NAG C Library Function Document

nag_pde_parab_1d_cd (d03pfc)

1 Purpose

nag_pde_parab_1d_cd (d03pfc) integrates a system of linear or nonlinear convection-diffusion equations in one space dimension, with optional source terms. The system must be posed in conservative form. Convection terms are discretised 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 PDEs to a system of ordinary differential equations (ODEs), and the resulting system is solved using a backward differentiation formula (BDF) method.

2 Specification

```
void nag_pde_parab_1d_cd (Integer npde, double *ts, double tout,
          void (*pdedef)(Integer npde, double t, double x, const double u[],
                const double ux[], double p[], double c[], double d[], double s[],
                Integer *ires, Nag_Comm *comm),
          void (*numflx)(Integer npde, double t, double x, 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 ibnd, double g[], Integer *ires,
                Nag_Comm *comm),
          double u[], Integer npts, const double x[], const double acc[], double tsmax,
          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 (d03pfc) 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 $i = 1, 2, \ldots, \mathbf{npde}$, $\quad a \leq x \leq b, \quad t \geq t_0$, where the vector U is the set of solution values

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

The functions $P_{i,j}$, F_i, C_i and S_i depend on x, t and U; and D_i depends on x, t, U and U_x, where U_x is the spatial derivative of U. Note that $P_{i,j}$, F_i, C_i and S_i must not depend on any space derivatives; and none of the functions may depend on time derivatives. In terms of conservation laws, F_i, $C_i\partial D_i/\partial x$ and S_i are the convective flux, diffusion and source terms respectively.

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 $U(x,t)$ 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 discretisation 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 the user 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 $x_{j-1/2} = (x_{j-1} + x_j)/2$ for $j = 2, 3, \ldots,$ **npts**. The left and right values are calculated by nag_pde_parab_1d_cd (d03pfc) 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

$$\frac{\partial U_i}{\partial t} + \frac{\partial F_i}{\partial y} = 0, \tag{3}$$

where $y = x - x_{j-1/2}$, i.e., $y = 0$ corresponds to $x = x_{j-1/2}$, 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 (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

$$\hat{F} = \tfrac{1}{2}(F_L + F_R) - \tfrac{1}{2}\sum_{k=1}^{\mathbf{npde}} \alpha_k |\lambda_k| e_k, \tag{4}$$

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

$$U_R - U_L = \sum_{k=1}^{\mathbf{npde}} \alpha_k e_k. \tag{5}$$

An example is given in Section 9.

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 a function **pdedef** supplied by the user. The numerical flux $\hat{F}_i$ must be supplied in a separate user-supplied function **numflx**. For problems in the form (2), the actual argument d03pfp may be used for **pdedef** (d03pfp is included in the NAG C Library; however, its name may be implementation-dependent: see the Users' Note for your implementation for details). d03pfp sets the matrix with entries $P_{i,j}$ to the identity matrix, and the functions C_i, D_i and S_i to zero.

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 discretisation 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). Both types of boundary conditions must be supplied by the user, 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 the user 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 only linear extrapolation is allowed in this function (for greater flexibility the function nag_pde_parab_1d_cd_ode (d03plc) should be used). For problems in which the solution is known to be uniform (in space) towards a boundary during the period of integration then extrapolation is unneccesary; the numerical boundary condition can be supplied as the known solution at the boundary. Examples can be found in Section 9.

The boundary conditions must be specified in a function **bndary** (provided by the user) in the form

$$G_i^L(x,t,U) = 0 \quad \text{at} \quad x = a, \quad i = 1,2,\ldots,\mathbf{npde}, \tag{6}$$

at the left-hand boundary, and

$$G_i^R(x,t,U) = 0 \quad \text{at} \quad x = b, \quad i = 1,2,\ldots,\mathbf{npde}, \tag{7}$$

at the right-hand boundary.

Note that spatial derivatives at the boundary are not passed explicitly to the function **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 problem is subject to the following restrictions:

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

(ii) $P_{i,j}$, F_i, C_i, D_i and S_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 function **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 non-zero so that there is a time derivative present in the PDE problem;

In total there are **npde** $\times$ **npts** ODEs in the time direction. This system is then integrated forwards in time using a BDF method.

For further details of the algorithm, 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* Birkhäuser 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

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*

pdedef must evaluate the functions $P_{i,j}$, C_i, D_i and S_i which partially define the system of PDEs. $P_{i,j}$, C_i and S_i may depend on x, t and U; D_i may depend on x, t, U and U_x. **pdedef** is called approximately midway between each pair of mesh points in turn by nag_pde_parab_1d_cd (d03pfc). The actual argument d03pfp may be used for **pdedef** for problems in the form (2) (d03pfp is included in the NAG C Library; however, its name may be implementation-dependent: see the Users' Note for your implementation for details).

Its specification is:

```
void pdedef (Integer npde, double t, double x, const double u[],
     const double ux[], 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 $\partial U_i(x,t)/\partial x$, for $i = 1,2,\ldots,$**npde**.

