NAG Library Function Document

nag_pde_parab_1d_cd_ode (d03plc)

1 Purpose

nag_pde_parab_1d_cd_ode (d03plc) 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. 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 Specification

```
#include <nag.h>
#include <nagd03.h>
void nag_pde_parab_1d_cd_ode (Integer npde, double *ts, double tout,
     void (*pdedef)(Integer npde, double t, double x, const double u[],
          const double ux[], Integer ncode, const double v[],
          const double vdot[], double p[], double c[], double d[],
          double s[], Integer *ires, Nag_Comm *comm),
     void (*numflx)(Integer npde, double t, double x, Integer ncode,
          const double v[], const double uleft[], const double uright[],
          double flux[], Integer *ires, Nag_Comm *comm, Nag_D03_Save *saved),
     void (*bndary)(Integer npde, Integer npts, double t, const double x[],
          const double u[], Integer ncode, const double v[],
          const double vdot[], Integer ibnd, double g[], Integer *ires,
          Nag_Comm *comm),
     double u[], Integer npts, const double x[], Integer ncode,
     void (*odedef)(Integer npde, double t, Integer ncode, const double v[],
          const double vdot[], Integer nxi, const double xi[],
          const double ucp[], const double ucpx[], const double ucpt[],
          double r[], Integer *ires, Nag_Comm *comm),
     Integer nxi, const double xi[], Integer neqn, const double rtol[],
     const double atol[], Integer itol, Nag_NormType norm,
     Nag_LinAlgOption laopt, const double algopt[], double rsave[],
     Integer lrsave, Integer isave[], Integer lisave, Integer itask,
     Integer itrace, const char *outfile, Integer *ind, Nag_Comm *comm,
     Nag_D03_Save *saved, NagError *fail)
```

3 Description

nag_pde_parab_1d_cd_ode (d03plc) 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 (1), $P_{i,j}$, 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, $\dfrac{C_i \partial D_i}{\partial x}$ 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 $x_1, x_2, \ldots, x_{\mathbf{npts}}$. The initial values of the functions $U(x,t)$ and $V(t)$ must be given at $t = t_0$.

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_parab_1d_fd (d03pcc), nag_pde_parab_1d_fd_ode (d03phc) and nag_pde_parab_1d_fd_ode_remesh (d03ppc), 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 vector, $\hat{F}_i$ 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_parab_1d_cd_ode (d03plc) 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

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

An example is given in Section 10 and in the nag_pde_parab_1d_cd (d03pfc) documentation.

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 a separate **numflx**. For problems in the form (2)) the NAG defined null void function pointer, NULLFN, can be supplied in the call to nag_pde_parab_1d_cd_ode (d03plc).

The boundary condition specification has sufficient flexibility to allow for different types of problems. For second-order problems, i.e., D_i depending on U_x, a boundary condition is required for each PDE at both boundaries for the problem to be well-posed. If there are no second-order 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 condition, 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 Section 10 and in the nag_pde_parab_1d_cd (d03pfc) documentation.

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

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

at the left-hand boundary, and

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

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**.

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 $P_{i,j}$, F_i, C_i and D_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 mesh points $x_1, x_2, \ldots, x_{\mathbf{npts}}$;

(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.

In total there are **npde** $\times$ **npts** + **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)).

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

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

Sod G A (1978) A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws *J. Comput. Phys.* **27** 1–31

5 Arguments

1: **npde** – Integer *Input*

On entry: the number of PDEs to be solved.

Constraint: **npde** ≥ 1.

2: **ts** – double * *Input/Output*

On entry: the initial value of the independent variable t.

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

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

3: **tout** – double *Input*

On entry: the final value of t to which the integration is to be carried out.

4: **pdedef** – function, supplied by the user *External Function*

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_parab_1d_cd_ode (d03plc). For problems in the form (2)) the NAG defined null void function pointer, NULLFN, can be supplied in the call to nag_pde_parab_1d_cd_ode (d03plc).

The specification of **pdedef** is:

```
void pdedef (Integer npde, double t, double x, const double u[],
     const double ux[], Integer ncode, const double v[],
     const double vdot[], double p[], double c[], double d[],
     double s[], Integer *ires, Nag_Comm *comm)
```

1: **npde** – Integer *Input*

On entry: the number of PDEs in the system.

2: **t** – double *Input*

On entry: the current value of the independent variable t.

3: **x** – double *Input*

On entry: the current value of the space variable x.

4: **u**[**npde**] – const double *Input*

On entry: **u**$[i-1]$ contains the value of the component $U_i(x,t)$, for $i = 1,2,\ldots,$ **npde**.

5: **ux**[**npde**] – const double *Input*

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

6: **ncode** – Integer *Input*

On entry: the number of coupled ODEs in the system.

7: **v**[**ncode**] – const double *Input*

On entry: if **ncode** > 0, **v**$[i-1]$ contains the value of the component $V_i(t)$, for $i = 1,2,\ldots,$ **ncode**.

8: **vdot**[**ncode**] – const double *Input*

On entry: if **ncode** > 0, **vdot**$[i-1]$ 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: **p**[**npde** $\times$ **npde**] – double *Output*

On exit: **p**[**npde** $\times (j-1) + i - 1$] 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**.

10: **c**[**npde**] – double *Output*

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

11: **d**[**npde**] – double *Output*

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

12: **s**[**npde**] – double *Output*

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

13: **ires** – Integer * *Input/Output*

On entry: set to -1 or 1.

On exit: 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 function with the error indicator set to **fail**.**code** = NE_USER_STOP.

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_parab_1d_cd_ode (d03plc) returns to the calling function with the error indicator set to **fail**.**code** = NE_FAILED_DERIV.

14: **comm** – Nag_Comm *

Pointer to structure of type Nag_Comm; the following members are relevant to **pdedef**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_pde_parab_1d_cd_ode (d03plc) you may allocate memory and initialize these pointers with various quantities for use by **pdedef** when called from nag_pde_parab_1d_cd_ode (d03plc) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

5: **numflx** – function, supplied by the user *External Function*

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_parab_1d_cd_ode (d03plc).

The specification of **numflx** is:

```
void numflx (Integer npde, double t, double x, Integer ncode,
     const double v[], const double uleft[], const double uright[],
     double flux[], Integer *ires, Nag_Comm *comm,
     Nag_D03_Save *saved)
```

1: **npde** – Integer *Input*

On entry: the number of PDEs in the system.

2: **t** – double *Input*

On entry: the current value of the independent variable t.

3: **x** – double *Input*

On entry: the current value of the space variable x.

4: **ncode** – Integer *Input*

On entry: the number of coupled ODEs in the system.

5: **v**[**ncode**] – const double *Input*

On entry: if **ncode** > 0, **v**$[i-1]$ contains the value of the component $V_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

6: **uleft**[**npde**] – const double *Input*

On entry: **uleft**$[i-1]$ contains the *left* value of the component $U_i(x)$, for $i = 1, 2, \ldots,$ **npde**.

7: **uright**[**npde**] – const double *Input*

On entry: **uright**$[i-1]$ contains the *right* value of the component $U_i(x)$, for $i = 1, 2, \ldots,$ **npde**.

8: **flux**[**npde**] – double *Output*

On exit: **flux**$[i-1]$ must be set to the numerical flux $\hat{F}_i$, for $i = 1, 2, \ldots,$ **npde**.

9: **ires** – Integer * *Input/Output*

On entry: set to -1 or 1.

On exit: 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 function with the error indicator set to **fail**.**code** = NE_USER_STOP.

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_parab_1d_cd_ode (d03plc) returns to the calling function with the error indicator set to **fail**.**code** = NE_FAILED_DERIV.

10: **comm** – Nag_Comm *

Pointer to structure of type Nag_Comm; the following members are relevant to **numflx**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_pde_parab_1d_cd_ode (d03plc) you may allocate memory and initialize these pointers with various quantities for use by **numflx** when called from nag_pde_parab_1d_cd_ode (d03plc) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

11: **saved** – Nag_D03_Save * *Communication Structure*

If **numflx** calls one of the approximate Riemann solvers nag_pde_parab_1d_euler_roe (d03puc), nag_pde_parab_1d_euler_osher (d03pvc), nag_pde_parab_1d_euler_hll (d03pwc) or nag_pde_parab_1d_euler_exact (d03pxc) then **saved** is used to pass data concerning the computation to the solver. You should not change the components of **saved**.

6: **bndary** – function, supplied by the user *External Function*

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).

