NAG Toolbox for MATLAB

d03rb

1 Purpose

d03rb integrates a system of linear or nonlinear, time-dependent partial differential equations (PDEs) in two space dimensions on a rectilinear domain. The method of lines is employed to reduce the PDEs to a system of ordinary differential equations (ODEs) which are solved using a backward differentiation formula (BDF) method. The resulting system of nonlinear equations is solved using a modified Newton method and a Bi-CGSTAB iterative linear solver with ILU preconditioning. Local uniform grid refinement is used to improve the accuracy of the solution. d03rb originates from the VLUGR2 package (see Blom and Verwer 1993 and Blom *et al.* 1996).

2 Syntax

```
[ts, dt, rwk, iwk, ind, ifail] = d03rb(ts, tout, dt, tols, tolt, inidom,
pdedef, bndary, pdeiv, monitr, opti, optr, rwk, iwk, lenlwk, itrace,
ind, 'npde', npde, 'lenrwk', lenrwk, 'leniwk', leniwk)
```

3 Description

d03rb integrates the system of PDEs:

$$F_j\big(t,x,y,u,u_t,u_x,u_y,u_{xx},u_{xy},u_{yy}\big)=0, \qquad j=1,2,\ldots,\mathbf{npde}, \qquad (x,y)\in\Omega, \qquad t_0\le t\le t_{\mathrm{out}}, \tag{1}$$

where Ω is an arbitrary rectilinear domain, i.e., a domain bounded by perpendicular straight lines. If the domain is rectangular then it is recommended that d03ra is used.

The vector u is the set of solution values

$$u(x,y,t)=\Big[u_1(x,y,t),\ldots,u_{\mathbf{npde}}(x,y,t)\Big]^{\mathrm{T}},$$

and u_t denotes partial differentiation with respect to t, and similarly for u_x, etc.

The functions F_j must be supplied by you in a user-supplied (sub)program **pdedef**. Similarly the initial values of the functions $u(x,y,t)$ for $(x,y)\in\Omega$ must be specified at $t=t_0$ in a user-supplied (sub)program **pdeiv**.

Note that whilst complete generality is offered by the master equations (1), d03rb is not appropriate for all PDEs. In particular, hyperbolic systems should not be solved using this function. Also, at least one component of u_t must appear in the system of PDEs.

The boundary conditions must be supplied by you in a user-supplied (sub)program **bndary** in the form

$$G_j\big(t,x,y,u,u_t,u_x,u_y\big)=0 \qquad j=1,2,\ldots,\mathbf{npde}, \qquad (x,y)\in\partial\Omega, \qquad t_0\le t\le t_{\mathrm{out}}. \tag{2}$$

The domain is covered by a uniform coarse base grid specified by you, and nested finer uniform subgrids are subsequently created in regions with high spatial activity. The refinement is controlled using a space monitor which is computed from the current solution and a user-supplied space tolerance **tols**. A number of optional parameters, e.g., the maximum number of grid levels at any time, and some weighting factors, can be specified in the arrays **opti** and **optr**. Further details of the refinement strategy can be found in Section 8.

The system of PDEs and the boundary conditions are discretized in space on each grid using a standard second-order finite difference scheme (centred on the internal domain and one-sided at the boundaries), and the resulting system of ODEs is integrated in time using a second-order, two-step, implicit BDF method with variable step size. The time integration is controlled using a time monitor computed at each grid level from the current solution and a user-supplied time tolerance **tolt**, and some further optional user-specified weighting factors held in **optr** (see Section 8 for details). The time monitor is used to compute a new step size, subject to restrictions on the size of the change between steps, and (optional) user-specified maximum and minimum step sizes held in **dt**. The step size is adjusted so that the remaining integration interval is an integer number times Δt. In this way a solution is obtained at $t = t_{\mathrm{out}}$.

A modified Newton method is used to solve the nonlinear equations arising from the time integration. You may specify (in **opti**) the maximum number of Newton iterations to be attempted. A Jacobian matrix is calculated at the beginning of each time step. If the Newton process diverges or the maximum number of iterations is exceeded, a new Jacobian is calculated using the most recent iterates and the Newton process is restarted. If convergence is not achieved after the (optional) user-specified maximum number of new Jacobian evaluations, the time step is retried with $\Delta t = \Delta t/4$. The linear systems arising from the Newton iteration are solved using a Bi-CGSTAB iterative method, in combination with ILU preconditioning. The maximum number of iterations can be specified by you in **opti**.

In order to define the base grid you must first specify a virtual uniform rectangular grid which contains the entire base grid. The position of the virtual grid in physical (x, y) space is given by the (x, y) co-ordinates of its boundaries. The number of points n_x and n_y in the x and y directions must also be given, corresponding to the number of columns and rows respectively. This is sufficient to determine precisely the (x, y) co-ordinates of all virtual grid points. Each virtual grid point is then referred to by integer co-ordinates $\left(v_x, v_y\right)$, where $(0, 0)$ corresponds to the lower-left corner and $\left(n_x - 1, n_y - 1\right)$ corresponds to the upper-right corner. v_x and v_y are also referred to as the virtual column and row indices respectively.

The base grid is then specified with respect to the virtual grid, with each base grid point coinciding with a virtual grid point. Each base grid point must be given an index, starting from 1, and incrementing row-wise from the leftmost point of the lowest row. Also, each base grid row must be numbered consecutively from the lowest row in the grid, so that row 1 contains grid point 1.

As an example, consider the domain consisting of the two separate squares shown in Figure 1. The left-hand diagram shows the virtual grid and its integer co-ordinates (i.e., its column and row indices), and the right-hand diagram shows the base grid point indices and the base row indices (in brackets).

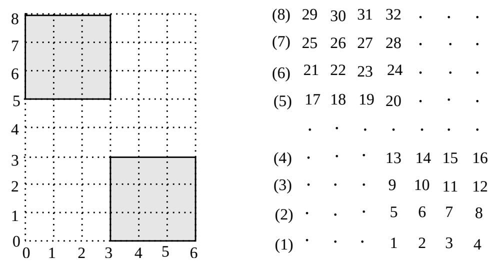

Figure 1

Hence the base grid point with index 6 say is in base row 2, virtual column 4, and virtual row 1, i.e., virtual grid integer co-ordinates $(4, 1)$; and the base grid point with index 19 say is in base row 5, virtual column 2, and virtual row 5, i.e., virtual grid integer co-ordinates $(2, 5)$.

The base grid must then be defined in the user-supplied (sub)program **inidom** by specifying the number of base grid rows, the number of base grid points, the number of boundaries, the number of boundary points, and the following integer arrays:

lrow contains the base grid indices of the starting points of the base grid rows;

irow contains the virtual row numbers v_y of the base grid rows;

icol contains the virtual column numbers v_x of the base grid points;

lbnd contains the grid indices of the boundary edges (without corners) and corner points;

llbnd contains the starting elements of the boundaries and corners in **lbnd**.