6: **p**[**npde** × **npde**] – double *Output*

Note: where $\mathbf{P}(i,j)$ appears in this document it refers to the array element **p**[**npde** $\times (j-1)+i-1$]. We recommend using a #define to make the same definition in your calling program.

On exit: $\mathbf{P}(i,j)$ must be set to the value of $P_{i,j}(x,t,U)$, for $i,j = 1,2,\ldots,$**npde**.

7: **c**[**npde**] – double *Output*

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

8: **d**[**npde**] – double *Output*

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

9: **s**[**npde**] – double *Output*

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

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

On entry: set to -1 or 1.

On exit: should usually remain unchanged. However, the user 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. The user may wish to set **ires** $= 3$ when a physically meaningless input or output value has been generated. If the user consecutively sets **ires** $= 3$, then nag_pde_parab_1d_cd (d03pfc) returns to the calling function with the error indicator set to **fail.code** = **NE_FAILED_DERIV**.

11: **comm** – NAG_Comm * *Input/Output*

The NAG communication parameter (see the Essential Introduction).

5: **numflx** *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 (d03pfc).

Its specification is:

```
void numflx (Integer npde, double t, double x, 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: **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**.

5: **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**.

6: **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**.

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

On entry: set to -1 or 1.

On exit: should usually remain unchanged. However, the user 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. The user may wish to set **ires** $= 3$ when a physically meaningless input or output value has been generated. If the user consecutively sets **ires** $= 3$, then nag_pde_parab_1d_cd (d03pfc) returns to the calling function with the error indicator set to **fail.code** = **NE_FAILED_DERIV**.

8: **comm** – NAG_Comm * *Input/Output*

The NAG communication parameter (see the Essential Introduction).

9: **saved** – Nag_D03_Save * *Input/Output*

On entry: contains the current state of saved data concerning the computation. 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** should be passed through unchanged to that function.

On exit: the user should not change the components of **saved**.

6: **bndary** *Function*

bndary must evaluate the functions G_i^L and G_i^R which describe the physical and numerical boundary conditions, as given by (6) and (7).

Its specification is:

```
void bndary (Integer npde, Integer npts, double t, const double x[],
     const double u[], 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**[3 $\times$ **npde**] – const double *Input*

Note: where $\mathbf{U}(i, j)$ appears in this document it refers to the array element **u**[**npde** $\times (j-1)+i-1$]. We recommend using a #define to make the same definition in your calling program.

On entry: contains the value of solution components in the boundary region. If **ibnd** $= 0$, then $\mathbf{U}(i, j)$ contains the value of the component $U_i(x, t)$ at $x = \mathbf{x}[j - 1]$, for $i = 1, 2, \ldots, \mathbf{npde}$; $j = 1, 2, 3$. If **ibnd** $\neq 0$, then $\mathbf{U}(i, j)$ contains the value of the component $U_i(x, t)$ at $x = \mathbf{x}[\mathbf{npts} - j]$, for $i = 1, 2, \ldots, \mathbf{npde}$; $j = 1, 2, 3$.

6: **ibnd** – Integer *Input*

On entry: specifies which boundary conditions are to be evaluated. If **ibnd** $= 0$, then **bndary** must evaluate the left-hand boundary condition at $x = a$. If **ibnd** $\neq 0$, then **bndary** must evaluate the right-hand boundary condition at $x = b$.

7: **g**[**npde**] – double *Output*

On exit: $\mathbf{g}[i - 1]$ must contain the ith component of either $\mathbf{g}^L$ or $\mathbf{g}^R$ in (6) and (7), depending on the value of **ibnd**, for $i = 1, 2, \ldots, \mathbf{npde}$.

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

On entry: set to -1 or 1.

On exit: should usually remain unchanged. However, the user 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. The user may wish to set **ires** $= 3$ when a physically meaningless input or output value has been generated. If the user consecutively sets **ires** $= 3$, then nag_pde_parab_1d_cd (d03pfc) returns to the calling function with the error indicator set to **fail.code** = **NE_FAILED_DERIV**.

9: **comm** – NAG_Comm * *Input/Output*

The NAG communication parameter (see the Essential Introduction).

7: **u**[**npde** × **npts**] – double *Input/Output*

Note: where $\mathbf{U}(i, j)$ appears in this document it refers to the array element $\mathbf{u}[\mathbf{npde} \times (j - 1) + i - 1]$. We recommend using a #define to make the same definition in your calling program.

On entry: $\mathbf{U}(i, j)$ must contain the initial value of $U_i(x, t)$ at $x = \mathbf{x}[j - 1]$ and $t = \mathbf{ts}$; for $i = 1, 2, \ldots, \mathbf{npde}$; $j = 1, 2, \ldots, \mathbf{npts}$.

On exit: $\mathbf{U}(i, j)$ will contain the computed solution $U_i(x, t)$ at $x = \mathbf{x}[j - 1]$ and $t = \mathbf{ts}$; for $i = 1, 2, \ldots, \mathbf{npde}$; $j = 1, 2, \ldots, \mathbf{npts}$.

