NAG Toolbox

nag_pde_1d_parab_convdiff_remesh (d03ps)

1 Purpose

nag_pde_1d_parab_convdiff_remesh (d03ps) integrates a system of linear or nonlinear convection-diffusion equations in one space dimension, with optional source terms and scope for coupled ordinary differential equations (ODEs). The system must be posed in conservative form. This function also includes the option of automatic adaptive spatial remeshing. Convection terms are discretized using a sophisticated upwind scheme involving a user-supplied numerical flux function based on the solution of a Riemann problem at each mesh point. The method of lines is employed to reduce the partial differential equations (PDEs) to a system of ODEs, and the resulting system is solved using a backward differentiation formula (BDF) method or a Theta method.

2 Syntax

```
[ts, u, x, rsave, isave, ind, ifail] = nag_pde_1d_parab_convdiff_remesh(npde,
ts, tout, pdedef, numflx, bndary, uvinit, u, x, ncode, odedef, xi, rtol, atol,
itol, norm_p, laopt, algopt, remesh, xfix, nrmesh, dxmesh, trmesh, ipminf,
monitf, rsave, isave, itask, itrace, ind, 'npts', npts, 'nxi', nxi, 'neqn',
neqn, 'nxfix', nxfix, 'xratio', xratio, 'con', con)
```

```
[ts, u, x, rsave, isave, ind, ifail] = d03ps(npde, ts, tout, pdedef, numflx,
bndary, uvinit, u, x, ncode, odedef, xi, rtol, atol, itol, norm_p, laopt,
algopt, remesh, xfix, nrmesh, dxmesh, trmesh, ipminf, monitf, rsave, isave,
itask, itrace, ind, 'npts', npts, 'nxi', nxi, 'neqn', neqn, 'nxfix', nxfix,
'xratio', xratio, 'con', con)
```

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

At Mark 22: *lrsave* and *lisave* were removed from the interface; **nxi** was made optional.

3 Description

nag_pde_1d_parab_convdiff_remesh (d03ps) integrates the system of convection-diffusion equations in conservative form:

$$\sum_{j=1}^{\mathbf{npde}} P_{i,j}\frac{\partial U_j}{\partial t}+\frac{\partial F_i}{\partial x}=C_i\frac{\partial D_i}{\partial x}+S_i, \tag{1}$$

or the hyperbolic convection-only system:

$$\frac{\partial U_i}{\partial t}+\frac{\partial F_i}{\partial x}=0, \tag{2}$$

for $i=1,2,\ldots,\mathbf{npde}$, $a\leq x\leq b$, $t\geq t_0$, where the vector U is the set of PDE solution values

$$U(x,t)=\left[U_1(x,t),\ldots,U_{\mathbf{npde}}(x,t)\right]^{\mathrm{T}}.$$

The optional coupled ODEs are of the general form

$$R_i\left(t,V,\dot{V},\xi,U^*,U_x^*,U_t^*\right)=0,\quad i=1,2,\ldots,\mathbf{ncode}, \tag{3}$$

where the vector V is the set of ODE solution values

$$V(t)=\left[V_1(t),\ldots,V_{\mathbf{ncode}}(t)\right]^{\mathrm{T}},$$

$\dot{V}$ denotes its derivative with respect to time, and U_x is the spatial derivative of U.

In (2), $P_{i,j}$, F_i and C_i depend on x, t, U and V; D_i depends on x, t, U, U_x and V; and S_i depends on x, t, U, V and **linearly** on $\dot{V}$. Note that $P_{i,j}$, F_i, C_i and S_i must not depend on any space derivatives, and $P_{i,j}$, F_i, C_i and D_i must not depend on any time derivatives. In terms of conservation laws, F_i, $\frac{C_i \partial D_i}{\partial x}$ and S_i are the convective flux, diffusion and source terms respectively.

In (3), ξ represents a vector of n_ξ spatial coupling points at which the ODEs are coupled to the PDEs. These points may or may not be equal to PDE spatial mesh points. U^*, U_x^* and U_t^* are the functions U, U_x and U_t evaluated at these coupling points. Each R_i may depend only linearly on time derivatives. Hence (3) may be written more precisely as

$$R = L - M\dot{V} - NU_t^*, \tag{4}$$

where $R = [R_1, \ldots, R_{\mathbf{ncode}}]^{\mathrm{T}}$, L is a vector of length **ncode**, M is an **ncode** by **ncode** matrix, N is an **ncode** by $\left(n_\xi \times \mathbf{npde}\right)$ matrix and the entries in L, M and N may depend on t, ξ, U^*, U_x^* and V. In practice you only need to supply a vector of information to define the ODEs and not the matrices L, M and N. (See Section 5 for the specification of **odedef**.)

The integration in time is from t_0 to t_{out}, over the space interval $a \leq x \leq b$, where $a = x_1$ and $b = x_{\mathbf{npts}}$ are the leftmost and rightmost points of a user-defined mesh $x_1, x_2, \ldots, x_{\mathbf{npts}}$ defined initially by you and (possibly) adapted automatically during the integration according to user-specified criteria.

The initial $(t = t_0)$ values of the functions $U(x,t)$ and $V(t)$ must be specified in **uvinit**. Note that **uvinit** will be called again following any initial remeshing, and so $U(x, t_0)$ should be specified for **all** values of x in the interval $a \leq x \leq b$, and not just the initial mesh points.

The PDEs are approximated by a system of ODEs in time for the values of U_i at mesh points using a spatial discretization method similar to the central-difference scheme used in nag_pde_1d_parab_fd (d03pc), nag_pde_1d_parab_dae_fd (d03ph) and nag_pde_1d_parab_remesh_fd (d03pp), but with the flux F_i replaced by a *numerical flux*, which is a representation of the flux taking into account the direction of the flow of information at that point (i.e., the direction of the characteristics). Simple central differencing of the numerical flux then becomes a sophisticated upwind scheme in which the correct direction of upwinding is automatically achieved.

The numerical flux, $\hat{F}_i$ say, must be calculated by you in terms of the *left* and *right* values of the solution vector U (denoted by U_L and U_R respectively), at each mid-point of the mesh $x_{j-\frac{1}{2}} = \left(x_{j-1} + x_j\right)/2$, for $j = 2, 3, \ldots, \mathbf{npts}$. The left and right values are calculated by nag_pde_1d_parab_convdiff_remesh (d03ps) from two adjacent mesh points using a standard upwind technique combined with a Van Leer slope-limiter (see LeVeque (1990)). The physically correct value for $\hat{F}_i$ is derived from the solution of the Riemann problem given by

$$\frac{\partial U_i}{\partial t} + \frac{\partial F_i}{\partial y} = 0, \tag{5}$$

where $y = x - x_{j-\frac{1}{2}}$, i.e., $y = 0$ corresponds to $x = x_{j-\frac{1}{2}}$, with discontinuous initial values $U = U_L$ for $y < 0$ and $U = U_R$ for $y > 0$, using an *approximate Riemann solver*. This applies for either of the systems (1) or (2); the numerical flux is independent of the functions $P_{i,j}$, C_i, D_i and S_i. A description of several approximate Riemann solvers can be found in LeVeque (1990) and Berzins *et al.* (1989). Roe's scheme (see Roe (1981)) is perhaps the easiest to understand and use, and a brief summary follows. Consider the system of PDEs $U_t + F_x = 0$ or equivalently $U_t + AU_x = 0$. Provided the system is linear in U, i.e., the Jacobian matrix A does not depend on U, the numerical flux $\hat{F}$ is given by

$$\hat{F} = \tfrac{1}{2}(F_L + F_R) - \tfrac{1}{2}\sum_{k=1}^{\mathbf{npde}} \alpha_k |\lambda_k| e_k, \tag{6}$$

where F_L (F_R) is the flux F calculated at the left (right) value of U, denoted by U_L (U_R); the λ_k are the eigenvalues of A; the e_k are the right eigenvectors of A; and the α_k are defined by

$$U_R - U_L = \sum_{k=1}^{\mathbf{npde}} \alpha_k e_k. \tag{7}$$

Examples are given in the documents for nag_pde_1d_parab_convdiff (d03pf) and nag_pde_1d_parab_ convdiff_dae (d03pl).

If the system is nonlinear, Roe's scheme requires that a linearized Jacobian is found (see Roe (1981)).

The functions $P_{i,j}$, C_i, D_i and S_i (but **not** F_i) must be specified in **pdedef**. The numerical flux $\hat{F}_i$ must be supplied in **numflx**. For problems in the form (2), the actual argument nag_pde_1d_parab_conv-diff_dae_sample_pdedef (d03plp) may be used for **pdedef**. nag_pde_1d_parab_convdiff_dae_sample_p-dedef (d03plp) is included in the NAG Toolbox and sets the matrix with entries $P_{i,j}$ to the identity matrix, and the functions C_i, D_i and S_i to zero.

For second-order problems, i.e., diffusion terms are present, a boundary condition is required for each PDE at both boundaries for the problem to be well-posed. If there are no diffusion terms present, then the continuous PDE problem generally requires exactly one boundary condition for each PDE, that is **npde** boundary conditions in total. However, in common with most discretization schemes for first-order problems, a *numerical boundary condition* is required at the other boundary for each PDE. In order to be consistent with the characteristic directions of the PDE system, the numerical boundary conditions must be derived from the solution inside the domain in some manner (see below). You must supply both types of boundary conditions, i.e., a total of **npde** conditions at each boundary point.

The position of each boundary condition should be chosen with care. In simple terms, if information is flowing into the domain then a physical boundary condition is required at that boundary, and a numerical boundary condition is required at the other boundary. In many cases the boundary conditions are simple, e.g., for the linear advection equation. In general you should calculate the characteristics of the PDE system and specify a physical boundary condition for each of the characteristic variables associated with incoming characteristics, and a numerical boundary condition for each outgoing characteristic.

A common way of providing numerical boundary conditions is to extrapolate the characteristic variables from the inside of the domain (note that when using banded matrix algebra the fixed bandwidth means that only linear extrapolation is allowed, i.e., using information at just two interior points adjacent to the boundary). For problems in which the solution is known to be uniform (in space) towards a boundary during the period of integration then extrapolation is unnecessary; the numerical boundary condition can be supplied as the known solution at the boundary. Another method of supplying numerical boundary conditions involves the solution of the characteristic equations associated with the outgoing characteristics. Examples of both methods can be found in the documents for nag_pde_1d_parab_convdiff (d03pf) and nag_pde_1d_parab_convdiff_dae (d03pl).