Finally, **ilbnd** contains the types of the boundaries and corners, as follows:

Boundaries:

1 – lower boundary

2 – left boundary

3 – upper boundary

4 – right boundary

External corners (90°):

12 – lower-left corner

23 – upper-left corner

34 – upper-right corner

41 – lower-right corner

Internal corners (270°):

21 – lower-left corner

32 – upper-left corner

43 – upper-right corner

14 – lower-right corner

Figure 2 shows the boundary types of a domain with a hole. Notice the logic behind the labelling of the corners: each one includes the types of the two adjacent boundary edges, in a clockwise fashion (outside the domain).

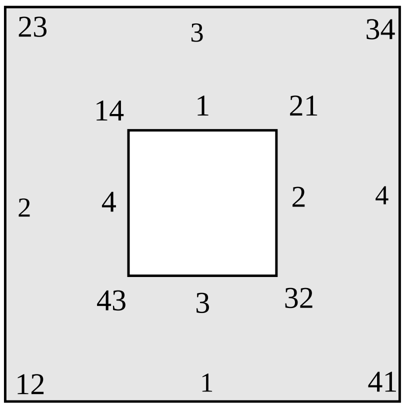

Figure 2

As an example, consider the domain shown in Figure 3. The left-hand diagram shows the physical domain and the right-hand diagram shows the base and virtual grids. The numbers outside the base grid are the indices of the left and rightmost base grid points, and the numbers inside the base grid are the boundary or corner numbers, indicating the order in which the boundaries are stored in **lbnd**.

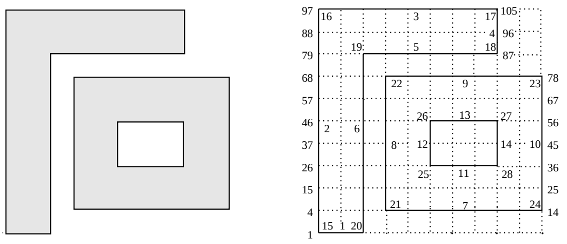

Figure 3

For this example we have

```
NROWS = 11
NPTS = 105
NBNDS = 28
NBPTS = 72
```

```
LROW = (1,4,15,26,37,46,57,68,79,88,97)

IROW = (0,1,2,3,4,5,6,7,8,9,10)

ICOL = (0,1,2,
        0,1,2,3,4,5,6,7,8,9,10,
        0,1,2,3,4,5,6,7,8,9,10,
        0,1,2,3,4,5,6,7,8,9,10,
        0,1,2,3,4,5,8,9,10,
        0,1,2,3,4,5,6,7,8,9,10,
        0,1,2,3,4,5,6,7,8,9,10,
        0,1,2,3,4,5,6,7,8,9,10,
        0,1,2,3,4,5,6,7,8,
        0,1,2,3,4,5,6,7,8,
        0,1,2,3,4,5,6,7,8)

LBND = (2,
        4,15,26,37,46,57,68,79,88,
        98,99,100,101,102,103,104,
        96,
        86,85,84,83,82,
        70,59,48,39,28,17,6,
        8,9,10,11,12,13,
        18,29,40,49,60,
        72,73,74,75,76,77,
        67,56,45,36,25,
        33,32,
        42,
        52,53,
        43,
        1,97,105,87,81,3,7,71,78,14,31,51,54,34)

LLBND = (1,2,11,18,19,24,31,37,42,48,53,55,56,58,59,60,
         61,62,63,64,65,66,67,68,69,70,71,72)

ILBND = (1,2,3,4,1,4,1,2,3,4,3,4,1,2,12,23,34,41,14,41,
         12,23,34,41,43,14,21,32)
```

This particular domain is used in the example in Section 9, and data statements are used to define the above arrays in that example program. For less complicated domains it is simpler to assign the values of the arrays in do-loops. This also allows flexibility in the number of base grid points.

The function d03ry can be called from user-supplied (sub)program **inidom** to obtain a simple graphical representation of the base grid, and to verify the data that you have specified in **inidom**.

Subgrids are stored internally using the same data structure, and solution information is communicated to you in the user-supplied (sub)programs **pdeiv**, **pdedef** and **bndary** in arrays according to the grid index on the particular level, e.g., $\mathbf{x}(i)$ and $\mathbf{y}(i)$ contain the (x,y) co-ordinates of grid point i, and $\mathbf{u}(i,j)$ contains the jth solution component u_j at grid point i.

The grid data and the solutions at all grid levels are stored in the workspace arrays, along with other information needed for a restart (i.e., a continuation call). It is not intended that you extract the solution from these arrays, indeed the necessary information regarding these arrays is not provided. The user-supplied monitor (user-supplied (sub)program **monitr**) should be used to obtain the solution at particular levels and times. **monitr** is called at the end of every time step, with the last step being identified via the input parameter **tlast**. The function d03rz should be called from **monitr** to obtain grid information at a particular level.

Further details of the underlying algorithm can be found in Section 8 and in Blom and Verwer 1993 and Blom *et al.* 1996 and the references therein.

4 References

Blom J G, Trompert R A and Verwer J G 1996 Algorithm 758. VLUGR2: A vectorizable adaptive grid solver for PDEs in 2D *Trans. Math. Software* **22** 302–328

Blom J G and Verwer J G 1993 VLUGR2: A vectorized local uniform grid refinement code for PDEs in 2D *Report NM-R9306* CWI, Amsterdam

Trompert R A 1993 Local uniform grid refinement and systems of coupled partial differential equations *Appl. Numer. Maths* **12** 331–355

Trompert R A and Verwer J G 1993 Analysis of the implicit Euler local uniform grid refinement method *SIAM J. Sci. Comput.* **14** 259–278

5 Parameters

5.1 Compulsory Input Parameters

1: **ts – double scalar**

The initial value of the independent variable t.

Constraint: **ts** $<$ **tout**.

2: **tout – double scalar**

The final value of t to which the integration is to be carried out.

3: **dt(3) – double array**

The initial, minimum and maximum time step sizes respectively.

dt(1)

Specifies the initial time step size to be used on the first entry, i.e., when **ind** $= 0$. If **dt**$(1) = 0.0$ then the default value **dt**$(1) = 0.01 \times ($**tout** $-$ **ts**$)$ is used. On subsequent entries (**ind** $= 1$), the value of **dt**(1) is not referenced.

dt(2)

Specifies the minimum time step size to be attempted by the integrator. If **dt**$(2) = 0.0$ the default value **dt**$(2) = 10.0 \times$ ***machine precision*** is used.

dt(3)

Specifies the maximum time step size to be attempted by the integrator. If **dt**$(3) = 0.0$ the default value **dt**$(3) =$ **tout** $-$ **ts** is used.