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. $\mathbf{x}[0]$ must specify the left-hand boundary, a, and $\mathbf{x}[\mathbf{npts} - 1]$ must specify the right-hand boundary, b.

Constraint: $\mathbf{x}[0] < \mathbf{x}[1] < \cdots < \mathbf{x}[\mathbf{npts} - 1]$.

10: **acc**[2] – const double *Input*

On entry: the components of **acc** contain the relative and absolute error tolerances used in the local error test in the time integration.

If $\mathrm{E}(i,j)$ is the estimated error for U_i at the jth mesh point, the error test is

$$\mathrm{E}(i,j) = \mathbf{acc}[0] \times \mathbf{U}(i,j) + \mathbf{acc}[1].$$

Constraint: **acc**[0] and **acc**[1] ≥ 0.0 (but not both zero).

11: **tsmax** – double *Input*

On entry: the maximum absolute step size to be allowed in the time integration. If **tsmax** $= 0.0$ then no maximum is imposed.

Constraint: **tsmax** ≥ 0.0.

12: **rsave**[**lrsave**] – double *Input/Output*

On entry: if **ind** $= 0$, **rsave** need not be set. If **ind** $= 1$ then it must be unchanged from the previous call to the function.

On exit: contains information about the iteration required for subsequent calls.

13: **lrsave** – Integer *Input*

On entry: the dimension of the array **rsave** as declared in the function from which nag_pde_parab_1d_cd (d03pfc) is called.

Constraint: **lrsave** $\geq (11 + 9 \times \mathbf{npde}) \times \mathbf{npde} \times \mathbf{npts} + (32 + 3 \times \mathbf{npde}) \times \mathbf{npde} + 7 \times \mathbf{npts} + 54$.

14: **isave**[**lisave**] – Integer *Input/Output*

On entry: if **ind** $= 0$, **isave** need not be set. If **ind** $= 1$ then it must be unchanged from the previous call to the function.

On exit: contains information about the iteration required for subsequent calls. In particular:

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 computing 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 last backward differentiation formula method used.

isave[4] contains the number of Newton iterations performed by the time integrator. Each iteration involves an ODE residual evaluation followed by a back-substitution using the *LU* decomposition of the Jacobian matrix.

15: **lisave** – Integer *Input*

On entry: the dimension of the array **isave** as declared in the function from which nag_pde_parab_1d_cd (d03pfc) is called.

Constraint: **lisave** $\geq \mathbf{npde} \times \mathbf{npts} + 24$.

16: **itask** – Integer *Input*

On entry: the task to be performed by the ODE integrator. The permitted values of **itask** and their meanings are detailed below:

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**.

Constraint: $1 \leq$ **itask** ≤ 3.

17: **itrace** – Integer *Input*

On entry: the level of trace information required from nag_pde_parab_1d_cd (d03pfc) and the underlying ODE solver. **itrace** may take the value -1, 0, 1, 2, or 3. If **itrace** < -1, then -1 is assumed and similarly if **itrace** > 3, then 3 is assumed. If **itrace** $= -1$, no output is generated. If **itrace** $= 0$, only warning messages from the PDE solver are printed. If **itrace** > 0, then 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. The advisory messages are given in greater detail as **itrace** increases.

18: **outfile** – 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.

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

On entry: **ind** must be set to 0 or 1.

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 parameters **tout** and **fail** should be reset between calls to nag_pde_parab_1d_cd (d03pfc).

Constraint: $0 \leq$ **ind** ≤ 1.

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

20: **comm** – NAG_Comm * *Input/Output*

The NAG communication parameter (see the Essential Introduction).

21: **saved** – Nag_D03_Save * *Input/Output*

Note: saved is a NAG defined structure. See Section 2.2.1.1 of the Essential Introduction.

On entry: if the current call to nag_pde_parab_1d_cd (d03pfc) follows a previous call to a Chapter d03 function then **saved** must contain the unchanged value output from that previous call.

On exit: data to be passed unchanged to any subsequent call to a Chapter d03 function.

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

The NAG error parameter (see the Essential Introduction).

6 Error Indicators and Warnings

NE_INT

ires set to an invalid value in call to **pdedef**, **numflx**, or **bndary**.

On entry, **npde** $= \langle value \rangle$.
Constraint: **npde** ≥ 1.

On entry, **npts** $= \langle value \rangle$.
Constraint: **npts** ≥ 3.

On entry, **ind** is not equal to 0 or 1: **ind** $= \langle value \rangle$.

On entry, **itask** is not equal to 1, 2, or 3: **itask** $= \langle value \rangle$.

NE_INT_2

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: $\langle value \rangle$.

NE_ACC_IN_DOUBT

Integration completed, but small changes in **acc** are unlikely to result in a changed solution. **acc**[0] $= \langle value \rangle$, **acc**[1] $= \langle value \rangle$.