The specification of **bndary** is:

```
void bndary (Integer npde, Integer npts, double t, const double x[],
     const double u[], Integer ncode, const double v[],
     const double vdot[], Integer ibnd, double g[], Integer *ires,
     Nag_Comm *comm)
```

1: **npde** – Integer *Input*

On entry: the number of PDEs in the system.

2: **npts** – Integer *Input*

On entry: the number of mesh points in the interval $[a,b]$.

3: **t** – double *Input*

On entry: the current value of the independent variable t.

4: **x**[**npts**] – const double *Input*

On entry: the mesh points in the spatial direction. **x**[0] corresponds to the left-hand boundary, a, and **x**[**npts** -1] corresponds to the right-hand boundary, b.

5: **u**[**npde** $\times$ **npts**] – const double *Input*

On entry: **u**[**npde** $\times (j-1)+i-1$] contains the value of the component $U_i(x,t)$ at $x =$ **x**$[j-1]$, 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 *Input*

On entry: the number of coupled ODEs in the system.

7: **v**[**ncode**] – const double *Input*

On entry: if **ncode** > 0, **v**$[i-1]$ contains the value of the component $V_i(t)$, for $i = 1,2,\ldots,$ **ncode**.

8: **vdot**[**ncode**] – const double *Input*

On entry: if **ncode** > 0, **vdot**$[i-1]$ 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 *Input*

On entry: 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: **g**[**npde**] – double *Output*

On exit: **g**$[i-1]$ 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**.

11: **ires** – Integer * *Input/Output*

On entry: set to -1 or 1.

On exit: 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 function with the error indicator set to **fail**.**code** = NE_USER_STOP.

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_parab_1d_cd_ode (d03plc) returns to the calling function with the error indicator set to **fail**.**code** = NE_FAILED_DERIV.

12: **comm** – Nag_Comm *

Pointer to structure of type Nag_Comm; the following members are relevant to **bndary**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_pde_parab_1d_cd_ode (d03plc) you may allocate memory and initialize these pointers with various quantities for use by **bndary** when called from nag_pde_parab_1d_cd_ode (d03plc) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

7: **u**[**neqn**] – double *Input/Output*

On entry: the initial values of the dependent variables defined as follows:

u[**npde** $\times (j-1) + i - 1$] contain $U_i(x_j, t_0)$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **npts**, and

u[**npts** $\times$ **npde** $+ k - 1$] contain $V_k(t_0)$, for $k = 1, 2, \ldots,$ **ncode**.

On exit: the computed solution $U_i(x_j, t)$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **npts**, and $V_k(t)$, for $k = 1, 2, \ldots,$ **ncode**, all evaluated at $t =$ **ts**.

8: **npts** – Integer *Input*

On entry: the number of mesh points in the interval $[a, b]$.

Constraint: **npts** ≥ 3.

9: **x**[**npts**] – const double *Input*

On entry: the mesh points in the space direction. **x**[0] must specify the left-hand boundary, a, and **x**[**npts** $- 1$] must specify the right-hand boundary, b.

Constraint: **x**[0] $<$ **x**[1] $< \cdots <$ **x**[**npts** $- 1$].

10: **ncode** – Integer *Input*

On entry: the number of coupled ODE components.

Constraint: **ncode** ≥ 0.

11: **odedef** – function, supplied by the user *External Function*

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

If **ncode** $= 0$, **odedef** will never be called and the NAG defined null void function pointer, NULLFN, can be supplied in the call to nag_pde_parab_1d_cd_ode (d03plc).

The specification of **odedef** is:

```
void odedef (Integer npde, double t, Integer ncode, const double v[],
     const double vdot[], Integer nxi, const double xi[],
     const double ucp[], const double ucpx[], const double ucpt[],
     double r[], Integer *ires, Nag_Comm *comm)
```

1: **npde** – Integer *Input*

On entry: the number of PDEs in the system.

2: **t** – double *Input*

On entry: the current value of the independent variable t.

3: **ncode** – Integer *Input*

On entry: the number of coupled ODEs in the system.

4: **v**[**ncode**] – const double *Input*

On entry: if **ncode** > 0, **v**$[i-1]$ contains the value of the component $V_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

5: **vdot**[**ncode**] – const double *Input*

On entry: if **ncode** > 0, **vdot**$[i-1]$ contains the value of component $\dot{V}_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

6: **nxi** – Integer *Input*

On entry: the number of ODE/PDE coupling points.

7: **xi**[**nxi**] – const double *Input*

On entry: if **nxi** > 0, **xi**$[i-1]$ contains the ODE/PDE coupling point, ξ_i, for $i = 1, 2, \ldots,$ **nxi**.

8: **ucp**[**npde** $\times$ **nxi**] – const double *Input*

On entry: if **nxi** > 0, **ucp**[**npde** $\times (j-1) + i - 1$] 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** $\times$ **nxi**] – const double *Input*

On entry: if **nxi** > 0, **ucpx**[**npde** $\times (j-1) + i - 1$] contains the value of $\dfrac{\partial U_i(x,t)}{\partial x}$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **nxi**.

10: **ucpt**[**npde** $\times$ **nxi**] – const double *Input*

On entry: if **nxi** > 0, **ucpt**[**npde** $\times (j-1) + i - 1$] contains the value of $\dfrac{\partial U_i}{\partial t}$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots,$ **npde** and $j = 1, 2, \ldots,$ **nxi**.

11: **r**[**ncode**] – double *Output*

On exit: **r**$[i-1]$ 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**.

12: **ires** – Integer * *Input/Output*

On entry: 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.

On exit: 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 function with the error indicator set to **fail**.**code** = NE_USER_STOP.

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_parab_1d_cd_ode (d03plc) returns to the calling function with the error indicator set to **fail**.**code** = NE_FAILED_DERIV.

13: **comm** – Nag_Comm *

Pointer to structure of type Nag_Comm; the following members are relevant to **odedef**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_pde_parab_1d_cd_ode (d03plc) you may allocate memory and initialize these pointers with various quantities for use by **odedef** when called from nag_pde_parab_1d_cd_ode (d03plc) (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

12: **nxi** – Integer *Input*

On entry: the number of ODE/PDE coupling points.

Constraints:

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

13: **xi**[dim] – const double *Input*

Note: the dimension, *dim*, of the array **xi** must be at least $\max(1, \textbf{nxi})$.

On entry: **xi**$[i-1]$, for $i = 1, 2, \ldots,$ **nxi**, must be set to the ODE/PDE coupling points.

Constraint: $\textbf{x}[0] \leq \textbf{xi}[0] < \textbf{xi}[1] < \cdots < \textbf{xi}[\textbf{nxi}-1] \leq \textbf{x}[\textbf{npts}-1]$.

14: **neqn** – Integer *Input*

On entry: the number of ODEs in the time direction.

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

15: **rtol**[dim] – const double *Input*

Note: the dimension, *dim*, of the array **rtol** must be at least

1 when **itol** $= 1$ or 2;
neqn when **itol** $= 3$ or 4.

On entry: the relative local error tolerance.

Constraint: **rtol**$[i-1] \geq 0.0$ for all relevant i.

16: **atol**[dim] – const double *Input*

Note: the dimension, *dim*, of the array **atol** must be at least

1 when **itol** $= 1$ or 3;
neqn when **itol** $= 2$ or 4.

On entry: the absolute local error tolerance.

Constraint: **atol**$[i-1] \geq 0.0$ for all relevant i.

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

17: **itol** – Integer *Input*

On entry: a value to indicate the form of the local error test. If e_i is the estimated local error for $\mathbf{u}[i-1]$, for $i = 1, 2, \ldots,$ **neqn**, and $\| \quad \|$ denotes the norm, then the error test to be satisfied is $\|e_i\| < 1.0$. **itol** indicates to nag_pde_parab_1d_cd_ode (d03plc) 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**):

itol	**rtol**	**atol**	w_i
1	scalar	scalar	$\mathbf{rtol}[0] \times \lvert\mathbf{u}[i-1]\rvert + \mathbf{atol}[0]$
2	scalar	vector	$\mathbf{rtol}[0] \times \lvert\mathbf{u}[i-1]\rvert + \mathbf{atol}[i-1]$
3	vector	scalar	$\mathbf{rtol}[i-1] \times \lvert\mathbf{u}[i-1]\rvert + \mathbf{atol}[0]$
4	vector	vector	$\mathbf{rtol}[i-1] \times \lvert\mathbf{u}[i-1]\rvert + \mathbf{atol}[i-1]$

Constraint: $1 \leq$ **itol** ≤ 4.