Constraints:

if **ind** $= 0$, **dt**$(1) \geq 0$;
if **ind** $= 0$ and **dt**$(1) > 0$, $10.0 \times$ ***machine precision*** $\times \max(|$**ts**$|, |$**tout**$|) \leq$ **dt**$(1) \leq$ **tout** $-$ **ts** and **dt**$(2) \leq$ **dt**$(1) \leq$ **dt**(3), where the values of **dt**(2) and **dt**(3) will have been reset to their default values if zero on entry;
$0 \leq$ **dt**$(2) \leq$ **dt**(3).

4: **tols – double scalar**

The space tolerance used in the grid refinement strategy (σ in equation (4)). See Section 8.2.

Constraint: **tols** > 0.0.

5: **tolt – double scalar**

The time tolerance used to determine the time step size (τ in equation (7)). See Section 8.3.

Constraint: **tolt** > 0.0.

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

inidom must specify the base grid in terms of the data structure described in Section 3. **inidom** is not referenced if, on entry, **ind** $= 1$. d03ry can be called from **inidom** to obtain a simple graphical representation of the base grid, and to verify the data that you have specified in **inidom**. d03rb also checks the validity of the data, but you are strongly advised to call d03ry to ensure that the base grid is exactly as required.

Note: the boundaries of the base grid should consist of as many points as are necessary to employ second-order space discretization, i.e., a boundary enclosing the internal part of the domain must include at least 3 grid points including the corners. If Neumann boundary conditions are to be applied the minimum is 4.

Its specification is:

```
[xmin, xmax, ymin, ymax, nx, ny, npts, nrows, nbnds, nbpts, lrow,
irow, icol, llbnd, ilbnd, lbnd, ierr] = inidom(maxpts, ierr)
```

Input Parameters

1: **maxpts – int32 scalar**

The maximum number of base grid points allowed by the available workspace.

2: **ierr – int32 scalar**

Will be initialized by d03rb to some value prior to internal calls to **inidom**.

If the required number of grid points is larger than **maxpts**, **ierr** must be set to -1 to force a termination of the integration and an immediate return to the calling program with **ifail** $= 3$. Otherwise, **ierr** should remain unchanged.

Output Parameters

1: **xmin – double scalar**

2: **xmax – double scalar**

The extents of the virtual grid in the x-direction, i.e., the x co-ordinates of the left and right boundaries respectively.

3: **ymin – double scalar**

4: **ymax – double scalar**

The extents of the virtual grid in the y-direction, i.e., the y co-ordinates of the left and right boundaries respectively.

5: **nx – int32 scalar**

6: **ny – int32 scalar**

The number of virtual grid points in the x- and y-direction respectively (including the boundary points).

7: **npts – int32 scalar**

The total number of points in the base grid. If the required number of points is greater than **maxpts** then **inidom** must be exited immediately with **ierr** set to -1 to avoid overwriting memory.

8: **nrows – int32 scalar**

The total number of rows of the virtual grid that contain base grid points. This is the maximum base row index.

9: **nbnds – int32 scalar**

The total number of physical boundaries and corners in the base grid.

10: **nbpts – int32 scalar**

The total number of boundary points in the base grid.

11: **lrow**($*$) – **int32 array**

Note: the dimension of the array **lrow** must be at least **nrows**.

lrow(i), for $i = 1, 2, \dots,$ **nrows**, must contain the base grid index of the first grid point in base grid row i.

12: **irow**($*$) – **int32 array**

Note: the dimension of the array **irow** must be at least **nrows**.

irow(i), for $i = 1, 2, \dots,$ **nrows**, must contain the virtual row number v_y that corresponds to base grid row i.

13: **icol**($*$) – **int32 array**

Note: the dimension of the array **icol** must be at least **npts**.

icol(i), for $i = 1, 2, \dots,$ **npts**, must contain the virtual column number v_x that contains base grid point i.

14: **llbnd**($*$) – **int32 array**

Note: the dimension of the array **llbnd** must be at least **nbnds**.

llbnd(i), for $i = 1, 2, \dots,$ **nbnds**, must contain the element of **lbnd** corresponding to the start of the ith boundary or corner.

Note: the order of the boundaries and corners in **llbnd** must be first all the boundaries and then all the corners. The end points of a boundary (i.e., the adjacent corner points) must **not** be included in the list of points on that boundary. Also, if a corner is shared by two pairs of physical boundaries then it has two types and must therefore be treated as two corners.

15: **ilbnd**($*$) – **int32 array**

Note: the dimension of the array **ilbnd** must be at least **nbnds**.

ilbnd(i), for $i = 1, 2, \dots,$ **nbnds**, must contain the type of the ith boundary (or corner), as given in Section 3.

16: **lbnd**($*$) – **int32 array**

Note: the dimension of the array **lbnd** must be at least **nbpts**.

lbnd(i), for $i = 1, 2, \dots,$ **nbpts**, must contain the grid index of the ith boundary point. The order of the boundaries is as specified in **llbnd**, but within this restriction the order of the points in **lbnd** is arbitrary.

17: **ierr** – **int32 scalar**

Will be initialized by d03rb to some value prior to internal calls to **inidom**.

If the required number of grid points is larger than **maxpts**, **ierr** must be set to -1 to force a termination of the integration and an immediate return to the calling program with **ifail** $= 3$. Otherwise, **ierr** should remain unchanged.

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

pdedef must evaluate the functions F_j, for $j = 1, 2, \dots,$ **npde**, in equation (1) which define the system of PDEs (i.e., the residuals of the resulting ODE system) at all interior points of the domain. Values at points on the boundaries of the domain are ignored and will be overwritten by the user-supplied (sub)program **bndary**. **pdedef** is called for each subgrid in turn.

Its specification is:

```
[res] = pdedef(npts, npde, t, x, y, u, ut, ux, uy, uxx, uxy, uyy)
```

Input Parameters

1: **npts – int32 scalar**

The number of grid points in the current grid.

2: **npde – int32 scalar**

The number of PDEs in the system.

3: **t – double scalar**

The current value of the independent variable t.

4: **x(npts) – double array**

x(i) contains the x co-ordinate of the ith grid point, for $i = 1, 2, \ldots,$ **npts**.

5: **y(npts) – double array**

y(i) contains the y co-ordinate of the ith grid point, for $i = 1, 2, \ldots,$ **npts**.

