NAG Toolbox

nag_pde_2d_gen_order2_rectangle (d03ra)

1 Purpose

nag_pde_2d_gen_order2_rectangle (d03ra) integrates a system of linear or nonlinear, time-dependent partial differential equations (PDEs) in two space dimensions on a rectangular 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. nag_pde_2d_gen_order2_rectangle (d03ra) originates from the VLUGR2 package (see Blom and Verwer (1993) and Blom *et al.* (1996)).

2 Syntax

```
[ts, dt, rwk, iwk, ind, ifail] = nag_pde_2d_gen_order2_rectangle(ts, tout, dt,
xmin, xmax, ymin, ymax, nx, ny, tols, tolt, pdedef, bndary, pdeiv, monitr,
opti, optr, rwk, iwk, itrace, ind, 'npde', npde, 'leniwk', leniwk, 'lenlwk',
lenlwk)
[ts, dt, rwk, iwk, ind, ifail] = d03ra(ts, tout, dt, xmin, xmax, ymin, ymax, nx,
ny, tols, tolt, pdedef, bndary, pdeiv, monitr, opti, optr, rwk, iwk, itrace,
ind, 'npde', npde, 'leniwk', leniwk, 'lenlwk', lenlwk)
```

Note: the interface to this routine has changed since earlier releases of the toolbox:

At Mark 22: *lenrwk* was removed from the interface; **lenlwk** was made optional.

3 Description

nag_pde_2d_gen_order2_rectangle (d03ra) integrates the system of PDEs:

$$F_j\big(t,x,y,u,u_t,u_x,u_y,u_{xx},u_{xy},u_{yy}\big)=0, \quad j=1,2,\ldots,\mathbf{npde}, \tag{1}$$

for x and y in the rectangular domain $x_{\min}\le x\le x_{\max}$, $y_{\min}\le y\le y_{\max}$, and time interval $t_0\le t\le t_{\text{out}}$, where 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 **pdedef**. Similarly the initial values of the functions $u(x,y,t)$ must be specified at $t=t_0$ in **pdeiv**.

Note that whilst complete generality is offered by the master equations (1), nag_pde_2d_gen_order2_rectangle (d03ra) 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 **bndary** in the form

$$G_j\big(t,x,y,u,u_t,u_x,u_y\big)=0, \tag{2}$$

for all y when $x_{\min}$ or $x_{\max}$ and for all x when $y=y_{\min}$ or $y=y_{\max}$ and $j=1,2,\ldots,\mathbf{npde}$

The domain is covered by a uniform coarse base grid of size $n_x\times n_y$ 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 9.

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 9 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 $t = t_{\text{out}}$.

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**.

The solution at all grid levels is 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 included. The user-supplied monitor **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 argument **tlast**.

Within **pdeiv**, **pdedef**, **bndary** and **monitr** the data structure is as follows. Each point on a particular grid is given an index (ranging from 1 to the total number of points on the grid) and all coordinate or solution information is stored in arrays according to this index, e.g., $\mathbf{x}(i)$ and $\mathbf{y}(i)$ contain the x- and y coordinate of point i, and $\mathbf{u}(i,j)$ contains the jth solution component u_j at point i.

Further details of the underlying algorithm can be found in Section 9 and in Blom and Verwer (1993) and Blom *et al.* (1996) and the references therein.

4 References

Adjerid S and Flaherty J E (1988) A local refinement finite element method for two-dimensional parabolic systems *SIAM J. Sci. Statist. Comput.* **9** 792–811

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

Brown P N, Hindmarsh A C and Petzold L R (1994) Using Krylov methods in the solution of large scale differential-algebraic systems *SIAM J. Sci. Statist. Comput.* **15** 1467–1488

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** – REAL (KIND=nag_wp)

The initial value of the independent variable t.

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

2: **tout** – REAL (KIND=nag_wp)

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

3: **dt**(**3**) – REAL (KIND=nag_wp) 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**$(1) \geq 0.0$;
if **ind** $= 0$ and **dt**$(1) > 0.0$,
$10.0 \times$ ***machine precision*** $\times \max(|\mathbf{ts}|, |\mathbf{tout}|) \leq$ **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: **xmin** – REAL (KIND=nag_wp)
5: **xmax** – REAL (KIND=nag_wp)

The extents of the rectangular domain in the x-direction, i.e., the x coordinates of the left and right boundaries respectively.

Constraint: **xmin** $<$ **xmax** and **xmax** must be sufficiently distinguishable from **xmin** for the precision of the machine being used.

6: **ymin** – REAL (KIND=nag_wp)
7: **ymax** – REAL (KIND=nag_wp)

The extents of the rectangular domain in the y-direction, i.e., the y coordinates of the lower and upper boundaries respectively.

Constraint: **ymin** $<$ **ymax** and **ymax** must be sufficiently distinguishable from **ymin** for the precision of the machine being used.

8: **nx** – INTEGER

The number of grid points in the x-direction (including the boundary points).

Constraint: **nx** ≥ 4.

9: **ny** – INTEGER

The number of grid points in the y-direction (including the boundary points).

Constraint: **ny** ≥ 4.

10: **tols** – REAL (KIND=nag_wp)

The space tolerance used in the grid refinement strategy (σ in equation (4)). See Section 9.2.

Constraint: **tols** > 0.0.

11: **tolt** – REAL (KIND=nag_wp)

The time tolerance used to determine the time step size (τ in equation (7)). See Section 9.3.

Constraint: **tolt** > 0.0.

12: **pdedef** – SUBROUTINE, supplied by the user.

pdedef must evaluate the functions F_j, for $j = 1, 2, \ldots,$ **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 **bndary**. **pdedef** is called for each subgrid in turn.

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

Input Parameters

1: **npts** – INTEGER

The number of grid points in the current grid.

2: **npde** – INTEGER

The number of PDEs in the system.

3: **t** – REAL (KIND=nag_wp)

The current value of the independent variable t.