18: **norm** – Nag_NormType *Input*

On entry: the type of norm to be used.

norm $=$ Nag_OneNorm
Averaged L_1 norm.

norm $=$ Nag_TwoNorm
Averaged L_2 norm.

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

$$U_{\text{norm}} = \frac{1}{\mathbf{neqn}} \sum_{i=1}^{\mathbf{neqn}} \mathbf{u}[i-1]/w_i,$$

and for the averaged L_2 norm

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

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

Constraint: **norm** $=$ Nag_OneNorm or Nag_TwoNorm.

19: **laopt** – Nag_LinAlgOption *Input*

On entry: the type of matrix algebra required.

laopt = Nag_LinAlgFull
Full matrix methods to be used.

laopt = Nag_LinAlgBand
Banded matrix methods to be used.

laopt = Nag_LinAlgSparse
Sparse matrix methods to be used.

Constraint: **laopt** = Nag_LinAlgFull, Nag_LinAlgBand or Nag_LinAlgSparse.

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.

20: **algopt**[**30**] – const double *Input*

On entry: may be set to control various options available in the integrator. If you wish to employ all the default options, then **algopt**[0] 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[0]
Selects the ODE integration method to be used. If **algopt**[0] = 1.0, a BDF method is used and if **algopt**[0] = 2.0, a Theta method is used. The default is **algopt**[0] = 1.0.

If **algopt**[0] = 2.0, then **algopt**[$i-1$], for $i = 2, 3, 4$, are not used.

algopt[1]
Specifies the maximum order of the BDF integration formula to be used. **algopt**[1] may be 1.0, 2.0, 3.0, 4.0 or 5.0. The default value is **algopt**[1] = 5.0.

algopt[2]
Specifies what method is to be used to solve the system of nonlinear equations arising on each step of the BDF method. If **algopt**[2] = 1.0 a modified Newton iteration is used and if **algopt**[2] = 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**[2] = 1.0.

algopt[3]
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**[3] = 1.0, then the Petzold test is used. If **algopt**[3] = 2.0, then the Petzold test is not used. The default value is **algopt**[3] = 1.0.

If **algopt**[0] = 1.0, then **algopt**[$i-1$], for $i = 5, 6, 7$, are not used.

algopt[4]
Specifies the value of Theta to be used in the Theta integration method. $0.51 \leq$ **algopt**[4] ≤ 0.99. The default value is **algopt**[4] = 0.55.

algopt[5]
Specifies what method is to be used to solve the system of nonlinear equations arising on each step of the Theta method. If **algopt**[5] = 1.0, a modified Newton iteration is used and if **algopt**[5] = 2.0, a functional iteration method is used. The default value is **algopt**[5] = 1.0.

algopt[6]
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[6] $= 1.0$, then switching is allowed and if **algopt**[6] $= 2.0$, then switching is not allowed. The default value is **algopt**[6] $= 1.0$.

algopt[10]
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**[0] $\neq 0.0$, a value of 0.0 for **algopt**[10], say, should be specified even if **itask** subsequently specifies that t_{crit} will not be used.

algopt[11]
Specifies the minimum absolute step size to be allowed in the time integration. If this option is not required, **algopt**[11] should be set to 0.0.

algopt[12]
Specifies the maximum 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 initial step size to be attempted by the integrator. If **algopt**[13] $= 0.0$, then the initial step size is calculated internally.

algopt[14]
Specifies the maximum number of steps to be attempted by the integrator in any one call. If **algopt**[14] $= 0.0$, then no limit is imposed.

algopt[22]
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**[22] $= 1.0$, a modified Newton iteration is used and if **algopt**[22] $= 2.0$, functional iteration is used. The default value is **algopt**[22] $= 1.0$.

algopt[28] and **algopt**[29] are used only for the sparse matrix algebra option, i.e., **laopt** = Nag_LinAlgSparse.

algopt[28]
Governs the choice of pivots during the decomposition of the first Jacobian matrix. It should lie in the range $0.0 <$ **algopt**[28] < 1.0, with smaller values biasing the algorithm towards maintaining sparsity at the expense of numerical stability. If **algopt**[28] lies outside the range then the default value is used. If the functions regard the Jacobian matrix as numerically singular, then increasing **algopt**[28] towards 1.0 may help, but at the cost of increased fill-in. The default value is **algopt**[28] $= 0.1$.

algopt[29]
Used as the relative pivot threshold during subsequent Jacobian decompositions (see **algopt**[28]) below which an internal error is invoked. **algopt**[29] must be greater than zero, otherwise the default value is used. If **algopt**[29] 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**[28]). The default value is **algopt**[29] $= 0.0001$.

21: **rsave**[**lrsave**] – double *Communication Array*

If **ind** $= 0$, **rsave** need not be set on entry.

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

22: **lrsave** – Integer *Input*

On entry: the dimension of the array **rsave**. Its size depends on the type of matrix algebra selected.

If **laopt** = Nag_LinAlgFull, **lrsave** $\geq$ **neqn** $\times$ **neqn** + **neqn** + $nwkres + lenode$.

If **laopt** = Nag_LinAlgBand, **lrsave** $\geq (3 \times mlu + 1) \times$ **neqn** + $nwkres + lenode$.

If **laopt** = Nag_LinAlgSparse, **lrsave** $\geq 4 \times$ **neqn** $+ 11 \times$ **neqn**$/2 + 1 + nwkres + lenode$.

Where

mlu is the lower or upper half bandwidths such that
$mlu = 3 \times \mathbf{npde} - 1$, for PDE problems only (no coupled ODEs); or
$mlu = \mathbf{neqn} - 1$, for coupled PDE/ODE problems.

$$nwkres = \begin{cases} \mathbf{npde} \times (2 \times \mathbf{npts} + 6 \times \mathbf{nxi} + 3 \times \mathbf{npde} + 26) + \mathbf{nxi} + \mathbf{ncode} + 7 \times \mathbf{npts} + 2, & \text{when } \mathbf{ncode} > 0 \text{ and } \mathbf{nxi} > 0; \\ \mathbf{npde} \times (2 \times \mathbf{npts} + 3 \times \mathbf{npde} + 32) + \mathbf{ncode} + 7 \times \mathbf{npts} + 3, & \text{when } \mathbf{ncode} > 0 \text{ and } \mathbf{nxi} = 0; \\ \mathbf{npde} \times (2 \times \mathbf{npts} + 3 \times \mathbf{npde} + 32) + 7 \times \mathbf{npts} + 4, & \text{when } \mathbf{ncode} = 0. \end{cases}$$

$$lenode = \begin{cases} (6 + \text{int}(\mathbf{algopt}[1])) \times \mathbf{neqn} + 50, & \text{when the BDF method is used; or} \\ 9 \times \mathbf{neqn} + 50, & \text{when the Theta method is used.} \end{cases}$$

Note: when **laopt** = Nag_LinAlgSparse, the value of **lrsave** may be too small when supplied to the integrator. An estimate of the minimum size of **lrsave** is printed on the current error message unit if **itrace** > 0 and the function returns with **fail.code** = NE_INT_2.

23: **isave**[**lisave**] – Integer *Communication Array*

If **ind** $= 0$, **isave** need not be set.

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[0]
Contains the number of steps taken in time.

isave[1]
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[2]
Contains the number of Jacobian evaluations performed by the time integrator.

isave[3]
Contains the order of the BDF method last used in the time integration, if applicable. When the Theta method is used, **isave**[3] contains no useful information.

isave[4]
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.

24: **lisave** – Integer *Input*

On entry: the dimension of the array **isave**. Its size depends on the type of matrix algebra selected:

if **laopt** = Nag_LinAlgFull, **lisave** ≥ 24;

if **laopt** = Nag_LinAlgBand, **lisave** $\geq$ **neqn** $+ 24$;

if **laopt** = Nag_LinAlgSparse, **lisave** $\geq 25 \times$ **neqn** $+ 24$.

Note: when using the sparse option, the value of **lisave** may be too small when supplied to the integrator. An estimate of the minimum size of **lisave** is printed if **itrace** > 0 and the function returns with **fail.code** = NE_INT_2.

25: **itask** – Integer *Input*

On entry: 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 $t = t_{\text{crit}}$ 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.

26: **itrace** – Integer *Input*

On entry: the level of trace information required from nag_pde_parab_1d_cd_ode (d03plc) 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 .

itrace > 0
Output from the underlying ODE solver is printed . 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.

