NAG C Library Function Document

nag_pde_parab_1d_keller_ode (d03pkc)

1 Purpose

nag_pde_parab_1d_keller_ode (d03pkc) integrates a system of linear or nonlinear, first-order, time-dependent partial differential equations (PDEs) in one space variable, with scope for coupled ordinary differential equations (ODEs). The spatial discretization is performed using the Keller box scheme and the method of lines is employed to reduce the PDEs to a system of ODEs. The resulting system is solved using a Backward Differentiation Formula (BDF) method or a Theta method (switching between Newton's method and functional iteration).

2 Specification

```
#include <nag.h>
#include <nagd03.h>

void nag_pde_parab_1d_keller_ode (Integer npde, double *ts, double tout,
     void (*pdedef)(Integer npde, double t, double x, const double u[],
          const double ut[], const double ux[], Integer ncode, const double v[],
          const double vdot[], double res[], Integer *ires, Nag_Comm *comm),
     void (*bndary)(Integer npde, double t, Integer ibnd, Integer nobc,
          const double u[], const double ut[], Integer ncode, const double v[],
          const double vdot[], double res[], Integer *ires, Nag_Comm *comm),
     double u[], Integer npts, const double x[], Integer nleft, 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 f[], 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_keller_ode (d03pkc) integrates the system of first-order PDEs and coupled ODEs

$$G_i\left(x,t,U,U_x,U_t,V,\dot{V}\right)=0, \quad i=1,2,\ldots,\mathbf{npde},\ a\le x\le b, t\ge t_0, \tag{1}$$

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

In the PDE part of the problem given by (1), the functions G_i must have the general form

$$G_i=\sum_{j=1}^{\mathbf{npde}}P_{i,j}\frac{\partial U_j}{\partial t}+\sum_{j=1}^{\mathbf{ncode}}Q_{i,j}\dot{V}_j+R_i=0, \quad i=1,2,\ldots,\mathbf{npde}, \tag{3}$$

where $P_{i,j}$, $Q_{i,j}$ and R_i depend on x,t,U,U_x and V.

The vector U is the set of PDE solution values

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

and the vector U_x is the partial derivative with respect to x. The vector V is the set of ODE solution values

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

and $\dot{V}$ denotes its derivative with respect to time.

In the ODE part given by (2), 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 some of the 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 F_i may only depend linearly on time derivatives. Hence equation (2) may be written more precisely as

$$F = A - B\dot{V} - CU_t^*, \tag{4}$$

where $F = \left[F_1, \ldots, F_{\mathbf{ncode}}\right]^{\mathrm{T}}$, A is a vector of length **ncode**, B is an **ncode** by **ncode** matrix, C is an **ncode** by $\left(n_\xi \times \mathbf{npde}\right)$ matrix. The entries in A, B and C 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 B and C. (See Section 5 for the specification of the user-supplied function **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 PDE system which is defined by the functions G_i must be specified in the user-supplied function **pdedef**.

The initial values of the functions $U(x,t)$ and $V(t)$ must be given at $t = t_0$.

For a first-order system of PDEs, only one boundary condition is required for each PDE component U_i. The **npde** boundary conditions are separated into n_a at the left-hand boundary $x = a$, and n_b at the right-hand boundary $x = b$, such that $n_a + n_b = \mathbf{npde}$. The position of the boundary condition for each component should be chosen with care; the general rule is that if the characteristic direction of U_i at the left-hand boundary (say) points into the interior of the solution domain, then the boundary condition for U_i should be specified at the left-hand boundary. Incorrect positioning of boundary conditions generally results in initialization or integration difficulties in the underlying time integration functions.

