NAG Toolbox

nag_ode_sl2_breaks_funs (d02ke)

1 Purpose

nag_ode_sl2_breaks_funs (d02ke) finds a specified eigenvalue of a regular or singular second-order Sturm–Liouville system on a finite or infinite interval, using a Pruefer transformation and a shooting method. It also reports values of the eigenfunction and its derivatives. Provision is made for discontinuities in the coefficient functions or their derivatives.

2 Syntax

```
[match, elam, delam, hmax, maxit, ifail] = nag_ode_sl2_breaks_funs(xpoint,
match, coeffn, bdyval, k, tol, elam, delam, hmax, monit, report, 'm', m,
'maxit', maxit, 'maxfun', maxfun)
```

```
[match, elam, delam, hmax, maxit, ifail] = d02ke(xpoint, match, coeffn, bdyval,
k, tol, elam, delam, hmax, monit, report, 'm', m, 'maxit', maxit, 'maxfun',
maxfun)
```

3 Description

nag_ode_sl2_breaks_funs (d02ke) has essentially the same purpose as nag_ode_sl2_breaks_vals (d02kd) with minor modifications to enable values of the eigenfunction to be obtained after convergence to the eigenvalue has been achieved.

It first finds a specified eigenvalue $\tilde{\lambda}$ of a Sturm–Liouville system defined by a self-adjoint differential equation of the second-order

$$(p(x)y')' + q(x;\lambda)y = 0, \quad a < x < b,$$

together with appropriate boundary conditions at the two, finite or infinite, end points a and b. The functions p and q, which are real-valued, are defined by **coeffn**. The boundary conditions must be defined by **bdyval**, and, in the case of a singularity at a or b, take the form of an asymptotic formula for the solution near the relevant end point.

When the final estimate $\lambda = \tilde{\lambda}$ of the eigenvalue has been found, the function integrates the differential equation once more with that value of λ, and with initial conditions chosen so that the integral

$$S = \int_a^b y(x)^2 \frac{\partial q}{\partial \lambda}(x;\lambda)\,dx$$

is approximately one. When $q(x;\lambda)$ is of the form $\lambda w(x) + q(x)$, which is the most common case, S represents the square of the norm of y induced by the inner product

$$\langle f, g \rangle = \int_a^b f(x)g(x)w(x)\,dx,$$

with respect to which the eigenfunctions are mutually orthogonal. This normalization of y is only approximate, but experience shows that S generally differs from unity by only one or two per cent.

During this final integration the **report** is called at each integration mesh point x. Sufficient information is returned to permit you to compute $y(x)$ and $y'(x)$ for printing or plotting. For reasons described in Section 9.2, nag_ode_sl2_breaks_funs (d02ke) passes across to **report**, not y and y', but the Pruefer variables β, ϕ and ρ on which the numerical method is based. Their relationship to y and y' is given by the equations

$$p(x)y' = \sqrt{\beta}\exp\left(\frac{\rho}{2}\right)\cos\left(\frac{\phi}{2}\right), \quad y = \frac{1}{\sqrt{\beta}}\exp\left(\frac{\rho}{2}\right)\sin\left(\frac{\phi}{2}\right).$$

A specimen **report** is given in Section 10 below.

For the theoretical basis of the numerical method to be valid, the following conditions should hold on the coefficient functions:

(a) $p(x)$ must be nonzero and must not change sign throughout the interval (a, b); and,

(b) $\dfrac{\partial q}{\partial \lambda}$ must not change sign throughout the interval (a, b) for all relevant values of λ, and must not be identically zero as x varies, for any λ.

Points of discontinuity in the functions p and q or their derivatives are allowed, and should be included as 'break-points' in the array **xpoint**.

A good account of the theory of Sturm–Liouville systems, with some description of Pruefer transformations, is given in Chapter X of Birkhoff and Rota (1962). An introduction to the use of Pruefer transformations for the numerical solution of eigenvalue problems arising from physics and chemistry is given in Bailey (1966).

The scaled Pruefer method is described in a short note by Pryce and Hargrave (1977) and in some detail in the technical report by Pryce (1981).

4 References

Abramowitz M and Stegun I A (1972) *Handbook of Mathematical Functions* (3rd Edition) Dover Publications

Bailey P B (1966) Sturm–Liouville eigenvalues via a phase function *SIAM J. Appl. Math.* **14** 242–249

Banks D O and Kurowski I (1968) Computation of eigenvalues of singular Sturm–Liouville Systems *Math. Comput.* **22** 304–310

Birkhoff G and Rota G C (1962) *Ordinary Differential Equations* Ginn & Co., Boston and New York

Pryce J D (1981) Two codes for Sturm–Liouville problems *Technical Report CS-81-01* Department of Computer Science, Bristol University

Pryce J D and Hargrave B A (1977) The scaled PrÏfer method for one-parameter and multi-parameter eigenvalue problems in ODEs *IMA Numerical Analysis Newsletter* **1(3)**

5 Parameters

5.1 Compulsory Input Parameters

1: **xpoint**(**m**) – REAL (KIND=nag_wp) array

The points where the boundary conditions computed by **bdyval** are to be imposed, and also any break-points, i.e., **xpoint**(1) to **xpoint**(m) must contain values $x_1, \ldots, x_m$ such that

$$x_1 \leq x_2 < x_3 < \cdots < x_{m-1} \leq x_m$$

with the following meanings:

(a) x_1 and x_m are the left- and right-hand end points, a and b, of the domain of definition of the Sturm–Liouville system if these are finite. If either a or b is infinite, the corresponding value x_1 or x_m may be a more-or-less arbitrarily 'large' number of appropriate sign.

(b) x_2 and x_{m-1} are the Boundary Matching Points (BMPs), that is the points at which the left and right boundary conditions computed in **bdyval** are imposed.