The boundary conditions must be specified in **bndary** in the form

$$G_i^L\big(x,t,U,V,\dot{V}\big)=0 \quad \text{at } x=a, \quad i=1,2,\ldots,\mathbf{npde}, \tag{8}$$

at the left-hand boundary, and

$$G_i^R\big(x,t,U,V,\dot{V}\big)=0 \quad \text{at } x=b, \quad i=1,2,\ldots,\mathbf{npde}, \tag{9}$$

at the right-hand boundary.

Note that spatial derivatives at the boundary are not passed explicitly to **bndary**, but they can be calculated using values of U at and adjacent to the boundaries if required. However, it should be noted that instabilities may occur if such one-sided differencing opposes the characteristic direction at the boundary.

The algebraic-differential equation system which is defined by the functions R_i must be specified in **odedef**. You must also specify the coupling points ξ (if any) in the array **xi**.

In total there are $\mathbf{npde}\times\mathbf{npts}+\mathbf{ncode}$ ODEs in the time direction. This system is then integrated forwards in time using a BDF or Theta method, optionally switching between Newton's method and functional iteration (see Berzins *et al.* (1989) and the references therein).

The adaptive space remeshing can be used to generate meshes that automatically follow the changing time-dependent nature of the solution, generally resulting in a more efficient and accurate solution using fewer mesh points than may be necessary with a fixed uniform or non-uniform mesh. Problems with travelling wavefronts or variable-width boundary layers for example will benefit from using a moving adaptive mesh. The discrete time-step method used here (developed by Furzeland (1984)) automatically creates a new mesh based on the current solution profile at certain time-steps, and the solution is then interpolated onto the new mesh and the integration continues.

The method requires you to supply a **monitf** which specifies in an analytical or numerical form the particular aspect of the solution behaviour you wish to track. This so-called monitor function is used by the function to choose a mesh which equally distributes the integral of the monitor function over the domain. A typical choice of monitor function is the second space derivative of the solution value at each point (or some combination of the second space derivatives if there is more than one solution component), which results in refinement in regions where the solution gradient is changing most rapidly.

You must specify the frequency of mesh updates together with certain other criteria such as adjacent mesh ratios. Remeshing can be expensive and you are encouraged to experiment with the different options in order to achieve an efficient solution which adequately tracks the desired features of the solution.

Note that unless the monitor function for the initial solution values is zero at all user-specified initial mesh points, a new initial mesh is calculated and adopted according to the user-specified remeshing criteria. **uvinit** will then be called again to determine the initial solution values at the new mesh points (there is no interpolation at this stage) and the integration proceeds.

The problem is subject to the following restrictions:

(i) In (1), $\dot{V}_j(t)$, for $j = 1, 2, \ldots,$ **ncode**, may only appear **linearly** in the functions S_i, for $i = 1, 2, \ldots,$ **npde**, with a similar restriction for G_i^L and G_i^R;

(ii) $P_{i,j}$, F_i, C_i and S_i must not depend on any space derivatives; and $P_{i,j}$, C_i, D_i and F_i must not depend on any time derivatives;

(iii) $t_0 < t_{\text{out}}$, so that integration is in the forward direction;

(iv) The evaluation of the terms $P_{i,j}$, C_i, D_i and S_i is done by calling the **pdedef** at a point approximately midway between each pair of mesh points in turn. Any discontinuities in these functions **must** therefore be at one or more of the **fixed** mesh points specified by **xfix**;

(v) At least one of the functions $P_{i,j}$ must be nonzero so that there is a time derivative present in the PDE problem.

For further details of the scheme, see Pennington and Berzins (1994) and the references therein.

4 References

Berzins M, Dew P M and Furzeland R M (1989) Developing software for time-dependent problems using the method of lines and differential-algebraic integrators *Appl. Numer. Math.* **5** 375–397

Furzeland R M (1984) The construction of adaptive space meshes *TNER.85.022* Thornton Research Centre, Chester

Hirsch C (1990) *Numerical Computation of Internal and External Flows, Volume 2: Computational Methods for Inviscid and Viscous Flows* John Wiley

LeVeque R J (1990) *Numerical Methods for Conservation Laws* BirkhÌuser Verlag

Pennington S V and Berzins M (1994) New NAG Library software for first-order partial differential equations *ACM Trans. Math. Softw.* **20** 63–99

Roe P L (1981) Approximate Riemann solvers, parameter vectors, and difference schemes *J. Comput. Phys.* **43** 357–372

5 Parameters

5.1 Compulsory Input Parameters

1: **npde** – INTEGER

The number of PDEs to be solved.

Constraint: **npde** ≥ 1.

2: **ts** – REAL (KIND=nag_wp)

The initial value of the independent variable t.

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

3: **tout** – REAL (KIND=nag_wp)

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

4: **pdedef** – SUBROUTINE, supplied by the NAG Library or the user.

pdedef must evaluate the functions $P_{i,j}$, C_i, D_i and S_i which partially define the system of PDEs. $P_{i,j}$ and C_i may depend on x, t, U and V; D_i may depend on x, t, U, U_x and V; and S_i may depend on x, t, U, V and linearly on $\dot{V}$. **pdedef** is called approximately midway between each pair of mesh points in turn by nag_pde_1d_parab_convdiff_remesh (d03ps). The actual argument nag_pde_1d_parab_convdiff_dae_sample_pdedef (d03plp) may be used for **pdedef** for problems in the form (2). (nag_pde_1d_parab_convdiff_dae_sample_pdedef (d03plp) is included in the NAG Toolbox.)

```
[p, c, d, s, ires] = pdedef(npde, t, x, u, ux, ncode, v, vdot, ires)
```

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.

3: **x** – REAL (KIND=nag_wp)

The current value of the space variable x.

4: **u**(**npde**) – REAL (KIND=nag_wp) array

u(i) contains the value of the component $U_i(x,t)$, for $i = 1, 2, \ldots,$ **npde**.

5: **ux**(**npde**) – REAL (KIND=nag_wp) array

ux(i) contains the value of the component $\dfrac{\partial U_i(x,t)}{\partial x}$, for $i = 1, 2, \ldots,$ **npde**.

6: **ncode** – INTEGER

The number of coupled ODEs in the system.

7: **v**(**ncode**) – REAL (KIND=nag_wp) array

If **ncode** > 0, **v**(i) contains the value of the component $V_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

8: **vdot**(**ncode**) – REAL (KIND=nag_wp) array

If **ncode** > 0, **vdot**(i) contains the value of component $\dot{V}_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

Note: $\dot{V}_i(t)$, for $i = 1, 2, \ldots,$ **ncode**, may only appear linearly in S_j, for $j = 1, 2, \ldots,$ **npde**.

9: **ires** – INTEGER

Set to -1 or 1.

Output Parameters

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

p(i, j) must be set to the value of $P_{i,j}(x, t, U, V)$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **npde**.

2: **c**(**npde**) – REAL (KIND=nag_wp) array

c(i) must be set to the value of $C_i(x, t, U, V)$, for $i = 1, 2, \ldots,$ **npde**.

3: **d**(**npde**) – REAL (KIND=nag_wp) array

d(i) must be set to the value of $D_i(x, t, U, U_x, V)$, for $i = 1, 2, \ldots,$ **npde**.

4: **s**(**npde**) – REAL (KIND=nag_wp) array

s(i) must be set to the value of $S_i\left(x, t, U, V, \dot{V}\right)$, for $i = 1, 2, \ldots,$ **npde**.

5: **ires** – INTEGER

Should usually remain unchanged. However, you may set **ires** to force the integration function to take certain actions as described below:

ires $= 2$
Indicates to the integrator that control should be passed back immediately to the calling (sub)routine with the error indicator set to **ifail** $= 6$.

ires $= 3$
Indicates to the integrator that the current time step should be abandoned and a smaller time step used instead. You may wish to set **ires** $= 3$ when a physically meaningless input or output value has been generated. If you consecutively set **ires** $= 3$, then nag_pde_1d_parab_convdiff_remesh (d03ps) returns to the calling function with the error indicator set to **ifail** $= 4$.

5: **numflx** – SUBROUTINE, supplied by the user.

numflx must supply the numerical flux for each PDE given the *left* and *right* values of the solution vector **u**. **numflx** is called approximately midway between each pair of mesh points in turn by nag_pde_1d_parab_convdiff_remesh (d03ps).

```
[flux, ires] = numflx(npde, t, x, ncode, v, uleft, uright, ires)
```

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.

3: **x** – REAL (KIND=nag_wp)

The current value of the space variable x.

4: **ncode** – INTEGER

The number of coupled ODEs in the system.

5: **v**(**ncode**) – REAL (KIND=nag_wp) array

If **ncode** > 0, **v**(i) contains the value of the component $V_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

6: **uleft**(**npde**) – REAL (KIND=nag_wp) array

uleft(i) contains the *left* value of the component $U_i(x)$, for $i = 1, 2, \ldots,$ **npde**.

7: **uright**(**npde**) – REAL (KIND=nag_wp) array

uright(i) contains the *right* value of the component $U_i(x)$, for $i = 1, 2, \ldots,$ **npde**.

8: **ires** – INTEGER

Set to -1 or 1.

Output Parameters

1: **flux**(**npde**) – REAL (KIND=nag_wp) array

flux(i) must be set to the numerical flux $\hat{F}_i$, for $i = 1, 2, \ldots,$ **npde**.

2: **ires** – INTEGER

Should usually remain unchanged. However, you may set **ires** to force the integration function to take certain actions as described below:

ires $= 2$
Indicates to the integrator that control should be passed back immediately to the calling (sub)routine with the error indicator set to **ifail** $= 6$.

ires $= 3$
Indicates to the integrator that the current time step should be abandoned and a smaller time step used instead. You may wish to set **ires** $= 3$ when a physically meaningless input or output value has been generated. If you consecutively set **ires** $= 3$, then nag_pde_1d_parab_convdiff_remesh (d03ps) returns to the calling function with the error indicator set to **ifail** $= 4$.

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

bndary must evaluate the functions G_i^L and G_i^R which describe the physical and numerical boundary conditions, as given by (8) and (9).

```
[g, ires] = bndary(npde, npts, t, x, u, ncode, v, vdot, ibnd, ires)
```

Input Parameters

1: **npde** – INTEGER

The number of PDEs in the system.

2: **npts** – INTEGER

The number of mesh points in the interval $[a, b]$.

3: **t** – REAL (KIND=nag_wp)

The current value of the independent variable t.