4: **x**(**npts**) – REAL (KIND=nag_wp) array

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

5: **y**(**npts**) – REAL (KIND=nag_wp) array

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

6: **u**(**npts**, **npde**) – REAL (KIND=nag_wp) 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**) – REAL (KIND=nag_wp) array

ut(i, j) contains the value of $\dfrac{\partial u}{\partial t}$ 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**) – REAL (KIND=nag_wp) array

ux(i, j) contains the value of $\dfrac{\partial u}{\partial x}$ 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**) – REAL (KIND=nag_wp) array

uy(i,j) contains the value of $\dfrac{\partial u}{\partial y}$ 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**) – REAL (KIND=nag_wp) array

uxx(i,j) contains the value of $\dfrac{\partial^2 u}{\partial x^2}$ 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**) – REAL (KIND=nag_wp) array

uxy(i,j) contains the value of $\dfrac{\partial^2 u}{\partial x \partial y}$ 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**) – REAL (KIND=nag_wp) array

uyy(i,j) contains the value of $\dfrac{\partial^2 u}{\partial y^2}$ 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**) – REAL (KIND=nag_wp) 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.

13: **bndary** – SUBROUTINE, supplied by the user.

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 function.

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

Input Parameters

1: **npts** – INTEGER

The number of grid points in the current grid.

2: **npde** – INTEGER

The number of PDEs in the system.

3: **t** – REAL (KIND=nag_wp)

The current value of the independent variable t.

4: **x**(**npts**) – REAL (KIND=nag_wp) array

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

5: **y**(**npts**) – REAL (KIND=nag_wp) array

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

6: **u**(**npts**, **npde**) – REAL (KIND=nag_wp) 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**) – REAL (KIND=nag_wp) array

ut(i,j) contains the value of $\dfrac{\partial u}{\partial t}$ 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**) – REAL (KIND=nag_wp) array

ux(i,j) contains the value of $\dfrac{\partial u}{\partial x}$ 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**) – REAL (KIND=nag_wp) array

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

10: **nbpts** – INTEGER

The number of boundary points in the grid.

11: **lbnd**(**nbpts**) – INTEGER array

lbnd(i) contains the grid index for the ith boundary point, for $i = 1, 2, \ldots,$ **nbpts**. Hence the ith boundary point has coordinates **x**(**lbnd**(i)) and **y**(**lbnd**(i)), and the corresponding solution values are **u**(**lbnd**(i), **npde**), etc.

12: **res**(**npts**, **npde**) – REAL (KIND=nag_wp) array

res(i,j) contains the value of F_j, for $i = 1, 2, \ldots,$ **npde**, at the ith grid point, for $i = 1, 2, \ldots,$ **npts**, as returned by **pdedef**. The residuals at the boundary points will be overwritten and so need not have been set by **pdedef**.

Output Parameters

1: **res**(**npts**, **npde**) – REAL (KIND=nag_wp) array

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 must **not** be altered.

14: **pdeiv** – SUBROUTINE, supplied by the user.

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

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

Input Parameters

1: **npts** – INTEGER

The number of grid points in the grid.

2: **npde** – INTEGER

The number of PDEs in the system.

3: **t** – REAL (KIND=nag_wp)

The (initial) value of the independent variable t.

4: **x**(**npts**) – REAL (KIND=nag_wp) array

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

5: **y**(**npts**) – REAL (KIND=nag_wp) array

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

Output Parameters

1: **u**(**npts**, **npde**) – REAL (KIND=nag_wp) 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**.

15: **monitr** – SUBROUTINE, supplied by the user.

monitr is called by nag_pde_2d_gen_order2_rectangle (d03ra) 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 argument **tlast**. The input arguments contain information about the grid and solution at all grid levels used.

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

```
[ierr] = monitr(npde, t, dt, dtnew, tlast, nlev, ngpts, xpts, ypts, lsol,
sol, ierr)
```

Input Parameters

1: **npde** – INTEGER

The number of PDEs in the system.

2: **t** – REAL (KIND=nag_wp)

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

3: **dt** – REAL (KIND=nag_wp)

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

4: **dtnew** – REAL (KIND=nag_wp)

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

5: **tlast** – LOGICAL

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** – INTEGER

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

7: **ngpts**(**nlev**) – INTEGER array

ngpts(l) contains the number of grid points at level l, for $l = 1, 2, \ldots,$ **nlev**.

8: **xpts**($lpts$) – REAL (KIND=nag_wp) array

Contains the x coordinates of the grid points in each level in turn, i.e., **x**(i), for $i = 1, 2, \ldots,$ **ngpts**(l) and $l = 1, 2, \ldots,$ **nlev**.

So for level l, **x**(i) = **xpts**($k + i$), where $k =$ **ngpts**(1) + **ngpts**(2) + $\cdots$ + **ngpts**($l - 1$), for $i = 1, 2, \ldots,$ **ngpts**(l) and $l = 1, 2, \ldots,$ **nlev**.

9: **ypts**($lpts$) – REAL (KIND=nag_wp) array

Contains the y coordinates of the grid points in each level in turn, i.e., **y**(i), for $i = 1, 2, \ldots,$ **ngpts**(l) and $l = 1, 2, \ldots,$ **nlev**.

So for level l, **y**(i) = **ypts**($k + i$), where $k =$ **ngpts**(1) + **ngpts**(2) + $\cdots$ + **ngpts**($l - 1$), for $i = 1, 2, \ldots,$ **ngpts**(l) and $l = 1, 2, \ldots,$ **nlev**.

10: **lsol**(**nlev**) – INTEGER 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**.)

11: **sol**(:) – REAL (KIND=nag_wp) array

Default: $lenrwk - 6 \times$ **npde**

Contains the solution **u**(**ngpts**(l), **npde**) at time **t** for each grid level l in turn, positioned according to **lsol**, i.e., for level l, **u**(i, j) = **sol**(**lsol**(l) + ($j - 1$) $\times$ **ngpts**(l) + i), for $i = 1, 2, \ldots,$ **ngpts**(l), $j = 1, 2, \ldots,$ **npde** and $l = 1, 2, \ldots,$ **nlev**.

12: **ierr** – INTEGER

Will be set to 0.

Output Parameters

1: **ierr** – INTEGER

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

16: **opti**(**4**) – INTEGER 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 $i = 2, 3, 4$, 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) ≥ 0. The default value is **opti**(4) $= 100$.