6: **u(npts,npde) – double array**

u(i,j) contains the value of the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

7: **ut(npts,npde) – double array**

ut(i,j) contains the value of $\frac{\partial u}{\partial t}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

8: **ux(npts,npde) – double array**

ux(i,j) contains the value of $\frac{\partial u}{\partial x}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

9: **uy(npts,npde) – double array**

uy(i,j) contains the value of $\frac{\partial u}{\partial y}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

10: **uxx(npts,npde) – double array**

uxx(i,j) contains the value of $\frac{\partial^2 u}{\partial x^2}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

11: **uxy(npts,npde) – double array**

uxy(i,j) contains the value of $\frac{\partial^2 u}{\partial x \partial y}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

12: **uyy(npts,npde) – double array**

uyy(i,j) contains the value of $\frac{\partial^2 u}{\partial y^2}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

Output Parameters

1: **res(npts,npde) – double array**

res(i,j) must contain the value of F_j, for $j = 1, 2, \ldots,$ **npde**, at the ith grid point for $i = 1, 2, \ldots,$ **npts**, although the residuals at boundary points will be ignored (and overwritten later on) and so they need not be specified here.

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

bndary must evaluate the functions G_j, for $j = 1, 2, \ldots,$ **npde**, in equation (2) which define the boundary conditions at all boundary points of the domain. Residuals at interior points must **not** be altered by this (sub)program.

Its specification is:

```
[res] = bndary(npts, npde, t, x, y, u, ut, ux, uy, nbnds, nbpts,
llbnd, ilbnd, lbnd, res)
```

Input Parameters

1: **npts – int32 scalar**

The number of grid points in the current grid.

2: **npde – int32 scalar**

The number of PDEs in the system.

3: **t – double scalar**

The current value of the independent variable t.

4: **x(npts) – double array**

x(i) contains the x co-ordinate of the ith grid point, for $i = 1, 2, \ldots,$ **npts**.

5: **y(npts) – double array**

y(i) contains the y co-ordinate of the ith grid point, for $i = 1, 2, \ldots,$ **npts**.

6: **u(npts,npde) – double array**

u(i,j) contains the value of the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

7: **ut(npts,npde) – double array**

ut(i,j) contains the value of $\frac{\partial u}{\partial t}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

8: **ux(npts,npde) – double array**

ux(i,j) contains the value of $\frac{\partial u}{\partial x}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

9: **uy(npts,npde) – double array**

uy(i,j) contains the value of $\frac{\partial u}{\partial y}$ for the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

10: **nbnds – int32 scalar**

The total number of physical boundaries and corners in the grid.

11: **nbpts – int32 scalar**

The total number of boundary points in the grid.

12: **llbnd(nbnds) – int32 array**

llbnd(i), for $i = 1, 2, \ldots,$ **nbnds**, contains the element of **lbnd** corresponding to the start of the ith boundary (or corner).

13: **ilbnd(nbnds) – int32 array**

ilbnd(i), for $i = 1, 2, \ldots,$ **nbnds**, contains the type of the ith boundary, as given in Section 3.

14: **lbnd(nbpts) – int32 array**

lbnd(i), for $i = 1, 2, \ldots,$ **nbpts**, contains the grid index of the ith boundary point, where the order of the boundaries is as specified in **llbnd**. Hence the ith boundary point has coordinates **x**(**lbnd**(i)) and **y**(**lbnd**(i)), and the corresponding solution values are **u**(**lbnd**$(i)j$), for $j = 1, 2, \ldots,$ **npde**.

15: **res(npts,npde) – double array**

Contains function values returned by user-supplied (sub)program **pdedef**.

res(**lbnd**$(i)j$) must contain the value of G_j, for $j = 1, 2, \ldots,$ **npde**, at the ith boundary point, for $i = 1, 2, \ldots,$ **nbpts**.

Note: elements of **res** corresponding to interior points, i.e., points not included in **lbnd**, must **not** be altered.

Output Parameters

1: **res(npts,npde) – double array**

Contains function values returned by user-supplied (sub)program **pdedef**.

res(**lbnd**$(i)j$) must contain the value of G_j, for $j = 1, 2, \ldots,$ **npde**, at the ith boundary point, for $i = 1, 2, \ldots,$ **nbpts**.

Note: elements of **res** corresponding to interior points, i.e., points not included in **lbnd**, must **not** be altered.

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

pdeiv must specify the initial values of the PDE components u at all points in the base grid. **pdeiv** is not referenced if, on entry, **ind** $= 1$.

Its specification is:

```
[u] = pdeiv(npts, npde, t, x, y)
```

Input Parameters

1: **npts – int32 scalar**

The number of grid points in the base grid.

2: **npde – int32 scalar**

The number of PDEs in the system.

3: **t – double scalar**

The (initial) value of the independent variable t.

4: **x(npts) – double array**

x(i) contains the x co-ordinate of the ith grid point, for $i = 1, 2, \ldots,$ **npts**.

5: **y(npts) – double array**

y(i) contains the y co-ordinate of the ith grid point, for $i = 1, 2, \ldots,$ **npts**.

Output Parameters

1: **u(npts,npde) – double array**

u(i,j) must contain the value of the jth PDE component at the ith grid point, for $i = 1, 2, \ldots,$ **npts** and $j = 1, 2, \ldots,$ **npde**.

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

monitr is called by d03rb at the end of every successful time step, and may be used to examine or print the solution or perform other tasks such as error calculations, particularly at the final time step, indicated by the parameter **tlast**.

The input arguments contain information about the grid and solution at all grid levels used. d03rz should be called from **monitr** in order to extract the number of points and their (x, y) co-ordinates on a particular grid.

monitr can also be used to force an immediate tidy termination of the solution process and return to the calling program.

Its specification is:

```
[ierr] = monitr(npde, t, dt, dtnew, tlast, nlev, xmin, ymin, dxb,
dyb, lgrid, istruc, lsol, sol, ierr)
```

Input Parameters

1: **npde – int32 scalar**

The number of PDEs in the system.

2: **t – double scalar**

The current value of the independent variable t, i.e., the time at the end of the integration step just completed.

3: **dt – double scalar**

The current time step size Δt, i.e., the time step size used for the integration step just completed.

4: **dtnew – double scalar**

The time step size that will be used for the next time step.

5: **tlast – logical scalar**

Indicates if intermediate or final time step. **tlast** = **false** for an intermediate step, **tlast** = **true** for the last call to **monitr** before returning to your program.

6: **nlev – int32 scalar**

The number of grid levels used at time **t**.

7: **xmin – double scalar**

8: **ymin – double scalar**

The (x, y) co-ordinates of the lower-left corner of the virtual grid.

9: **dxb – double scalar**

10: **dyb – double scalar**

The sizes of the base grid spacing in the x- and y-direction respectively.