NE_FAILED_DERIV

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

NE_FAILED_START

Values in **acc** are too small to start integration: **acc**[0] $= \langle value \rangle$, **acc**[1] $= \langle value \rangle$.

NE_FAILED_STEP

Repeated errors in an attempted step of underlying ODE solver. Integration was successful as far as **ts**: **ts** $= \langle value \rangle$.

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

Underlying ODE solver cannot make further progress from the point **ts** with the supplied values of **acc**. **ts** $= \langle value \rangle$, **acc**[0] $= \langle value \rangle$, **acc**[1] $= \langle value \rangle$.

NE_INCOMPAT_PARAM

On entry, **acc**[0] and **acc**[1] are both zero.

NE_INTERNAL_ERROR

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

NE_NOT_STRICTLY_INCREASING

On entry, mesh points **x** appear to be badly ordered: $i = \langle value \rangle$, **x**$[i-1] = \langle value \rangle$ $j = \langle value \rangle$, **x**$[j-1] = \langle value \rangle$.

NE_REAL

On entry, **tsmax** $= \langle value \rangle$.
Constraint: **tsmax** ≥ 0.0.

On entry, **acc**[1] < 0.0: **acc**[1] $= \langle value \rangle$.

On entry, **acc**[0] < 0.0: **acc**[0] $= \langle value \rangle$.

NE_REAL_2

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

On entry, **tout** $\leq$ **ts**: **tout** $= \langle value \rangle$, **ts** $= \langle value \rangle$.

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 **pdedef**, **numflx**, or **bndary**. Integration is successful as far as **ts**: **ts** $= \langle$*value*$\rangle$.

NE_ALLOC_FAIL

Memory allocation failed.

NE_BAD_PARAM

On entry, parameter $\langle$*value*$\rangle$ had an illegal value.

NE_NOT_WRITE_FILE

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

NE_NOT_CLOSE_FILE

Cannot close file $\langle$*value*$\rangle$.

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 consult NAG for assistance.

7 Accuracy

The function 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. The user should therefore test the effect of varying the components of the accuracy parameter, **acc**.

8 Further Comments

The function 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 users are advised to use one of the central-difference schemes for such problems.

Users 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 **tsmax**. It is worth experimenting with this parameter, 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.

9 Example

For this function two examples are presented, with a main program and two example problems given in the functions $ex1$ and $ex2$.

Example 1 ($ex1$**)**

This example is a simple first-order system which illustrates the calculation of the numerical flux using Roe's approximate Riemann solver, and the specification of numerical boundary conditions using extrapolated characteristic variables. The PDEs are

$$\begin{aligned}\frac{\partial U_1}{\partial t}+\frac{\partial U_1}{\partial x}+\frac{\partial U_2}{\partial x} &= 0,\\ \frac{\partial U_2}{\partial t}+4\frac{\partial U_1}{\partial x}+\frac{\partial U_2}{\partial x} &= 0,\end{aligned}$$

for $x \in [0,1]$ and $t \geq 0$. The PDEs have an exact solution given by

$$\begin{aligned}U_1(x,t) &= \tfrac{1}{2}\{\exp(x+t)+\exp(x-3t)\}+\tfrac{1}{4}\Big\{\sin(2\pi(x-3t)^2)-\sin(2\pi(x+t)^2)\Big\}+2t^2-2xt,\\ U_2(x,t) &= \exp(x-3t)-\exp(x+t)+\tfrac{1}{2}\Big\{\sin(2\pi(x-3t)^2)+\sin(2\pi(x-3t)^2)\Big\}+x^2+5t^2-2xt.\end{aligned}$$

The initial conditions are given by the exact solution. The characteristic variables are $2U_1+U_2$ and $2U_1-U_2$ corresponding to the characteristics given by $dx/dt=3$ and $dx/dt=-1$ respectively. Hence a physical boundary condition is required for $2U_1+U_2$ at the left-hand boundary, and for $2U_1-U_2$ at the right-hand boundary (corresponding to the incoming characteristics); and a numerical boundary condition is required for $2U_1-U_2$ at the left-hand boundary, and for $2U_1+U_2$ at the right-hand boundary (outgoing characteristics). The physical boundary conditions are obtained from the exact solution, and the numerical boundary conditions are calculated by linear extrapolation of the appropriate characteristic variable. The numerical flux is calculated using Roe's approximate Riemann solver: Using the notation in Section 3, the flux vector F and the Jacobian matrix A are

$$F=\begin{bmatrix}U_1+U_2\\4U_1+U_2\end{bmatrix}\quad\text{and}\quad A=\begin{bmatrix}1&1\\4&1\end{bmatrix},$$

and the eigenvalues of A are 3 and -1 with right eigenvectors $[1\quad 2]^T$ and $[-1\quad 2]^T$ respectively. Using equation (4) the α_k are given by

$$\begin{bmatrix}U_{1R}-U_{1L}\\U_{2R}-U_{2L}\end{bmatrix}=\alpha_1\begin{bmatrix}1\\2\end{bmatrix}+\alpha_2\begin{bmatrix}-1\\2\end{bmatrix},$$

that is

$$\alpha_1=\tfrac{1}{4}(2U_{1R}-2U_{1L}+U_{2R}-U_{2L})\quad\text{and}\quad\alpha_2=\tfrac{1}{4}(-2U_{1R}+2U_{1L}+U_{2R}-U_{2L}).$$