The boundary conditions have the form:

$$G_i^L\left(x, t, U, U_t, V, \dot{V}\right) = 0 \quad \text{at } x = a, \quad i = 1, 2, \ldots, n_a, \tag{5}$$

at the left-hand boundary, and

$$G_i^R\left(x, t, U, U_t, V, \dot{V}\right) = 0 \quad \text{at } x = b, \quad i = 1, 2, \ldots, n_b, \tag{6}$$

at the right-hand boundary.

Note that the functions G_i^L and G_i^R must not depend on U_x, since spatial derivatives are not determined explicitly in the Keller box scheme. If the problem involves derivative (Neumann) boundary conditions then it is generally possible to restate such boundary conditions in terms of permissible variables. Also note that G_i^L and G_i^R must be linear with respect to time derivatives, so that the boundary conditions have the general form:

$$\sum_{j=1}^{\mathbf{npde}} E_{i,j}^L \frac{\partial U_j}{\partial t} + \sum_{j=1}^{\mathbf{ncode}} H_{i,j}^L \dot{V}_j + S_i^L = 0, \quad i = 1, 2, \ldots, n_a, \tag{7}$$

at the left-hand boundary, and

$$\sum_{j=1}^{\mathbf{npde}} E_{i,j}^R \frac{\partial U_j}{\partial t} + \sum_{j=1}^{\mathbf{ncode}} H_{i,j}^R \dot{V}_j + S_i^R = 0, \quad i = 1, 2, \ldots, n_b, \tag{8}$$

at the right-hand boundary, where $E_{i,j}^L$, $E_{i,j}^R$, $H_{i,j}^L$, $H_{i,j}^R$, S_i^L and S_i^R depend on x, t, U and V only.

The boundary conditions must be specified in a function **bndary** provided by you.

The problem is subject to the following restrictions:

(i) $P_{i,j}$, $Q_{i,j}$ and R_i must not depend on any time derivatives;

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

(iii) The evaluation of the function G_i is done approximately at the mid-points of the mesh $\mathbf{x}[i-1]$, for $i = 1, 2, \ldots,$ **npts**, by calling the function **pdedef** for each mid-point in turn. Any discontinuities in the function **must** therefore be at one or more of the mesh points $x_1, x_2, \ldots, x_{\mathbf{npts}}$;

(iv) 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.

The algebraic-differential equation system which is defined by the functions F_i must be specified in the user-supplied function **odedef**. You must also specify the coupling points ξ in the array **xi**.

The parabolic equations are approximated by a system of ODEs in time for the values of U_i at mesh points. In this method of lines approach the Keller box scheme (see Keller (1970)) is applied to each PDE in the space variable only, resulting in a system of ODEs in time for the values of U_i at each mesh point. In total there are **npde** $\times$ **npts** + **ncode** ODEs in time direction. This system is then integrated forwards in time using a Backward Differentiation Formula (BDF) or a Theta method.

4 References

Berzins M (1990) Developments in the NAG Library software for parabolic equations *Scientific Software Systems* (ed J C Mason and M G Cox) 59–72 Chapman and Hall

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

Berzins M and Furzeland R M (1992) An adaptive theta method for the solution of stiff and nonstiff differential equations *Appl. Numer. Math.* **9** 1–19

Keller H B (1970) A new difference scheme for parabolic problems *Numerical Solutions of Partial Differential Equations* (ed J Bramble) **2** 327–350 Academic Press

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

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.

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

On exit: the value of t corresponding to the solution in **u**. Normally **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 G_i which define the system of PDEs. **pdedef** is called approximately midway between each pair of mesh points in turn by nag_pde_parab_1d_keller_ode (d03pkc).

Its specification is:

```
void pdedef (Integer npde, double t, double x, const double u[],
      const double ut[], const double ux[], Integer ncode, const double v[],
      const double vdot[], double res[], 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: **ut**[**npde**] – const double *Input*

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

6: **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**.

7: **ncode** – Integer *Input*

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

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

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

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

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

10: **res**[**npde**] – double *Output*

On exit: **res**$[i-1]$ must contain the ith component of G, for $i = 1,2,\ldots,$ **npde**, where G is defined as

$$G_i = \sum_{j=1}^{\mathbf{npde}} P_{i,j}\frac{\partial U_j}{\partial t} + \sum_{j=1}^{\mathbf{ncode}} Q_{i,j}\dot{V}_j, \tag{9}$$

i.e., only terms depending explicitly on time derivatives, or

$$G_i = \sum_{j=1}^{\mathbf{npde}} P_{i,j}\frac{\partial U_j}{\partial t} + \sum_{j=1}^{\mathbf{ncode}} Q_{i,j}\dot{V}_j + R_i, \tag{10}$$

i.e., all terms in equation (3).

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

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

On entry: the form of G_i that must be returned in the array **res**. If **ires** $= -1$, then equation (9) above must be used. If **ires** $= 1$, then equation (10) above must be used.

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_keller_ode (d03pkc) returns to the calling function with the error indicator set to **fail.code** = **NE_FAILED_DERIV**.

12: **comm** – Nag_Comm * *Communication Structure*

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_keller_ode (d03pkc) these pointers may be allocated memory by the user and initialized with various quantities for use by **pdedef** when called from nag_pde_parab_1d_keller_ode (d03pkc).

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

bndary must evaluate the functions G_i^L and G_i^R which describe the boundary conditions, as given in (5) and (6).