11: **lgrid**$(*)$ **– int32 array**

Note: the dimension of the array **lgrid** must be at least **nlev** + 1.

Contains pointers to the start of the grid structures in **istruc**, and must be passed unchanged to d03rz in order to extract the grid information.

12: **istruc**$(*)$ **– int32 array**

Note: the dimension of the array **istruc** must be at least $leniwk - 15 \times maxlev - 5$, where *leniwk* is the value of **leniwk** passed to d03rb, and *maxlev* is the maximum number of grid levels allowed, as defined by **opti**(1).

Contains the grid structures for each grid level and must be passed unchanged to d03rz in order to extract the grid information.

13: **lsol**(**nlev**) **– int32 array**

lsol(l) contains the pointer to the solution in **sol** at grid level l and time **t**. (**lsol**(l) actually contains the array index immediately preceding the start of the solution in **sol**.)

14: **sol**$(ntot \times$ **npde** $\times$ **nlev**$)$ **– double array**

Contains the solution u at time **t** for each grid level l in turn, positioned according to **lsol**. More precisely

$$\mathbf{u}(i,j) = \mathbf{sol}(\mathbf{lsol}(l) + (j-1) \times n_l + i)$$

represents the jth component of the solution at the ith grid point in the lth level, for $i = 1, \ldots, n_l$, $j = 1, \ldots,$ **npde** and $l = 1, \ldots,$ **nlev**, where n_l is the number of grid points at level l (obtainable by a call to d03rz).

15: **ierr – int32 scalar**

Will be initialized by d03rb to some value prior to internal calls to user-supplied (sub)program **inidom**.

Should be set to 1 to force a termination of the integration and an immediate return to the calling program with **ifail** = 4. **ierr** should remain unchanged otherwise.

Output Parameters

1: **ierr – int32 scalar**

Will be initialized by d03rb to some value prior to internal calls to user-supplied (sub)program **inidom**.

Should be set to 1 to force a termination of the integration and an immediate return to the calling program with **ifail** $= 4$. **ierr** should remain unchanged otherwise.

11: **opti(4) – int32 array**

May be set to control various options available in the integrator.

opti$(1) = 0$

All the default options are employed.

opti$(1) > 0$

The default value of **opti**(i), for $i = 2, 3$ or 4, can be obtained by setting **opti**$(i) = 0$.

opti(1)

Specifies the maximum number of grid levels allowed (including the base grid). **opti**$(1) \geq 0$. The default value is **opti**$(1) = 3$.

opti(2)

Specifies the maximum number of Jacobian evaluations allowed during each nonlinear equations solution. **opti**$(2) \geq 0$. The default value is **opti**$(2) = 2$.

opti(3)

Specifies the maximum number of Newton iterations in each nonlinear equations solution. **opti**$(3) \geq 0$. The default value is **opti**$(3) = 10$.

opti(4)

Specifies the maximum number of iterations in each linear equations solution. **opti**$(4) \geq 0$. The default value is **opti**$(4) = 100$.

Constraint: **opti**$(1) \geq 0$ and if **opti**$(1) > 0$, **opti**$(i) \geq 0$, for $i = 2, 3$ or 4.

12: **optr(3,npde) – double array**

May be used to specify the optional vectors $u^{\max}$, w^s and w^t in the space and time monitors (see Section 8).

If an optional vector is not required then all its components should be set to 1.0.

optr$(1,j)$, for $j = 1, 2, \ldots,$ **npde**, specifies $u_j^{\max}$, the approximate maximum absolute value of the jth component of u, as used in (4) and (7). **optr**$(1,j) > 0.0$, for $j = 1, 2, \ldots,$ **npde**.

optr$(2,j)$, for $j = 1, 2, \ldots,$ **npde**, specifies w_j^s, the weighting factors used in the space monitor (see (4)) to indicate the relative importance of the jth component of u on the space monitor. **optr**$(2,j) \geq 0.0$, for $j = 1, 2, \ldots,$ **npde**.

optr$(3,j)$, for $j = 1, 2, \ldots,$ **npde**, specifies w_j^t, the weighting factors used in the time monitor (see (6)) to indicate the relative importance of the jth component of u on the time monitor. **optr**$(3,j) \geq 0.0$, for $j = 1, 2, \ldots,$ **npde**.

Constraint: **optr**$(1,j) > 0.0$, for $j = 1, 2, \ldots,$ **npde**, and **optr**$(i,j) \geq 0.0$, for $i = 2, 3$ and $j = 1, 2, \ldots,$ **npde**.

13: **rwk(lenrwk) – double array**

The required value of **lenrwk** cannot be determined exactly in advance, but a suggested value is

$$\textbf{lenrwk} = maxpts \times \textbf{npde} \times (5 \times l + 18 \times \textbf{npde} + 9) + 2 \times maxpts,$$

where $l = \textbf{opti}(1)$ if $\textbf{opti}(1) \neq 0$ and $l = 3$ otherwise, and $maxpts$ is the expected maximum number of grid points at any one level. If during the execution the supplied value is found to be too small then the function returns with **ifail** $= 3$ and an estimated required size is printed on the current error message unit (see x04aa).

Note: the size of **lenrwk** cannot be checked upon initial entry to d03rb since the number of grid points on the base grid is not known.

14: **iwk(leniwk) – int32 array**

If **ind** $= 0$, **iwk** need not be set. Otherwise **iwk** must remain unchanged from a previous call to d03rb.

15: **lenlwk – int32 scalar**

The required value of **lenlwk** cannot be determined exactly in advance, but a suggested value is

$$\textbf{lenlwk} = maxpts + 1,$$

where $maxpts$ is the expected maximum number of grid points at any one level. If during the execution the supplied value is found to be too small then the function returns with **ifail** $= 3$ and an estimated required size is printed on the current error message unit (see x04aa).

Note: the size of **lenlwk** cannot be checked upon initial entry to d03rb since the number of grid points on the base grid is not known.

16: **itrace – int32 scalar**

The level of trace information required from d03rb. **itrace** may take the value -1, 0, 1, 2 or 3.

itrace $= -1$

No output is generated.

itrace $= 0$

Only warning messages are printed.

itrace > 0

Output from the underlying solver is printed on the current advisory message unit (see x04ab). This output contains details of the time integration, the nonlinear iteration and the linear solver.

If **itrace** < -1, then -1 is assumed and similarly if **itrace** > 3, then 3 is assumed.

The advisory messages are given in greater detail as **itrace** increases. Setting **itrace** $= 1$ allows you to monitor the progress of the integration without possibly excessive information.