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

The mesh points in the spatial direction. **x**(1) corresponds to the left-hand boundary, a, and **x**(**npts**) corresponds to the right-hand boundary, b.

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

u(i,j) contains the value of the component $U_i(x,t)$ at $x =$ **x**(j), for $i = 1,2,\ldots,$ **npde** and $j = 1,2,\ldots,$ **npts**.

Note: if banded matrix algebra is to be used then the functions G_i^L and G_i^R may depend on the value of $U_i(x,t)$ at the boundary point and the two adjacent points only.

6: **ncode** – INTEGER

The number of coupled ODEs in the system.

7: **v**(**ncode**) – REAL (KIND=nag_wp) array

If **ncode** > 0, **v**(i) contains the value of the component $V_i(t)$, for $i = 1,2,\ldots,$ **ncode**.

8: **vdot**(**ncode**) – REAL (KIND=nag_wp) array

If **ncode** > 0, **vdot**(i) contains the value of component $\dot{V}_i(t)$, for $i = 1,2,\ldots,$ **ncode**.

Note: $\dot{V}_i(t)$, for $i = 1,2,\ldots,$ **ncode**, may only appear linearly in G_j^L and G_j^R, for $j = 1,2,\ldots,$ **npde**.

9: **ibnd** – INTEGER

Specifies which boundary conditions are to be evaluated.

ibnd $= 0$
: **bndary** must evaluate the left-hand boundary condition at $x = a$.

ibnd $\neq 0$
: **bndary** must evaluate the right-hand boundary condition at $x = b$.

10: **ires** – INTEGER

Set to -1 or 1.

Output Parameters

1: **g**(**npde**) – REAL (KIND=nag_wp) array

g(i) must contain the ith component of either G_i^L or G_i^R in (8) and (9), depending on the value of **ibnd**, for $i = 1,2,\ldots,$ **npde**.

2: **ires** – INTEGER

Should usually remain unchanged. However, you may set **ires** to force the integration function to take certain actions as described below:

ires $= 2$
: Indicates to the integrator that control should be passed back immediately to the calling (sub)routine with the error indicator set to **ifail** $= 6$.

ires $= 3$
: Indicates to the integrator that the current time step should be abandoned and a smaller time step used instead. You may wish to set **ires** $= 3$ when a physically

meaningless input or output value has been generated. If you consecutively set **ires** $= 3$, then nag_pde_1d_parab_convdiff_remesh (d03ps) returns to the calling function with the error indicator set to **ifail** $= 4$.

7: **uvinit** – SUBROUTINE, supplied by the user.

uvinit must supply the initial $(t = t_0)$ values of $U(x,t)$ and $V(t)$ for all values of x in the interval $a \leq x \leq b$.

```
[u, v] = uvinit(npde, npts, nxi, x, xi, ncode)
```

Input Parameters

1: **npde** – INTEGER

The number of PDEs in the system.

2: **npts** – INTEGER

The number of mesh points in the interval $[a,b]$.

3: **nxi** – INTEGER

The number of ODE/PDE coupling points.

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

The current mesh. **x**(i) contains the value of x_i, for $i = 1, 2, \ldots,$ **npts**.

5: **xi**(**nxi**) – REAL (KIND=nag_wp) array

If **nxi** > 0, **xi**(i) contains the ODE/PDE coupling point, ξ_i, for $i = 1, 2, \ldots,$ **nxi**.

6: **ncode** – INTEGER

The number of coupled ODEs in the system.

Output Parameters

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

If **nxi** > 0, **u**(i,j) contains the value of the component $U_i(x_j, t_0)$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **npts**.

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

If **ncode** > 0, **v**(i) must contain the value of component $V_i(t_0)$, for $i = 1, 2, \ldots,$ **ncode**.

8: **u**(**neqn**) – REAL (KIND=nag_wp) array

If **ind** $= 1$ the value of **u** must be unchanged from the previous call.

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

The mesh points in the space direction. **x**(1) must specify the left-hand boundary, a, and **x**(**npts**) must specify the right-hand boundary, b.

Constraint: **x**$(1) <$ **x**$(2) < \cdots <$ **x**(**npts**).

10: **ncode** – INTEGER

The number of coupled ODE components.

Constraint: **ncode** ≥ 0.

11: **odedef** – SUBROUTINE, supplied by the NAG Library or the user.

odedef must evaluate the functions R, which define the system of ODEs, as given in (4).

If you wish to compute the solution of a system of PDEs only (i.e., **ncode** $= 0$), **odedef** must be the string `nag_pde_1d_parab_dae_keller_remesh_fd_dummy_odedef (d03pek)`. (nag_pde_1d_parab_dae_keller_remesh_fd_dummy_odedef (d03pek) is included in the NAG Toolbox.)

```
[r, ires] = odedef(npde, t, ncode, v, vdot, nxi, xi, ucp, ucpx, ucpt, ires)
```

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.

3: **ncode** – INTEGER

The number of coupled ODEs in the system.

4: **v**(**ncode**) – REAL (KIND=nag_wp) array

If **ncode** > 0, **v**(i) contains the value of the component $V_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

5: **vdot**(**ncode**) – REAL (KIND=nag_wp) array

If **ncode** > 0, **vdot**(i) contains the value of component $\dot{V}_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

6: **nxi** – INTEGER

The number of ODE/PDE coupling points.

7: **xi**(**nxi**) – REAL (KIND=nag_wp) array

If **nxi** > 0, **xi**(i) contains the ODE/PDE coupling point, ξ_i, for $i = 1, 2, \ldots,$ **nxi**.

8: **ucp**(**npde**, :) – REAL (KIND=nag_wp) array

The second dimension of the array **ucp** must be at least $\max(1,$ **nxi**$)$.

If **nxi** > 0, **ucp**(i, j) contains the value of $U_i(x, t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **nxi**.

9: **ucpx**(**npde**, :) – REAL (KIND=nag_wp) array

The second dimension of the array **ucpx** must be at least $\max(1,$ **nxi**$)$.

If **nxi** > 0, **ucpx**(i, j) contains the value of $\frac{\partial U_i(x,t)}{\partial x}$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **nxi**.

10: **ucpt**(**npde**, :) – REAL (KIND=nag_wp) array

The second dimension of the array **ucpt** must be at least $\max(1,$ **nxi**$)$.

If **nxi** > 0, **ucpt**(i, j) contains the value of $\frac{\partial U_i}{\partial t}$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **nxi**.

11: **ires** – INTEGER

The form of R that must be returned in the array **r**.

ires $= 1$
: Equation (10) must be used.

ires $= -1$
: Equation (11) must be used.

Output Parameters

1: **r**(**ncode**) – REAL (KIND=nag_wp) array

r(i) must contain the ith component of R, for $i = 1, 2, \ldots,$ **ncode**, where R is defined as

$$R = L - M\dot{V} - NU_t^*, \tag{10}$$

or

$$R = -M\dot{V} - NU_t^*. \tag{11}$$

The definition of R is determined by the input value of **ires**.

2: **ires** – INTEGER

Should usually remain unchanged. However, you may reset **ires** to force the integration function to take certain actions, as described below:

ires $= 2$
: Indicates to the integrator that control should be passed back immediately to the calling (sub)routine with the error indicator set to **ifail** $= 6$.

ires $= 3$
: Indicates to the integrator that the current time step should be abandoned and a smaller time step used instead. You may wish to set **ires** $= 3$ when a physically meaningless input or output value has been generated. If you consecutively set **ires** $= 3$, then nag_pde_1d_parab_convdiff_remesh (d03ps) returns to the calling function with the error indicator set to **ifail** $= 4$.

12: **xi**(**nxi**) – REAL (KIND=nag_wp) array

If **nxi** > 0, **xi**(i), for $i = 1, 2, \ldots,$ **nxi**, must be set to the ODE/PDE coupling points.

Constraint: **x**$(1) \leq$ **xi**$(1) <$ **xi**$(2) < \cdots <$ **xi**(**nxi**) $\leq$ **x**(**npts**).

13: **rtol**(:) – REAL (KIND=nag_wp) array

The dimension of the array **rtol** must be at least 1 if **itol** $= 1$ or 2 and at least **neqn** if **itol** $= 3$ or 4

The relative local error tolerance.

Constraint: **rtol**$(i) \geq 0.0$ for all relevant i.

14: **atol**(:) – REAL (KIND=nag_wp) array

The dimension of the array **atol** must be at least 1 if **itol** $= 1$ or 3 and at least **neqn** if **itol** $= 2$ or 4

The absolute local error tolerance.

Constraint: **atol**$(i) \geq 0.0$ for all relevant i.

Note: corresponding elements of **rtol** and **atol** cannot both be 0.0.

15: **itol** – INTEGER

A value to indicate the form of the local error test. If e_i is the estimated local error for $\mathbf{u}(i)$, for $i = 1, 2, \ldots, \mathbf{neqn}$, and $\| \quad \|$, denotes the norm, then the error test to be satisfied is $\|e_i\| < 1.0$. **itol** indicates to nag_pde_1d_parab_convdiff_remesh (d03ps) whether to interpret either or both of **rtol** and **atol** as a vector or scalar in the formation of the weights w_i used in the calculation of the norm (see the description of **norm_p**):

itol	**rtol**	**atol**	w_i
1	scalar	scalar	$\mathbf{rtol}(1) \times \lvert\mathbf{u}(i)\rvert + \mathbf{atol}(1)$
2	scalar	vector	$\mathbf{rtol}(1) \times \lvert\mathbf{u}(i)\rvert + \mathbf{atol}(i)$
3	vector	scalar	$\mathbf{rtol}(i) \times \lvert\mathbf{u}(i)\rvert + \mathbf{atol}(1)$
4	vector	vector	$\mathbf{rtol}(i) \times \lvert\mathbf{u}(i)\rvert + \mathbf{atol}(i)$

Constraint: **itol** $= 1$, 2, 3 or 4.

16: **norm_p** – CHARACTER(1)

The type of norm to be used.

norm_p = '1'
Averaged L_1 norm.

norm_p = '2'
Averaged L_2 norm.

If U_{norm} denotes the norm of the vector **u** of length **neqn**, then for the averaged L_1 norm

$$U_{\text{norm}} = \frac{1}{\mathbf{neqn}} \sum_{i=1}^{\mathbf{neqn}} \mathbf{u}(i)/w_i,$$

and for the averaged L_2 norm

$$U_{\text{norm}} = \sqrt{\frac{1}{\mathbf{neqn}} \sum_{i=1}^{\mathbf{neqn}} (\mathbf{u}(i)/w_i)^2},$$

See the description of **itol** for the formulation of the weight vector w.