27: **outfile** – const char * *Input*

On entry: the name of a file to which diagnostic output will be directed. If **outfile** is **NULL** the diagnostic output will be directed to standard output.

28: **ind** – Integer * *Input/Output*

On entry: 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** and **fail** should be reset between calls to nag_pde_parab_1d_cd_ode (d03plc).

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

On exit: **ind** $= 1$.

29: **comm** – Nag_Comm *

The NAG communication argument (see Section 2.3.1.1 in How to Use the NAG Library and its Documentation).

30: **saved** – Nag_D03_Save * *Communication Structure*

saved must remain unchanged following a previous call to a Chapter d03 function and prior to any subsequent call to a Chapter d03 function.

31: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_ACC_IN_DOUBT

Integration completed, but small changes in **atol** or **rtol** are unlikely to result in a changed solution.

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 2.3.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

NE_FAILED_DERIV

In setting up the ODE system an internal auxiliary was unable to initialize the derivative. This could be due to your setting **ires** = 3 in **pdedef**, **numflx**, **bndary** or **odedef**.

NE_FAILED_START

atol and **rtol** were too small to start integration.

NE_FAILED_STEP

Error during Jacobian formulation for ODE system. Increase **itrace** for further details.

Repeated errors in an attempted step of underlying ODE solver. Integration was successful as far as **ts**: **ts** = ⟨*value*⟩.

Underlying ODE solver cannot make further progress from the point **ts** with the supplied values of **atol** and **rtol**. **ts** = ⟨*value*⟩.

NE_INT

ires set to an invalid value in a call to user-supplied functions **pdedef**, **numflx**, **bndary** or **odedef**.

On entry, **ind** = ⟨*value*⟩.
Constraint: **ind** = 0 or 1.

On entry, **itask** = ⟨*value*⟩.
Constraint: **itask** = 1, 2, 3, 4 or 5.

On entry, **itol** = ⟨*value*⟩.
Constraint: **itol** = 1, 2, 3 or 4.

On entry, **ncode** = ⟨*value*⟩.
Constraint: **ncode** ≥ 0.

On entry, **npde** = ⟨*value*⟩.
Constraint: **npde** ≥ 1.

On entry, **npts** = ⟨*value*⟩.
Constraint: **npts** ≥ 3.

NE_INT_2

On entry, corresponding elements **atol**$[I-1]$ and **rtol**$[J-1]$ are both zero: $I = \langle value \rangle$ and $J = \langle value \rangle$.

On entry, **lisave** is too small: **lisave** $= \langle value \rangle$. Minimum possible dimension: $\langle value \rangle$.

On entry, **lrsave** is too small: **lrsave** $= \langle value \rangle$. Minimum possible dimension: $\langle value \rangle$.

On entry, **ncode** $= \langle value \rangle$ and **nxi** $= \langle value \rangle$.
Constraint: **nxi** $= 0$ when **ncode** $= 0$.

On entry, **ncode** $= \langle value \rangle$ and **nxi** $= \langle value \rangle$.
Constraint: **nxi** ≥ 0 when **ncode** > 0.

When using the sparse option **lisave** or **lrsave** is too small: **lisave** $= \langle value \rangle$, **lrsave** $= \langle value \rangle$.

NE_INT_4

On entry, **neqn** $= \langle value \rangle$, **npde** $= \langle value \rangle$, **npts** $= \langle value \rangle$ and **ncode** $= \langle value \rangle$.
Constraint: **neqn** $=$ **npde** $\times$ **npts** $+$ **ncode**.

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 2.7.6 in How to Use the NAG Library and its Documentation for further information.

Serious error in internal call to an auxiliary. Increase **itrace** for further details.

NE_ITER_FAIL

In solving ODE system, the maximum number of steps **algopt**$[14]$ has been exceeded. **algopt**$[14] = \langle value \rangle$.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 2.7.5 in How to Use the NAG Library and its Documentation for further information.

NE_NOT_CLOSE_FILE

Cannot close file $\langle value \rangle$.

NE_NOT_STRICTLY_INCREASING

On entry, $I = \langle value \rangle$, **xi**$[I] = \langle value \rangle$ and **xi**$[I-1] = \langle value \rangle$.
Constraint: **xi**$[I] >$ **xi**$[I-1]$.

On entry, mesh points **x** badly ordered: $I = \langle value \rangle$, **x**$[I-1] = \langle value \rangle$, $J = \langle value \rangle$ and **x**$[J-1] = \langle value \rangle$.

NE_NOT_WRITE_FILE

Cannot open file $\langle value \rangle$ for writing.

NE_REAL_2

On entry, at least one point in **xi** lies outside $[\mathbf{x}[0], \mathbf{x}[\mathbf{npts}-1]]$: **x**$[0] = \langle value \rangle$ and **x**$[\mathbf{npts}-1] = \langle value \rangle$.

On entry, **tout** $= \langle value \rangle$ and **ts** $= \langle value \rangle$.
Constraint: **tout** $>$ **ts**.

On entry, **tout** $-$ **ts** is too small: **tout** $= \langle value \rangle$ and **ts** $= \langle value \rangle$.

NE_REAL_ARRAY

On entry, $I = \langle value \rangle$ and **atol**$[I-1] = \langle value \rangle$.
Constraint: **atol**$[I-1] \geq 0.0$.

On entry, $I = \langle value \rangle$ and **rtol**$[I-1] = \langle value \rangle$.
Constraint: **rtol**$[I-1] \geq 0.0$.

NE_SING_JAC

Singular Jacobian of ODE system. Check problem formulation.

NE_TIME_DERIV_DEP

The functions P, D, or C appear to depend on time derivatives.

NE_USER_STOP

In evaluating residual of ODE system, **ires** $= 2$ has been set in user-supplied functions **pdedef**, **numflx**, **bndary** or **odedef**. Integration is successful as far as **ts**: **ts** $= \langle value \rangle$.

NE_ZERO_WTS

Zero error weights encountered during time integration.

7 Accuracy

nag_pde_parab_1d_cd_ode (d03plc) 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 Parallelism and Performance

nag_pde_parab_1d_cd_ode (d03plc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_pde_parab_1d_cd_ode (d03plc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

Please consult the x06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

nag_pde_parab_1d_cd_ode (d03plc) 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 schemes 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**[12]. 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 and on the accuracy requested. For a given system and a fixed accuracy it is approximately proportional to **neqn**.

10 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 first-order system with coupled ODEs arising from the use of the characteristic equations for the numerical boundary conditions.

The PDEs are

$$\frac{\partial U_1}{\partial t} + \frac{\partial U_1}{\partial x} + 2\frac{\partial U_2}{\partial x} = 0,$$

$$\frac{\partial U_2}{\partial t} + 2\frac{\partial U_1}{\partial x} + \frac{\partial U_2}{\partial x} = 0,$$

for $x \in [0,1]$ and $t \geq 0$.

The PDEs have an exact solution given by

$$U_1(x,t) = f(x-3t) + g(x+t), \quad U_2(x,t) = f(x-3t) - g(x+t),$$

where $f(z) = \exp(\pi z)\sin(2\pi z)$, $g(z) = \exp(-2\pi z)\cos(2\pi z)$.

The initial conditions are given by the exact solution.

The characteristic variables are $W_1 = U_1 - U_2$ and $W_2 = U_1 + U_2$, corresponding to the characteristics given by $dx/dt = -1$ and $dx/dt = 3$ respectively. Hence we require a physical boundary condition for W_2 at the left-hand boundary and for W_1 at the right-hand boundary (corresponding to the incoming characteristics), and a numerical boundary condition for W_1 at the left-hand boundary and for W_2 at the right-hand boundary (outgoing characteristics).

The physical boundary conditions are obtained from the exact solution, and the numerical boundary conditions are supplied in the form of the characteristic equations for the outgoing characteristics, that is

$$\frac{\partial W_1}{\partial t} - \frac{\partial W_1}{\partial x} = 0$$

at the left-hand boundary, and

$$\frac{\partial W_2}{\partial t} + 3\frac{\partial W_2}{\partial x} = 0$$

at the right-hand boundary.

In order to specify these boundary conditions, two ODE variables V_1 and V_2 are introduced, defined by

$$V_1(t) = W_1(0,t) = U_1(0,t) - U_2(0,t),$$
$$V_2(t) = W_2(1,t) = U_1(1,t) + U_2(1,t).$$

The coupling points are therefore at $x = 0$ and $x = 1$.