F_L is given by

$$F_L=[U_{1L}+U_{2L}\,4U_{1L}+U_{2L}],$$

and similarly for F_R. From equation (4), the numerical flux vector is

$$\hat{F}=\tfrac{1}{2}\begin{bmatrix}U_{1L}+U_{2L}+U_{1R}+U_{2R}\\4U_{1L}+U_{2L}+4U_{1R}+U_{2R}\end{bmatrix}-\tfrac{1}{2}\alpha_1|3|\begin{bmatrix}1\\2\end{bmatrix}-\tfrac{1}{2}\alpha_2|-1|\begin{bmatrix}-1\\2\end{bmatrix},$$

that is

$$\hat{F}=\tfrac{1}{2}\begin{bmatrix}3U_{1L}-U_{1R}+\tfrac{3}{2}U_{2L}+\tfrac{1}{2}U_{2R}\\6U_{1L}+2U_{1R}+3U_{2L}-U_{2R}\end{bmatrix}.$$

Example 2 (*ex*2)

This example is an advection-diffusion equation in which the flux term depends explicitly on x:

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

for $x \in [-1, 1]$ and $0 \leq t \leq 10$. The parameter ϵ is taken to be 0.01. The two physical boundary conditions are $U(-1, t) = 3.0$ and $U(1, t) = 5.0$ and the initial condition is $U(x, 0) = x + 4$. The integration is run to steady state at which the solution is known to be $U = 4$ across the domain with a narrow boundary layer at both boundaries. In order to write the PDE in conservative form, a source term must be introduced, i.e.,

$$\frac{\partial U}{\partial t} + \frac{\partial(xU)}{\partial x} = \epsilon\frac{\partial^2 U}{\partial x^2} + U.$$

As in Example 1, the numerical flux is calculated using the Roe approximate Riemann solver. The Riemann problem to solve locally is

$$\frac{\partial U}{\partial t} + \frac{\partial(xU)}{\partial x} = 0.$$

The x in the flux term is assumed to be constant at a local level, and so using the notation in Section 3, $F = xU$ and $A = x$. The eigenvalue is x and the eigenvector (a scalar in this case) is 1. The numerical flux is therefore

$$\hat{F} = \begin{cases} xU_L & \text{if} \quad x \geq 0, \\ xU_R & \text{if} \quad x < 0. \end{cases}$$