Its specification is:

```
void bndary (Integer npde, double t, Integer ibnd, Integer nobc,
      const double u[], const double ut[], Integer ncode, const double v[],
      const double vdot[], double res[], 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: **ibnd** – Integer *Input*

On entry: specifies which boundary conditions are to be evaluated.

ibnd = 0

bndary must compute the left-hand boundary condition at $x = a$.

ibnd $\neq 0$

bndary must compute the right-hand boundary condition at $x = b$.

4: **nobc** – Integer *Input*

On entry: specifies the number of boundary conditions at the boundary specified by **ibnd**.

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

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

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

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

7: **ncode** – Integer *Input*

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

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

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

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

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

Note: **vdot**$[i-1]$, for $i = 1, 2, \ldots,$ **ncode**, may only appear linearly as in (7) and (8).

10: **res**[**nobc**] – double *Output*

On exit: **res**$[i-1]$ must contain the ith component of G^L or G^R, depending on the value of **ibnd**, for $i = 1, 2, \ldots,$ **nobc**, where G^L is defined as

$$G_i^L = \sum_{j=1}^{\mathbf{npde}} E_{i,j}^L \frac{\partial U_j}{\partial t} + \sum_{j=1}^{\mathbf{ncode}} H_{i,j}^L \dot{V}_j, \tag{11}$$

i.e., only terms depending explicitly on time derivatives, or

$$G_i^L = \sum_{j=1}^{\mathbf{npde}} E_{i,j}^L \frac{\partial U_j}{\partial t} + \sum_{j=1}^{\mathbf{ncode}} H_{i,j}^L \dot{V}_j + S_i^L, \tag{12}$$

i.e., all terms in equation (7), and similarly for G_i^R.

The definitions of G^L and G^R are determined by the input value of **ires**.

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

On entry: the form of G_i^L (or G_i^R) that must be returned in the array **res**. If **ires** $= -1$, then equation (11) above must be used. If **ires** $= 1$, then equation (12) above must be used.

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_keller_ode (d03pkc) returns to the calling function with the error indicator set to **fail.code** = **NE_FAILED_DERIV**.

12: **comm** – Nag_Comm * *Communication Structure*

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_keller_ode (d03pkc) these pointers may be allocated memory by the user and initialized with various quantities for use by **bndary** when called from nag_pde_parab_1d_keller_ode (d03pkc).

6: **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**; $j = 1, 2, \ldots,$ **npts** and
u[**npts** $\times$ **npde** $+ i - 1$] contain $V_i(t_0)$, for $i = 1, 2, \ldots,$ **ncode**.

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

7: **npts** – Integer *Input*

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

Constraint: **npts** ≥ 3.

8: **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** $- 1$].

9: **nleft** – Integer *Input*

On entry: the number n_a of boundary conditions at the left-hand mesh point **x**[0].

Constraint: $0 \leq$ **nleft** $\leq$ **npde**.

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 F, which define the system of ODEs, as given in (4). If you wish to compute the solution of a system of PDEs only (i.e., **ncode** $= 0$), **odedef** must be the dummy function d03pek. (d03pek 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 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 f[], 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: **v**$[i-1]$ contains the value of component $V_i(t)$, for $i = 1, 2, \ldots,$ **ncode**.

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

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

8: **ucp**[**npde** × **nxi**] – const double *Input*

On entry: **ucp**[**npde** $\times j + i$] contains the value of $U_i(x, t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots,$ **npde**; $j = 1, 2, \ldots,$ **nxi**.

9: **ucpx**[**npde** × **nxi**] – const double *Input*

On entry: **ucpx**[**npde** $\times j + i$] 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**; $j = 1, 2, \ldots,$ **nxi**.

10: **ucpt**[**npde** × **nxi**] – const double *Input*

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

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

On exit: **f**$[i-1]$ must contain the ith component of **f**, for $i = 1, 2, \ldots,$ **ncode**, where **f** is defined as

$$F = -B\dot{V} - CU_t^*, \tag{13}$$

i.e., only terms depending explicitly on time derivatives, or

$$F = A - B\dot{V} - CU_t^*, \tag{14}$$

i.e., all terms in equation (4). The definition of **f** is determined by the input value of **ires**.

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

On entry: the form of **f** that must be returned in the array **f**. If **ires** $= -1$, then equation (13) above must be used. If **ires** $= 1$, then equation (14) above 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_keller_ode (d03pkc) returns to the calling function with the error indicator set to **fail.code** $=$ **NE_FAILED_DERIV**.

13: **comm** – Nag_Comm * *Communication Structure*