The numerical boundary conditions are now

$$\dot{V}_1 - \frac{\partial W_1}{\partial x} = 0$$

at the left-hand boundary, and

$$\dot{V}_2 + 3\frac{\partial W_2}{\partial x} = 0$$

at the right-hand boundary.

The spatial derivatives are evaluated at the appropriate boundary points in **bndary** using one-sided differences (into the domain and therefore consistent with the characteristic directions).

The numerical flux is calculated using Roe's approximate Riemann solver (see Section 3 for details), giving

$$\hat{F} = \tfrac{1}{2}\begin{bmatrix} 3U_{1L} - U_{1R} + 3U_{2L} + U_{2R} \\ 3U_{1L} + U_{1R} + 3U_{2L} - U_{2R} \end{bmatrix}.$$

Example 2 (ex2)

This example is the standard shock-tube test problem proposed by Sod (1978) for the Euler equations of gas dynamics. The problem models the flow of a gas in a long tube following the sudden breakdown of a diaphragm separating two initial gas states at different pressures and densities. There is an exact solution to this problem which is not included explicitly as the calculation is quite lengthy. The PDEs are

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \frac{\partial m}{\partial x} &= 0, \\ \frac{\partial m}{\partial t} + \frac{\partial}{\partial x}\left(\frac{m^2}{\rho} + (\gamma - 1)\left(e - \frac{m^2}{2\rho}\right)\right) &= 0, \\ \frac{\partial e}{\partial t} + \frac{\partial}{\partial x}\left(\frac{me}{\rho} + \frac{m}{\rho}(\gamma - 1)\left(e - \frac{m^2}{2\rho}\right)\right) &= 0, \end{aligned}$$

where ρ is the density; m is the momentum, such that $m = \rho u$, where u is the velocity; e is the specific energy; and γ is the (constant) ratio of specific heats. The pressure p is given by

$$p = (\gamma - 1)\left(e - \frac{\rho u^2}{2}\right).$$

The solution domain is $0 \leq x \leq 1$ for $0 < t \leq 0.2$, with the initial discontinuity at $x = 0.5$, and initial conditions

$$\begin{array}{llll} \rho(x,0) = 1, & m(x,0) = 0, & e(x,0) = 2.5, & \text{for } x < 0.5, \\ \rho(x,0) = 0.125, & m(x,0) = 0, & e(x,0) = 0.25, & \text{for } x > 0.5. \end{array}$$

The solution is uniform and constant at both boundaries for the spatial domain and time of integration stated, and hence the physical and numerical boundary conditions are indistinguishable and are both given by the initial conditions above. The evaluation of the numerical flux for the Euler equations is not trivial; the Roe algorithm given in Section 3 cannot be used directly as the Jacobian is nonlinear. However, an algorithm is available using the argument-vector method (see Roe (1981)), and this is provided in the utility function nag_pde_parab_1d_euler_roe (d03puc). An alternative Approxiate Riemann Solver using Osher's scheme is provided in nag_pde_parab_1d_euler_osher (d03pvc). Either nag_pde_parab_1d_euler_roe (d03puc) or nag_pde_parab_1d_euler_osher (d03pvc) can be called from **numflx**.