If the left-hand end point is a regular point then you should set $x_2 = x_1$ $(= a)$, while if it is a singular point you must set $x_2 > x_1$. Similarly $x_{m-1} = x_m$ $(= b)$ if the right-hand end point is regular, and $x_{m-1} < x_m$ if it is singular.

(c) The remaining $m - 4$ points $x_3, \ldots, x_{m-2}$, if any, define 'break-points' which divide the interval $[x_2, x_{m-1}]$ into $m - 3$ sub-intervals

$$i_1 = [x_2, x_3], \ldots, i_{m-3} = [x_{m-2}, x_{m-1}].$$

Numerical integration of the differential equation is stopped and restarted at each break-point. In simple cases no break-points are needed. However, if $p(x)$ or $q(x;\lambda)$ are given by different formulae in different parts of the interval, then integration is more efficient if the range is broken up by break-points in the appropriate way. Similarly points where any jumps occur in $p(x)$ or $q(x;\lambda)$, or in their derivatives up to the fifth-order, should appear as break-points.

Examples are given in Section 9 and Section 10. **xpoint** determines the position of the Shooting Matching Point (SMP), as explained in Section 9.3.

Constraint: $\mathbf{xpoint}(1) \leq \mathbf{xpoint}(2) < \cdots < \mathbf{xpoint}(\mathbf{m}-1) \leq \mathbf{xpoint}(\mathbf{m})$.

2: **match** – INTEGER

Must be set to the index of the 'break-point' to be used as the matching point (see Section 9.3). If **match** is set to a value outside the range $[2, m-1]$ then a default value is taken, corresponding to the break-point nearest the centre of the interval $[\mathbf{xpoint}(2), \mathbf{xpoint}(m-1)]$.

3: **coeffn** – SUBROUTINE, supplied by the user.

coeffn must compute the values of the coefficient functions $p(x)$ and $q(x;\lambda)$ for given values of x and λ. Section 3 states the conditions which p and q must satisfy. See Section 9.4 and Section 10 for examples.

```
[p, q, dqdl] = coeffn(x, elam, jint)
```

Input Parameters

1: **x** – REAL (KIND=nag_wp)

The current value of x.

2: **elam** – REAL (KIND=nag_wp)

The current trial value of the eigenvalue argument λ.

3: **jint** – INTEGER

The index j of the sub-interval i_j (see specification of **xpoint**) in which x lies.

Output Parameters

1: **p** – REAL (KIND=nag_wp)

The value of $p(x)$ for the current value of x.

2: **q** – REAL (KIND=nag_wp)

The value of $q(x;\lambda)$ for the current value of x and the current trial value of λ.

3: **dqdl** – REAL (KIND=nag_wp)

The value of $\frac{\partial q}{\partial \lambda}(x;\lambda)$ for the current value of x and the current trial value of λ. However **dqdl** is only used in error estimation and, in the rare cases where it may be difficult to evaluate, an approximation (say to within 20%) will suffice.

4: **bdyval** – SUBROUTINE, supplied by the user.

bdyval must define the boundary conditions. For each end point, **bdyval** must return (in **yl** or **yr**) values of $y(x)$ and $p(x)y'(x)$ which are consistent with the boundary conditions at the end points; only the ratio of the values matters. Here x is a given point (**xl** or **xr**) equal to, or close to, the end point.

For a **regular** end point (a, say), $x = a$, a boundary condition of the form

$$c_1 y(a) + c_2 y'(a) = 0$$

can be handled by returning constant values in **yl**, e.g., **yl**$(1) = c_2$ and **yl**$(2) = -c_1 p(a)$.

For a **singular** end point however, **yl**(1) and **yl**(2) will in general be functions of **xl** and **elam**, and **yr**(1) and **yr**(2) functions of **xr** and **elam**, usually derived analytically from a power-series or asymptotic expansion. Examples are given in Section 9.5 and Section 10.

```
[yl, yr] = bdyval(xl, xr, elam)
```

Input Parameters

1: **xl** – REAL (KIND=nag_wp)

If a is a regular end point of the system (so that $a = x_1 = x_2$), then **xl** contains a. If a is a singular point (so that $a \leq x_1 < x_2$), then **xl** contains a point x such that $x_1 < x \leq x_2$.

2: **xr** – REAL (KIND=nag_wp)

If b is a regular end point of the system (so that $x_{m-1} = x_m = b$), then **xr** contains b. If b is a singular point (so that $x_{m-1} < x_m \leq b$), then **xr** contains a point x such that $x_{m-1} \leq x < x_m$.

3: **elam** – REAL (KIND=nag_wp)

The current trial value of λ.

Output Parameters

1: **yl**(**3**) – REAL (KIND=nag_wp) array

yl(1) and **yl**(2) should contain values of $y(x)$ and $p(x)y'(x)$ respectively (not both zero) which are consistent with the boundary condition at the left-hand end point, given by $x =$ **xl**. **yl**(3) should not be set.

2: **yr**(**3**) – REAL (KIND=nag_wp) array

yr(1) and **yr**(2) should contain values of $y(x)$ and $p(x)y'(x)$ respectively (not both zero) which are consistent with the boundary condition at the right-hand end point, given by $x =$ **xr**. **yr**(3) should not be set.

5: **k** – INTEGER

k, the index of the required eigenvalue when the eigenvalues are ordered

$$\lambda_0 < \lambda_1 < \lambda_2 < \cdots < \lambda_k < \cdots .$$

Constraint: **k** ≥ 0.

6: **tol** – REAL (KIND=nag_wp)

The tolerance argument which determines the accuracy of the computed eigenvalue. The error estimate held in **delam** on exit satisfies the mixed absolute/relative error test

$$\mathbf{delam} \leq \mathbf{tol} \times \max(1.0, |\mathbf{elam}|), \quad (1)$$

where **elam** is the final estimate of the eigenvalue. **delam** is usually somewhat smaller than the right-hand side of (1) but not several orders of magnitude smaller.

Constraint: **tol** > 0.0.