17: **ind – int32 scalar**

Must be set to 0 or 1.

ind $= 0$

Starts the integration in time.

ind $= 1$

Continues the integration after an earlier exit from the function. In this case, only the following parameters may be reset between calls to d03rb: **tout**, **dt**(2), **dt**(3), **tols**, **tolt**, **opti**, **optr**, **itrace** and **ifail**.

Constraint: $0 \leq \textbf{ind} \leq 1$.

5.2 Optional Input Parameters

1: **npde – int32 scalar**

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

the number of PDEs in the system.

Constraint: **npde** ≥ 1.

2: **lenrwk – int32 scalar**

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

The required value of **lenrwk** cannot be determined exactly in advance, but a suggested value is

$$\textbf{lenrwk} = maxpts \times \textbf{npde} \times (5 \times l + 18 \times \textbf{npde} + 9) + 2 \times maxpts,$$

where $l = \textbf{opti}(1)$ if $\textbf{opti}(1) \neq 0$ and $l = 3$ otherwise, and $maxpts$ is the expected maximum number of grid points at any one level. If during the execution the supplied value is found to be too small then the function returns with **ifail** $= 3$ and an estimated required size is printed on the current error message unit (see x04aa).

Note: the size of **lenrwk** cannot be checked upon initial entry to d03rb since the number of grid points on the base grid is not known.

3: **leniwk – int32 scalar**

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

The required value of **leniwk** cannot be determined exactly in advance, but a suggested value is

$$\textbf{leniwk} = maxpts \times (14 + 5 \times m) + 7 \times m + 2,$$

where $maxpts$ is the expected maximum number of grid points at any one level and $m = \textbf{opti}(1)$ if $\textbf{opti}(1) > 0$ and $m = 3$ otherwise. If during the execution the supplied value is found to be too small then the function returns with **ifail** $= 3$ and an estimated required size is printed on the current error message unit (see x04aa).

Note: the size of **leniwk** cannot be checked upon initial entry to d03rb since the number of grid points on the base grid is not known.

5.3 Input Parameters Omitted from the MATLAB Interface

lwk

5.4 Output Parameters

1: **ts – double scalar**

The value of t which has been reached. Normally **ts** $=$ **tout**.

2: **dt**(3) **– double array**

dt(1) contains the time step size for the next time step. **dt**(2) and **dt**(3) are unchanged or set to their default values if zero on entry.

3: **rwk**$($**lenrwk**$)$ **– double array**

4: **iwk**$($**leniwk**$)$ **– int32 array**

The following components of the array **iwk** concern the efficiency of the integration. Here, m is the maximum number of grid levels allowed ($m = \textbf{opti}(1)$ if $\textbf{opti}(1) > 1$ and $m = 3$ otherwise), and l is a grid level taking the values $l = 1, 2, \ldots, nl$, where nl is the number of levels used.

iwk(1)

Contains the number of steps taken in time.

iwk(2)

Contains the number of rejected time steps.

iwk$(2+l)$

Contains the total number of residual evaluations performed (i.e., the number of times user-supplied (sub)program **pdedef** was called) at grid level l.

iwk$(2+m+l)$

Contains the total number of Jacobian evaluations performed at grid level l.

iwk$(2+2\times m+l)$

Contains the total number of Newton iterations performed at grid level l.

iwk$(2+3\times m+l)$

Contains the total number of linear solver iterations performed at grid level l.

iwk$(2+4\times m+l)$

Contains the maximum number of Newton iterations performed at any one time step at grid level l.

iwk$(2+5\times m+l)$

Contains the maximum number of linear solver iterations performed at any one time step at grid level l.

Note: the total and maximum numbers are cumulative over all calls to d03rb. If the specified maximum number of Newton or linear solver iterations is exceeded at any stage, then the maximums above are set to the specified maximum plus one.

5: **ind – int32 scalar**

ind $=1$.

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 $=1$

On entry, **npde** <1,
or **tout** $\leq$ **ts**,
or **tout** is too close to **ts**,
or **ind** $=0$ and **dt**$(1)<0.0$,
or **dt**$(i)<0.0$, for $i=2$ or 3,
or **dt**$(2)>$ **dt**(3),
or **ind** $=0$ and $0.0<$ **dt**$(1)<10\times$ ***machine precision*** $\times\max(|$**ts**$|,|$**tout**$|)$,
or **ind** $=0$ and **dt**$(1)>$ **tout** $-$ **ts**,
or **ind** $=0$ and **dt**$(1)<$ **dt**(2) or **dt**$(1)>$ **dt**(3),
or **tols** or **tolt** ≤ 0.0,
or **opti**$(1)<0$,
or **opti**$(1)>0$ and **opti**$(j)<0$, for $j=2$, 3 or 4,
or **optr**$(1,j)\leq 0.0$, for some $j=1,2,\ldots,$ **npde**,
or **optr**$(2,j)<0.0$, for some $j=1,2,\ldots,$ **npde**,
or **optr**$(3,j)<0.0$, for some $j=1,2,\ldots,$ **npde**,

or **ind** $\neq 0$ or 1,
or **ind** $= 1$ on initial entry to d03rb.

ifail $= 2$

The time step size to be attempted is less than the specified minimum size. This may occur following time step failures and subsequent step size reductions caused by one or more of the following:

the requested accuracy could not be achieved, i.e., **tolt** is too small,

the maximum number of linear solver iterations, Newton iterations or Jacobian evaluations is too small,

ILU decomposition of the Jacobian matrix could not be performed, possibly due to singularity of the Jacobian.

Setting **itrace** to a higher value may provide further information.

In the latter two cases you are advised to check their problem formulation in user-supplied (sub)program **pdedef** and/or user-supplied (sub)program **bndary**, and the initial values in user-supplied (sub)program **pdeiv** if appropriate.

ifail $= 3$

One or more of the workspace arrays is too small for the required number of grid points. At the initial time step this error may result because you set **ierr** to -1 in the user-supplied (sub)program **inidom** or the internal check on the number of grid points following the call to **inidom**. An estimate of the required sizes for the current stage is output, but more space may be required at a later stage.

ifail $= 4$

ierr was set to 1 in the user-supplied (sub)program **monitr**, forcing control to be passed back to calling program. Integration was successful as far as **t** $=$ **ts**.

ifail $= 5$

The integration has been completed but the maximum number of levels specified in **opti**(1) was insufficient at one or more time steps, meaning that the requested space accuracy could not be achieved. To avoid this warning either increase the value of **opti**(1) or decrease the value of **tols**.