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

17: **optr**(**3**, **npde**) – REAL (KIND=nag_wp) array

May be used to specify the optional vectors $u^{\max}$, w^{s} and w^{t} in the space and time monitors (see Section 9).

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**.

Constraints:

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

18: **rwk**($lenrwk$) – REAL (KIND=nag_wp) array

$lenrwk$, the dimension of the array, must satisfy the constraint $lenrwk \geq$ **nx** $\times$ **ny** $\times$ **npde** $\times (14 + 18 \times$ **npde**$) + 2 \times$ **nx** $\times$ **ny** (the required size for the initial grid).

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

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

where $l =$ **opti**(1) if **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 nag_file_set_unit_error (x04aa)).

Constraint: $lenrwk \geq$ **nx** $\times$ **ny** $\times$ **npde** $\times (14 + 18 \times$ **npde**$) + 2 \times$ **nx** $\times$ **ny** (the required size for the initial grid).

19: **iwk**(**leniwk**) – INTEGER array

If **ind** $= 0$, **iwk** need not be set. Otherwise **iwk** must remain unchanged from a previous call to nag_pde_2d_gen_order2_rectangle (d03ra).

20: **itrace** – INTEGER

The level of trace information required from nag_pde_2d_gen_order2_rectangle (d03ra). **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 nag_file_set_unit_advisory (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.

21: **ind** – INTEGER

Must be set to 0 or 1, alternatively 10 or 11.

ind $= 0$
Starts the integration in time. **pdedef** is assumed to be serial.

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 nag_pde_2d_gen_order2_rectangle (d03ra): **tout**, **dt**, **tols**, **tolt**, **opti**, **optr**, **itrace** and **ifail**. **pdedef** is assumed to be serial.

ind $= 10$
Equivalent to **ind** $= 0$. This option is included only for compatibility with other NAG library products.

ind $= 11$
Equivalent to **ind** $= 1$. This option is included only for compatibility with other NAG library products.

Constraint: $0 \leq$ **ind** ≤ 1 or $10 \leq$ **ind** ≤ 11.

5.2 Optional Input Parameters

1: **npde** – INTEGER

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

The number of PDEs in the system.

Constraint: **npde** ≥ 1.

2: **leniwk** – INTEGER

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

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 $m =$ **opti**(1) if **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 nag_file_set_unit_error (x04aa)).

Constraint: **leniwk** $\geq 19 \times$ **nx** $\times$ **ny** $+ 9$ (the required size for the initial grid).

3: **lenlwk** – INTEGER

Default: $maxpts + 1$

The dimension of the array **lwk**.

The required value of **lenlwk** cannot be determined exactly in advanced, 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 nag_file_set_unit_error (x04aa)).

Constraint: **lenlwk** $\geq$ **nx** $\times$ **ny** $+ 1$ (the required size for the initial grid).

5.3 Output Parameters

1: **ts** – REAL (KIND=nag_wp)

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

2: **dt**(**3**) – REAL (KIND=nag_wp) 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**$(lenrwk)$ – REAL (KIND=nag_wp) array

$lenrwk =$ **nx** $\times$ **ny** $\times$ **npde** $\times (14 + 18 \times$ **npde**$) + 2 \times$ **nx** $\times$ **ny** (the required size for the initial grid).

Communication array, used to store information between calls to nag_pde_2d_gen_order2_rectangle (d03ra).

4: **iwk**(**leniwk**) – INTEGER array

The following components of the array **iwk** concern the efficiency of the integration. Here, m is the maximum number of grid levels allowed ($m =$ **opti**(1) if **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 **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 nag_pde_2d_gen_order2_rectangle (d03ra). 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** – INTEGER

ind $= 1$, if **ind** on input was 0 or 1, or **ind** $= 11$, if **ind** on input was 10 or 11.

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 $= 1$

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*** $\times \max(|\mathbf{ts}|, |\mathbf{tout}|)$,
or **ind** $= 0$ and **dt**$(1) >$ **tout** $-$ **ts**,
or **ind** $= 0$ and **dt**$(1) <$ **dt**(2) or **dt**$(1) >$ **dt**(3),
or **xmin** $\geq$ **xmax**,
or **xmax** too close to **xmin**,
or **ymin** $\geq$ **ymax**,
or **ymax** too close to **ymin**,
or **nx** or **ny** < 4,
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 $lenrwk$, **leniwk** or **lenlwk** too small for initial grid level,
or **ind** $\neq 0$ or 1,
or **ind** $= 1$ on initial entry to nag_pde_2d_gen_order2_rectangle (d03ra).

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 **pdedef** and/or **bndary**, and the initial values in **pdeiv** if appropriate.

ifail $= 3$

One or more of the workspace arrays is too small for the required number of grid points. An estimate of the required sizes for the current stage is output, but more space may be required at a later stage.

ifail $= 4$ (*warning*)

ierr was set to 1 in **monitr**, forcing control to be passed back to calling program. Integration was successful as far as **t** $=$ **ts**.

ifail $= 5$ (*warning*)

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 $= -99$

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.

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 arguments **tols** and **tolt** described in the following section, and you should test the effects of varying these arguments. 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 summarised 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

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

where Δx and Δy are the grid widths in the x and y directions; and x_i, y_i are the x and y coordinates at grid point i. The argument γ_j is obtained from

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

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 value 1.0 if little information about the solution is known.

A new level of refinement is created if

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

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 9.1.

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

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

where $\mathbf{ngpts}(l)$ is the total number of points on grid level l; $N = \mathbf{ngpts}(l) \times \mathbf{npde}$; 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

For this function two examples are presented, with a main program and two example problems given in Example 1 (EX1) and Example 2 (EX2).