Constraint: **norm_p** = '1' or '2'.

17: **laopt** – CHARACTER(1)

The type of matrix algebra required.

laopt = 'F'
Full matrix methods to be used.

laopt = 'B'
Banded matrix methods to be used.

laopt = 'S'
Sparse matrix methods to be used.

Constraint: **laopt** = 'F', 'B' or 'S'.

Note: you are recommended to use the banded option when no coupled ODEs are present (**ncode** $= 0$). Also, the banded option should not be used if the boundary conditions involve solution components at points other than the boundary and the immediately adjacent two points.

18: **algopt**(**30**) – REAL (KIND=nag_wp) array

May be set to control various options available in the integrator. If you wish to employ all the default options, then **algopt**(1) should be set to 0.0. Default values will also be used for any other elements of **algopt** set to zero. The permissible values, default values, and meanings are as follows:

algopt(1)
: Selects the ODE integration method to be used. If **algopt**$(1) = 1.0$, a BDF method is used and if **algopt**$(1) = 2.0$, a Theta method is used. The default is **algopt**$(1) = 1.0$.

If **algopt**$(1) = 2.0$, then **algopt**(i), for $i = 2, 3, 4$, are not used.

algopt(2)
: Specifies the maximum order of the BDF integration formula to be used. **algopt**(2) may be 1.0, 2.0, 3.0, 4.0 or 5.0. The default value is **algopt**$(2) = 5.0$.

algopt(3)
: Specifies what method is to be used to solve the system of nonlinear equations arising on each step of the BDF method. If **algopt**$(3) = 1.0$ a modified Newton iteration is used and if **algopt**$(3) = 2.0$ a functional iteration method is used. If functional iteration is selected and the integrator encounters difficulty, then there is an automatic switch to the modified Newton iteration. The default value is **algopt**$(3) = 1.0$.

algopt(4)
: Specifies whether or not the Petzold error test is to be employed. The Petzold error test results in extra overhead but is more suitable when algebraic equations are present, such as $P_{i,j} = 0.0$, for $j = 1, 2, \ldots,$ **npde**, for some i or when there is no $\dot{V}_i(t)$ dependence in the coupled ODE system. If **algopt**$(4) = 1.0$, then the Petzold test is used. If **algopt**$(4) = 2.0$, then the Petzold test is not used. The default value is **algopt**$(4) = 1.0$.

If **algopt**$(1) = 1.0$, then **algopt**(i), for $i = 5, 6, 7$, are not used.

algopt(5)
: Specifies the value of Theta to be used in the Theta integration method. $0.51 \leq$ **algopt**$(5) \leq 0.99$. The default value is **algopt**$(5) = 0.55$.

algopt(6)
: Specifies what method is to be used to solve the system of nonlinear equations arising on each step of the Theta method. If **algopt**$(6) = 1.0$, a modified Newton iteration is used and if **algopt**$(6) = 2.0$, a functional iteration method is used. The default value is **algopt**$(6) = 1.0$.

algopt(7)
: Specifies whether or not the integrator is allowed to switch automatically between modified Newton and functional iteration methods in order to be more efficient. If **algopt**$(7) = 1.0$, then switching is allowed and if **algopt**$(7) = 2.0$, then switching is not allowed. The default value is **algopt**$(7) = 1.0$.

algopt(11)
: Specifies a point in the time direction, t_{crit}, beyond which integration must not be attempted. The use of t_{crit} is described under the argument **itask**. If **algopt**$(1) \neq 0.0$, a value of 0.0 for **algopt**(11), say, should be specified even if **itask** subsequently specifies that t_{crit} will not be used.

algopt(12)
: Specifies the minimum absolute step size to be allowed in the time integration. If this option is not required, **algopt**(12) should be set to 0.0.

algopt(13)
: Specifies the maximum absolute step size to be allowed in the time integration. If this option is not required, **algopt**(13) should be set to 0.0.

algopt(14)
: Specifies the initial step size to be attempted by the integrator. If **algopt**$(14) = 0.0$, then the initial step size is calculated internally.

algopt(15)
: Specifies the maximum number of steps to be attempted by the integrator in any one call. If **algopt**(15) = 0.0, then no limit is imposed.

algopt(23)
: Specifies what method is to be used to solve the nonlinear equations at the initial point to initialize the values of U, U_t, V and $\dot{V}$. If **algopt**(23) = 1.0, a modified Newton iteration is used and if **algopt**(23) = 2.0, functional iteration is used. The default value is **algopt**(23) = 1.0.

algopt(29) and **algopt**(30) are used only for the sparse matrix algebra option, i.e., **laopt** = 'S'.

algopt(29)
: Governs the choice of pivots during the decomposition of the first Jacobian matrix. It should lie in the range 0.0 < **algopt**(29) < 1.0, with smaller values biasing the algorithm towards maintaining sparsity at the expense of numerical stability. If **algopt**(29) lies outside the range then the default value is used. If the functions regard the Jacobian matrix as numerically singular, then increasing **algopt**(29) towards 1.0 may help, but at the cost of increased fill-in. The default value is **algopt**(29) = 0.1.

algopt(30)
: Used as the relative pivot threshold during subsequent Jacobian decompositions (see **algopt**(29)) below which an internal error is invoked. **algopt**(30) must be greater than zero, otherwise the default value is used. If **algopt**(30) is greater than 1.0 no check is made on the pivot size, and this may be a necessary option if the Jacobian matrix is found to be numerically singular (see **algopt**(29)). The default value is **algopt**(30) = 0.0001.

19: **remesh** – LOGICAL

Indicates whether or not spatial remeshing should be performed.

remesh = *true*
: Indicates that spatial remeshing should be performed as specified.

remesh = *false*
: Indicates that spatial remeshing should be suppressed.

Note: **remesh** should **not** be changed between consecutive calls to nag_pde_1d_parab_convdiff_remesh (d03ps). Remeshing can be switched off or on at specified times by using appropriate values for the arguments **nrmesh** and **trmesh** at each call.

20: **xfix**(:) – REAL (KIND=nag_wp) array

The dimension of the array **xfix** must be at least max(1, **nxfix**)

xfix(i), for $i = 1, 2, \ldots,$ **nxfix**, must contain the value of the x coordinate at the ith fixed mesh point.

Constraints:

xfix(i) < **xfix**($i + 1$), for $i = 1, 2, \ldots,$ **nxfix** $- 1$;
each fixed mesh point must coincide with a user-supplied initial mesh point, that is **xfix**(i) = **x**(j) for some j, $2 \leq j \leq$ **npts** $- 1$..

Note: the positions of the fixed mesh points in the array **x**(**npts**) remain fixed during remeshing, and so the number of mesh points between adjacent fixed points (or between fixed points and end points) does not change. You should take this into account when choosing the initial mesh distribution.

21: **nrmesh** – INTEGER

Specifies the spatial remeshing frequency and criteria for the calculation and adoption of a new mesh.

nrmesh < 0
: Indicates that a new mesh is adopted according to the argument **dxmesh**. The mesh is tested every $|$**nrmesh**$|$ timesteps.

nrmesh $= 0$
: Indicates that remeshing should take place just once at the end of the first time step reached when $t >$ **trmesh**.

nrmesh > 0
: Indicates that remeshing will take place every **nrmesh** time steps, with no testing using **dxmesh**.

Note: **nrmesh** may be changed between consecutive calls to nag_pde_1d_parab_convdiff_remesh (d03ps) to give greater flexibility over the times of remeshing.

22: **dxmesh** – REAL (KIND=nag_wp)

Determines whether a new mesh is adopted when **nrmesh** is set less than zero. A possible new mesh is calculated at the end of every $|$**nrmesh**$|$ time steps, but is adopted only if

$$x_i^{\text{new}} > x_i^{\text{old}} + \mathbf{dxmesh} \times \left(x_{i+1}^{\text{old}} - x_i^{\text{old}}\right)$$

or

$$x_i^{\text{new}} < x_i^{\text{old}} - \mathbf{dxmesh} \times \left(x_i^{\text{old}} - x_{i-1}^{\text{old}}\right)$$

dxmesh thus imposes a lower limit on the difference between one mesh and the next.

Constraint: **dxmesh** ≥ 0.0.

23: **trmesh** – REAL (KIND=nag_wp)

Specifies when remeshing will take place when **nrmesh** is set to zero. Remeshing will occur just once at the end of the first time step reached when t is greater than **trmesh**.

Note: **trmesh** may be changed between consecutive calls to nag_pde_1d_parab_convdiff_remesh (d03ps) to force remeshing at several specified times.

24: **ipminf** – INTEGER

The level of trace information regarding the adaptive remeshing. Details are directed to the current advisory message unit (see nag_file_set_unit_advisory (x04ab)).

ipminf $= 0$
: No trace information.

ipminf $= 1$
: Brief summary of mesh characteristics.

ipminf $= 2$
: More detailed information, including old and new mesh points, mesh sizes and monitor function values.

Constraint: **ipminf** $= 0$, 1 or 2.

25: **monitf** – SUBROUTINE, supplied by the NAG Library or the user.

monitf must supply and evaluate a remesh monitor function to indicate the solution behaviour of interest.

If you specify **remesh** $=$ *false*, i.e., no remeshing, then **monitf** will not be called and the string `nag_pde_1d_parab_dae_keller_remesh_fd_dummy_monitf (d03pel)` may be used for

monitf. (nag_pde_1d_parab_dae_keller_remesh_fd_dummy_monitf (d03pel) is included in the NAG Toolbox.)

```
[fmon] = monitf(t, npts, npde, x, u)
```

Input Parameters

1: **t** – REAL (KIND=nag_wp)

The current value of the independent variable t.

2: **npts** – INTEGER

The number of mesh points in the interval $[a, b]$.

3: **npde** – INTEGER

The number of PDEs in the system.

4: $\mathbf{x}(\mathbf{npts})$ – REAL (KIND=nag_wp) array

The current mesh. $\mathbf{x}(i)$ contains the value of x_i, for $i = 1, 2, \ldots, \mathbf{npts}$.

5: $\mathbf{u}(\mathbf{npde}, \mathbf{npts})$ – REAL (KIND=nag_wp) array

$\mathbf{u}(i, j)$ contains the value of $U_i(x, t)$ at $x = \mathbf{x}(j)$ and time t, for $i = 1, 2, \ldots, \mathbf{npde}$ and $j = 1, 2, \ldots, \mathbf{npts}$.