ifail $= 6$

One or more of the output arguments of user-supplied (sub)program **inidom** was incorrectly specified, i.e.,

xmin $\geq$ **xmax**,

xmax too close to **xmin**,

ymin $\geq$ **ymax**,

ymax too close to **ymin**,

nx or **ny** < 4,

nrows < 4,

nrows $>$ **ny**,

npts $>$ **nx** $\times$ **ny**,

nbnds < 8,

nbpts < 12,

nbpts $\geq$ **npts**,

lrow$(i) < 1$ or **lrow**$(i) >$ **npts**, for some $i = 1, 2, \ldots,$ **nrows**,

lrow$(i) \leq$ **lrow**$(i-1)$, for some $i = 2, 3, \ldots,$ **nrows**,

irow$(i) < 0$ or **irow**$(i) >$ **ny**, for some $i = 1, 2, \ldots,$ **nrows**,

irow$(i) \leq$ **irow**$(i-1)$, for some $i = 2, 3, \ldots,$ **nrows**,

icol$(i) < 0$ or **icol**$(i) >$ **nx**, for some $i = 1, 2, \ldots,$ **npts**,

llbnd$(i) < 1$ or **llbnd**$(i) >$ **nbpts**, for some $i = 1, 2, \ldots,$ **nbnds**,

llbnd$(i) \leq$ **llbnd**$(i-1)$, for some $i = 2, 3, \ldots,$ **nbnds**,

ilbnd$(i) \neq 1$, 2, 3, 4, 12, 23, 34, 41, 21, 32, 43 or 14, for some $i = 1, 2, \ldots,$ **nbnds**,

lbnd$(i) < 1$ or **lbnd**$(i) >$ **npts**, for some $i = 1, 2, \ldots,$ **nbpts**.

7 Accuracy

There are three sources of error in the algorithm: space and time discretization, and interpolation (linear) between grid levels. The space and time discretization errors are controlled separately using the parameters **tols** and **tolt** described in Section 8, and you should test the effects of varying these parameters. Interpolation errors are generally implicitly controlled by the refinement criterion since in areas where interpolation errors are potentially large, the space monitor will also be large. It can be shown that the global spatial accuracy is comparable to that which would be obtained on a uniform grid of the finest grid size. A full error analysis can be found in Trompert and Verwer 1993.

8 Further Comments

8.1 Algorithm Outline

The local uniform grid refinement method is summarized as follows.

1. Initialize the course base grid, an initial solution and an initial time step.
2. Solve the system of PDEs on the current grid with the current time step.
3. If the required accuracy in space and the maximum number of grid levels have not yet been reached:
   (a) Determine new finer grid at forward time level.
   (b) Get solution values at previous time level(s) on new grid.
   (c) Interpolate internal boundary-values from old grid at forward time.
   (d) Get initial values for the Newton process at forward time.
   (e) Go to 2.
4. Update the coarser grid solution using the finer grid values.
5. Estimate error in time integration. If time error is acceptable advance time level.
6. Determine new step size then go to 2 with coarse base as current grid.

8.2 Refinement Strategy

For each grid point i a space monitor μ_i^s is determined by

$$\mu_i^s = \max_{j=1,\mathbf{npde}} \left\{ \gamma_j \left(\left| \Delta x^2 \frac{\partial^2}{\partial x^2} u_j(x_i, y_i, t) \right| + \left| \Delta y^2 \frac{\partial^2}{\partial y^2} u_j(x_i, y_i, t) \right| \right) \right\}, \tag{3}$$

where Δx and Δy are the grid widths in the x and y directions; and x_i, y_i are the (x, y) co-ordinates at grid point i. The parameter γ_j is obtained from

$$\gamma_j = \frac{w_j^s}{u_j^{\max} \sigma}, \tag{4}$$

where σ is the user-supplied space tolerance; w_j^s is a weighting factor for the relative importance of the jth PDE component on the space monitor; and $u_j^{\max}$ is the approximate maximum absolute value of the jth component. A value for σ must be supplied by you. Values for w_j^s and $u_j^{\max}$ must also be supplied but may be set to the values 1.0 if little information about the solution is known.

A new level of refinement is created if

$$\max_i\{\mu_i^s\} > 0.9 \qquad \text{or} \qquad 1.0, \tag{5}$$

depending on the grid level at the previous step in order to avoid fluctuations in the number of grid levels between time steps. If (5) is satisfied then all grid points for which $\mu_i^s > 0.25$ are flagged and surrounding cells are quartered in size.

No derefinement takes place as such, since at each time step the solution on the base grid is computed first and new finer grids are then created based on the new solution. Hence derefinement occurs implicitly. See Section 8.1.

8.3 Time Integration

The time integration is controlled using a time monitor calculated at each level l up to the maximum level used, given by

$$\mu_l^t = \sqrt{\frac{1}{N}\sum_{j=1}^{\mathbf{npde}} w_j^t \sum_{i=1}^{ngpts(l)} \left(\frac{\Delta t}{\alpha_{ij}} u_t(x_i, y_i, t)\right)^2} \tag{6}$$

where $ngpts(l)$ is the total number of points on grid level l; $N = ngpts(l) \times \mathbf{npde}$; Δt is the current time step; u_t is the time derivative of u which is approximated by first-order finite differences; w_j^t is the time equivalent of the space weighting factor w_j^s; and α_{ij} is given by

$$\alpha_{ij} = \tau\left(\frac{u_j^{\max}}{100} + |u(x_i, y_i, t)|\right) \tag{7}$$

where $u_j^{\max}$ is as before, and τ is the user-specified time tolerance.

An integration step is rejected and retried at all levels if

$$\max_l\{\mu_l^t\} > 1.0. \tag{8}$$

9 Example

d03rb_bndary.m

```
function [res] = ...
     bndary(npts, npde, t, x, y, u, ut, ux, uy, nbnds, nbpts, llbnd,
ilbnd, lbnd, res)

      eps = 1e-3;

      for k = llbnd(1):nbpts
         i = lbnd(k);
         a = (-4.0d0*x(i)+4.0d0*y(i)-t)/(32.0d0*eps);
         if (a <= 0.0d0)
            res(i,1) = u(i,1) - (0.75d0-0.25d0/(1.0d0+exp(a)));
            res(i,2) = u(i,2) - (0.75d0+0.25d0/(1.0d0+exp(a)));
         else
                    res(i,1)  =  u(i,1)  -  (0.75d0-0.25d0*exp(-a)/(exp(-
a)+1.0d0));
                    res(i,2)  =  u(i,2)  -  (0.75d0+0.25d0*exp(-a)/(exp(-
a)+1.0d0));
         end
      end
```