Example 1 (EX1)

This example stems from combustion theory and is a model for a single, one-step reaction of a mixture of two chemicals (see Adjerid and Flaherty (1988)). The PDE for the temperature of the mixture u is

$$\frac{\partial u}{\partial t} = d\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + D(1 + \alpha - u)\exp\left(-\frac{\delta}{u}\right)$$

for $0 \leq x, y \leq 1$ and $t \geq 0$, with initial conditions $u(x, y, 0) = 1$ for $0 \leq x, y \leq 1$, and boundary conditions

$$u_x(0, y, t) = 0, u(1, y, t) = 1 \quad \text{for } 0 \leq y \leq 1,$$
$$u_y(x, 0, t) = 0, u(x, 1, t) = 1 \quad \text{for } 0 \leq x \leq 1.$$

The heat release argument $\alpha = 1$, the Damkohler number $D = R\exp(\delta)/(\alpha\delta)$, the activation energy $\delta = 20$, the reaction rate $R = 5$, and the diffusion argument $d = 0.1$.

For small times the temperature gradually increases in a circular region about the origin, and at about $t = 0.24$ 'ignition' occurs causing the temperature to suddenly jump from near unity to $1 + \alpha$, and a reaction front forms and propagates outwards, becoming steeper. Thus during the solution, just one grid level is used up to the ignition point, then two levels, and then three as the reaction front steepens.

Example 2 (EX2)

This example is taken from a multispecies food web model, in which predator-prey relationships in a spatial domain are simulated (see Brown *et al.* (1994)). In this example there is just one species each of prey and predator, and the two PDEs for the concentrations c_1 and c_2 of the prey and the predator respectively are

$$\frac{\partial c_1}{\partial t} = c_1(b_1 + a_{11}c_1 + a_{12}c_2) + d_1\left(\frac{\partial^2 c_1}{\partial x^2} + \frac{\partial^2 c_1}{\partial y^2}\right),$$

$$0 = c_2(b_2 + a_{21}c_1 + a_{22}c_2) + d_2\left(\frac{\partial^2 c_2}{\partial x^2} + \frac{\partial^2 c_2}{\partial y^2}\right),$$

with

$$\begin{aligned} &a_{11} = a_{22} = -1, \\ &a_{12} = -0.5 \times 10^{-6}, \text{ and} \\ &a_{21} = 10^4, \text{ and} \\ &b_1 = 1 + \alpha xy + \beta\sin(4\pi x)\sin(4\pi y), \end{aligned}$$

where $\alpha = 50$ and $\beta = 300$, and $b_2 = -b_1$.

The initial conditions are taken to be simple peaked functions which satisfy the boundary conditions and very nearly satisfy the PDEs:

$$c_1 = 10 + (16x(1-x)y(1-y))^2,$$

$$c_2 = b_2 + a_{21}c_1,$$

and the boundary conditions are of Neumann type, i.e., zero normal derivatives everywhere.

During the solution a number of peaks and troughs develop across the domain, and so the number of levels required increases with time. Since the solution varies rapidly in space across the whole of the domain, refinement at intermediate levels tends to occur at all points of the domain.