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_keller_ode (d03pkc) these pointers may be allocated memory by the user and initialized with various quantities for use by **odedef** when called from nag_pde_parab_1d_keller_ode (d03pkc).

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, ξ_i.

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**$[i-1] \geq 0$ 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**$[i-1] \geq 0$ for all relevant i.

17: **itol** – Integer *Input*

On entry: a value to indicate the form of the local error test. **itol** indicates to nag_pde_parab_1d_keller_ode (d03pkc) whether to interpret either or both of **rtol** or **atol** as a vector or scalar. The error test to be satisfied is $\|e_i/w_i\| < 1.0$, where w_i is defined as follows:

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

In the above, e_i denotes the estimated local error for the ith component of the coupled PDE/ODE system in time, $\mathbf{u}[i-1]$, for $i = 1, 2, \ldots,$ **neqn**.

The choice of norm used is defined by the argument **norm**, see below.

Constraint: $1 \leq$ **itol** ≤ 4.

18: **norm** – Nag_NormType *Input*

On entry: the type of norm to be used. Two options are available:

norm = **Nag_MaxNorm**

Maximum norm.

norm = **Nag_TwoNorm**

Averaged L_2 norm.

If $\mathbf{u}_{\text{norm}}$ denotes the norm of the vector **u** of length **neqn**, then for the averaged L_2 norm

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

while for the maximum norm

$$\mathbf{u}_{\text{norm}} = \max_i |\mathbf{u}[i-1]/w_i|.$$

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

Constraint: **norm** = **Nag_MaxNorm** 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 (i.e., **ncode** = 0).

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 value is **algopt**[0] = 1.0.

If **algopt**[0] = 2.0, then **algopt**[i], for $i = 1, 2, 3$ 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], for $i = 4, 5, 6$ 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 this 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 a 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 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** as declared in the function from which nag_pde_parab_1d_keller_ode (d03pkc) is called. 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** $\geq (2 \times ml + mu + 2) \times$ **neqn** $+ nwkres + lenode$;
if **laopt** $=$ **Nag_LinAlgSparse**, **lrsave** $\geq 4 \times$ **neqn** $+ 11 \times$ **neqn**$/2 + 1 + nwkres + lenode$;

where

ml and mu are the lower and upper half bandwidths given by **npde** $+$ **nleft** $- 1$, and
$mu = 2 \times$ **npde** $-$ **nleft** $- 1$, for problems involving PDEs only, and
$ml = mu =$ **neqn** $- 1$, for coupled PDE/ODE problems.

$nwkres =$ **npde** $\times (2 \times$ **npts** $+ 6 \times$ **nxi** $+ 3 \times$ **npde** $+ 26) +$ **nxi** $+$ **ncode** $+ 7 \times$ **npts** $+ 2$, when **ncode** > 0 and **nxi** > 0, and
$nwkres =$ **npde** $\times (2 \times$ **npts** $+ 3 \times$ **npde** $+ 32) +$ **ncode** $+ 7 \times$ **npts** $+ 3$, when **ncode** > 0 and **nxi** $= 0$, and
$nwkres =$ **npde** $\times (2 \times$ **npts** $+ 3 \times$ **npde** $+ 32) + 7 \times$ **npts** $+ 4$, when **ncode** $= 0$.

$lenode = (6 +$ int(**algopt**[1])$) \times$ **neqn** $+ 50$, when the BDF method is used, and
$lenode = 9 \times$ **neqn** $+ 50$, when the Theta method is used.

Note: when using the sparse option, 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 ODE method last used in the time integration.

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** as declared in the function from which nag_pde_parab_1d_keller_ode (d03pkc) is called. 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 $t = t_{\mathrm{crit}}$, where t_{crit} is described under the argument **algopt**.

itask $= 5$

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

Constraint: $1 \leq$ **itask** ≤ 5.

26: **itrace** – Integer *Input*

On entry: the level of trace information required from nag_pde_parab_1d_keller_ode (d03pkc) and the underlying ODE solver as follows:

itrace ≤ -1

No output is generated.

itrace $= 0$

Only warning messages from the PDE solver are printed .

itrace $= 1$

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.

itrace $= 2$

Output from the underlying ODE solver is similar to that produced when **itrace** $= 1$, except that the advisory messages are given in greater detail.

itrace ≥ 3

Output from the underlying ODE solver is similar to that produced when **itrace** $= 2$, except that the advisory messages are given in greater detail.

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: 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 arguments **tout** and **fail** should be reset between calls to nag_pde_parab_1d_keller_ode (d03pkc).

Constraint: $0 \leq$ **ind** ≤ 1.

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

29: **comm** – Nag_Comm * *Communication Structure*

The NAG communication argument (see Section 2.2.1.1 of the Essential Introduction).

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

Note: **saved** is a NAG defined type (see Section 2.2.1.1 of the Essential Introduction).

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

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

The NAG error argument (see Section 2.6 of the Essential Introduction).