7: **elam** – REAL (KIND=nag_wp)

An initial estimate of the eigenvalue $\tilde{\lambda}$.

8: **delam** – REAL (KIND=nag_wp)

An indication of the scale of the problem in the λ-direction. **delam** holds the initial 'search step' (positive or negative). Its value is not critical, but the first two trial evaluations are made at **elam** and **elam** + **delam**, so the function will work most efficiently if the eigenvalue lies between these values. A reasonable choice (if a closer bound is not known) is half the distance between adjacent eigenvalues in the neighbourhood of the one sought. In practice, there will often be a problem, similar to the one in hand but with known eigenvalues, which will help one to choose initial values for **elam** and **delam**.

If **delam** $= 0.0$ on entry, it is given the default value of $0.25 \times \max(1.0, |$**elam**$|)$.

9: **hmax**$(\mathbf{2}, \mathbf{m})$ – REAL (KIND=nag_wp) array

hmax$(1, j)$ should contain a maximum step size to be used by the differential equation code in the jth sub-interval i_j (as described in the specification of argument **xpoint**), for $j = 1, 2, \ldots, m - 3$. If it is zero the function generates a maximum step size internally.

It is recommended that **hmax**$(1, j)$ be set to zero unless the coefficient functions p and q have features (such as a narrow peak) within the jth sub-interval that could be 'missed' if a long step were taken. In such a case **hmax**$(1, j)$ should be set to about half the distance over which the feature should be observed. Too small a value will increase the computing time for the function. See Section 9 for further suggestions.

The rest of the array is used as workspace.

10: **monit** – SUBROUTINE, supplied by the NAG Library or the user.

monit is called by nag_ode_sl2_breaks_funs (d02ke) at the end of each root-finding iteration and allows you to monitor the course of the computation by printing out the arguments (see Section 10 for an example).

If no monitoring is required, the dummy (sub)program nag_ode_sl2_reg_finite_dummy_monit (d02kay) may be used. (nag_ode_sl2_reg_finite_dummy_monit (d02kay) is included in the NAG Toolbox.)

```
monit(nit, iflag, elam, finfo)
```

Input Parameters

1: **nit** – INTEGER

The current value of the argument **maxit** of nag_ode_sl2_breaks_funs (d02ke), this is decreased by one at each iteration.

2: **iflag** – INTEGER

Describes what phase the computation is in.

iflag < 0
An error occurred in the computation at this iteration; an error exit from nag_ode_sl2_breaks_funs (d02ke) with **ifail** $= -$**iflag** will follow.

iflag $= 1$
The function is trying to bracket the eigenvalue $\tilde{\lambda}$.

iflag $= 2$
The function is converging to the eigenvalue $\tilde{\lambda}$ (having already bracketed it).

3: **elam** – REAL (KIND=nag_wp)

The current trial value of λ.

4: **finfo**(**15**) – REAL (KIND=nag_wp) array

Information about the behaviour of the shooting method, and diagnostic information in the case of errors. It should not normally be printed in full if no error has occurred (that is, if **iflag** > 0), though the first few components may be of interest to you. In case of an error (**iflag** < 0) all the components of **finfo** should be printed.

The contents of **finfo** are as follows:

finfo(1)
The current value of the 'miss-distance' or 'residual' function $f(\lambda)$ on which the shooting method is based. (See Section 9.2 for further information.) **finfo**(1) is set to zero if **iflag** < 0.

finfo(2)
An estimate of the quantity $\partial\lambda$ defined as follows. Consider the perturbation in the miss-distance $f(\lambda)$ that would result if the local error in the solution of the differential equation were always positive and equal to its maximum permitted value. Then $\partial\lambda$ is the perturbation in λ that would have the same effect on $f(\lambda)$. Thus, at the zero of $f(\lambda), |\partial\lambda|$ is an approximate bound on the perturbation of the zero (that is the eigenvalue) caused by errors in numerical solution. If $\partial\lambda$ is very large then it is possible that there has been a programming error in **coeffn** such that q is independent of λ. If this is the case, an error exit with **ifail** $= 5$ should follow. **finfo**(2) is set to zero if **iflag** < 0.

finfo(3)
The number of internal iterations, using the same value of λ and tighter accuracy tolerances, needed to bring the accuracy (that is, the value of $\partial\lambda$) to an acceptable value. Its value should normally be 1.0, and should almost never exceed 2.0.

finfo(4)
The number of calls to **coeffn** at this iteration.

finfo(5)
The number of successful steps taken by the internal differential equation solver at this iteration. A step is successful if it is used to advance the integration.

finfo(6)
The number of unsuccessful steps used by the internal integrator at this iteration.

finfo(7)
The number of successful steps at the maximum step size taken by the internal integrator at this iteration.

finfo(8)
Not used.

finfo(9) to **finfo**(15)
Set to zero, unless **iflag** < 0 in which case they hold the following values describing the point of failure:

finfo(9)
The index of the sub-interval where failure occurred, in the range 1 to $m - 3$. In case of an error in **bdyval**, it is set to 0 or $m - 2$ depending on whether the left or right boundary condition caused the error.

finfo(10)
The value of the independent variable, x, the point at which the error occurred. In case of an error in **bdyval**, it is set to the value of **xl** or **xr** as appropriate (see the specification of **bdyval**).

finfo(11), **finfo**(12), **finfo**(13)
The current values of the Pruefer dependent variables β, ϕ and ρ respectively. These are set to zero in case of an error in **bdyval**.

finfo(14)
The local-error tolerance being used by the internal integrator at the point of failure. This is set to zero in the case of an error in **bdyval**.

finfo(15)
The last integration mesh point. This is set to zero in the case of an error in **bdyval**.

11: **report** – SUBROUTINE, supplied by the user.

report provides the means by which you may compute the eigenfunction $y(x)$ and its derivative at each integration mesh point x. (See Section 9 for an example.)

```
report(x, v, jint)
```

Input Parameters

1: **x** – REAL (KIND=nag_wp)

The current value of the independent variable x. See Section 9.3 for the order in which values of x are supplied.