9.1 Program Text

```
    function d03ra_example

fprintf('d03ra example results\n\n');

global ssav1 ssav2;
global xsav ysav tsav;

ex1;
fig1 = figure;
ex1_plot;

ex2;
fig2 = figure;
ex2_plot(1);
fig3 = figure;
ex2_plot(2);

function ex1
% First d03ra example.  Temperature distribution for a single, one-step
% reaction of a mixture of two chemicals.
global heat_release activ_energy diffusion damkohler;
global iout isav;
```

```
% Set values for problem parameters.
mxlev = 3;
npde = 1;
npts = 2000;
xmin = 0;    xmax = 1;
ymin = 0;    ymax = 1;
tols = 0.5;
tolt = 0.01;
nx   = nag_int(21);
ny   = nx;

reaction_rate =  5;
activ_energy  = 20;
heat_release  =  1;
diffusion     =  0.1;
damkohler = reaction_rate*exp(activ_energy)/(heat_release*activ_energy);

leniwk = npts*(14 + 5*mxlev) + 7*mxlev + 2;
lenlwk = npts + 1;
lenrwk = npts*npde*(5*mxlev + 18*npde + 9) + 2*npts;

% The above are the recommended values from the documentation, but d03ra
% returns with ifail = 3 and the minumum workspace size requirements if
% these are used.  So we beef them up here with some fudge factors.
leniwk = 2*leniwk;
lenlwk = nag_int(2*lenlwk);
lenrwk = 2*lenrwk;

% Create some work arrays - see NAG documentation.
rwk = zeros(lenrwk, 1);
iwk = zeros(leniwk, 1, nag_int_name);

% Initialize some problem variables.
ind    = nag_int(0);
itrace = nag_int(0);
ts     = 0;
dt     = [0.1e-2; 0.0; 0.0];
twant  = [0.24; 0.25];
opti = zeros(4, 1, nag_int_name);
optr = ones(3, npde);

% index of next set of saved results
isav = 1;

% We run through the problem twice - once from t=0 up to t=twant(1), and
% then onwards to t=twant(2).
warning('Off');
fprintf('Example 1\n==========\n\n');
for iout = 1:2
  tout = twant(iout);
  [ts, dt, rwk, iwk, ind, ifail] = ...
  d03ra( ...
         ts, tout, dt, xmin, xmax, ymin, ymax, nx, ny, tols, tolt, ...
         @ex1_pdedef, @ex1_bndary, @ex1_pdeiv, @ex1_monitr, opti, ...
         optr, rwk, iwk, itrace, ind, 'lenlwk', lenlwk);

  fprintf('\nTime =%8.4f \n', ts);
  fprintf('Total number of accepted timesteps %5d \n', iwk(1));
  fprintf('Total number of rejected timesteps %5d \n', iwk(2));
  fprintf('\n              Total (max) number of\n');
  fprintf('          Residual  Jacobian    Newton   Lin sys\n');
  fprintf('             evals     evals     iters     iters\n');
  fprintf('At level\n');
  maxlev = 3;
  for j = 1:maxlev
    if iwk(j+2) ~= 0
      evals = iwk(j+2:maxlev:j+2+5*maxlev);
      fprintf('%8d%10d%10d%6d(%2d)%6d(%2d)\n', j, evals);
    end;
  end;
end
```

```
function [res] = ex1_pdedef(npts, npde, t, x, y, u, ut, ux, uy, uxx, uxy, uyy)
  % Evaluate the system of PDEs for this problem.

  global heat_release activ_energy diffusion damkohler;

  res = zeros(npts, npde);
  for i = 1:npts
    e = exp(-activ_energy/u(i,1));
    res(i,1) = ut(i,1) - diffusion*(uxx(i,1) + uyy(i,1)) - ...
               damkohler*(1.0 + heat_release - u(i,1))*e;
  end

function [u] = ex1_pdeiv(npts, npde, t, x, y)
  % Evaluate initial conditions for PDEs for this system.

  u = ones(npts, npde);

function [res] = ex1_bndary(npts, npde, t, x, y, u, ut, ux, uy, ...
                            nbpts, lbnd, res)
  % Implement boundary conditions for the domain.

  tol = 10*x02aj();
  for i = 1:nbpts
    j = lbnd(i);
    if (abs(x(j)) <= tol)
      res(j,1) = ux(j,1);
    elseif (abs(x(j)-1) <= tol)
      res(j,1) = u(j,1) - 1;
    elseif (abs(y(j)) <= tol)
      res(j,1) = uy(j,1);
    elseif (abs(y(j)-1) <= tol)
      res(j,1) = u(j,1) - 1;
    end
end

function [ierr] = ex1_monitr(npde, t, dt, dtnew, tlast, nlev, ...
                             ngpts, xpts, ypts, lsol, sol, ierr)
  % Save the results at specified times.
  times = [0.001; 0.228; 0.240; 0.25];

  global iout isav;
  global xsav ysav ssav1 tsav;

  % Save this time step only if the current time is equal to
  % or just past the next output time
  if (t < times(isav))
    return;
  end
  fprintf('Saving set %d at time = %f\n', isav, t);

  % Specify the grid level for extracting the solution.
  level = 1;
  npts = ngpts(level);
  ipsol = lsol(level);

  % Allocate space for saves the first time through.
  nside = round(sqrt(double(npts)));
  if isav == 1
    nsav = length(times);
    xsav = zeros(nside, 1);
    ysav = zeros(nside, 1);
    ssav1 = zeros(nside, nside, nsav);
    tsav = zeros(nsav, 1);
  end

  % Save this time step.
  tsav(isav) = t;
  for i = 1:nside
    for j = 1:nside
      k = (i-1)*nside + j;
```

```
      xsav(j) = xpts(k);
      ysav(i) = ypts(k);
      ssav1(j, i, isav) = sol(ipsol+k);
    end
  end

  % Look for the next time step, unless called for very last time.
  if ~(tlast && iout == 2)
    isav = isav+1;
  end

function ex1_plot
  % Plot the results.
  global xsav ysav ssav1 tsav isav;
  % Set the colours for each surface, and the array of handles for the plots.
  colours = [[1 0 0]; [0 1 0]; [1 0 1]; [0 0 1]; [0 1 1]; [2 1 0]; [2 0 1];
             [1 0 2]; [1 2 0]; [0 1 2]; [0 2 1]; [3 2 1]; [2 3 1]; [2 1 3];
             [3 1 2]; [1 3 2]; [1 2 3]; [0.3 2 1]; [2 0.3 1]; [2 1 0.3];
             [0.3 1 2]; [1 0.3 2]; [1 2 0.3]; [0.3 0.2 1]; [0.2 0.3 1];
             [0.2 1 0.3]; [0.3 1 0.2]; [1 0.3 0.2]; [1 0.2 0.3]];

  h = zeros(isav,1);
  % Check the allocation of colours against the number to be plotted.
  if (length(colours(:,1)) < isav)
    fprintf(['Not enough colours allocated in example1_plot: ', ...
             '%d needed\n'], isav);
  end
  % Plot all the surfaces, and colour each one.
  for i = 1:isav
    h(i) = mesh(xsav, ysav, ssav1(:,:,i));
    hold on;
    set(h(i), 'EdgeColor', colours(i,:));
  end
  hold off;
  % Label the axes, and set the title.
  xlabel('x');
  ylabel('y');
  zlabel('U(x,y,t)');
  title({'Model for a Single, One-step Reaction', ...
         'of a Mixture of Two Chemicals'});
  % Add a legend, using the plot handles and the array of time values.
  hleg = legend(h, num2str(tsav(1:isav)));
  % Set the view to something nice (determined empirically).
  view(15, 30);


function ex2
  % Second d03ra example.  Concentration distribution in a multispecies
  % (predator-prey) food web model.
  global alpha beta;
  global iout icount;

  % Set values for problem parameters.
  npde = nag_int(2);
  npts = nag_int(4000);
  xmin = 0;  xmax = 1;
  ymin = 0;  ymax = 1;
  tols = 0.075;
  tolt = 0.1;
  nx = nag_int(21);
  ny = nag_int(21);
  alpha = 50.0;
  beta = 300.0;

  maxlev = 4;
  leniwk = npts*(14 + 5*maxlev) + 7*maxlev + 2;
  lenlwk = npts + 1;
  lenrwk = npts*npde*(5*maxlev + 18*npde + 9) + 2*npts;

  % The above are the recommended values from the documentation, but d03ra
  % returns with ifail = 3 and the minumum workspace size requirements if
```

```
  % these are used.  So we beef them up here with some fudge factors.
  leniwk = 2*leniwk;
  lenlwk = nag_int(3*lenlwk);
  lenrwk = 3*lenrwk;

  % Create some work arrays - see NAG documentation.
  rwk = zeros(lenrwk, 1);
  iwk = zeros(leniwk, 1, nag_int_name);

% Initialize some problem variables.
  ind    = nag_int(0);
  itrace = nag_int(0);
  ts     = 0;
  dt     = [0.5e-3; 1.0e-6; 0.0];
  twant  = [0.01; 0.025];
  opti    = zeros(4, 1, nag_int_name);
  opti(1) = 4;
  optr      = ones(3, npde);
  optr(1,1) = 250;
  optr(1,2) = 1.5e+6;

  % Counts how many times monitor routine is called.
  icount = 0;
  fprintf('\n\nExample 2\n=========\n\n');

  % We run through the problem twice - once from t=0 up to t=twant(1), and
  % then onwards to t=twant(2).
  for iout = 1:2
    tout = twant(iout);
    [ts, dt, rwk, iwk, ind, ifail] = ...
    d03ra( ...
           ts, tout, dt, xmin, xmax, ymin, ymax, nx, ny, tols, tolt, ...
           @ex2_pdedef, @ex2_bndary, @ex2_pdeiv, @ex2_monitr, opti, optr, ...
           rwk, iwk, itrace, ind, 'lenlwk', lenlwk);

    % Output some statistics.
    fprintf('\nTime =%8.4f \n', ts);
    fprintf('Total number of accepted timesteps %5d \n', iwk(1));
    fprintf('Total number of rejected timesteps %5d \n', iwk(2));
    fprintf('\n              Total (max) number of\n');
    fprintf('          Residual  Jacobian    Newton   Lin sys\n');
    fprintf('             evals     evals     iters     iters\n');
    fprintf('At level\n');
    for j = 1:maxlev
      if iwk(j+2) ~= 0
        evals = iwk(j+2:maxlev:j+2+5*maxlev);
        fprintf('%8d%10d%10d%6d(%2d)%6d(%2d)\n', j, evals);
      end;
    end;
end

function [res] = ex2_pdedef(npts, npde, t, x, y, u, ut, ux, uy, ...
                            uxx, uxy, uyy)
  % Evaluate the system of PDEs for this problem.

  global alpha beta;

  fp = 4*pi;
  res = zeros(npts, npde);
  for i = 1:npts
    b1 = 1 + alpha*x(i)*y(i) + beta*sin(fp*x(i))*sin(fp*y(i));
    res(i,1) = ut(i,1) - uxx(i,1) - uyy(i,1) - ...
               u(i,1)*(b1 - u(i,1) - 0.5e-6*u(i,2));
    res(i,2) = -0.05*(uxx(i,2) + uyy(i,2)) - ...
               u(i,2)*(-b1 + 1.0e4*u(i,1) - u(i,2));
  end

function [u] = ex2_pdeiv(npts, npde, t, x, y)
  % Evaluate initial conditions for PDEs for this system.

  global alpha beta;
```