10.1 Program Text

```
/* nag_pde_parab_1d_cd_ode (d03plc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */
```

```
#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagd03.h>
#include <nagx01.h>
#include <math.h>

/* Structure to communicate with user-supplied function arguments */

struct user
{
  double elo, ero, gamma, rlo, rro;
};

#ifdef __cplusplus
extern "C"
{
#endif
  static void NAG_CALL pdedef(Integer, double, double, const double[],
                              const double[], Integer, const double[],
                              const double[], double[], double[], double[],
                              double[], Integer *, Nag_Comm *);

  static void NAG_CALL bndry1(Integer, Integer, double, const double[],
                              const double[], Integer, const double[],
                              const double[], Integer, double[], Integer *,
                              Nag_Comm *);

  static void NAG_CALL bndry2(Integer, Integer, double, const double[],
                              const double[], Integer, const double[],
                              const double[], Integer, double[], Integer *,
                              Nag_Comm *);

  static void NAG_CALL nmflx1(Integer, double, double, Integer,
                              const double[], const double[], const double[],
                              double[], Integer *, Nag_Comm *,
                              Nag_D03_Save *);

  static void NAG_CALL nmflx2(Integer, double, double, Integer,
                              const double[], const double[], const double[],
                              double[], Integer *, Nag_Comm *,
                              Nag_D03_Save *);

  static void NAG_CALL odedef(Integer, double, Integer, const double[],
                              const double[], Integer, const double[],
                              const double[], const double[], const double[],
                              double[], Integer *, Nag_Comm *);
#ifdef __cplusplus
}
#endif

static void init1(double, double *, Integer, double *, Integer, Integer);
static void init2(Integer, Integer, double *, double *, Nag_Comm *);
static void exact(double, double *, Integer, const double *, Integer);
static int ex1(void);
static int ex2(void);

#define P(I, J)    p[npde*((J) -1)+(I) -1]
#define UCP(I, J)  ucp[npde*((J) -1)+(I) -1]
#define U(I, J)    u[npde*((J) -1)+(I) -1]
#define UE(I, J)    ue[npde*((J) -1)+(I) -1]

int main(void)
{
  Integer exit_status_ex1 = 0;
  Integer exit_status_ex2 = 0;

  printf("nag_pde_parab_1d_cd_ode (d03plc) Example Program Results\n");
  exit_status_ex1 = ex1();
  exit_status_ex2 = ex2();
```

```
  return (exit_status_ex1 == 0 && exit_status_ex2 == 0) ? 0 : 1;
}

int ex1(void)
{
  /* Constants */
  const Integer npde = 2, npts = 201, ncode = 2, nxi = 2;
  const Integer neqn = npde*npts + ncode, lrsave = 100000;
  const Integer lisave = 100000;
  static double ruser1[4] = { -1.0, -1.0, -1.0, -1.0 };

  /* Scalars */
  double        tout, ts, errmax, lerr, lwgt;
  Integer       exit_status = 0, i, ind, itask, itol, itrace, j;

  /* Arrays */
  double        *algopt = 0, *atol = 0, *rsave = 0, *rtol = 0, *u = 0, *ue = 0;
  double        *x = 0, *xi = 0;
  Integer       *isave = 0;

  /* Nag Types */
  NagError      fail;
  Nag_Comm      comm;
  Nag_D03_Save  saved;

  INIT_FAIL(fail);

  /* For communication with user-supplied functions: */
  comm.user = ruser1;

  printf("\n\nExample 1\n\n");

  /* Allocate memory */

  if (!(algopt = NAG_ALLOC(30, double)) ||
      !(atol = NAG_ALLOC(1, double)) ||
      !(rsave = NAG_ALLOC(lrsave, double)) ||
      !(rtol = NAG_ALLOC(1, double)) ||
      !(u = NAG_ALLOC(neqn, double)) ||
      !(ue = NAG_ALLOC(npde * npts, double)) ||
      !(x = NAG_ALLOC(npts, double)) ||
      !(xi = NAG_ALLOC(nxi, double)) ||
      !(isave = NAG_ALLOC(lisave, Integer)))
  {
    printf("Allocation failure\n");
    exit_status = 1;
    goto END;
  }

  itrace = 0;
  itol = 1;
  atol[0] = 1e-5;
  rtol[0] = 2.5e-4;

  printf(" Method parameters:\n");
  printf("  Number of mesh points used = %4" NAG_IFMT "\n", npts);
  printf("  Relative tolerance used    = %12.3e\n", rtol[0]);
  printf("  Absolute tolerance used    = %12.3e\n\n", atol[0]);

  /* Initialize mesh */

  for (i = 0; i < npts; ++i)
    x[i] = i / (npts - 1.0);
  xi[0] = 0.0;
  xi[1] = 1.0;

  /* Set initial values */

  ts = 0.0;
  init1(ts, u, npde, x, npts, ncode);
```

```
  ind = 0;
  itask = 1;

  for (i = 0; i < 30; ++i)
    algopt[i] = 0.0;

  /* BDF integration */

  algopt[0] = 1.0;

  /* Sparse matrix algebra parameters */

  algopt[28] = 0.1;
  algopt[29] = 1.1;

  tout = 0.5;
  /* nag_pde_parab_1d_cd_ode (d03plc).
   * General system of convection-diffusion PDEs with source
   * terms in conservative form, coupled DAEs, method of
   * lines, upwind scheme using numerical flux function based
   * on Riemann solver, one space variable
   */
  nag_pde_parab_1d_cd_ode(npde, &ts, tout, pdedef, nmflx1, bndry1, u, npts, x,
                          ncode, odedef, nxi, xi, neqn, rtol, atol, itol,
                          Nag_OneNorm, Nag_LinAlgSparse, algopt, rsave,
                          lrsave, isave, lisave, itask, itrace, 0, &ind,
                          &comm, &saved, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_pde_parab_1d_cd_ode (d03plc).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }

  /* Check against exact solution */
  exact(tout, ue, npde, &x[0], npts);
  errmax = 0.0;
  for (i=1; i<npts; i++) {
    lerr = 0.0;
    for (j=0; j<npde; j++) {
      lwgt = rtol[0]*fabs(ue[i*npde+j]) + atol[0];
      lerr += fabs(u[i*npde+j]-ue[i*npde+j])/lwgt;
    }
    lerr = lerr/(double) npde;
    errmax = MAX(errmax,lerr);
  }
  errmax = MAX(100.0*round(errmax/100.0),50.0);
  printf("\n Integration Results:\n");
  printf("  Global error is less than %4" NAG_IFMT ""
         " times the local error tolerance.\n", (Integer)(errmax));

  /* Print integration statistics (reasonably rounded) */
  printf("\n Integration Statistics:\n");
  printf("  %-30s (nearest %3d) = %6" NAG_IFMT "\n",
         "Number of time steps", 50, 50*((isave[0]+25)/50));
  printf("  %-30s (nearest %3d) = %6" NAG_IFMT "\n",
         "Number of function evaluations", 100, 100*((isave[1]+50)/100));
  printf("  %-30s (nearest %3d) = %6" NAG_IFMT "\n",
         "Number of Jacobian evaluations", 20, 20*((isave[2]+10)/20));
  printf("  %-30s (nearest %3d) = %6" NAG_IFMT "\n",
         "Number of iterations", 100, 100*((isave[4]+50)/100));
END:
  NAG_FREE(algopt);
  NAG_FREE(atol);
  NAG_FREE(rsave);
  NAG_FREE(rtol);
  NAG_FREE(u);
  NAG_FREE(ue);
  NAG_FREE(x);
  NAG_FREE(xi);
  NAG_FREE(isave);
```

```
  return exit_status;
}

static void NAG_CALL pdedef(Integer npde, double t, double x,
                            const double u[], const double ux[],
                            Integer ncode, const double v[],
                            const double vdot[], double p[], double c[],
                            double d[], double s[], Integer *ires,
                            Nag_Comm *comm)
{
  Integer i, j;

  if (comm->user[2] == -1.0) {
    /* printf("(User-supplied callback pdedef, first invocation.)\n"); */
    comm->user[2] = 0.0;
  }
  for (i = 1; i <= npde; ++i) {
    c[i - 1] = 1.0;
    d[i - 1] = 0.0;
    s[i - 1] = 0.0;
    for (j = 1; j <= npde; ++j) {
      if (i == j) {
        P(i, j) = 1.0;
      }
      else {
        P(i, j) = 0.0;
      }
    }
  }
  return;
}

static void NAG_CALL bndry1(Integer npde, Integer npts, double t,
                            const double x[], const double u[], Integer ncode,
                            const double v[], const double vdot[],
                            Integer ibnd, double g[], Integer *ires,
                            Nag_Comm *comm)
{
  double dudx;
  double *ue = 0;

  if (comm->user[0] == -1.0) {
    /* printf("(User-supplied callback bndry1, first invocation.)\n"); */
    comm->user[0] = 0.0;
  }

  /* Allocate memory */

  if (!(ue = NAG_ALLOC(npde, double)))
  {
    printf("Allocation failure\n");
    goto END;
  }

  if (ibnd == 0) {
    exact(t, ue, npde, &x[0], 1);
    g[0] = U(1, 1) + U(2, 1) - UE(1, 1) - UE(2, 1);
    dudx = (U(1, 2) - U(2, 2) - U(1, 1) + U(2, 1)) / (x[1] - x[0]);
    g[1] = vdot[0] - dudx;
  }
  else {
    exact(t, ue, npde, &x[npts - 1], 1);
    g[0] = U(1, npts) - U(2, npts) - UE(1, 1) + UE(2, 1);
    dudx = (U(1, npts) + U(2, npts) - U(1, npts - 1) - U(2, npts - 1)) /
           (x[npts - 1] - x[npts - 2]);
    g[1] = vdot[1] + 3.0 * dudx;
  }
END:
  NAG_FREE(ue);
```

```
  return;
}

static void NAG_CALL nmflx1(Integer npde, double t, double x, Integer ncode,
                            const double v[], const double uleft[],
                            const double uright[], double flux[],
                            Integer *ires, Nag_Comm *comm,
                            Nag_D03_Save *saved)
{
  if (comm->user[1] == -1.0) {