2: **v**(**3**) – REAL (KIND=nag_wp) array

v(1), **v**(2), **v**(3) hold the current values of the Pruefer variables β, ϕ, ρ respectively.

3: **jint** – INTEGER

Indicates the sub-interval between break-points in which **x** lies exactly as for **coeffn**, **except** that at the extreme left-hand end point (when $x =$ **xpoint**(2)) **jint** is set to 0 and at the extreme right-hand end point (when $x = x_r =$ **xpoint**$(m-1)$) **jint** is set to $m-2$.

5.2 Optional Input Parameters

1: **m** – INTEGER

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

The number of points in the array **xpoint**.

Constraint: **m** ≥ 4.

2: **maxit** – INTEGER

Suggested value: **maxit** $= 0$.

Default: 0

A bound on n_r, the number of root-finding iterations allowed, that is the number of trial values of λ that are used. If **maxit** ≤ 0, no such bound is assumed. (See also **maxfun**.)

3: **maxfun** – INTEGER

Suggested value: **maxfun** $= 0$.

maxfun and **maxit** may be used to limit the computational cost of a call to nag_ode_sl2_breaks_funs (d02ke), which is roughly proportional to $n_r \times n_f$.

Default: 0

A bound on n_f, the number of calls to **coeffn** made in any one root-finding iteration. If **maxfun** ≤ 0, no such bound is assumed.

5.3 Output Parameters

1: **match** – INTEGER

The index of the break-point actually used as the matching point.

2: **elam** – REAL (KIND=nag_wp)

The final computed estimate, whether or not an error occurred.

3: **delam** – REAL (KIND=nag_wp)

If **ifail** $= 0$, **delam** holds an estimate of the absolute error in the computed eigenvalue, that is $|\tilde{\lambda} - \mathbf{elam}| \simeq \mathbf{delam}$. (In Section 9.2 we discuss the assumptions under which this is true.) The true error is rarely more than twice, or less than a tenth, of the estimated error.

If **ifail** $\neq 0$, **delam** may hold an estimate of the error, or its initial value, depending on the value of **ifail**. See Section 6 for further details.

4: **hmax**$(\mathbf{2}, \mathbf{m})$ – REAL (KIND=nag_wp) array

hmax$(1, m-1)$ and **hmax**$(1, m)$ contain the sensitivity coefficients σ_l, σ_r, described in Section 9.6. Other entries contain diagnostic output in the case of an error exit (see Section 6).

5: **maxit** – INTEGER

Suggested value: **maxit** $= 0$.

Default: 0

Will have been decreased by the number of iterations actually performed, whether or not it was positive on entry.

6: **ifail** – INTEGER

ifail $= 0$ unless the function detects an error (see Section 5).

6 Error Indicators and Warnings

Errors or warnings detected by the function:

ifail $= 1$

A argument error. All arguments (except **ifail**) are left unchanged. The reason for the error is shown by the value of **hmax**$(2, 1)$ as follows:

hmax$(2, 1) = 1$: **m** < 4;

hmax$(2, 1) = 2$: **k** < 0;

hmax$(2, 1) = 3$: **tol** ≤ 0.0;

hmax$(2, 1) = 4$: **xpoint**(1) to **xpoint**(m) are not in ascending order. **hmax**$(2, 2)$ gives the position i in **xpoint** where this was detected.

ifail $= 2$

At some call to **bdyval**, invalid values were returned, that is, either **yl**$(1) =$ **yl**$(2) = 0.0$, or **yr**$(1) =$ **yr**$(2) = 0.0$ (a programming error in **bdyval**). See the last call of **monit** for details.

This error exit will also occur if $p(x)$ is zero at the point where the boundary condition is imposed. Probably **bdyval** was called with **xl** equal to a singular end point a or with **xr** equal to a singular end point b.

ifail $= 3$

At some point between **xl** and **xr** the value of $p(x)$ computed by **coeffn** became zero or changed sign. See the last call of **monit** for details.

ifail $= 4$

maxit > 0 on entry, and after **maxit** iterations the eigenvalue had not been found to the required accuracy.

ifail $= 5$

The 'bracketing' phase (with argument **iflag** of the **monit** equal to 1) failed to bracket the eigenvalue within ten iterations. This is caused by an error in formulating the problem (for example, q is independent of λ), or by very poor initial estimates of **elam** and **delam**.

On exit, **elam** and **elam** $+$ **delam** give the end points of the interval within which no eigenvalue was located by the function.

ifail $= 6$

maxfun > 0 on entry, and the last iteration was terminated because more than **maxfun** calls to **coeffn** were used. See the last call of **monit** for details.

ifail $= 7$

To obtain the desired accuracy the local error tolerance was set so small at the start of some sub-interval that the differential equation solver could not choose an initial step size large enough to make significant progress. See the last call of **monit** for diagnostics.

ifail $= 8$

At some point inside a sub-interval the step size in the differential equation solver was reduced to a value too small to make significant progress (for the same reasons as with **ifail** $= 7$). This could be due to pathological behaviour of $p(x)$ and $q(x;\lambda)$ or to an unreasonable accuracy requirement or to the current value of λ making the equations 'stiff'. See the last call of **monit** for details.

ifail $= 9$ (*warning*)

tol is too small for the problem being solved and the ***machine precision*** is being used. The final value of **elam** should be a very good approximation to the eigenvalue.

ifail $= 10$

nag_roots_contfn_brent_rcomm (c05az), called by nag_ode_sl2_breaks_funs (d02ke), has terminated with the error exit corresponding to a pole of the residual function $f(\lambda)$. This error exit should not occur, but if it does, try solving the problem again with a smaller value for **tol**.

ifail $= 11$
ifail $= 12$

A serious error has occurred in an internal call. Check all (sub)program calls and array dimensions. Seek expert help.

ifail $= -99$

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

ifail $= -399$

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

ifail $= -999$

Dynamic memory allocation failed.