d03rb_inidom.m

```
function [xmin, xmax, ymin, ymax, nx, ny, npts, nrows, nbnds, nbpts,
...
          lrow, irow, icol, llbnd, ilbnd, lbnd, ierr] = inidom(maxpts,
ierr)

  nrows = int32(11);
  npts = int32(105);
  nbnds = int32(28);
  nbpts = int32(72);

  lrow = zeros(nrows, 1, 'int32');
  irow = zeros(nrows, 1, 'int32');
  icol = zeros(npts, 1, 'int32');
  llbnd = zeros(nbnds, 1, 'int32');
  ilbnd = zeros(nbnds, 1, 'int32');
  lbnd = zeros(nbpts, 1, 'int32');



  icold = [int32(0),
           int32(1),
           int32(2),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(9),
           int32(10),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(9),
           int32(10),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(9),
           int32(10),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(8),
           int32(9),
           int32(10),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(9),
           int32(10),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(9),
           int32(10),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(9),
           int32(10),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8)];

  ilbndd = [int32(1),
            int32(2),
            int32(3),
            int32(4),
            int32(1),
            int32(4),
            int32(1),
            int32(2),
            int32(3),
            int32(4),
            int32(3),
            int32(4),
            int32(1),
            int32(2),
            int32(12),
            int32(23),
            int32(34),
            int32(41),
            int32(14),
            int32(41),
            int32(12),
            int32(23),
            int32(34),
            int32(41),
            int32(43),
            int32(14),
            int32(21),
            int32(32)];

  irowd = [int32(0),
           int32(1),
           int32(2),
           int32(3),
           int32(4),
           int32(5),
           int32(6),
           int32(7),
           int32(8),
           int32(9),
           int32(10)];

  lbndd = [int32(2),
           int32(4),
           int32(15),
           int32(26),
           int32(37),
           int32(46),
           int32(57),
           int32(68),
           int32(79),
           int32(88),
           int32(98),
           int32(99),
           int32(100),
           int32(101),
           int32(102),
           int32(103),
           int32(104),
           int32(96),
           int32(86),
           int32(85),
           int32(84),
           int32(83),
           int32(82),
           int32(70),
           int32(59),
           int32(48),
           int32(39),
           int32(28),
           int32(17),
           int32(6),
           int32(8),
           int32(9),
           int32(10),
           int32(11),
           int32(12),
           int32(13),
           int32(18),
           int32(29),
           int32(40),
           int32(49),
           int32(60),
           int32(72),
           int32(73),
           int32(74),
           int32(75),
           int32(76),
           int32(77),
           int32(67),
           int32(56),
           int32(45),
           int32(36),
           int32(25),
           int32(33),
           int32(32),
           int32(42),
           int32(52),
           int32(53),
           int32(43),
           int32(1),
           int32(97),
           int32(105),
           int32(87),
           int32(81),
           int32(3),
           int32(7),
           int32(71),
           int32(78),
           int32(14),
           int32(31),
           int32(51),
           int32(54),
           int32(34)];

  llbndd = [int32(1),
            int32(2),
            int32(11),
            int32(18),
            int32(19),
            int32(24),
            int32(31),
            int32(37),
            int32(42),
            int32(48),
            int32(53),
            int32(55),
            int32(56),
            int32(58),
            int32(59),
            int32(60),
            int32(61),
            int32(62),
            int32(63),
            int32(64),
            int32(65),
            int32(66),
            int32(67),
            int32(68),
            int32(69),
            int32(70),
            int32(71),
            int32(72)];

  lrowd = [int32(1),
           int32(4),
           int32(15),
           int32(26),
           int32(37),
           int32(46),
           int32(57),
           int32(68),
           int32(79),
           int32(88),
           int32(97)];
```

```
  nx = int32(11);
  ny = int32(11);

% check maxpts against rough estimate of npts
  if (maxpts < nx*ny)
     ierr = -1;
     return;
  end

  xmin = 0.0d0;
  ymin = 0.0d0;
  xmax = 1.0d0;
  ymax = 1.0d0;

  for i = 1:nrows
    lrow(i) = lrowd(i);
    irow(i) = irowd(i);
  end

  for i = 1:nbnds
    llbnd(i) = llbndd(i);
    ilbnd(i) = ilbndd(i);
  end

  for i = 1:nbpts
    lbnd(i) = lbndd(i);
  end

  for i = 1:npts
    icol(i) = icold(i);
  end
```

```
d03rb_monitr.m
```

```
function [ierr] = ...
      monitr(npde, t, dt, dtnew, tlast, nlev, xmin, ymin, dxb, dyb,
lgrid, ...
            istruc, lsol, sol, ierr)
```

```
d03rb_pdedef.m
```

```
function [res] = pdedef(npts, npde, t, x, y, u, ut, ux, uy, uxx, uxy,
uyy)
  res = zeros(npts, npde);

  eps = 1e-3;
  for i = 1:npts
                         res(i,1)      =      ut(i,1)-(-u(i,1)*ux(i,1)-
u(i,2)*uy(i,1)+eps*(uxx(i,1)+uyy(i,1)));
                         res(i,2)      =      ut(i,2)-(-u(i,1)*ux(i,2)-
u(i,2)*uy(i,2)+eps*(uxx(i,2)+uyy(i,2)));
  end
```

```
d03rb_pdeiv.m
```

```
function [u] = pdeiv(npts, npde, t, x, y)
  u = zeros(npts, npde);

  eps = 1e-3;

  for i = 1:npts
    a = (-4.0d0*x(i)+4.0d0*y(i)-t)/(32.0d0*eps);
    if (a <= 0.0d0)
       u(i,1) = 0.75d0 - 0.25d0/(1.0d0+exp(a));
       u(i,2) = 0.75d0 + 0.25d0/(1.0d0+exp(a));
    else
       u(i,1) = 0.75d0 - 0.25d0*exp(-a)/(exp(-a)+1.0d0);
       u(i,2) = 0.75d0 + 0.25d0*exp(-a)/(exp(-a)+1.0d0);
    end
  end
```

```
ts = 0;
tout = 0.25;
dt = [0.001;
     1e-07;
     0];
tols = 0.1;
tolt = 0.05;
opti = [int32(5);
     int32(0);
     int32(0);
     int32(0)];
optr = [1, 1;
     1, 1;
     1, 1];
rwk = zeros(426000, 1);
iwk = zeros(117037, 1, 'int32');
lenlwk = int32(6000);
itrace = int32(0);
ind = int32(0);
[tsOut, dtOut, rwkOut, iwkOut, indOut, ifail] = ...
    d03rb(ts, tout, dt, tols, tolt, 'd03rb_inidom', 'd03rb_pdedef', ...
    'd03rb_bndary', 'd03rb_pdeiv', 'd03rb_monitr', opti, optr, ...
    rwk, iwk, lenlwk, itrace, ind)
```

```
tsOut =
    0.2500
dtOut =
    0.0249
    0.0000
    0.2500
rwkOut =
     array elided
iwkOut =
     array elided
indOut =
           1
ifail =
           0
```