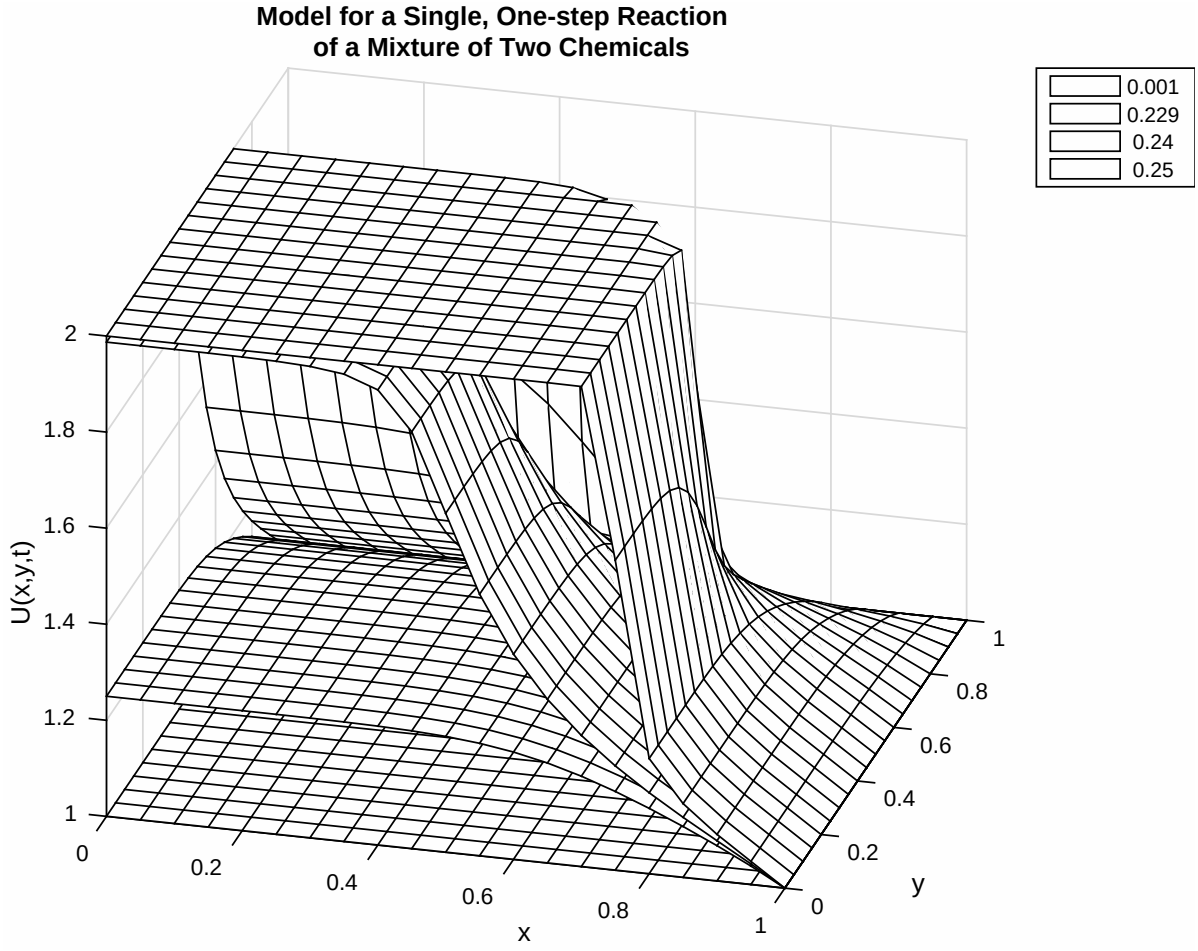
```
  fp = 4*pi;
  u = zeros(npts, npde);
  for i = 1:npts
    u(i,1) = 10 + (16*x(i)*(1 - x(i))*y(i)*(1 - y(i)))^2;
    u(i,2) = -1 - alpha*x(i)*y(i) - beta*sin(fp*x(i))*sin(fp*y(i)) + ...
             1.0e4*u(i,1);
  end

function [res] = ex2_bndary(npts, npde, t, x, y, u, ut, ux, uy, ...
                            nbpts, lbnd, res)
  % Implement boundary conditions for the domain.

  tol = 10*x02aj;
  for i = 1:nbpts
    j = lbnd(i);
    if (abs(x(j)) <= tol || abs(x(j)-1) <= tol)
      res(j,1) = ux(j,1);
      res(j,2) = ux(j,2);
    elseif (abs(y(j)) <= tol || abs(y(j)-1) <= tol)
      res(j,1) = uy(j,1);
      res(j,2) = uy(j,2);
    end
  end

function [ierr] = ex2_monitr(npde, t, dt, dtnew, tlast, nlev, ...
                             ngpts, xpts, ypts, lsol, sol, ierr)
  % Monitor the results at specified intervals.

  global iout icount;
  global xsav ysav ssav2 tsav isav;

  % Specify the maximum number of times the results get saved, and the
  % interval for saves.
%  nsav = 5;
%  nint = 10;
 size(xpts);
 size(ypts);

  % Do we want to save this time?
  icount = icount+1;
 % if mod(icount,nint) ~= 1 && ~tlast
 %   return;
 % end

 % isav = 1 + round(icount/nint);
 % if isav > nsav
 %   fprintf('Not enough save space allocated in monitr2: %d\n', ...
 %           nsav);
 %   return;
 % end
  isav = icount;

  % Specify the grid level for extracting solution.
  level = 1;
  npts = ngpts(level);
  ipsol = lsol(level);

  % Allocate space for saves the first time through.
  nside = round(sqrt(double(npts)));

  % Save this time step.  For this problem, calculating two
  % concentrations as a function of x and y (and time).
  tsav(isav) = t;
  if (icount==1)
    xsav(1:nside) = xpts(1:nside);
    ysav(1:nside) = ypts(1:nside:nside^2);
  end

  s1 = reshape(sol(ipsol+1:ipsol+nside^2),[nside,nside]);
  s2 = reshape(sol(ipsol+npts+1:ipsol+npts+nside^2),[nside,nside]);
```