Note: error exits with **ifail** $= 2$, 3, 6, 7, 8 or 11 are caused by being unable to set up or solve the differential equation at some iteration and will be immediately preceded by a call of **monit** giving diagnostic information. For other errors, diagnostic information is contained in **hmax**$(2,j)$, for $j = 1, 2, \ldots, m$, where appropriate.

7 Accuracy

See the discussion in Section 9.2.

8 Further Comments

8.1 Timing

The time taken by nag_ode_sl2_breaks_funs (d02ke) depends on the complexity of the coefficient functions, whether they or their derivatives are rapidly changing, the tolerance demanded, and how many iterations are needed to obtain convergence. The amount of work per iteration is roughly doubled when **tol** is divided by 16. To make the most economical use of the function, one should try to obtain good initial values for **elam** and **delam**, and, where appropriate, good asymptotic formulae. Also the boundary matching points should not be set unnecessarily close to singular points. The extra time needed to compute the eigenfunction is principally the cost of one additional integration once the eigenvalue has been found.

8.2 General Description of the Algorithm

A shooting method, for differential equation problems containing unknown parameters, relies on the construction of a 'miss-distance function', which for given trial values of the parameters measures how far the conditions of the problem are from being met. The problem is then reduced to one of finding the values of the parameters for which the miss-distance function is zero, that is to a root-finding process. Shooting methods differ mainly in how the miss-distance is defined.

nag_ode_sl2_breaks_funs (d02ke) defines a miss-distance $f(\lambda)$ based on the rotation about the origin of the point $\mathbf{p}(x) = (p(x)y'(x), y(x))$ in the Phase Plane as the solution proceeds from a to b. The **boundary conditions** define the ray (i.e., two-sided line through the origin) on which $p(x)$ should start, and the ray on which it should finish. The **eigenvalue** k defines the total number of half-turns it should make. Numerical solution is actually done by 'shooting forward' from $x = a$ and 'shooting backward' from $x = b$ to a matching point $x = c$. Then $f(\lambda)$ is taken as the angle between the rays to the two resulting points $P_a(c)$ and $P_b(c)$. A relative scaling of the py' and y axes, based on the behaviour of the coefficient functions p and q, is used to improve the numerical behaviour.

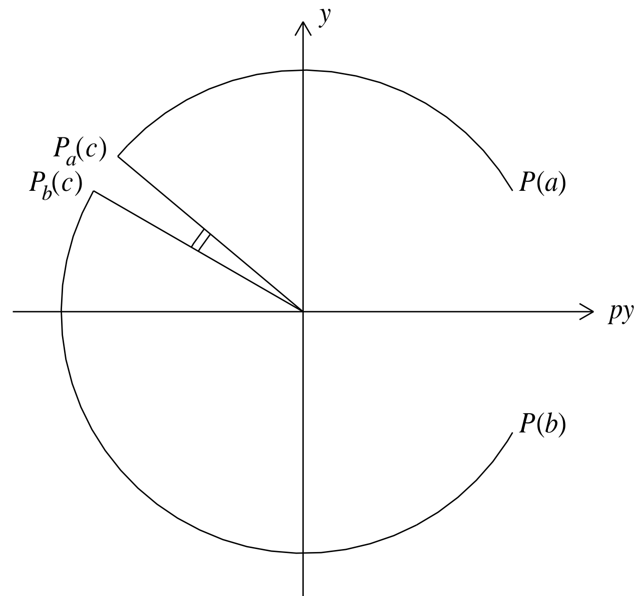

Figure 1

The resulting function $f(\lambda)$ is monotonic over $-\infty < \lambda < \infty$, increasing if $\frac{\partial q}{\partial \lambda} > 0$ and decreasing if $\frac{\partial q}{\partial \lambda} < 0$, with a unique zero at the desired eigenvalue $\tilde{\lambda}$. The function measures $f(\lambda)$ in units of a half-turn. This means that as λ increases, $f(\lambda)$ varies by about 1 as each eigenvalue is passed. (This feature implies that the values of $f(\lambda)$ at successive iterations – especially in the early stages of the iterative process – can be used with suitable extrapolation or interpolation to help the choice of initial estimates for eigenvalues near to the one currently being found.)

The function actually computes a value for $f(\lambda)$ with errors, arising from the local errors of the differential equation code and from the asymptotic formulae provided by you if singular points are involved. However, the error estimate output in **delam** is usually fairly realistic, in that the actual error $\left|\tilde{\lambda} - \mathbf{elam}\right|$ is within an order of magnitude of **delam**.

We pass the values of β, ϕ, ρ across through **report** rather than converting them to values of y, y' inside nag_ode_sl2_breaks_funs (d02ke), for the following reasons. First, there may be cases where auxiliary quantities can be more accurately computed from the Pruefer variables than from y and y'. Second, in singular problems on an infinite interval y and y' may underflow towards the end of the range, whereas the Pruefer variables remain well-behaved. Third, with high-order eigenvalues (and therefore highly oscillatory eigenfunctions) the eigenfunction may have a complete oscillation (or more than one oscillation) between two mesh points, so that values of y and y' at mesh points give a very poor representation of the curve. The probable behaviour of the Pruefer variables in this case is that β and ρ vary slowly whilst ϕ increases quickly: for all three Pruefer variables linear interpolation between the values at adjacent mesh points is probably sufficiently accurate to yield acceptable intermediate values of β, ϕ, ρ (and hence of y, y') for graphical purposes.

Similar considerations apply to the exponentially decaying 'tails' of the eigenfunctions that often occur in singular problems. Here ϕ has approximately constant value whilst ρ increases rapidly in the direction of integration, though the step length is generally fairly small over such a range.