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_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** or **bndary**.

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

On entry, **ind** is not equal to 0 or 1: **ind** = ⟨*value*⟩.

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

On entry, **itask** is not equal to 1, 2, 3, 4 or 5: **itask** = ⟨*value*⟩.

On entry, **itol** is not equal to 1, 2, 3, or 4: **itol** = ⟨*value*⟩.

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

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

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

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

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

NE_INT_2

On entry, corresponding elements **atol**$[i-1]$ and **rtol**$[j-1]$ are both zero. i = ⟨*value*⟩, j = ⟨*value*⟩.

On entry, **lisave** is too small: **lisave** = ⟨*value*⟩. Minimum possible dimension: ⟨*value*⟩.

On entry, **lrsave** is too small: **lrsave** = ⟨*value*⟩. Minimum possible dimension: ⟨*value*⟩.

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

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

On entry, **nleft** = ⟨*value*⟩, **npde** = ⟨*value*⟩.
Constraint: 0 ≤ **nleft** ≤ **npde**.

On entry, **nleft** > **npde**: **nleft** = ⟨*value*⟩, **npde** = ⟨*value*⟩.

When using the sparse option **lisave** or **lrsave** is too small: **lisave** = ⟨*value*⟩, **lrsave** = ⟨*value*⟩.

NE_INT_4

On entry, **npde** = ⟨*value*⟩, **npts** = ⟨*value*⟩, **ncode** = ⟨*value*⟩, **neqn** = ⟨*value*⟩.
Constraint: **neqn** = **npde** × **npts** + **ncode**.

NE_INTERNAL_ERROR

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] = ⟨*value*⟩.

NE_NOT_CLOSE_FILE

Cannot close file ⟨*value*⟩.

NE_NOT_STRICTLY_INCREASING

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

On entry, **xi**$[i] \leq$ **xi**$[i-1]$: $i = \langle value \rangle$, **xi**$[i] = \langle value \rangle$, **xi**$[i-1] = \langle value \rangle$.

NE_NOT_WRITE_FILE

Cannot open file ⟨*value*⟩ for writing.

NE_REAL_2

On entry, at least one point in **xi** lies outside [**x**[0], **x**[**npts** − 1]]: **x**[0] = ⟨*value*⟩, **x**[**npts** − 1] = ⟨*value*⟩.

On entry, **tout** − **ts** is too small: **tout** = ⟨*value*⟩, **ts** = ⟨*value*⟩.

On entry, **tout** ≤ **ts**: **tout** = ⟨*value*⟩, **ts** = ⟨*value*⟩.

NE_REAL_ARRAY

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

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

NE_SING_JAC

Singular Jacobian of ODE system. Check problem formulation.

NE_USER_STOP

In evaluating residual of ODE system, **ires** = 2 has been set in **pdedef**, **bndary**, or **odedef**. Integration is successful as far as **ts**: **ts** = ⟨*value*⟩.

NE_ZERO_WTS

Zero error weights encountered during time integration.

7 Accuracy

nag_pde_parab_1d_keller_ode (d03pkc) controls the accuracy of the integration in the time direction but not the accuracy of the approximation in space. The spatial accuracy depends on both the number of mesh points and on their distribution in space. In the time integration only the local error over a single step is controlled and so the accuracy over a number of steps cannot be guaranteed. You should therefore test the effect of varying the accuracy arguments, **atol** and **rtol**.

8 Further Comments

The Keller box scheme can be used to solve higher-order problems which have been reduced to first-order by the introduction of new variables (see the example in Section 9 below). In general, a second-order problem can be solved with slightly greater accuracy using the Keller box scheme instead of a finite-difference scheme (see nag_pde_parab_1d_fd (d03pcc) or nag_pde_parab_1d_fd_ode (d03phc) for example), but at the expense of increased CPU time due to the larger number of function evaluations required.

It should be noted that the Keller box scheme, in common with other central-difference schemes, may be unsuitable for some hyperbolic first-order problems such as the apparently simple linear advection equation $U_t + aU_x = 0$, where a is a constant, resulting in spurious oscillations due to the lack of dissipation. This type of problem requires a discretization scheme with upwind weighting (nag_pde_parab_1d_cd_ode (d03plc) for example), or the addition of a second-order artificial dissipation term.

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**.

9 Example

This problem provides a simple coupled system of two PDEs and one ODE.

$$(V_1)^2\frac{\partial U_1}{\partial t} - xV_1\dot{V}_1U_2 - \frac{\partial U_2}{\partial x} = 0,$$

$$U_2 - \frac{\partial U_1}{\partial x} = 0,$$

$$\dot{V}_1 - V_1U_1 - U_2 - 1 - t = 0,$$

for $t \in \left[10^{-4}, 0.1 \times 2^i\right]$, for $i = 1, 2, \ldots, 5, x \in [0, 1]$. The left boundary condition at $x = 0$ is

$$U_2 = -V_1 \exp t,$$

and the right boundary condition at $x = 1$ is

$$U_2 = -V_1\dot{V}_1.$$

The initial conditions at $t = 10^{-4}$ are defined by the exact solution:

$$V_1 = t, U_1(x,t) = \exp\{t(1-x)\} - 1.0 \quad \text{and} \quad U_2(x,t) = -t\exp\{t(1-x)\}, x \in [0,1],$$

and the coupling point is at $\xi_1 = 1.0$.

This problem is exactly the same as the nag_pde_parab_1d_fd_ode (d03phc) example problem, but reduced to first-order by the introduction of a second PDE variable (as mentioned in Section 8).

9.1 Program Text