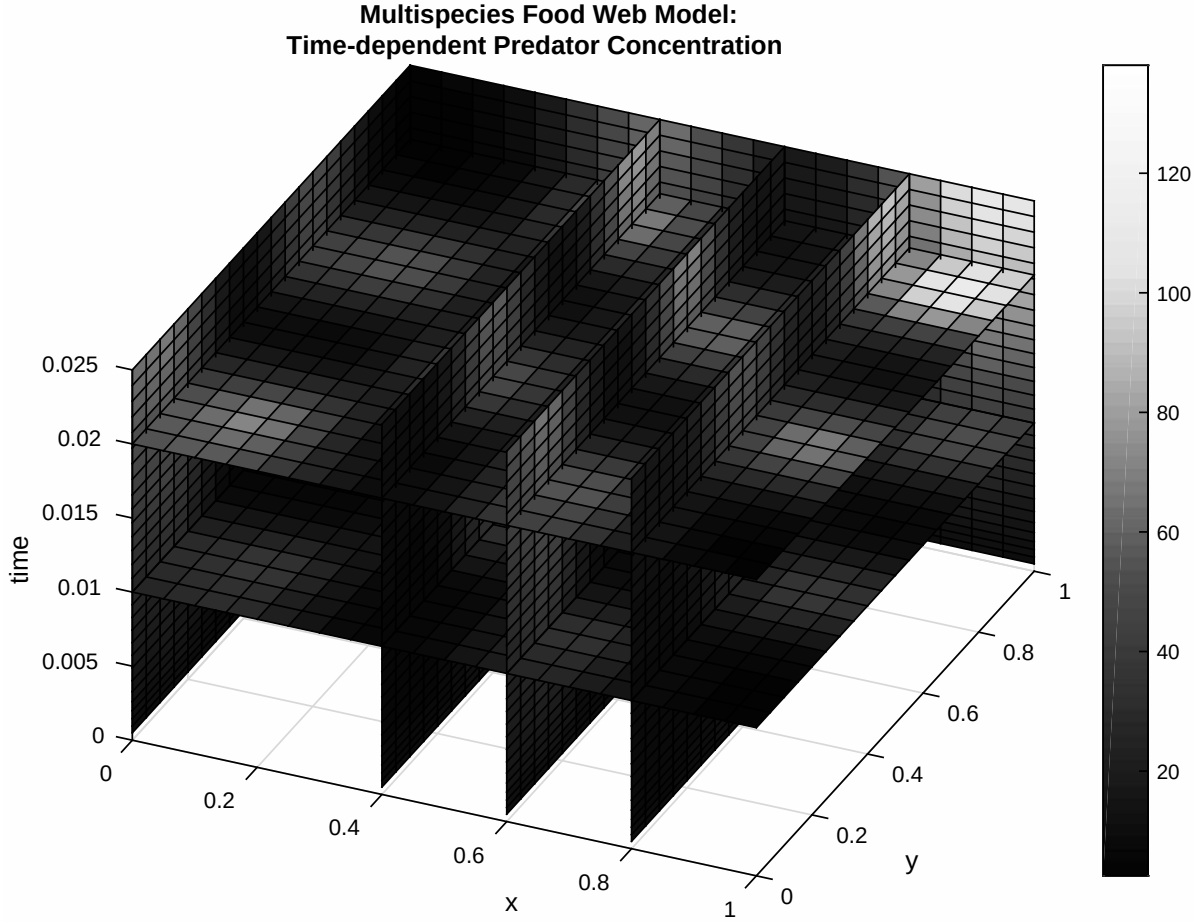
```
  ssav2(:,:,isav,1) = s1';
  ssav2(:,:,isav,2) = s2';

function ex2_plot(id)
  % Plot the results.
  global xsav ysav ssav2 tsav isav;
  % Check the value of the array identifier.
  if (id ~= 1 && id ~= 2)
    fprintf('Illegal value for array identifier in plot: %d\n', id);
    return;
  end
  % Label the axes, and set the title according to the array identifier.
  v = ssav2(:,:,:,id);
  slice(xsav, ysav, tsav, v ,[0,0.4,0.6,0.8],[1],[0.01,0.02]);
  colormap hot;
  colorbar;
  xlabel('x');
  ylabel('y');
  zlabel('time');
  if (id == 1)
    title({'Multispecies Food Web Model:', ...
           'Time-dependent Predator Concentration'});
  else
    title({'Multispecies Food Web Model:', ...
           'Time-dependent Prey Concentration'});
  end
  % Add a legend, using the plot handles and the array of time values.
  % Set the view to something nice (determined empirically).
  view(24,42);
```