If the solution is output through **report** at x values which are too widely spaced, the step length can be controlled by choosing **hmax** suitably, or, preferably, by reducing **tol**. Both these choices will lead to more accurate eigenvalues and eigenfunctions but at some computational cost.

8.3 The Position of the Shooting Matching Point *c*

This point is always one of the values x_i in array **xpoint**. It may be specified using the argument **match**. The default value is chosen to be the value of that x_i, $2 \leq i \leq m-1$, that lies closest to the middle of the interval $[x_2, x_{m-1}]$. If there is a tie, the rightmost candidate is chosen. In particular if there

are no break-points, then $c = x_{m-1}$ $(= x_3)$; that is, the shooting is from left to right in this case. A break-point may be inserted purely to move c to an interior point of the interval, even though the form of the equations does not require it. This often speeds up convergence especially with singular problems.

Note that the shooting method used by the code integrates first from the left-hand end x_l, then from the right-hand end x_r, to meet at the matching point c in the middle. This will of course be reflected in printed or graphical output. The diagram shows a possible sequence of nine mesh points τ_1 through τ_9 in the order in which they appear, assuming there are just two sub-intervals (so $m = 5$).

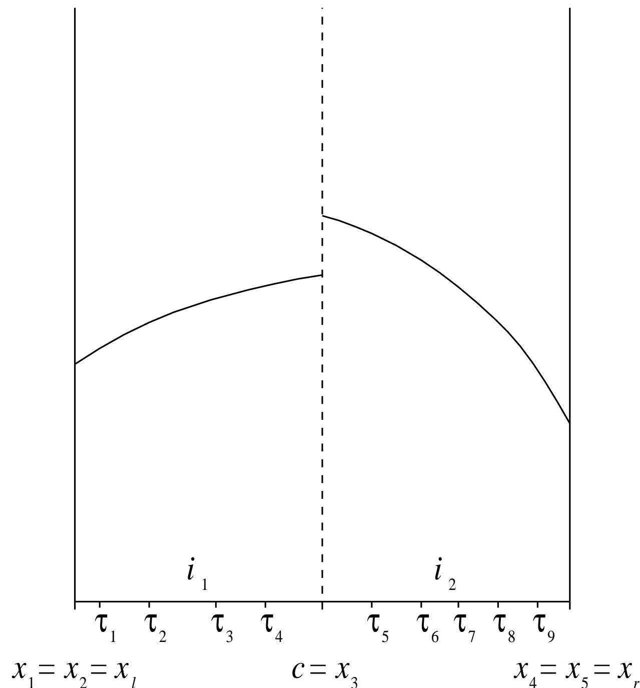

Figure 2

Since the shooting method usually fails to match up the two 'legs' of the curve exactly, there is bound to be a jump in y, or in $p(x)y'$ or both, at the matching point c. The code in fact 'shares' the discrepancy out so that both y and $p(x)y'$ have a jump. A large jump does **not** imply an inaccurate eigenvalue, but implies either

(a) a badly chosen matching point: if $q(x;\lambda)$ has a 'humped' shape, c should be chosen near the maximum value of q, especially if q is negative at the ends of the interval;

(b) an inherently ill-conditioned problem, typically one where another eigenvalue is pathologically close to the one being sought. In this case it is extremely difficult to obtain an accurate eigenfunction.

In Section 10, we find the 11th eigenvalue and corresponding eigenfunction of the equation

$$y'' + \left(\lambda - x - 2/x^2\right)y = 0 \quad \text{on} \quad 0 < x < \infty,$$

the boundary conditions being that y should remain bounded as x tends to 0 and x tends to ∞. The coding of this problem is discussed in detail in Section 9.5.

The choice of matching point c is open. If we choose $c = 30.0$ as in nag_ode_sl2_breaks_vals (d02kd) example program we find that the exponentially increasing component of the solution dominates and we get extremely inaccurate values for the eigenfunction (though the eigenvalue is determined accurately). The values of the eigenfunction calculated with $c = 30.0$ are given schematically in Figure 3.

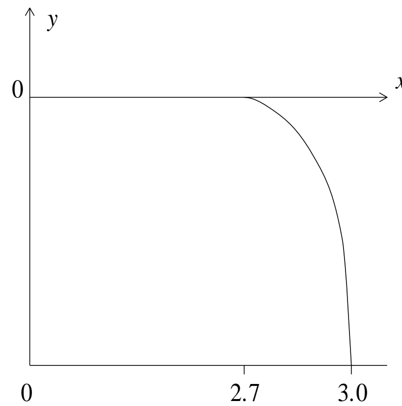

Figure 3

If we choose c as the maximum of the hump in $q(x;\lambda)$ (see item (a) above) we instead obtain the accurate results given in Figure 4

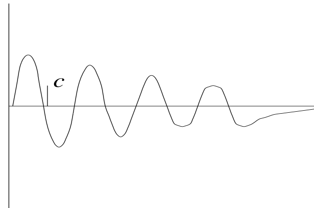

Figure 4

8.4 Examples of Coding the coeffn

Coding **coeffn** is straightforward except when break-points are needed. The examples below show:

(a) a simple case,

(b) a case in which discontinuities in the coefficient functions or their derivatives necessitate break-points, and

(c) a case where break-points together with the **hmax** argument are an efficient way to deal with a coefficient function that is well-behaved except over one short interval.

(Some of these cases are among the examples in Section 10.)

Example A

The modified Bessel equation

$$x(xy')' + \left(\lambda x^2 - \nu^2\right)y = 0.$$

Assuming the interval of solution does not contain the origin and dividing through by x, we have $p(x) = x$ and $q(x;\lambda) = \lambda x - \nu^2/x$. The code could be

```
function [p, q, dqdl] = coeffn(x, elam, jint)
   global nu;
   ...
   p = x;
   q = elam*x - nu*nu/x
   dqdl = x;
```

where nu (standing for ν) is a double global variable declared in the calling program.