```
/* nag_pde_parab_1d_keller_ode (d03pkc) Example Program.
 *
 * Copyright 2001 Numerical Algorithms Group.
 *
 * Mark 7, 2001.
 * Mark 7b revised, 2004.
 */

#include <stdio.h>
```

```
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagd03.h>
static void pdedef(Integer, double, double, const double[], const double[],
                           const double[], Integer, const double[],
                           const double[], double[], Integer *, Nag_Comm *);
static void bndary(Integer npde, double t, Integer ibnd, Integer nobc,
                           const double u[], const double ut[], Integer ncode,
                           const double v[], const double vdot[], double res[],
                           Integer *ires, Nag_Comm *comm);
static void odedef(Integer, double, Integer, const double[], const double[],
                           Integer, const double[], const double[],
                           const double[], const double[], double[],
                           Integer *, Nag_Comm *);
static void uvinit(Integer npde, Integer npts, double *x, double *u,
                           Integer ncode, Integer neqn, double ts);
static void exact(double, Integer, Integer, double *, double *);


#define UCP(I,J) ucp[npde*((J)-1)+(I)-1]


int main(void)
{
  const Integer npde=2, npts=21, ncode=1, nxi=1, nleft=1,
    neqn=npde*npts+ncode, lisave=24,
    nwkres=npde*(npts+6*nxi+3*npde+15)+ncode+nxi+7*npts+2,
    lenode=11*neqn+50, lrsave=neqn*neqn+neqn+nwkres+lenode;
  double tout, ts;
  Integer exit_status, i, ind, it, itask, itol, itrace;
  Nag_Boolean theta;
  double *algopt=0, *atol=0, *exy=0, *rsave=0, *rtol=0,
    *u=0, *x=0, *xi=0;
  Integer *isave=0;
  NagError fail;
  Nag_Comm comm;
  Nag_D03_Save saved;

  /* Allocate memory */

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

  Vprintf("nag_pde_parab_1d_keller_ode (d03pkc) Example Program Results\n\n");
  INIT_FAIL(fail);
  exit_status = 0;

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

  Vprintf("  Accuracy requirement =%10.3e", atol[0]);
  Vprintf(" Number of points = %3ld\n\n", npts);