Output Parameters

1: $\mathbf{fmon}(\mathbf{npts})$ – REAL (KIND=nag_wp) array

$\mathbf{fmon}(i)$ must contain the value of the monitor function $F^{\mathrm{mon}}(x)$ at mesh point $x = \mathbf{x}(i)$.

26: $\mathbf{rsave}(lrsave)$ – REAL (KIND=nag_wp) array

If $\mathbf{ind} = 0$, **rsave** need not be set on entry.

If $\mathbf{ind} = 1$, **rsave** must be unchanged from the previous call to the function because it contains required information about the iteration.

27: $\mathbf{isave}(lisave)$ – INTEGER array

If $\mathbf{ind} = 0$, **isave** need not be set.

If $\mathbf{ind} = 1$, **isave** must be unchanged from the previous call to the function because it contains required information about the iteration. In particular the following components of the array **isave** concern the efficiency of the integration:

$\mathbf{isave}(1)$
: Contains the number of steps taken in time.

$\mathbf{isave}(2)$
: Contains the number of residual evaluations of the resulting ODE system used. One such evaluation involves evaluating the PDE functions at all the mesh points, as well as one evaluation of the functions in the boundary conditions.

$\mathbf{isave}(3)$
: Contains the number of Jacobian evaluations performed by the time integrator.

$\mathbf{isave}(4)$
: Contains the order of the BDF method last used in the time integration, if applicable. When the Theta method is used, $\mathbf{isave}(4)$ contains no useful information.

isave(5)
Contains the number of Newton iterations performed by the time integrator. Each iteration involves residual evaluation of the resulting ODE system followed by a back-substitution using the *LU* decomposition of the Jacobian matrix.

28: **itask** – INTEGER

The task to be performed by the ODE integrator.

itask $= 1$
Normal computation of output values **u** at $t =$ **tout** (by overshooting and interpolating).

itask $= 2$
Take one step in the time direction and return.

itask $= 3$
Stop at first internal integration point at or beyond $t =$ **tout**.

itask $= 4$
Normal computation of output values **u** at $t =$ **tout** but without overshooting $t = t_{\mathrm{crit}}$ where t_{crit} is described under the argument **algopt**.

itask $= 5$
Take one step in the time direction and return, without passing t_{crit}, where t_{crit} is described under the argument **algopt**.

Constraint: **itask** $= 1$, 2, 3, 4 or 5.

29: **itrace** – INTEGER

The level of trace information required from nag_pde_1d_parab_convdiff_remesh (d03ps) and the underlying ODE solver. **itrace** may take the value -1, 0, 1, 2 or 3.

itrace $= -1$
No output is generated.

itrace $= 0$
Only warning messages from the PDE solver are printed on the current error message unit (see nag_file_set_unit_error (x04aa)).

itrace > 0
Output from the underlying ODE solver is printed on the current advisory message unit (see nag_file_set_unit_advisory (x04ab)). This output contains details of Jacobian entries, the nonlinear iteration and the time integration during the computation of the ODE system.

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. You are advised to set **itrace** $= 0$, unless you are experienced with Sub-chapter D02M–N.

30: **ind** – INTEGER

Indicates whether this is a continuation call or a new integration.

ind $= 0$
Starts or restarts the integration in time.

ind $= 1$
Continues the integration after an earlier exit from the function. In this case, only the arguments **tout**, **ifail**, **nrmesh** and **trmesh** may be reset between calls to nag_pde_1d_parab_convdiff_remesh (d03ps).

Constraint: **ind** $= 0$ or 1.

5.2 Optional Input Parameters

1: **npts** – INTEGER

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

The number of mesh points in the interval $[a, b]$.

Constraint: **npts** ≥ 3.

2: **nxi** – INTEGER

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

The number of ODE/PDE coupling points.

Constraints:

if **ncode** $= 0$, **nxi** $= 0$;
if **ncode** > 0, **nxi** ≥ 0.

3: **neqn** – INTEGER

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

The number of ODEs in the time direction.

Constraint: **neqn** $=$ **npde** $\times$ **npts** $+$ **ncode**.

4: **nxfix** – INTEGER

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

The number of fixed mesh points.

Constraint: $0 \leq$ **nxfix** $\leq$ **npts** $- 2$.

Note: the end points **x**(1) and **x**$($**npts**$)$ are fixed automatically and hence should not be specified as fixed points.

5: **xratio** – REAL (KIND=nag_wp)

Suggested value: **xratio** $= 1.5$.

Default: 1.5

An input bound on the adjacent mesh ratio (greater than 1.0 and typically in the range 1.5 to 3.0). The remeshing functions will attempt to ensure that

$$(x_i - x_{i-1})/\mathbf{xratio} < x_{i+1} - x_i < \mathbf{xratio} \times (x_i - x_{i-1}).$$

Constraint: **xratio** > 1.0.

6: **con** – REAL (KIND=nag_wp)

Suggested value: **con** $= 2.0/($**npts** $- 1)$.

Default: $2.0/($**npts** $- 1)$

An input bound on the sub-integral of the monitor function $F^{\mathrm{mon}}(x)$ over each space step. The remeshing functions will attempt to ensure that

$$\int_{x_i}^{x_{i+1}} F^{\mathrm{mon}}(x)\,dx \leq \mathbf{con} \int_{x_1}^{x_{\mathbf{npts}}} F^{\mathrm{mon}}(x)\,dx,$$

(see Furzeland (1984)). **con** gives you more control over the mesh distribution, e.g., decreasing **con** allows more clustering. A typical value is $2.0/($**npts** $- 1)$, but you are encouraged to experiment with different values. Its value is not critical and the mesh should be qualitatively correct for all values in the range given below.

Constraint: $0.1/(\mathbf{npts} - 1) \leq \mathbf{con} \leq 10.0/(\mathbf{npts} - 1)$.

5.3 Output Parameters

1: **ts** – REAL (KIND=nag_wp)

The value of t corresponding to the solution values in **u**. Normally **ts** = **tout**.

2: **u**(**neqn**) – REAL (KIND=nag_wp) array

$\mathbf{u}(\mathbf{npde} \times (j - 1) + i)$ contains the computed solution $U_i(x_j, t)$, for $i = 1, 2, \ldots, \mathbf{npde}$ and $j = 1, 2, \ldots, \mathbf{npts}$, and $\mathbf{u}(\mathbf{npts} \times \mathbf{npde} + k)$ contains $V_k(t)$, for $k = 1, 2, \ldots, \mathbf{ncode}$, all evaluated at $t = \mathbf{ts}$.

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

The final values of the mesh points.

4: **rsave**(*lrsave*) – REAL (KIND=nag_wp) array

If **ind** = 1, **rsave** must be unchanged from the previous call to the function because it contains required information about the iteration.

5: **isave**(*lisave*) – INTEGER array

If **ind** = 1, **isave** must be unchanged from the previous call to the function because it contains required information about the iteration. In particular the following components of the array **isave** concern the efficiency of the integration:

isave(1)
Contains the number of steps taken in time.

isave(2)
Contains the number of residual evaluations of the resulting ODE system used. One such evaluation involves evaluating the PDE functions at all the mesh points, as well as one evaluation of the functions in the boundary conditions.

isave(3)
Contains the number of Jacobian evaluations performed by the time integrator.

isave(4)
Contains the order of the BDF method last used in the time integration, if applicable. When the Theta method is used, **isave**(4) contains no useful information.

isave(5)
Contains the number of Newton iterations performed by the time integrator. Each iteration involves residual evaluation of the resulting ODE system followed by a back-substitution using the *LU* decomposition of the Jacobian matrix.

6: **ind** – INTEGER

ind = 1.

7: **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, **ts** $\geq$ **tout**,
or **tout** $-$ **ts** is too small,
or **itask** $\neq 1$, 2, 3, 4 or 5,
or at least one of the coupling points defined in array **xi** is outside the interval $[\mathbf{x}(1), \mathbf{x}(\mathbf{npts})]$,
or the coupling points are not in strictly increasing order,
or **npts** < 3,
or **npde** < 1,
or **laopt** $\neq$ 'F', 'B' or 'S',
or **itol** $\neq 1$, 2, 3 or 4,
or **ind** $\neq 0$ or 1,
or mesh points $\mathbf{x}(i)$ are badly ordered,
or *lrsave* is too small,
or *lisave* is too small,
or **ncode** and **nxi** are incorrectly defined,
or **ind** $= 1$ on initial entry to nag_pde_1d_parab_convdiff_remesh (d03ps),
or **neqn** $\neq$ **npde** $\times$ **npts** $+$ **ncode**,
or an element of **rtol** or **atol** < 0.0,
or corresponding elements of **rtol** and **atol** are both 0.0,
or **norm_p** $\neq$ '1' or '2',
or **nxfix** not in the range 0 to **npts** $- 2$,
or fixed mesh point(s) do not coincide with any of the user-supplied mesh points,
or **dxmesh** < 0.0,
or **ipminf** $\neq 0$, 1 or 2,
or **xratio** ≤ 1.0,
or **con** not in the range $0.1/($**npts** $- 1)$ to $10.0/($**npts** $- 1)$.

ifail $= 2$ (*warning*)

The underlying ODE solver cannot make any further progress, with the values of **atol** and **rtol**, across the integration range from the current point $t =$ **ts**. The components of **u** contain the computed values at the current point $t =$ **ts**.

ifail $= 3$ (*warning*)

In the underlying ODE solver, there were repeated error test failures on an attempted step, before completing the requested task, but the integration was successful as far as $t =$ **ts**. The problem may have a singularity, or the error requirement may be inappropriate. Incorrect specification of boundary conditions may also result in this error.

ifail $= 4$

In setting up the ODE system, the internal initialization function was unable to initialize the derivative of the ODE system. This could be due to the fact that **ires** was repeatedly set to 3 in one of **pdedef**, **numflx**, **bndary** or **odedef**, when the residual in the underlying ODE solver was being evaluated. Incorrect specification of boundary conditions may also result in this error.

ifail $= 5$

In solving the ODE system, a singular Jacobian has been encountered. Check the problem formulation.

ifail $= 6$ (*warning*)

When evaluating the residual in solving the ODE system, **ires** was set to 2 in at least one of **pdedef**, **numflx**, **bndary** or **odedef**. Integration was successful as far as $t =$ **ts**.

ifail $= 7$

The values of **atol** and **rtol** are so small that the function is unable to start the integration in time.

ifail $= 8$

In one of **pdedef**, **numflx**, **bndary** or **odedef**, **ires** was set to an invalid value.

ifail $= 9$ (nag_ode_ivp_stiff_imp_revcom (d02nn))