Example B

The Schroedinger equation

$$y'' + (\lambda + q(x))y = 0,$$

where

$$q(x) = \begin{cases} x^2 - 10 & (|x| \leq 4), \\ \dfrac{6}{|x|} & (|x| > 4), \end{cases}$$

over some interval 'approximating to $(-\infty, \infty)$', say $[-20, 20]$. Here we need break-points at ± 4, forming three sub-intervals $i_1 = [-20, -4]$, $i_2 = [-4, 4]$, $i_3 = [4, 20]$. The code could be

```
function [p, q, dqdl] = coeffn(x, elam, jint)
   ...
   p = 1;
   dqdl = 1;
   if (jint == 2)
     q = elam + x*x - 10
   else
     q = elam + 6/abs(x)
   end
```

The array **xpoint** would contain the values x_1, -20.0, -4.0, $+4.0$, $+20.0$, x_6 and m would be 6. The choice of appropriate values for x_1 and x_6 depends on the form of the asymptotic formula computed by **bdyval** and the technique is discussed in Section 9.5.

Example C

$$y'' + \lambda\left(1 - 2e^{-100x^2}\right)y = 0, \quad -10 \leq x \leq 10.$$

Here $q(x; \lambda)$ is nearly constant over the range except for a sharp inverted spike over approximately $-0.1 \leq x \leq 0.1$. There is a danger that the function will build up to a large step size and 'step over' the spike without noticing it. By using break-points – say at ± 0.5 – one can restrict the step size near the spike without impairing the efficiency elsewhere.

The code for **coeffn** could be

```
function [p, q, dqdl] = coeffn(x, elam, jint)
 ...
 p = 1;
 dqdl = 1 - 2*exp(-100*x*x);
 q = elam * dqdl;
```

xpoint might contain -10.0, -10.0, -0.5, 0.5, 10.0, 10.0 (assuming ± 10 are regular points) and m would be 6. **hmax**$(1, j)$, for $j = 1, 2, 3$, might contain 0.0, 0.1 and 0.0.

8.5 Examples of Boundary Conditions at Singular Points

Quoting from page 243 of Bailey (1966): 'Usually ... the differential equation has two essentially different types of solution near a singular point, and the boundary condition there merely serves to distinguish one kind from the other. This is the case in all the standard examples of mathematical physics.'

In most cases the behaviour of the ratio $p(x)y'/y$ near the point is quite different for the two types of solution. Essentially what you provide through the **bdyval** is an approximation to this ratio, valid as x tends to the singular point (SP).

You must decide (a) how accurate to make this approximation or asymptotic formula, for example how many terms of a series to use, and (b) where to place the boundary matching point (BMP) at which the numerical solution of the differential equation takes over from the asymptotic formula. Taking the BMP closer to the SP will generally improve the accuracy of the asymptotic formula, but will make the computation more expensive as the Pruefer differential equations generally become progressively more ill-behaved as the SP is approached. You are strongly recommended to experiment with placing the BMPs. In many singular problems quite crude asymptotic formulae will do. To help you avoid needlessly accurate formulae, nag_ode_sl2_breaks_funs (d02ke) outputs two 'sensitivity coefficients' σ_l, σ_r which estimate how much the errors at the BMPs affect the computed eigenvalue. They are described in detail in Section 9.6.

Example of coding bdyval:

The example below illustrates typical situations:

$$y'' + \left(\lambda - x - \frac{2}{x^2}\right)y = 0, \quad 0 < x < \infty$$

the boundary conditions being that y should remain bounded as x tends to 0 and x tends to ∞.

At the end $x = 0$ there is one solution that behaves like x^2 and another that behaves like x^{-1}. For the first of these solutions $p(x)y'/y$ is asymptotically $2/x$ while for the second it is asymptotically $-1/x$. Thus the desired ratio is specified by setting

$$\mathbf{yl}(1) = x \quad \text{and} \quad \mathbf{yl}(2) = 2.0.$$

At the end $x = \infty$ the equation behaves like Airy's equation shifted through λ, i.e., like $y'' - ty = 0$ where $t = x - \lambda$, so again there are two types of solution. The solution we require behaves as

$$\exp\left(-\frac{2}{3}t^{\frac{3}{2}}\right)/\sqrt[4]{t}$$

and the other as

$$\exp\left(+\frac{2}{3}t^{\frac{3}{2}}\right)/\sqrt[4]{t}.$$

Hence, the desired solution has $p(x)y'/y \sim -\sqrt{t}$ so that we could set $\mathbf{yr}(1) = 1.0$ and $\mathbf{yr}(2) = -\sqrt{x - \lambda}$. The complete function might thus be

```
function [yl, yr] = bdyval(xl, xr, elam)
   yl(1) = xl;
   yl(2) = 2;
   yr(1) = 1;
   yr(2) = -sqrt(xr-elam);
```

Clearly for this problem it is essential that any value given by nag_ode_sl2_breaks_funs (d02ke) to **xr** is well to the right of the value of **elam**, so that you must vary the right-hand BMP with the eigenvalue index k. One would expect λ_k to be near the kth zero of the Airy function $\mathrm{Ai}(x)$, so there is no problem estimating **elam**.

More accurate asymptotic formulae are easily found: near $x = 0$ by the standard Frobenius method, and near $x = \infty$ by using standard asymptotics for $\mathrm{Ai}(x)$, $\mathrm{Ai}'(x)$, (see page 448 of Abramowitz and Stegun (1972)).

