having multiple singular values. To remedy the error it should only be necessary to change the initial parameters. If the error persists it is likely that the problem has not been correctly formulated.

ifail $= 14$

The Newton iteration has failed to converge after exercising all its options. You are strongly recommended to monitor the progress of the iteration via the (sub)program **monit**. There are many possible reasons for the iteration not converging. Amongst the most likely are:

(a) there is no solution;

(b) the initial parameters are too far away from the correct parameters;

(c) the problem is too ill-conditioned as an initial value problem for Newton's method to choose suitable corrections;

(d) the accuracy requirements for convergence are too restrictive, that is some of the components of **pe** (and maybe **pf**) are too small – in this case the final value of this norm output via (sub)program **monit** will usually be very small; or

(e) the initial parameters are so close to the solution parameters p that the Newton iteration cannot find improved parameters. The norm output by (sub)program **monit** should be very small.

ifail = 15

The number of iterations permitted by **icount** has been exceeded (in the case when **icount** > 0 on entry).

ifail = 16
ifail = 17
ifail = 18
ifail = 19

These indicate that there has been a serious error in one of the auxiliary functions "temp_tag_xref_d02saz d02saw d02sax d02sau"done**d02saz d02saw d02sax d02sau**missing entity d02saz_d02saw_d02sax_d02sau respectively. Check all (sub)program calls and array dimensions. Seek expert help.

7 Accuracy

If the iteration converges, the accuracy to which the unknown parameters are determined is usually close to that specified by you. The accuracy of the solution (output via (sub)program **prsol**) depends on the error tolerances $\mathbf{e}(i)$, for $i = 1, 2, \ldots, n$. You are strongly recommended to vary all tolerances to check the accuracy of the parameters p and the solution y.

8 Further Comments

The time taken by d02sa depends on the complexity of the system of differential equations and on the number of iterations required. In practice, the integration of the differential system (1) is usually by far the most costly process involved. The computing time for integrating the differential equations can sometimes depend critically on the quality of the initial estimates for the parameters p. If it seems that too much computing time is required and, in particular, if the values of the residuals (output in (sub)program **monit**) are much larger than expected given your knowledge of the expected solution, then the coding of the user-supplied (sub)programs **fcn**, **eqn**, **range** and **bc** should be checked for errors. If no errors can be found then an independent attempt should be made to improve the initial estimates p.

In the case of an error exit in the integration of the differential system indicated by **ifail** = 4, 5, 9 or 10 you are strongly recommended to perform trial integrations with d02pd to determine the effects of changes of the local error tolerances and of changes to the initial choice of the parameters p_i, for $i = 1, 2 \ldots, m$ (that is the initial choice of p).

It is possible that by following the advice given in Section 6 an error exit with **ifail** = 7, 8, 9, 10 or 11 might be followed by one with **ifail** = 12 (or vice-versa) where the advice given is the opposite. If you are unable to refine the choice of $\mathbf{dp}(i)$, for $i = 1, 2, \ldots, n$, such that both these types of exits are avoided then the problem should be rescaled if possible or the method must be abandoned.

The choice of the 'floor' values $\mathbf{pf}(i)$, for $i = 1, 2, \ldots, m$, may be critical in the convergence of the Newton iteration. For each value i, the initial choice of p_i and the choice of $\mathbf{pf}(i)$ should not both be very small unless it is expected that the final parameter p_i will be very small and that it should be determined accurately in a **relative** sense.

For many problems it is critical that a good initial estimate be found for the parameters p or the iteration will not converge or may even break down with an error exit. There are many mathematical techniques which obtain good initial estimates for p in simple cases but which may fail to produce useful estimates in harder cases. If no such technique is available it is recommended that you try a continuation (homotopy)

technique preferably based on a physical parameter (e.g., the Reynolds or Prandtl number is often a suitable continuation parameter). In a continuation method a sequence of problems is solved, one for each choice of the continuation parameter, starting with the problem of interest. At each stage the parameters p calculated at earlier stages are used to compute a good initial estimate for the parameters at the current stage (see Hall and Watt 1976 for more details).

9 Example

d02sa_bc.m

```
function [g1, g2] = bc(p, m, n)
    g1 = [0, 0.5, p(1)];
    g2 = [0, 0.45, -1.2];
```

d02sa_constr.m

```
function result = constr(p,n)
  result = true;
  for i=1:n
    if p(i) < 0
      result = false;
    end
  end
  if p(3) > 5
    result = false;
  end
```

d02sa_eqn.m

```
function eq = eqn(q,p,m)
  eq = zeros(q,1);
  eq(1) =  0.02 - p(4) - 1.0d-5*p(3);
```

d02sa_fcn.m

```
function f = fcn(x, y, n, p, m, interval)
      f = zeros(n,1);
      f(1) = tan(y(3));
      if (interval == 1)
         f(2) = -0.032d0*tan(y(3))/y(2) - 0.02d0*y(2)/cos(y(3));
         f(3) = -0.032d0/y(2)^2;
      else
         f(2) = -p(2)*tan(y(3))/y(2) - p(4)*y(2)/cos(y(3));
         f(3) = -p(2)/y(2)^2;
      end
```