A serious error has occurred in an internal call to the specified function. Check the problem specification and all arguments and array dimensions. Setting **itrace** $= 1$ may provide more information. If the problem persists, contact NAG.

ifail $= 10$ (*warning*)

The required task has been completed, but it is estimated that a small change in **atol** and **rtol** is unlikely to produce any change in the computed solution. (Only applies when you are not operating in one step mode, that is when **itask** $\neq 2$ or 5.)

ifail $= 11$

An error occurred during Jacobian formulation of the ODE system (a more detailed error description may be directed to the current advisory message unit when **itrace** ≥ 1). If using the sparse matrix algebra option, the values of **algopt**(29) and **algopt**(30) may be inappropriate.

ifail $= 12$

In solving the ODE system, the maximum number of steps specified in **algopt**(15) have been taken.

ifail $= 13$ (*warning*)

Some error weights w_i became zero during the time integration (see the description of **itol**). Pure relative error control (**atol**$(i) = 0.0$) was requested on a variable (the ith) which has become zero. The integration was successful as far as $t =$ **ts**.

ifail $= 14$

One or more of the functions $P_{i,j}$, D_i or C_i was detected as depending on time derivatives, which is not permissible.

ifail $= 15$

When using the sparse option, the value of *lisave* or *lrsave* was not sufficient (more detailed information may be directed to the current error message unit, see nag_file_set_unit_error (x04aa)).

ifail $= 16$

remesh has been changed between calls to nag_pde_1d_parab_convdiff_remesh (d03ps).

ifail $= 17$

fmon is negative at one or more mesh points, or zero mesh spacing has been obtained due to an inappropriate choice of monitor function.

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

nag_pde_1d_parab_convdiff_remesh (d03ps) controls the accuracy of the integration in the time direction but not the accuracy of the approximation in space. The spatial accuracy depends on both the number of mesh points and on their distribution in space. In the time integration only the local error over a single step is controlled and so the accuracy over a number of steps cannot be guaranteed. You should therefore test the effect of varying the accuracy arguments, **atol** and **rtol**.

8 Further Comments

nag_pde_1d_parab_convdiff_remesh (d03ps) is designed to solve systems of PDEs in conservative form, with optional source terms which are independent of space derivatives, and optional second-order diffusion terms. The use of the function to solve systems which are not naturally in this form is discouraged, and you are advised to use one of the central-difference scheme functions for such problems.

You should be aware of the stability limitations for hyperbolic PDEs. For most problems with small error tolerances the ODE integrator does not attempt unstable time steps, but in some cases a maximum time step should be imposed using **algopt**(13). It is worth experimenting with this argument, particularly if the integration appears to progress unrealistically fast (with large time steps). Setting the maximum time step to the minimum mesh size is a safe measure, although in some cases this may be too restrictive.

Problems with source terms should be treated with caution, as it is known that for large source terms stable and reasonable looking solutions can be obtained which are in fact incorrect, exhibiting non-physical speeds of propagation of discontinuities (typically one spatial mesh point per time step). It is essential to employ a very fine mesh for problems with source terms and discontinuities, and to check for non-physical propagation speeds by comparing results for different mesh sizes. Further details and an example can be found in Pennington and Berzins (1994).

The time taken depends on the complexity of the system, the accuracy requested, and the frequency of the mesh updates. For a given system with fixed accuracy and mesh-update frequency it is approximately proportional to **neqn**.

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 is a simple model of the advection and diffusion of a cloud of material:

$$\frac{\partial U}{\partial t} + W\frac{\partial U}{\partial x} = C\frac{\partial^2 U}{\partial x^2},$$

for $x \in [0,1]$ and $t \leq 0 \leq 0.3$. In this example the constant wind speed $W = 1$ and the diffusion coefficient $C = 0.002$.

The cloud does not reach the boundaries during the time of integration, and so the two (physical) boundary conditions are simply $U(0,t) = U(1,t) = 0.0$, and the initial condition is

$$U(x,0) = \sin\left(\pi\frac{x-a}{b-a}\right), \quad a \leq x \leq b,$$

and $U(x,0) = 0$ elsewhere, where $a = 0.2$ and $b = 0.4$.

The numerical flux is simply $\hat{F} = WU_L$.

The monitor function for remeshing is taken to be the absolute value of the second derivative of U.

Example 2 (EX2)

This example is a linear advection equation with a nonlinear source term and discontinuous initial profile:

$$\frac{\partial u}{\partial t}+\frac{\partial u}{\partial x}=-pu(u-1)\left(u-\tfrac{1}{2}\right),$$

for $0 \leq x \leq 1$ and $t \geq 0$. The discontinuity is modelled by a ramp function of width 0.01 and gradient 100, so that the exact solution at any time $t \geq 0$ is

$$u(x,t)=1.0+\max(\min(\delta,0),-1),$$

where $\delta=100(0.1-x+t)$. The initial profile is given by the exact solution. The characteristic points into the domain at $x=0$ and out of the domain at $x=1$, and so a physical boundary condition $u(0,t)=1$ is imposed at $x=0$, with a numerical boundary condition at $x=1$ which can be specified as $u(1,t)=0$ since the discontinuity does not reach $x=1$ during the time of integration.

The numerical flux is simply $\hat{F}=U_L$ at all times.

The remeshing monitor function (described below) is chosen to create an increasingly fine mesh towards the discontinuity in order to ensure good resolution of the discontinuity, but without loss of efficiency in the surrounding regions. However, refinement must be limited so that the time step required for stability does not become unrealistically small. The region of refinement must also keep up with the discontinuity as it moves across the domain, and hence it cannot be so small that the discontinuity moves out of the refined region between remeshing.

The above requirements mean that the use of the first or second spatial derivative of U for the monitor function is inappropriate; the large relative size of either derivative in the region of the discontinuity leads to extremely small mesh-spacing in a very limited region, and the solution is then far more expensive than for a very fine fixed mesh.

An alternative monitor function based on a cosine function proves very successful. It is only semi-automatic as it requires some knowledge of the solution (for problems without an exact solution an initial approximate solution can be obtained using a coarse fixed mesh). On each call to **monitf** the discontinuity is located by finding the maximum spatial derivative of the solution. On the first call the desired width of the region of nonzero monitor function is set (this can be changed at a later time if desired). Then on each call the monitor function is assigned using a cosine function so that it has a value of one at the discontinuity down to zero at the edges of the predetermined region of refinement, and zero outside the region. Thus the monitor function and the subsequent refinement are limited, and the region is large enough to ensure that there is always sufficient refinement at the discontinuity.

9.1 Program Text

```
     function d03ps_example

fprintf('d03ps example results\n\n');

global usav1 usav2;
fprintf(' Example 1\n\n');
ex1;
% Plot results.
fig1 = figure;
ex1_plot;

fprintf('\n\n\n Example 2\n\n');
ex2;
% Plot results.
fig2 = figure;
ex2_plot;

function ex1
  % First d03ps example.  Advection and diffusion of a cloud of material.
  global xsav1 tsav1 usav1;

  % Set values for problem parameters.
  ncode  = nag_int(0);
```

```
nxi    = nag_int(0);
npde   = nag_int(1);

% Number of points on calculation mesh, and on interpolated mesh.
npts   = 61;

neqn   = npde*npts + ncode;
lisave = 25 + neqn;
lrsave = 5000;

% Define some arrays.
algopt = zeros(30, 1);
u      = zeros(npde, npts);
x      = [0:1/(npts-1):1];
rsave  = zeros(lrsave, 1);
isave  = zeros(lisave, 1, nag_int_name);

itrace = nag_int(0);
itol   = nag_int(1);
norm_p = '1';
atol   = [0.0001];
rtol   = [0.0001];

xfix   = [];

remesh = true;
nrmesh = nag_int(3);
dxmesh = 0.0;
trmesh = 0.0;
con    = 2.0/(npts-1.0);
xratio = 1.5;
ipminf = nag_int(0);
xi     = [0.0];
laopt  = 'B';

% b.d.f. integration.
algopt = zeros(30,1);
algopt(1)  = 1.0;
algopt(13) = 0.005;

% We run through the calculation twice - once to collect the results
% for plotting, and once to do interpolation to a coarser grid and
% and output the results.  Set ind = 0 to (re)start the integration
% in time, and set time to its initial value. Set itask = 2, so that
% the integrator takes one time-step at a time.
ind   = nag_int(0);
ts    = 0.0;
itask = nag_int(2);
tout  = 0.3;
tnext = 0.0;

fprintf(' npts = %4d atol = %10.3e  rtol = %10.3e \n\n', npts, atol, rtol);
isav  = 0;
while ts < tout
  [ts, u, x, rsave, isave, ind, ifail] = ...
  d03ps( ...
         npde, ts, tout, @ex1_pdedef, @ex1_numflx, @ex1_bndary, ...
         @ex1_uvinit, u, x, ncode, 'd03pek', xi, rtol, atol, itol, ...
         norm_p, laopt, algopt, remesh, xfix, nrmesh, dxmesh, ...
         trmesh, ipminf, @ex1_monitf, rsave, isave, itask, itrace, ...
         ind, 'nxi', nxi, 'xratio', xratio, 'con', con);

  if ts >= tnext
    % Save this timestep, and this set of results.  Note we have to
    % save the x coordinates at this timestep as well, because the
    % algorithm is automatically remeshing the spatial grid.
    isav = isav+1;
    tsav1(isav) = ts;
    for i=1:npts
      usav1(isav,i) = u(1,i);
      xsav1(isav,i) = x(i);
      end
      tnext = tnext + 0.01;
    end
  end

  % Now prepare to run through the calculation again.  First, set up
  % the points on the interpolation grid.
  intpts = 7;
  for i = 0:intpts - 1
      xinterp(i+1) = 0.2 + i*0.1;
  end

  % Set ind = 0 to (re)start the integration in time, set the time to its
  % initial value, and set itask = 1 to compute solution at t=tout by
  % overshooting & interpolating. Reset the calculation mesh, which had been
  % modified by the earlier calls to d03ps.
  ind   = nag_int(0);
  itask = nag_int(1);
  ts = 0.0;
  dx = 1/(npts-1);
  x = [0:dx:1];

  for it = 1:3
    tout = 0.1*it;
    [ts, u, x, rsave, isave, ind, ifail] = ...
    d03ps( ...
           npde, ts, tout, @ex1_pdedef, @ex1_numflx, @ex1_bndary, ...
           @ex1_uvinit, u, x, ncode, 'd03pek', xi, rtol, atol, itol, ...
           norm_p, laopt, algopt, remesh, xfix, nrmesh, dxmesh, trmesh, ...
           ipminf, @ex1_monitf, rsave, isave, itask, itrace, ...
           ind, 'nxi', nxi, 'xratio', xratio, 'con', con);

    fprintf(' t = %6.3f\n', ts);
    fprintf(' x       ');
    fprintf('%9.4f', xinterp);
    fprintf('\n');

    % Call d03pz to do interpolation of results onto coarser grid.
    m = nag_int(0);
    itype = nag_int(1);
    [uinterp, ifail] = d03pz( ...
                            m, u, x, xinterp, itype);
    fprintf(' approx u');
    fprintf('%9.4f', uinterp);
    fprintf('\n\n');
  end

  % Output some statistics.
  fprintf([' Number of integration steps in time = %6d\n', ...
           ' Number of function evaluations = %6d\n', ...
           ' Number of Jacobian evaluations = %6d\n', ...
           ' Number of iterations = %6d\n'], ...
          isave(1), isave(2), isave(3), isave(5));

function [p, c, d, s, ires] = ex1_pdedef(npde, t, x, u, ux, ncode, v, ...
                                         vdot, ires)
  p = zeros(npde, npde);
  c = zeros(npde, 1);
  d = zeros(npde, 1);
  s = zeros(npde, 1);

  p(1,1) = 1;
  c(1) = 0.2d-2;
  d(1) = ux(1);

function [flux, ires] = ex1_numflx(npde, t, x, ncode, v, uleft, uright, ires)
  flux = zeros(npde, 1);
  flux(1) = uleft(1);

function [g, ires] = ex1_bndary(npde, npts, t, x, u, ncode, v, vdot, ibnd, ires)
  g = zeros(npde, 1);
  if (ibnd == 0)
    g(1) = u(1,1);
  else
    g(1) = u(1,npts);
  end

function [u, v] = ex1_uvinit(npde, npts, nxi, x, xi, ncode)
  u = zeros(npde, npts);
  v = zeros(ncode, 1);

  for i = 1:double(npts)
    if (x(i) > 0.2 && x(i) <= 0.4)
      u(1,i) = sin(pi*(5*x(i)-1));
    end
  end

function [r, ires] = ex1_odedef(npde, t, ncode, v, vdot, nxi, xi, ucp, ucpx, ...
                                 ucpt, ires)
  r = zeros(ncode, 1);

function [fmon] = ex1_monitf(t, npts, npde, x, u)
  fmon = zeros(npts, 1);

  for i = 2:double(npts) - 1
    h1 = x(i) - x(i-1);
    h2 = x(i+1) - x(i);
    h3 = (x(i+1)-x(i-1))/2;
    % second derivatives ..
    fmon(i) = abs(((u(1,i+1)-u(1,i))/h2-(u(1,i)-u(1,i-1))/h1)/h3);
  end
  fmon(1) = fmon(2);
  fmon(npts) = fmon(double(npts)-1);

function ex2
  % Second d03ps example. Linear advection equation w/ non-linear source term.
  global xsav2 tsav2 usav2;

  % Set values for problem parameters.
  ncode = nag_int(0);
  nxi   = nag_int(0);
  npde  = nag_int(1);

  % Number of points on calculation mesh, and on interpolated mesh.
  npts = 61;

  neqn   = npde*npts + ncode;
  lisave = 25 + neqn;
  lrsave = 5000;

  % Define some arrays.
  algopt = zeros(30, 1);
  u = zeros(npde, npts);
  x = zeros(npts, 1);
  rsave = zeros(lrsave, 1);
  isave = zeros(lisave, 1, nag_int_name);

  itrace = nag_int(0);
  itol = nag_int(1);
  norm_p = '1';
  atol = [0.0005];
  rtol = [0.05];

  % Initialize calculation mesh.
  dx = 1/(npts-1);
  x = [0:dx:1];
  xfix = [];

  % Set remesh parameters.
  remesh = true;
  nrmesh = nag_int(5);
```