  /* Set spatial-mesh points */

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

```
  xi[0] = 1.0;
  ind = 0;
  itask = 1;

  /* Set THETA to TRUE if the Theta integrator is required */

  theta = Nag_FALSE;
  for (i = 0; i < 30; ++i) algopt[i] = 0.0;
  if (theta)
    {
      algopt[0] = 2.0;
    } else {
      algopt[0] = 0.0;
    }
  algopt[0] = 1.0;
  algopt[12] = 0.005;

  /* Loop over output value of t */

  ts = 1e-4;
  tout = 0.0;
  Vprintf("  x        %9.3f%9.3f%9.3f%9.3f%9.3f\n\n",
          x[0], x[4], x[8], x[12], x[20]);

  uvinit(npde, npts, x, u, ncode, neqn, ts);

  for (it = 0; it < 5; ++it)
    {
      tout = 0.1*pow(2.0, (it+1.0));
      /* nag_pde_parab_1d_keller_ode (d03pkc).
       * General system of first-order PDEs, coupled DAEs, method
       * of lines, Keller box discretisation, one space variable
       */
      nag_pde_parab_1d_keller_ode(npde, &ts, tout, pdedef, bndary, u, npts, x,
                                  nleft, ncode, odedef, nxi, xi, neqn, rtol,
                                  atol, itol, Nag_TwoNorm, Nag_LinAlgFull,
                                  algopt, rsave, lrsave, isave, lisave, itask,
                                  itrace, 0, &ind, &comm, &saved, &fail);

      if (fail.code != NE_NOERROR)
        {
          Vprintf("Error from nag_pde_parab_1d_keller_ode (d03pkc).\n%s\n",
                  fail.message);
          exit_status = 1;
          goto END;
        }

      /* Check against the exact solution */

      exact(tout, neqn, npts, x, exy);

      Vprintf(" t = %6.3f\n", ts);
      Vprintf(" App.  sol.  %7.3f%9.3f%9.3f%9.3f%9.3f",
              u[0], u[8], u[16], u[24], u[40]);
      Vprintf("  ODE sol. =%8.3f\n", u[42]);
      Vprintf(" Exact sol.  %7.3f%9.3f%9.3f%9.3f%9.3f",
              exy[0], exy[8], exy[16], exy[24], exy[40]);
      Vprintf("  ODE sol. =%8.3f\n\n", ts);
    }
  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 (algopt) NAG_FREE(algopt);
  if (atol) NAG_FREE(atol);
  if (exy) NAG_FREE(exy);
  if (rsave) NAG_FREE(rsave);
  if (rtol) NAG_FREE(rtol);
  if (u) NAG_FREE(u);
```

```
  if (x) NAG_FREE(x);
  if (xi) NAG_FREE(xi);
  if (isave) NAG_FREE(isave);

  return exit_status;
}
static void uvinit(Integer npde, Integer npts, double *x,
                           double *u, Integer ncode, Integer neqn,
                           double ts)
{
  Integer i, k;


  /* Routine for PDE initial values */

  k = 0;
  for (i = 0; i < npts; ++i)
    {
      u[k] = exp(ts*(1.0-x[i])) - 1.0;
      u[k+1] = -ts*exp(ts*(1.0-x[i]));
      k += 2;
    }
  u[neqn-1] = ts;

  return;
}
static 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 f[], Integer *ires,
                           Nag_Comm *comm)
{
  if (*ires == -1) {
    f[0] = vdot[0];
  } else {
    f[0] = vdot[0] - v[0]*UCP(1, 1) - UCP(2, 1) - 1.0 - t;
  }
  return;
}
static void pdedef(Integer npde, double t, double x, const double u[],
                           const double ut[], const double ux[], Integer ncode,
                           const double v[], const double vdot[], double res[],
                           Integer *ires, Nag_Comm *comm)
{
  if (*ires == -1)
    {
      res[0] = v[0]*v[0]*ut[0] - x*u[1]*v[0]*vdot[0];
      res[1] = 0.0;
    } else {
      res[0] = v[0]*v[0]*ut[0] - x*u[1]*v[0]*vdot[0] - ux[1];
      res[1] = u[1] - ux[0];
    }
  return;
}
static void bndary(Integer npde, double t, Integer ibnd,
                           Integer nobc, const double u[], const double ut[],
                           Integer ncode, const double v[],
                           const double vdot[], double res[], Integer *ires,
                           Nag_Comm *comm)
{
  if (ibnd == 0) {
    if (*ires == -1) {
      res[0] = 0.0;
    } else {
      res[0] = u[1] + v[0]*exp(t);
    }
  } else {
    if (*ires == -1) {
      res[0] = v[0]*vdot[0];
    } else {
      res[0] = u[1] + v[0]*vdot[0];
```

```
    }
  }
  return;
}
static void exact(double time, Integer neqn, Integer npts,
                          double *x, double *u)
{
  /* Exact solution (for comparison purposes) */

  Integer i, k;


  k = 0;
  for (i = 0; i < npts; ++i) {
    u[k] = exp(time*(1.0-x[i])) - 1.0;
    k += 2;
  }
  return;
}
```

9.2 Program Data

None.

9.3 Program Results

```
nag_pde_parab_1d_keller_ode (d03pkc) Example Program Results

  Accuracy requirement = 1.000e-04 Number of points =  21

  x            0.000    0.200    0.400    0.600    1.000

 t =  0.200
 App.  sol.    0.222    0.174    0.128    0.084    0.000  ODE sol. =   0.200
 Exact sol.    0.221    0.174    0.127    0.083    0.000  ODE sol. =   0.200

 t =  0.400
 App.  sol.    0.492    0.377    0.271    0.174    0.000  ODE sol. =   0.400
 Exact sol.    0.492    0.377    0.271    0.174    0.000  ODE sol. =   0.400

 t =  0.800
 App.  sol.    1.226    0.896    0.616    0.377   -0.000  ODE sol. =   0.800
 Exact sol.    1.226    0.896    0.616    0.377    0.000  ODE sol. =   0.800

 t =  1.600
 App.  sol.    3.952    2.595    1.610    0.895   -0.001  ODE sol. =   1.600
 Exact sol.    3.953    2.597    1.612    0.896    0.000  ODE sol. =   1.600

 t =  3.200
 App.  sol.   23.522   11.918    5.807    2.588   -0.004  ODE sol. =   3.197
 Exact sol.   23.533   11.936    5.821    2.597    0.000  ODE sol. =   3.200

 Number of integration steps in time =    642
 Number of function evaluations =   3022
 Number of Jacobian evaluations =    39
 Number of iterations =   1328
```