9.1 Program Text

```
/* nag_pde_parab_1d_cd (d03pfc) Example Program.
 *
 * Copyright 2001 Numerical Algorithms Group.
 *
 * Mark 7, 2001.
 */

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

int ex1(void);
int ex2(void);

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

static void bndary1(Integer, Integer, double, const double[],
                    const double[], Integer, double[], Integer *,
                    Nag_Comm *);

static void bndary2(Integer, Integer, double, const double[],
                    const double[], Integer, double[], Integer *,
                    Nag_Comm *);

static void numflx1(Integer, double, double, const double[],
                    const double[], double[], Integer *, Nag_Comm *,
                    Nag_D03_Save *);

static void numflx2(Integer, double, double, const double[],
                    const double[], double[], Integer *, Nag_Comm *,
                    Nag_D03_Save *);

static void exact(double, double *, Integer, const double *, Integer);
```

```
int main(void)
{
  Vprintf("d03pfc Example Program Results\n");
  ex1();
  ex2();
  return 0;
}

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

int ex1(void)
{
  double tout, ts, tsmax;
  const Integer npde=2, npts=101, outpts=7, inter=20, lisave=npde*npts+24,
    lrsave=(11+9*npde)*npde*npts+(32+3*npde)*npde+7*npts+54;
  Integer exit_status, i, ind, it, itask, itrace, j, nop;
  double *acc=0, *rsave=0, *u=0, *ue=0, *x=0, *xout=0;
  Integer *isave=0;
  NagError fail;
  Nag_Comm comm;
  Nag_D03_Save saved;

  /* Allocate memory */

  if ( !(acc = NAG_ALLOC(2, double)) ||
       !(rsave = NAG_ALLOC(lrsave, double)) ||
       !(u = NAG_ALLOC(npde*npts, double)) ||
       !(ue = NAG_ALLOC(npde*outpts, double)) ||
       !(x = NAG_ALLOC(npts, double)) ||
       !(xout = NAG_ALLOC(outpts, double)) ||
       !(isave = NAG_ALLOC(lisave, Integer)) )
    {
      Vprintf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  Vprintf("\n\nExample 1\n\n\n");
  INIT_FAIL(fail);
  exit_status = 0;

  itrace = 0;
  acc[0] = 1.0e-4;
  acc[1] = 1.0e-5;
  tsmax = 0.0;

  Vprintf(" npts = %4ld acc[0] = %10.3e acc[1] = %10.3e\n\n",
          npts, acc[0], acc[1]);
  Vprintf("        x        Approx u    Exact u     Approx v    Exact v\n");

  /* Initialise mesh */

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

  /* Set initial values */

  ts = 0.0;
  exact(ts, u, npde, x, npts);

  ind = 0;
  itask = 1;

  for (it = 1; it <= 2; ++it)
    {
      tout = 0.1*it;

      d03pfc(npde, &ts, tout, d03pfp, numflx1, bndary1, u, npts,
             x, acc, tsmax, rsave, lrsave, isave, lisave, itask,
              itrace, 0, &ind, &comm, &saved, &fail);

      if (fail.code != NE_NOERROR)
        {
          Vprintf("Error from d03pfc.\n%s\n", fail.message);
          exit_status = 1;
          goto END;
        }

      /* Set output points */

      nop = 0;
      for (i = 0; i < 101; i += inter)
        {
          ++nop;
          xout[nop - 1] = x[i];
        }

      Vprintf("\n t = %6.3f\n\n", ts);

      /* Check against exact solution */

      exact(tout, ue, npde, xout, nop);

      for (i = 1; i <= nop; ++i)
        {
          j = (i-1)*inter+1;
          Vprintf("      %9.4f %9.4f %9.4f %9.4f %9.4f\n",
                  xout[i-1], U(1,j), UE(1,i), U(2,j), UE(2,i));
        }
    }
  Vprintf("\n");
  Vprintf(" Number of integration steps in time = %6ld\n", isave[0]);
  Vprintf(" Number of function evaluations = %6ld\n", isave[1]);
  Vprintf(" Number of Jacobian evaluations = %6ld\n", isave[2]);
  Vprintf(" Number of iterations = %6ld\n\n", isave[4]);

 END:

  if (acc) NAG_FREE(acc);
  if (rsave) NAG_FREE(rsave);
  if (u) NAG_FREE(u);
  if (ue) NAG_FREE(ue);
  if (x) NAG_FREE(x);
  if (xout) NAG_FREE(xout);
  if (isave) NAG_FREE(isave);

  return exit_status;
}

static void bndary1(Integer npde, Integer npts, double t, const double x[],
            const double u[], Integer ibnd, double g[], Integer *ires,
            Nag_Comm *comm)
{
  double c, exu1, exu2, pi;
  double ue[2];

  pi = nag_pi;

  if (ibnd == 0)
    {
      exact(t, ue, npde, &x[0], 1);
      c = (x[1] - x[0])/(x[2] - x[1]);
      exu1 = (c + 1.0)*U(1, 2) - c*U(1, 3);
      exu2 = (c + 1.0)*U(2, 2) - c*U(2, 3);
      g[0] = 2.0*U(1, 1) + U(2, 1) - 2.0*UE(1, 1) - UE(2, 1);
      g[1] = 2.0*U(1, 1) - U(2, 1) - 2.0*exu1 + exu2;
    } else {
      exact(t, ue, npde, &x[npts-1], 1);
      c = (x[npts-1] - x[npts - 2])/(x[npts - 2] - x[npts - 3]);
      exu1 = (c + 1.0)*U(1, 2) - c*U(1, 3);