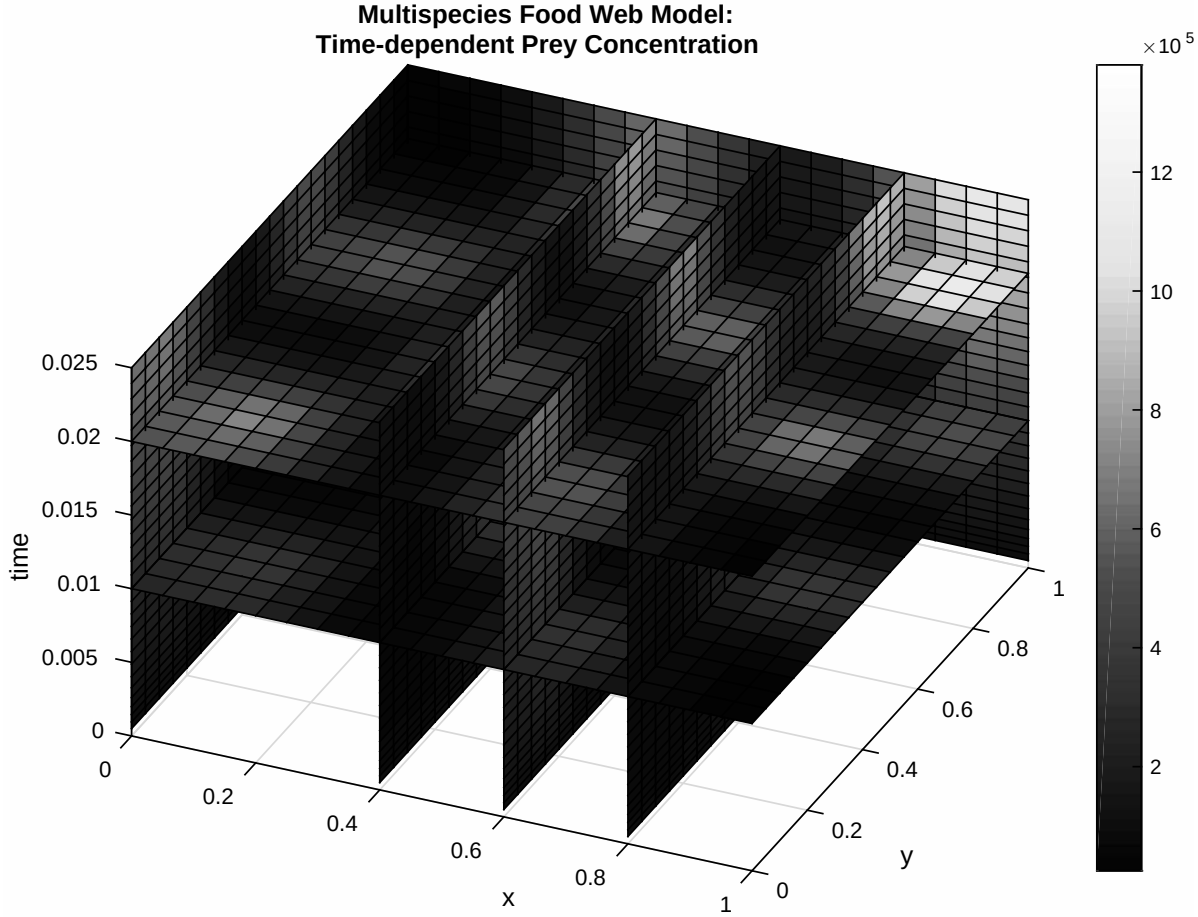
9.2 Program Results

```
     d03ra example results

Example 1
==========

Saving set 1 at time = 0.001000
Saving set 2 at time = 0.228996
Saving set 3 at time = 0.240000

Time =  0.2400
Total number of accepted timesteps    75
Total number of rejected timesteps     0

              Total (max) number of
          Residual  Jacobian    Newton   Lin sys
             evals     evals     iters     iters
At level
       1       600        75   150(159)     2( 2)
Saving set 4 at time = 0.250000

Time =  0.2500
Total number of accepted timesteps   180
Total number of rejected timesteps     1

              Total (max) number of
          Residual  Jacobian    Newton   Lin sys
             evals     evals     iters     iters
At level
       1      1468       181   382(391)     4( 2)
       2       662        82   170(170)     4( 1)
       3       177        22    45(45)     3( 1)

Example 2
=========

Time =  0.0100
Total number of accepted timesteps    14
Total number of rejected timesteps     0
```

```
                Total (max) number of
           Residual  Jacobian    Newton   Lin sys
              evals     evals     iters     iters
At level
       1       196        14    28(39)     2( 2)
       2        84         6    12(19)     2( 3)

Time =  0.0250
Total number of accepted timesteps    29
Total number of rejected timesteps     0

                Total (max) number of
           Residual  Jacobian    Newton   Lin sys
              evals     evals     iters     iters
At level
       1       406        29    58(84)     2( 2)
       2       294        21    42(64)     2( 3)
       3        98         7    14(28)     2( 3)
```




Multispecies Food Web Model:
Time-dependent Predator Concentration
time
0.025
0.02
0.015
0.01
0.005
0
0
0.2
0.4
0.6
0.8
1
x
0
0.2
0.4
0.6
0.8
1
y
120
100
80
60
40
20


Multispecies Food Web Model:
Time-dependent Prey Concentration
time
0.025
0.02
0.015
0.01
0.005
0
0
0.2
0.4
0.6
0.8
1
x
y
×10^5
12
10
8
6
4
2