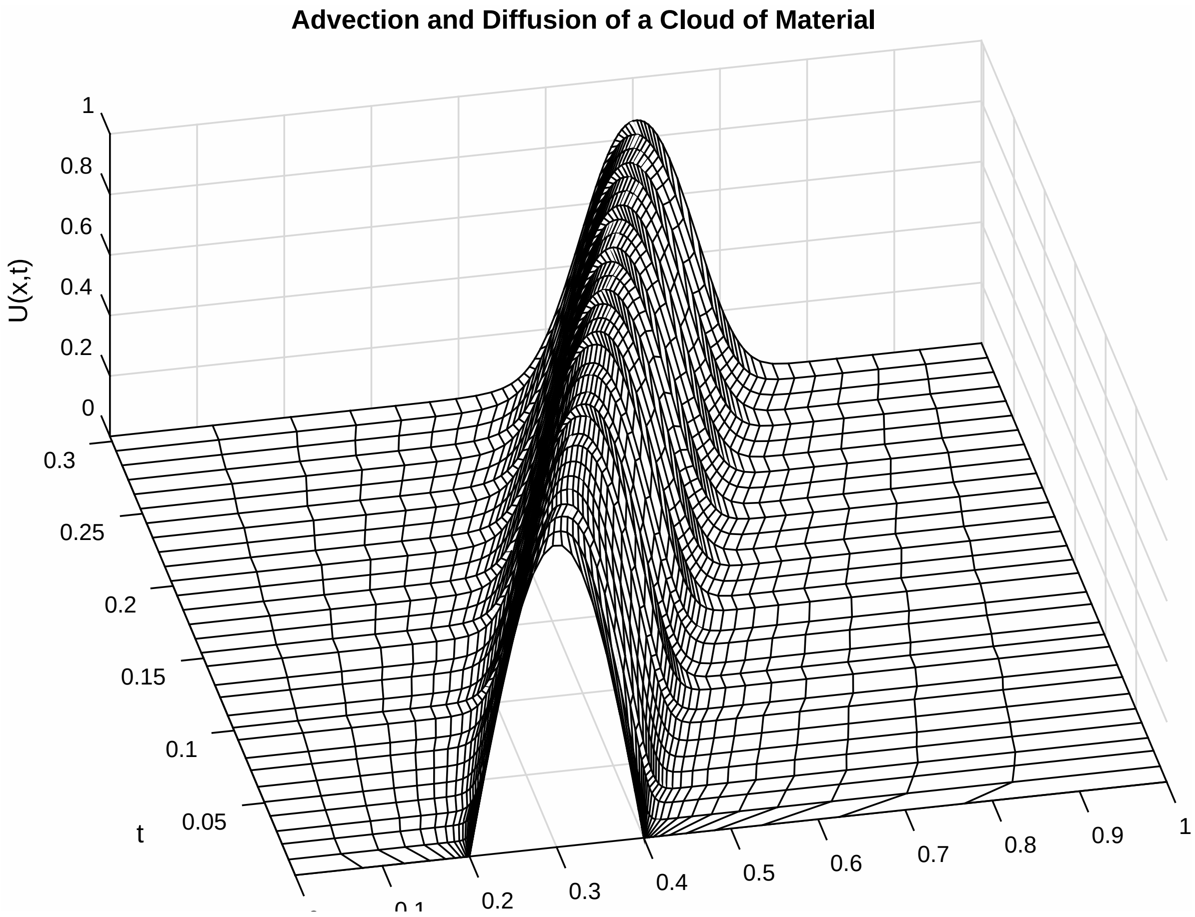
```
dxmesh = 0;
trmesh = 0;
con = dx;
xratio = 1.5;
ipminf = nag_int(0);
xi = [0.0];
laopt = 'B';

% b.d.f. integration.
algopt = zeros(30,1);
% Theta integration.
algopt(1) = 2.0;
algopt(6) = 2.0;
algopt(7) = 2.0;
% Maximum time step.
algopt(13) = 0.0025;

% We run through the calculation twice - once to collect the results for
% plotting, and once to do interpolation to a coarser grid and and output
% the results.  Set ind = 0 to (re)start the integration in time, and set
% the time to its initial value. Set itask = 2, tp take one time-step at
% a time.
ind = nag_int(0);
ts = 0.0;
itask = nag_int(2);
tout = 0.4;
tnext = 0.0;

isav = 0;
while ts < tout
  [ts, u, x, rsave, isave, ind, ifail] = ...
  d03ps( ...
         npde, ts, tout, @ex2_pdedef, @ex2_numflx, @ex2_bndary, ...
         @ex2_uvinit, u, x, ncode, 'd03pek', xi, rtol, atol, itol, ...
         norm_p, laopt, algopt, remesh, xfix, nrmesh, dxmesh, trmesh, ...
         ipminf, @ex2_monitf, rsave, isave, itask, itrace, ...
         ind, 'nxi', nxi, 'xratio', xratio, 'con', con);

  if ts >= tnext
    % Save this timestep, and this set of results.  Note we have to
    % save the x coordinates at this timestep as well, because the
    % algorithm is automatically remeshing the spatial grid.
    isav = isav+1;
    tsav2(isav) = ts;
    usav2(isav,1:npts) = u(1,:);
    xsav2(isav,1:npts) = x;
    tnext = tnext + 0.01;
  end
end

% Now prepare to run through the calculation again.  First, set up
% the points on the interpolation grid.
intpts = 7;
xinterp = [0.2:0.1:0.8];

% We set ind = 0 to (re)start the integration in time, set the time to its
% initial value, and set itask = 1 to compte solution at t=tout by
% overshooting & interpolating.
% We also need to reset the calculation mesh, since this has been modified
% by the earlier calls to d03ps.
ind   = nag_int(0);
itask = nag_int(1);
ts = 0;
dx = 1/(npts-1);
x = [0:dx:1];

for it = 1:2
  tout = 0.2*it;
  [ts, u, x, rsave, isave, ind, ifail] = ...
  d03ps( ...
         npde, ts, tout, @ex2_pdedef, @ex2_numflx, @ex2_bndary, ...
```