      exu2 = (c + 1.0)*U(2, 2) - c*U(2, 3);
      g[0] = 2.0*U(1, 1) - U(2, 1) - 2.0*UE(1, 1) + UE(2, 1);
      g[1] = 2.0*U(1, 1) + U(2, 1) - 2.0*exu1 - exu2;
    }

  return;
}

static void numflx1(Integer npde, double t, double x, const double uleft[],
             const double uright[], double flux[], Integer *ires,
             Nag_Comm *comm, Nag_D03_Save *saved)
{
  flux[0] = 0.5*(-uright[0] + 3.0*uleft[0] + 0.5*uright[1] + 1.5*uleft[1]);
  flux[1] = 0.5*( 2.0*uright[0] + 6.0*uleft[0] - uright[1] + uleft[1]*3.);

  return;
}

static void exact(double t, double *u, Integer npde,
                  const double *x, Integer npts)
{
  double x1, x2, pi;
  Integer i;

  pi = nag_pi;

  /* Exact solution (for comparison and b.c. purposes) */

  for (i = 1; i <= npts; ++i)
    {
      x1 = x[i-1] + t;
      x2 = x[i-1] - 3.0*t;

      U(1,i) = 0.5*(exp(x1) + exp(x2))
        + 0.25*(sin(2.0*pi*(x2*x2)) - sin(2.0*pi*(x1*x1)))
        + 2.0*t*t - 2.0*x[i-1]*t;

      U(2,i) = exp(x2) - exp(x1) +
        0.5*(sin(2.0*pi*(x2*x2)) + sin(2.0*pi*(x1*x1))) +
        x[i-1]*x[i-1] + 5.0*t*t - 2.0*x[i-1]*t;
    }
  return;
}

int ex2(void)
{
  double tout, ts, tsmax;
  const Integer npde=1, npts=151, outpts=7, lisave=npde*npts+24,
    lrsave=(11+9*npde)*npde*npts+(32+3*npde)*npde+7*npts+54;
  Integer exit_status=0, i, ind, it, itask, itrace;
  double *acc=0, *rsave=0, *u=0, *x=0, *xout=0;
  Integer *isave=0;
  NagError fail;
  Nag_Comm comm;
  Nag_D03_Save saved;

  /* Allocate memory */

  if ( !(acc = NAG_ALLOC(2, double)) ||
       !(rsave = NAG_ALLOC(lrsave, double)) ||
       !(u = NAG_ALLOC(npde*npts, double)) ||
       !(x = NAG_ALLOC(npts, double)) ||
       !(xout = NAG_ALLOC(outpts, double)) ||
       !(isave = NAG_ALLOC(lisave, Integer)) )
    {
      Vprintf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  Vprintf("\n\nExample 2\n\n\n");
  INIT_FAIL(fail);

  itrace = 0;
  acc[0] = 1e-5;
  acc[1] = 1e-5;

  Vprintf(" npts = %4ld  acc[0] = %10.3e acc[1] = %10.3e\n\n",
          npts, acc[0], acc[1]);

  /* Initialise mesh */

  for (i = 0; i < npts; ++i) x[i] = -1.0 + 2.0*i/(npts-1.0);

  /* Set initial values */

  for (i = 1; i <= npts; ++i) U(1, i) = x[i-1] + 4.0;

  ind = 0;
  itask = 1;
  tsmax = 0.02;

  /* Set output points */

  xout[0] = x[0];
  xout[1] = x[3];
  xout[2] = x[36];
  xout[3] = x[75];
  xout[4] = x[111];
  xout[5] = x[147];
  xout[6] = x[150];

  Vprintf(" x    ");

  for (i = 0; i < 7; ++i)
    {
      Vprintf("%9.4f", xout[i]);
      Vprintf((i+1)%7 == 0 || i == 6 ?"\n":"");
    }
  Vprintf("\n");

  /* Loop over output value of t */

  ts = 0.0;
  tout = 1.0;
  for (it = 0; it < 2; ++it)
    {
      if (it == 1) tout = 10.0;

      d03pfc(npde, &ts, tout, pdedef, numflx2, bndary2, u, npts,
             x, acc, tsmax, rsave, lrsave, isave, lisave, itask,
             itrace, 0, &ind, &comm, &saved, &fail);

      if (fail.code != NE_NOERROR)
        {
          Vprintf("Error from d03pfc.\n%s\n", fail.message);
          exit_status = 1;
          goto END;
        }

      Vprintf(" t = %6.3f\n", ts);
      Vprintf(" u    %9.4f%9.4f%9.4f%9.4f%9.4f%9.4f%9.4f\n\n",
              U(1,1), U(1,4), U(1,37), U(1,76),
              U(1,112), U(1,148), U(1,151));
    }

  Vprintf(" Number of integration steps in time = %6ld\n", isave[0]);
  Vprintf(" Number of function evaluations = %6ld\n", isave[1]);
  Vprintf(" Number of Jacobian evaluations = %6ld\n", isave[2]);
  Vprintf(" Number of iterations = %6ld\n\n", isave[4]);

 END:
  if (acc) NAG_FREE(acc);
  if (rsave) NAG_FREE(rsave);
  if (u) NAG_FREE(u);
  if (x) NAG_FREE(x);
  if (xout) NAG_FREE(xout);
  if (isave) NAG_FREE(isave);

  return exit_status;
}

static void pdedef(Integer npde, double t, double x, const double u[],
            const double ux[], double p[], double c[], double d[],
            double s[], Integer *ires, Nag_Comm *comm)
{
  P(1, 1) = 1.0;
  c[0] = 0.01;
  d[0] = ux[0];
  s[0] = u[0];

  return;
}

static void bndary2(Integer npde, Integer npts, double t, const double x[],
             const double u[], Integer ibnd, double g[], Integer *ires,
             Nag_Comm *comm)
{
  if (ibnd == 0)
    {
      g[0] = U(1, 1) - 3.0;
    } else {
      g[0] = U(1, 1) - 5.0;
    }
  return;
}

static void numflx2(Integer npde, double t, double x, const double uleft[],
             const double uright[], double flux[], Integer *ires,
             Nag_Comm *comm, Nag_D03_Save *saved)
{
  if (x >= 0.0)
    {
      flux[0] = x * uleft[0];
    } else {
      flux[0] = x * uright[0];
    }
  return;
}
```