    /* printf("(User-supplied callback nmflx1, first invocation.)\n"); */
    comm->user[1] = 0.0;
  }
  flux[0] = 0.5 * (3.0 * uleft[0] - uright[0] + 3.0 * uleft[1] + uright[1]);
  flux[1] = 0.5 * (3.0 * uleft[0] + uright[0] + 3.0 * uleft[1] - uright[1]);
  return;
}

static void NAG_CALL odedef(Integer npde, double t, Integer ncode,
                            const double v[], const double vdot[],
                            Integer nxi, const double xi[],
                            const double ucp[], const double ucpx[],
                            const double ucpt[], double r[], Integer *ires,
                            Nag_Comm *comm)
{
  if (comm->user[3] == -1.0) {
    /* printf("(User-supplied callback odedef, first invocation.)\n"); */
    comm->user[3] = 0.0;
  }
  if (*ires == -1) {
    r[0] = 0.0;
    r[1] = 0.0;
  }
  else {
    r[0] = v[0] - UCP(1, 1) + UCP(2, 1);
    r[1] = v[1] - UCP(1, 2) - UCP(2, 2);
  }
  return;
}

static void exact(double t, double *u, Integer npde, const double *x,
                  Integer npts)
{
  /* Exact solution (for comparison and b.c. purposes) */

  double f, g, x1, x2;
  Integer i;

  for (i = 0; i < npts; ++i) {
    x1 = x[i] - 3.0 * t;
    x2 = x[i] + t;
    f = exp(nag_pi * x1) * sin(2.0 * nag_pi * x1);
    g = exp(-2.0 * nag_pi * x2) * cos(2.0 * nag_pi * x2);
    u[npde*i]   = f + g;
    u[npde*i+1] = f - g;
  }
  return;
}

static void init1(double t, double *u, Integer npde, double *x, Integer npts,
                  Integer ncode)
{
  /* Initial solution */

  double f, g, x1, x2;
  Integer i, neqn;

  neqn = npde * npts + ncode;
  for (i = 0; i < npts; ++i) {
    x1 = x[i] - 3.0 * t;
    x2 = x[i] + t;
```

```
    f = exp(nag_pi * x1) * sin(2.0 * nag_pi * x1);
    g = exp(-2.0 * nag_pi * x2) * cos(2.0 * nag_pi * x2);
    u[npde*i]   = f + g;
    u[npde*i+1] = f - g;
  }
  u[neqn - 2] = u[0] - u[1];
  u[neqn - 1] = u[neqn - 3] + u[neqn - 4];

  return;
}

int ex2(void)
{
  const Integer npde = 3, npts = 141, ncode = 0, nxi = 0;
  const Integer neqn = npde * npts + ncode, lisave = neqn + 24;
  const Integer lrsave = 16392;
  static double ruser[2] = { -1.0, -1.0 };
  double d, p, tout, ts, v;
  Integer exit_status = 0, i, ind, it, itask, itol, itrace;
  double *algopt = 0, *atol = 0, *rsave = 0, *rtol = 0, *u = 0;
  double *x = 0, *xi = 0;
  Integer *isave = 0;
  NagError fail;
  Nag_Comm comm;
  Nag_D03_Save saved;
  struct user data;

  INIT_FAIL(fail);

  /* For communication with user-supplied functions: */
  comm.user = ruser;

  printf("\n\nExample 2\n\n");

  /* Allocate memory */

  if (!(algopt = NAG_ALLOC(30, double)) ||
      !(atol = NAG_ALLOC(1, double)) ||
      !(rsave = NAG_ALLOC(lrsave, double)) ||
      !(rtol = NAG_ALLOC(1, double)) ||
      !(u = NAG_ALLOC(npde * npts, double)) ||
      !(x = NAG_ALLOC(npts, double)) ||
      !(xi = NAG_ALLOC(1, double)) || !(isave = NAG_ALLOC(447, Integer)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

  /* Problem parameters */

  data.elo = 2.5;
  data.ero = 0.25;
  data.gamma = 1.4;
  data.rlo = 1.0;
  data.rro = 0.125;
  comm.p = (Pointer) &data;

  itrace = 0;
  itol = 1;
  atol[0] = 0.005;
  rtol[0] = 5e-4;

  printf(" Problem parameters and initial conditions:\n");
  printf("  gamma          = %5.3f\n", data.gamma);
  printf("      e(x<0.5,0) = %5.3f", data.elo);
  printf("      e(x>0.5,0) = %5.3f\n", data.ero);
  printf("    rho(x<0.5,0) = %5.3f", data.rlo);
  printf("    rho(x>0.5,0) = %5.3f\n\n", data.rro);
  printf(" Method parameters:\n");
  printf("  Number of mesh points used = %4" NAG_IFMT "\n", npts);
```

```
  printf("  Relative tolerance used    = %12.3e\n", rtol[0]);
  printf("  Absolute tolerance used    = %12.3e\n\n", atol[0]);

  /* Initialize mesh */

  for (i = 0; i < npts; ++i)
    x[i] = i / (npts - 1.0);

  /* Initial values of variables */

  init2(npde, npts, x, u, &comm);

  xi[0] = 0.0;
  ind = 0;
  itask = 1;
  for (i = 0; i < 30; ++i)
    algopt[i] = 0.0;

  /* Theta integration */

  algopt[0] = 2.0;
  algopt[5] = 2.0;
  algopt[6] = 2.0;

  /* Max. time step */

  algopt[12] = 0.005;
  ts = 0.0;

  printf(" Solution\n%4s%9s%9s%9s%9s\n", "t", "x", "d", "v", "p");
  for (it = 0; it < 2; ++it) {
    tout = 0.1 * (it + 1);

    /* nag_pde_parab_1d_cd_ode (d03plc), see above. */
    nag_pde_parab_1d_cd_ode(npde, &ts, tout, NULLFN, nmflx2, bndry2, u, npts,
                            x, ncode, NULLFN, nxi, xi, neqn, rtol, atol,
                            itol, Nag_TwoNorm, Nag_LinAlgBand, algopt, rsave,
                            lrsave, isave, lisave, itask, itrace, 0,
                            &ind, &comm, &saved, &fail);

    if (fail.code != NE_NOERROR) {
      printf("Error from nag_pde_parab_1d_cd_ode (d03plc).\n%s\n",
             fail.message);
      exit_status = 1;
      goto END;
    }

    /* Calculate density, velocity and pressure */

    for (i = 1; i <= npts; i += 14) {
      d = U(1, i);
      v = U(2, i) / d;
      p = d*(data.gamma - 1.0)*(U(3, i)/d - 0.5*v*v);
      if (i==1) {
        printf("%6.3f  %7.4f  %7.4f  %7.4f  %7.4f\n",
               ts, x[i-1], d, v, p);
      } else {
        printf("%6s  %7.4f  %7.4f  %7.4f  %7.4f\n",
               "", x[i-1], d, v, p);
      }
    }
    printf("\n");
  }

  /* Print integration statistics (reasonably rounded) */
  printf("\n Integration Statistics:\n");
  printf("  %-30s (nearest %3" NAG_IFMT ") = %6" NAG_IFMT "\n",
         "Number of time steps", 50, 50*((isave[0]+25)/50));
  printf("  %-30s (nearest %3" NAG_IFMT ") = %6" NAG_IFMT "\n",
         "Number of function evaluations", 50, 50*((isave[1]+25)/50));
  printf("  %-30s (nearest %3" NAG_IFMT ") = %6" NAG_IFMT "\n",
```

```
          "Number of Jacobian evaluations", 1, isave[2]);
  printf("  %-30s (nearest %3" NAG_IFMT ") = %6" NAG_IFMT "\n",
          "Number of iterations", 1, isave[4]);

END:
  NAG_FREE(algopt);
  NAG_FREE(atol);
  NAG_FREE(rsave);
  NAG_FREE(rtol);
  NAG_FREE(u);
  NAG_FREE(x);
  NAG_FREE(xi);
  NAG_FREE(isave);

  return exit_status;
}

static void init2(Integer npde, Integer npts, double *x, double *u,
                  Nag_Comm *comm)
{
  Integer i, j;
  struct user *data = (struct user *) comm->p;

  j = 0;
  for (i = 0; i < npts; ++i) {
    if (x[i] < 0.5) {
      u[j] = data->rlo;
      u[j + 1] = 0.0;
      u[j + 2] = data->elo;
    }
    else if (x[i] == 0.5) {
      u[j] = 0.5 * (data->rlo + data->rro);
      u[j + 1] = 0.0;
      u[j + 2] = 0.5 * (data->elo + data->ero);
    }
    else {
      u[j] = data->rro;
      u[j + 1] = 0.0;
      u[j + 2] = data->ero;
    }
    j += 3;
  }
  return;
}

static void NAG_CALL bndry2(Integer npde, Integer npts, double t,
                            const double x[], const double u[], Integer ncode,
                            const double v[], const double vdot[],
                            Integer ibnd, double g[], Integer *ires,
                            Nag_Comm *comm)
{
  struct user *data = (struct user *) comm->p;

  if (comm->user[0] == -1.0) {
    /* printf("(User-supplied callback bndry2, first invocation.)\n"); */
    comm->user[0] = 0.0;
  }
  if (ibnd == 0) {
    g[0] = U(1, 1) - data->rlo;
    g[1] = U(2, 1);
    g[2] = U(3, 1) - data->elo;
  }
  else {
    g[0] = U(1, npts) - data->rro;
    g[1] = U(2, npts);
    g[2] = U(3, npts) - data->ero;
  }
  return;
}

static void NAG_CALL nmflx2(Integer npde, double t, double x, Integer ncode,
```

```
                              const double v[], const double uleft[],
                              const double uright[], double flux[],
                              Integer *ires, Nag_Comm *comm,
                              Nag_D03_Save *saved)
{
  char solver;
  NagError fail;
  struct user *data = (struct user *) comm->p;

  if (comm->user[1] == -1.0) {
    /* printf("(User-supplied callback nmflx2, first invocation.)\n"); */
    comm->user[1] = 0.0;
  }

  INIT_FAIL(fail);

  solver = 'R';
  if (solver == 'R') {
    /* ROE SCHEME */

    /* nag_pde_parab_1d_euler_roe (d03puc).
     * Roe's approximate Riemann solver for Euler equations in
     * conservative form, for use with nag_pde_parab_1d_cd
     * (d03pfc), nag_pde_parab_1d_cd_ode (d03plc) and
     * nag_pde_parab_1d_cd_ode_remesh (d03psc)
     */
    nag_pde_parab_1d_euler_roe(uleft, uright, data->gamma, flux, saved,
                               &fail);
  }
  else {
    /* OSHER SCHEME */

    /* nag_pde_parab_1d_euler_osher (d03pvc).
     * Osher's approximate Riemann solver for Euler equations in
     * conservative form, for use with nag_pde_parab_1d_cd
     * (d03pfc), nag_pde_parab_1d_cd_ode (d03plc) and
     * nag_pde_parab_1d_cd_ode_remesh (d03psc)
     */
    nag_pde_parab_1d_euler_osher(uleft, uright, data->gamma,
                                 Nag_OsherPhysical, flux, saved, &fail);
  }

  if (fail.code != NE_NOERROR) {
    printf("Error from nag_pde_parab_1d_euler_osher (d03pvc).\n%s\n",
           fail.message);
  }

  return;
}
```