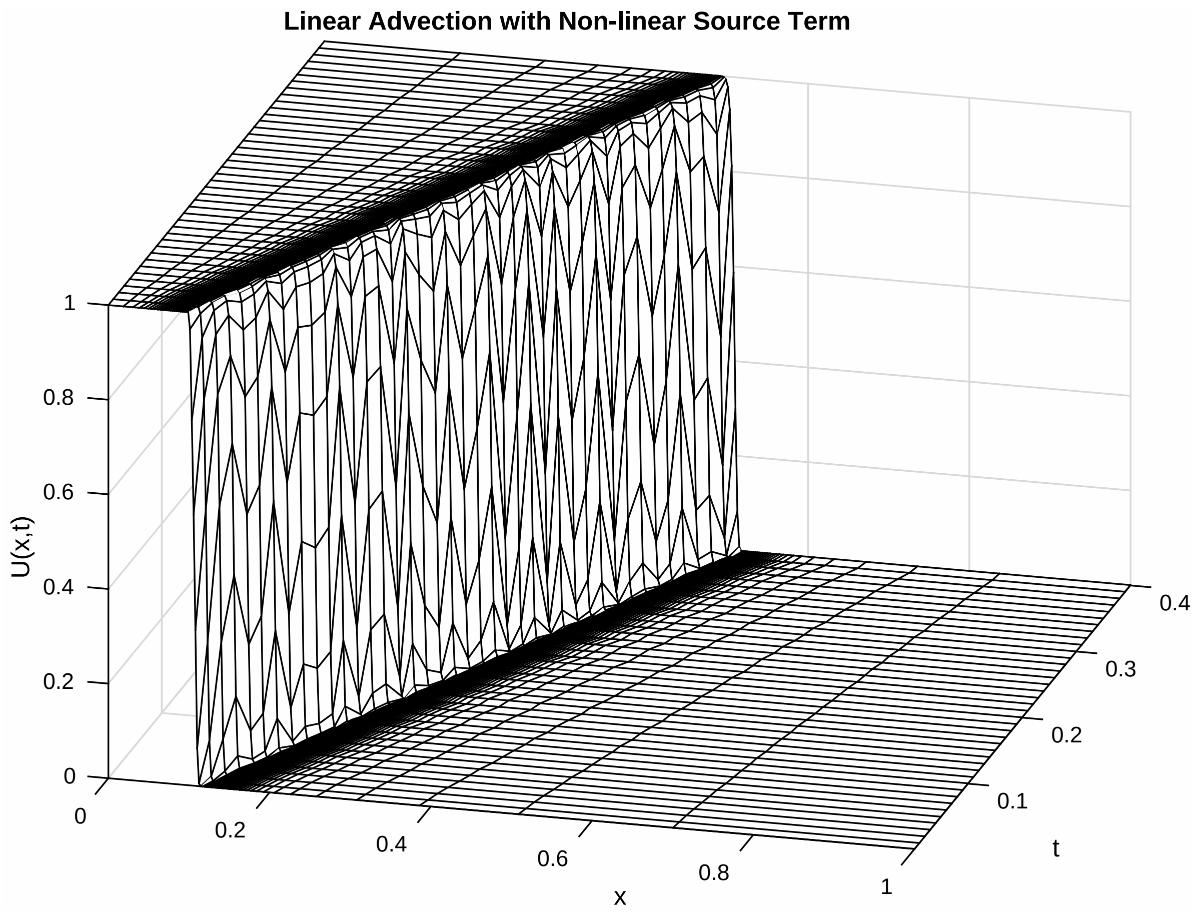
```
          @ex2_uvinit, u, x, ncode, 'd03pek', xi, rtol, atol, itol, ...
          norm_p, laopt, algopt, remesh, xfix, nrmesh, dxmesh, trmesh, ...
          ipminf, @ex2_monitf, rsave, isave, itask, itrace, ...
          ind, 'nxi', nxi, 'xratio', xratio, 'con', con);

    if it == 1
      fprintf(' npts = %4d atol = %10.3e  rtol = %10.3e \n\n', npts, ...
             atol, rtol)
    end
    fprintf(' t = %6.3f\n', ts);

    % Call d03pz to do interpolation of results onto coarser grid.
    m = nag_int(0);
    itype = nag_int(1);
    [uinterp, ifail] = d03pz(m, u, x, xinterp, itype);

    % Check against exact solution.
    [ue] = ex2_exact(tout, xinterp, intpts);
    fprintf('       x      approx u    exact u\n');
    for i=1:intpts
      fprintf('%9.4f   %9.4f   %9.4f\n', xinterp(i), uinterp(i), ue(i));
    end
    fprintf('\n\n');
  end

  % Output some statistics.
  fprintf([' Number of integration steps in time = %6d\n', ...
          ' Number of function evaluations = %6d\n', ...
          ' Number of Jacobian evaluations = %6d\n', ...
          ' Number of iterations = %6d\n'], ...
         isave(1), isave(2), isave(3), isave(5));

function [p, c, d, s, ires] = ex2_pdedef(npde, t, x, u, ux, ...
      ncode, v, vdot, ires)
  % Evaluate pij, ci, di, si to partially define the system of PDEs.

  p = zeros(npde, npde);
  c = zeros(npde, 1);
  d = zeros(npde, 1);
  s = zeros(npde, 1);

  p(1,1) = 1;
  c(1) = 0;
  d(1) = 0;
  s(1) = -100*u(1)*(u(1) - 1)*(u(1) - 0.5);

function [fmon] = ex2_monitf(t, npts, npde, x, u)
  % Supplies and evaluates a remesh monitor function.
  persistent icount;
  persistent xa;

  fmon = zeros(npts, 1);

  % Locate shock.
  uxmax = 0;
  xmax = 0;
  for i = 2:npts-1
    h1 = x(i) - x(i-1);
    ux = abs((u(1,i) - u(1,i-1))/h1);
    if (ux > uxmax)
      uxmax = ux;
      xmax = x(i);
    end
  end

  % Assign width (on first call only).
  if (isempty(icount))
    icount = 1;
    xleft = xmax - x(1);
    xright = x(npts) - xmax;
    if (xleft > xright)
```

```
      xa = xright;
    else
      xa = xleft;
    end
  end

  % Assign monitor function.
  xl = xmax - xa;
  xr = xmax + xa;
  for i = 1:npts
    if ((x(i) > xl) && (x(i) < xr))
      fmon(i) = 1.0 + cos(pi*(x(i) - xmax)/xa);
    else
      fmon(i) = 0.0;
    end
  end

function [g, ires] = ex2_bndary(npde, npts, t, x, u, ncode, v, ...
                                vdot, ibnd, ires)
  % Evaluate giL and giR to describe the physical & numerical boundary
  % conditions.

  % Solution known to be constant at both boundaries.
  if (ibnd == 0)
    [ue] = ex2_exact(t, x(1), 1);
    g(1) = ue(1,1) - u(1,1);
  else
    [ue] = ex2_exact(t, x(npts), 1);
    g(1) = ue(1,1) - u(1,npts);
  end

function [flux, ires] = ex2_numflx(npde, t, x, ncode, v, uleft, ...
                                   uright, ires)
  % Supplies numerical flux for each PDE given the L & R values of u.
  flux = zeros(npde, 1);
  flux(1) = uleft(1);

function [u, v] = ex2_uvinit(npde, npts, nxi, x, xi, ncode)
  % Specify initial values (at t=t0) of u(x,t) and v(t) for all x.

  u = zeros(npde, npts);
  v = zeros(ncode, 1);
  t = 0.0;
  [u] = ex2_exact(t, x, npts);

function [u] = ex2_exact(t, x, npts)
  % Calculate exact solution (for comparison and use in boundary conditions).

  s = 0.1;
  del = 0.01;
  rm = -1.0/del;
  rn = 1.0 + s/del;

  for i = 1:npts
    psi = x(i) - t;
    if (psi < s)
      u(1,i) = 1.0;
    elseif (psi > (del+s))
      u(1,i) = 0.0;
    else
      u(1,i) = rm*psi + rn;
    end
  end

function ex1_plot
  % Plot the results.
  global xsav1 tsav1 usav1;

  % Plot array as a mesh.
  mesh(xsav1, tsav1, usav1);
  % Label the axes, and set the title.
  xlabel('x');
  ylabel('t');
  zlabel('U(x,t)');
  title('Advection and Diffusion of a Cloud of Material');

  % Set the axes limits tight to the x and y range.
  axis([xsav1(1) xsav1(end) tsav1(1) tsav1(end) 0 1]);
  view(-12, 56);

function ex2_plot
  % Plot the results.
  global xsav2 tsav2 usav2;

  % Plot array as a mesh.
  mesh(xsav2, tsav2, usav2);

  % Label the axes, and set the title.
  xlabel('x');
  ylabel('t');
  zlabel('U(x,t)');
  title('Linear Advection with Non-linear Source Term');

  % Set the axes limits tight to the x and y range.
  axis([xsav2(1) xsav2(end) tsav2(1) tsav2(end) 0 1]);
  view(15, 30);
```

9.2 Program Results

```
     d03ps example results

 Example 1

 npts =   61 atol =  1.000e-04  rtol =  1.000e-04

 t =  0.100
 x          0.2000   0.3000   0.4000   0.5000   0.6000   0.7000   0.8000
 approx u   0.0000   0.1198   0.9461   0.1182   0.0000   0.0000   0.0000

 t =  0.200
 x          0.2000   0.3000   0.4000   0.5000   0.6000   0.7000   0.8000
 approx u   0.0000   0.0007   0.1631   0.9015   0.1629   0.0001   0.0000

 t =  0.300
 x          0.2000   0.3000   0.4000   0.5000   0.6000   0.7000   0.8000
 approx u   0.0000   0.0000   0.0025   0.1924   0.8596   0.1946   0.0002

 Number of integration steps in time =     92
 Number of function evaluations =    443
 Number of Jacobian evaluations =     39
 Number of iterations =    231

 Example 2

 npts =   61 atol =  5.000e-04  rtol =  5.000e-02

 t =  0.200
        x      approx u    exact u
   0.2000      1.0000      1.0000
   0.3000      0.9536      1.0000
   0.4000      0.0000      0.0000
   0.5000      0.0000      0.0000
   0.6000      0.0000      0.0000
   0.7000     -0.0000      0.0000
   0.8000      0.0000      0.0000

 t =  0.400
        x      approx u    exact u
   0.2000      1.0000      1.0000
   0.3000      1.0000      1.0000
   0.4000      1.0000      1.0000
   0.5000      0.9750      1.0000
   0.6000     -0.0000      0.0000
   0.7000      0.0000      0.0000
   0.8000      0.0000      0.0000

 Number of integration steps in time =    672
 Number of function evaluations =   1515
 Number of Jacobian evaluations =      1
 Number of iterations =      2
```




Linear Advection with Non-linear Source Term
U(x,t)
0
0.2
0.4
0.6
0.8
1
x
0
0.2
0.4
0.6
0.8
1
t
0.1
0.2
0.3
0.4