d02sa_monit.m

```
function monit(istate, iflag, ifail1, p, m, f, pnorm, pnorm1, eps, d)
  % Dummy
```

d02sa_prsol.m

```
function z = prsol(z, y, n)
  global prsolCalled;

  if (size(prsolCalled) == 0)
    fprintf('\n   x     ');
    for i=1:n
```

```
      fprintf('   y(%d)  ', i);
    end
    fprintf('\n');
    prsolCalled = true;
  end

  fprintf(' %+6.4f ',z);
  for i=1:n
    fprintf(' %+6.4f ', y(i));
  end
  fprintf('\n');

  z = z+0.5;
  if (abs(z-5) < 0.25)
    z = 5;
  end
```

d02sa_range.m

```
function x = range(npoint, p, m)
      x=zeros(npoint,1);
      x(1) = 0;
      x(2) = p(3);
      x(3) = 5;
```

```
p = [1.2;
     0.032;
     2.5;
     0.02];
n1 = int32(3);
pe = [0.001;
     0.001;
     0.001;
     0.001];
pf = [1e-06;
     1e-06;
     1e-06;
     1e-06];
e = [1e-05;
     1e-05;
     1e-05];
dp = [0;
     0;
     0;
     0];
npoint = int32(3);
swp = zeros(3, 6);
icount = int32(0);
ymax = 0;
[pOut, pfOut, dpOut, swpOut, w, ifail] = ...
    d02sa(p, n1, pe, pf, e, dp, npoint, swp, icount, 'd02sa_range', ...
    'd02sa_bc', 'd02sa_fcn', 'd02sa_eqn', 'd02sa_constr', ymax, ...
    'd02sa_monit', 'd02sa_prsol')
```

```
x        y(1)     y(2)     y(3)
 +0.0000  +0.0000  +0.5000  +1.1753
 +0.5000  +1.0881  +0.4127  +1.0977
 +1.0000  +1.9501  +0.3310  +0.9802
 +1.5000  +2.5768  +0.2582  +0.7918
 +2.0000  +2.9606  +0.2019  +0.4796
 +2.5000  +3.0958  +0.1773  +0.0245
 +3.0000  +2.9861  +0.1935  -0.4353
 +3.5000  +2.6289  +0.2409  -0.7679
 +4.0000  +2.0181  +0.3047  -0.9767
 +4.5000  +1.1454  +0.3759  -1.1099
 +5.0000  -0.0000  +0.4500  -1.2000
```

```
pOut =
    1.1753
    0.0305
    2.3302
    0.0200
pfOut =
   1.0e-06 *
    1.0000
    1.0000
    1.0000
    1.0000
dpOut =
   1.0e-04 *
    0.1200
    0.0032
    0.2500
    0.0020
swpOut =
    0.1744         0         0         0         0         0
    0.0749         0         0    2.3302    2.4121         0
         0         0         0    5.0000    5.0000         0
w =
  Columns 1 through 7
   -0.0000   -0.0000    1.1753    0.0000    0.0000    0.0000  115.5327
    0.0000    0.0000    0.0305   -0.0000    0.0000   -0.0000    2.0191
   -0.0000   -0.0004    2.3298    0.0001   -0.0000    0.0000    0.9570
    0.0000    0.0000    0.0200   -0.0000   -0.0001    0.0000    0.0034
  Columns 8 through 14
    0.0087   -0.9882   -0.0875    0.1102    0.0605    0.2248    0.9734
    0.4953    0.1117   -0.9087    0.3686   -0.1611    0.6096   -0.1767
    1.0449   -0.1048   -0.1420   -0.6437   -0.7447   -0.7601    0.1461
  290.5169   -0.0004   -0.3827   -0.6616    0.6449    0.0067   -0.0004
  Columns 15 through 21
    0.0010    0.0453   -0.0000         0   -2.5722    0.0327   -2.5576
    0.0025    0.7727    0.4500    0.4500    0.1493    0.4481    0.1492
   -0.0066    0.6331   -1.2000   -1.2000   -0.1504   -1.1981   -0.1517
   -1.0000   -0.0022    0.0200   -0.0000    1.0000    0.0000         0
  Columns 22 through 28
    0.0684   -2.5416   -0.0000    0.0000    0.0000   -0.0000   -0.0000
    0.4460    0.1492    0.0000    0.0000    0.0000    0.4500    0.0000
   -1.1959   -0.1531    0.0000    0.0000    0.0000   -1.2000    0.0000
         0         0         0         0         0         0         0
  Columns 29 through 35
    0.0000    0.0000         0  -25.8525 -111.0799   -0.1128   -5.2395
    0.0000    0.0000         0    1.5154   12.9417    0.0063   -0.6098
    0.0000    0.0000         0   -2.4285  -11.8253   -0.0059   -1.1598
         0         0         0         0         0   -0.0000   -1.0000
ifail =
           0
```