10.2 Program Data

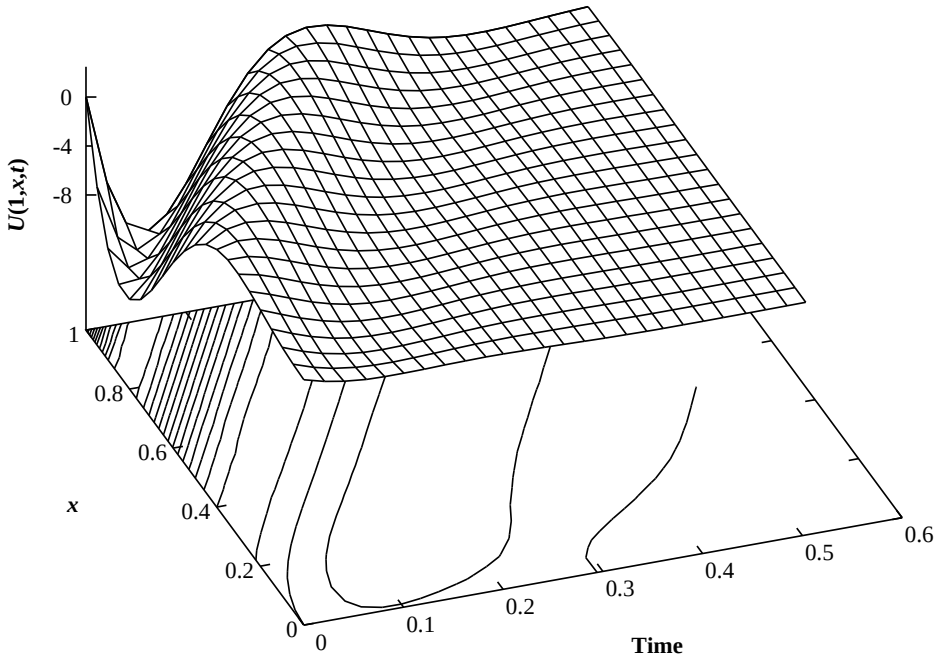
None.

10.3 Program Results

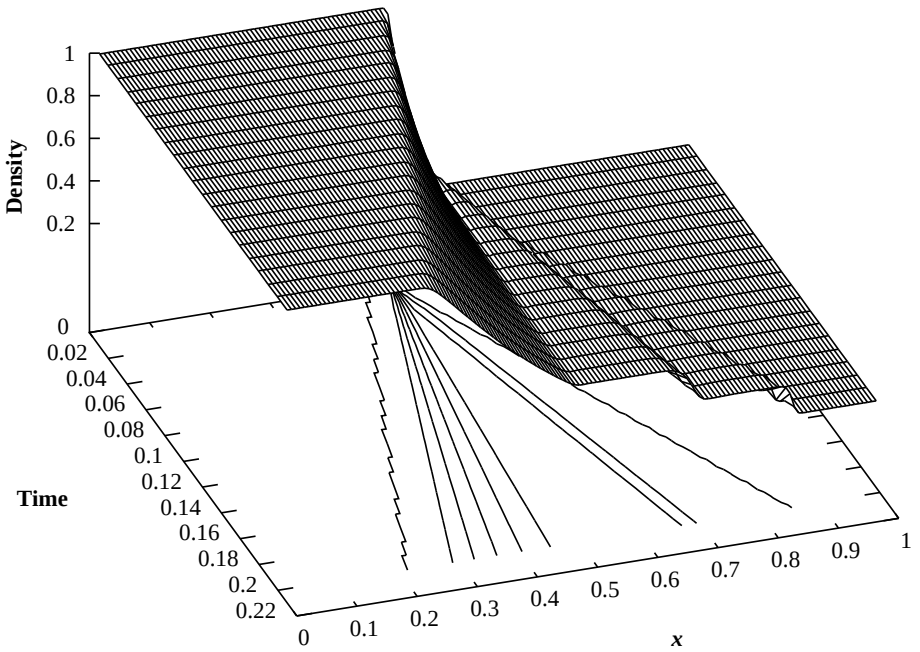
```
nag_pde_parab_1d_cd_ode (d03plc) Example Program Results


Example 1

 Method parameters:
  Number of mesh points used =  201
  Relative tolerance used    =    2.500e-04
  Absolute tolerance used    =    1.000e-05


 Integration Results:
  Global error is less than  100 times the local error tolerance.
```

```
 Integration Statistics:
  Number of time steps           (nearest  50) =    150
  Number of function evaluations (nearest 100) =   1400
  Number of Jacobian evaluations (nearest  20) =     20
  Number of iterations           (nearest 100) =    400


Example 2

 Problem parameters and initial conditions:
  gamma          = 1.400
      e(x<0.5,0) = 2.500      e(x>0.5,0) = 0.250
    rho(x<0.5,0) = 1.000    rho(x>0.5,0) = 0.125

 Method parameters:
  Number of mesh points used =  141
  Relative tolerance used    =    5.000e-04
  Absolute tolerance used    =    5.000e-03

 Solution
   t        x        d        v        p
 0.100   0.0000   1.0000   0.0000   1.0000
         0.1000   1.0000  -0.0000   1.0000
         0.2000   1.0000  -0.0000   1.0000
         0.3000   1.0000  -0.0000   1.0000
         0.4000   0.8668   0.1665   0.8188
         0.5000   0.4299   0.9182   0.3071
         0.6000   0.2969   0.9274   0.3028
         0.7000   0.1250   0.0000   0.1000
         0.8000   0.1250  -0.0000   0.1000
         0.9000   0.1250  -0.0000   0.1000
         1.0000   0.1250   0.0000   0.1000

 0.200   0.0000   1.0000   0.0000   1.0000
         0.1000   1.0000  -0.0000   1.0000
         0.2000   1.0000  -0.0000   1.0000
         0.3000   0.8718   0.1601   0.8253
         0.4000   0.6113   0.5543   0.5022
         0.5000   0.4245   0.9314   0.3014
         0.6000   0.4259   0.9277   0.3030
         0.7000   0.2772   0.9272   0.3031
         0.8000   0.2657   0.9276   0.3032
         0.9000   0.1250  -0.0000   0.1000
         1.0000   0.1250   0.0000   0.1000


 Integration Statistics:
  Number of time steps           (nearest  50) =    150
  Number of function evaluations (nearest  50) =    400
  Number of Jacobian evaluations (nearest   1) =      1
  Number of iterations           (nearest   1) =      2
```

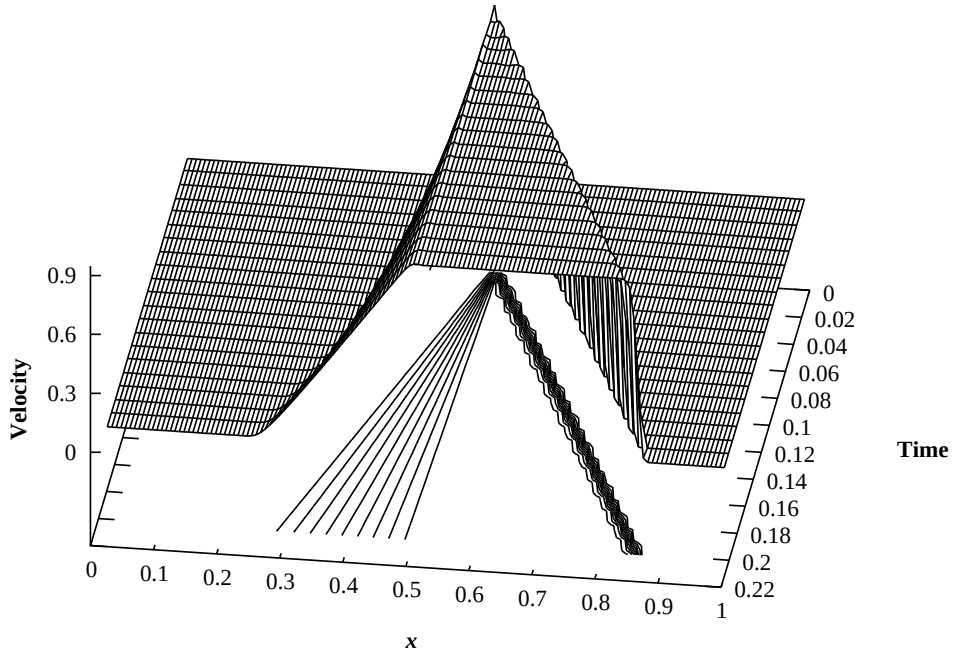
Example Program 1
First-order System with Coupled ODEs
Solution $U(1,x,t)$



Example Program 2
Shock Tube Test Problem of Euler Equations in Gas Dynamics
DENSITY



Shock Tube Test Problem of Euler Equations in Gas Dynamics
VELOCITY



Shock Tube Test Problem of Euler Equations in Gas Dynamics
PRESSURE