9.2 Program Data

None.

9.3 Program Results

```
d03pfc Example Program Results

Example 1

 npts =  101 acc[0] =  1.000e-04 acc[1] =  1.000e-05

       x        Approx u    Exact u     Approx v    Exact v

 t =  0.100

        0.0000    1.0615    1.0613   -0.0155   -0.0150
        0.2000    0.9892    0.9891   -0.0953   -0.0957
        0.4000    1.0826    1.0826    0.1180    0.1178
        0.6000    1.7001    1.7001   -0.0751   -0.0746
        0.8000    2.3959    2.3966   -0.2453   -0.2458
        1.0000    2.1029    2.1025    0.3760    0.3753
 t =  0.200

         0.0000    1.0957    1.0956    0.0368    0.0370
         0.2000    1.0808    1.0811    0.1826    0.1828
         0.4000    1.1102    1.1100   -0.2935   -0.2938
         0.6000    1.6461    1.6454   -1.2921   -1.2908
         0.8000    1.7913    1.7920   -0.8510   -0.8525
         1.0000    2.2050    2.2050   -0.4222   -0.4221

 Number of integration steps in time =     56
 Number of function evaluations =    229
 Number of Jacobian evaluations =      7
 Number of iterations =    143

Example 2

 npts =  151  acc[0] =  1.000e-05 acc[1] =  1.000e-05

 x      -1.0000  -0.9600  -0.5200   0.0000   0.4800   0.9600   1.0000

 t =  1.000
 u       3.0000   3.6221   3.8087   4.0000   4.1766   4.3779   5.0000

 t = 10.000
 u       3.0000   3.9592   4.0000   4.0000   4.0000   4.0408   5.0000

 Number of integration steps in time =    503
 Number of function evaluations =   1190
 Number of Jacobian evaluations =     28
 Number of iterations =   1035
```

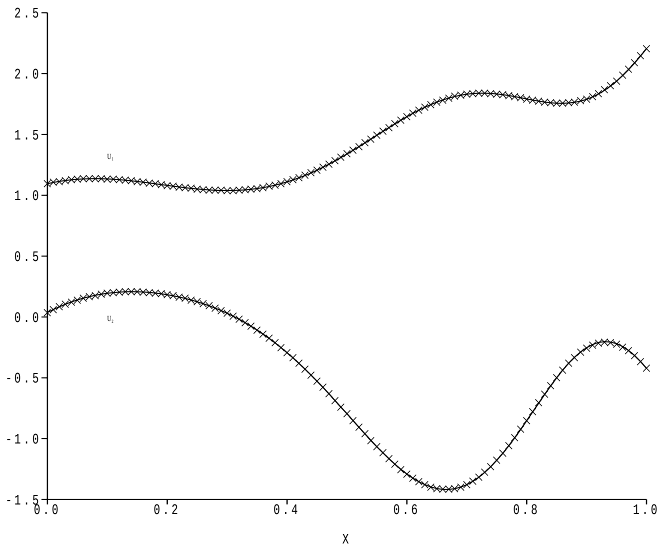

Figure 1
Solution to Example 1

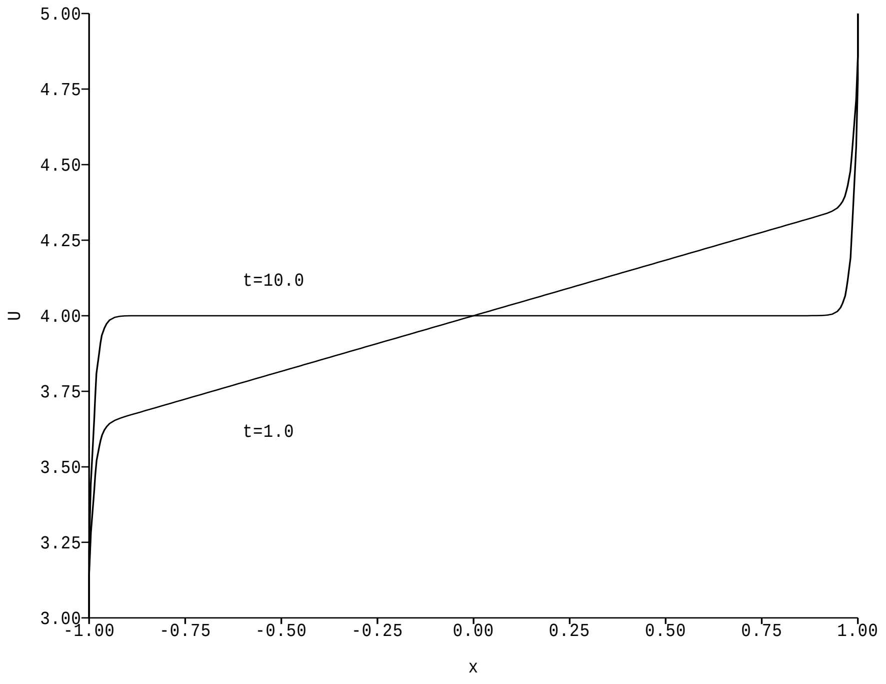

Figure 2
Solution to Example 2