For example, by the Frobenius method the solution near $x = 0$ has the expansion

$$y = x^2\left(c_0 + c_1 x + c_2 x^2 + \cdots\right)$$

with

$$c_0 = 1, c_1 = 0, c_2 = \frac{-\lambda}{10}, c_3 = \tfrac{1}{18}, \ldots, c_n = \frac{c_{n-3} - \lambda c_{n-2}}{n(n+3)}.$$

This yields

$$\frac{p(x)y'}{y} = \frac{2 - \frac{2}{5}\lambda x^2 + \cdots}{x\left(1 - \frac{\lambda}{10}x^2 + \cdots\right)}.$$

8.6 The Sensitivity Parameters σ_l and σ_r

The sensitivity parameters σ_l, σ_r (held in $\mathbf{hmax}(1, m-1)$ and $\mathbf{hmax}(1, m)$ on output) estimate the effect of errors in the boundary conditions. For sufficiently small errors Δy, $\Delta py'$ in y and py' respectively, the relations

$$\Delta\lambda \simeq (y.\Delta py' - py'.\Delta y)_l \sigma_l$$
$$\Delta\lambda \simeq (y.\Delta py' - py'.\Delta y)_r \sigma_r$$

are satisfied, where the subscripts l, r denote errors committed at the left- and right-hand BMPs respectively, and $\Delta\lambda$ denotes the consequent error in the computed eigenvalue.

8.7 ‘Missed Zeros’

This is a pitfall to beware of at a singular point. If the BMP is chosen so far from the SP that a zero of the desired eigenfunction lies in between them, then the function will fail to ‘notice’ this zero. Since the index of k of an eigenvalue is the number of zeros of its eigenfunction, the result will be that

(a) the wrong eigenvalue will be computed for the given index k – in fact some $\lambda_{k+k'}$ will be found where $k' \geq 1$;

(b) the same index k can cause convergence to any of several eigenvalues depending on the initial values of **elam** and **delam**.

It is up to you to take suitable precautions – for instance by varying the position of the BMPs in the light of knowledge of the asymptotic behaviour of the eigenfunction at different eigenvalues.

9 Example

This example finds the 11th eigenvalue and eigenfunction of the example of Section 9.5, using the simple asymptotic formulae for the boundary conditions.

Comparison of the results from this example program with the corresponding results from nag_ode_sl2_breaks_vals (d02kd) example program shows that similar output is produced from **monit**, followed by the eigenfunction values from **report**, and then a further line of information from **monit** (corresponding to the integration to find the eigenfunction). Final information is printed within the example program exactly as with nag_ode_sl2_breaks_vals (d02kd).

Note the discrepancy at the matching point c ($= \sqrt[3]{4}$, the maximum of $q(x;\lambda)$, in this case) between the solutions obtained by integrations from left- and right-hand end points.

9.1 Program Text

```
     function d02ke_example

fprintf('d02ke example results\n\n');

% For communication with display_plot.
global ykeep ncall xkeep pkeep;

% Set up initial values.
xpoint = [0; 0.1; 4d0^(1d0/3d0); 30; 30];
match = 0;
k = 11;
tol = 1d-4;
elam = 14;
delam = 1;
m = 5;
hmax = zeros(2,m);

ncall = 0;
ykeep = zeros(1,1);
xkeep = zeros(1,1);
pkeep = zeros(1,1);

% report outputs intermediate results.
[match, elam, delam, hmax, maxit, ifail] = ...
d02ke( ...
       xpoint, nag_int(match), @coeffn, @bdyval, ...
       nag_int(k), tol, elam, delam, hmax, 'd02kay', @report);

% Output final results.
fprintf('\n\n');
fprintf('Eigenvalue number, k = %d\n',k);
fprintf(' estimated value     = %6.3f\n',elam);
fprintf(' estimated error     = %9.2e\n',delam);
```

```
fprintf(' sensitivity to left  boundary = %6.3f\n',hmax(1,m-1));
fprintf(' sensitivity to right boundary = %6.3f\n\n', hmax(1,m));
fig1 = figure;
display_plot(xkeep,pkeep,ykeep)

function [yl, yr] = bdyval(xl, xr, elam)
  % Define the boundary conditions.
  yl(1) = xl;
  yl(2) = 2;
  yl(3) = 0;
  yr(1) = 1;
  yr(2) = -sqrt(xr-elam);
  yr(3) = 0;

function [p, q, dqdl] = coeffn(x, elam, jint)
  % Compute the coefficient functions.
  p = 1;
  dqdl = 1;
  q = elam - x - 2/x/x;

function report(x, v, jint)
  % For communication with main routine.
  global ykeep ncall xkeep pkeep;

% Compute eigenfunction and its dervative at this mesh point.
  if (jint == 0)
    fprintf('\n A singular problem\n');
  end
  sqrtb = sqrt(v(1));
  if (0.5*v(3) >= log(x02am))
    r = exp(0.5*v(3));
  else
    r = 0d0;
  end
  pyp = r*sqrtb*cos(0.5*v(2));
  y = r/sqrtb*sin(0.5*v(2));

  % Output results and store them for plotting.
  ncall = ncall+1;
  ykeep(ncall,1) = y;
  xkeep(ncall,1) = x;
  pkeep(ncall,1) = pyp;

function display_plot(xkeep, pkeep, ykeep)

  % Re-order the arrays before plotting.
  n = length(xkeep);
  for i = 2:n
    if (xkeep(i-1) > xkeep(i))
      x1 = [xkeep(i-1:n);rot90(xkeep(1:i-2)')];
      p1 = [pkeep(i-1:n);rot90(pkeep(1:i-2)')];
      y1 = [ykeep(i-1:n);rot90(ykeep(1:i-2)')];
        break
    end
  end

  % Plot the two curves.
  plot(x1,y1,'-',x1,p1,'--');
  % Add title.
  title({'Regular Singular Second-order Sturm-Liouville System',...
         '11th Eigenfunction and Derivative'});
  % Label the axes.
  xlabel('x'); ylabel('Eigenfunction and Derivative');
  % Add a legend.
  legend('y','p(x)y''','Location','Best');
```

9.2 Program Results

```
     d02ke example results

 A singular problem

Eigenvalue number, k = 11
 estimated value     = 14.946
 estimated error     =  9.60e-04
 sensitivity to left  boundary = -0.015
 sensitivity to right boundary =  0.000
```