Module 13.3: nag_pde_parab_1d
Parabolic Partial Differential Equations in One Space Variable

nag_pde_parab_1d provides procedures for the solution of linear or nonlinear parabolic partial differential equations (PDEs), in one space variable with optional scope for coupled ordinary differential equations (ODEs). The spatial discretisation is performed using either a Chebyshev C^0 collocation method or finite differences, and optional automatic adaptive spatial remeshing. The module also provides procedures for the interpolation of the solution obtained using these two methods.

Contents

Introduction

This module provides two procedures `nag_pde_parab_1d_fd` and `nag_pde_parab_1d_coll`, for solving parabolic equations in one-dimensional space. It also provides two procedures `nag_pde_interp_1d_fd` and `nag_pde_interp_1d_coll`, for interpolation of the solutions and derivatives at a set of user-specified points.

The procedures `nag_pde_parab_1d_fd` and `nag_pde_parab_1d_coll` solve systems of parabolic (and possibly elliptic) equations:

$$\sum_{j=1}^{n} P_{i,j}(x,t,U,\frac{\partial U}{\partial x},V)\frac{\partial U_j}{\partial t} + Q_i(x,t,U,\frac{\partial U}{\partial x},V,\dot{V}) = x^{-m}\frac{\partial}{\partial x}(x^m R_i(x,t,U,\frac{\partial U}{\partial x},V)),$$

where $i = 1,2,\ldots,n$, $a \le x \le b$, with $t \ge t_0$ and n is the number of PDEs. There is optional scope to include coupled differential algebraic systems:

$$F_i(t,V,\dot{V},\xi,U^*,\frac{\partial U^*}{\partial x},R^*,\frac{\partial U^*}{\partial t},\frac{\partial^2 U^*}{\partial x \partial t}) = 0,$$

where $i = 1,2,\ldots,p$, $a \le x \le b$, with $t \ge t_0$ and p is the number of ODEs. The first equation defines the PDE part and the second one generalizes the coupled ODE part of the problem. The PDE part may include non-linear terms, but the time derivatives should occur linearly and a second order derivative should normally be present. The procedures in this module can handle Cartesian, cylindrical polar and spherical polar coordinate systems. The procedure `nag_pde_parab_1d_fd` uses a finite difference spatial discretisation while `nag_pde_parab_1d_coll` uses collocation.

In the second equation, 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^*, $\frac{\partial U^*}{\partial x}$, R^*, $\frac{\partial U^*}{\partial t}$, and $\frac{\partial^2 U^*}{\partial x \partial t}$ are the functions U, $\frac{\partial U}{\partial x}$, R, $\frac{\partial U}{\partial t}$, and $\frac{\partial^2 U}{\partial x \partial t}$ evaluated at these coupling points. Each F_i may only depend linearly on time derivatives. Hence the coupled ODE part of the equations may be written more precisely as:

$$F = G - A\dot{V} - B\begin{pmatrix} U_t^* \\ U_{xt}^* \end{pmatrix},$$

where $F = [F_1,\ldots,F_p]^T$, G is a vector of length p, A is a p by p matrix, B is a p by $(n_\xi \times n)$ matrix and the entries in G, A and B may depend on t, ξ, U^*, $\frac{\partial U^*}{\partial x}$, and V. In practice the user only needs to supply a vector of information to define the ODEs and not the matrices A and B. (See Section 3.2 for the specification of the optional user-supplied procedure `ode_coef`).

This extended functionality allows for the solution of more complex and more general problems, such as problems with periodic boundary conditions or integro-differential equations.

The procedure `nag_pde_parab_1d_fd` also allows spatial remeshing. This facility can be very useful when the nature of the solution in the spatial direction varies considerably over time.

In their first call the procedures `nag_pde_parab_1d_fd` and `nag_pde_parab_1d_coll` compute and store a structure of the derived type `nag_pde_parab_1d_comm_wp`. This structure can then be passed to a subsequent call to those procedures respectively.

The procedure `nag_pde_interp_1d_fd` is an interpolation procedure for evaluating the solution and first spatial derivative of a system of partial differential equations (possibly with coupled ordinary differential equations) at a set of user-specified points. The solution must have been computed using a finite difference scheme, for example using `nag_pde_parab_1d_fd`.

The procedure `nag_pde_interp_1d_coll` performs the same function for a solution obtained using `nag_pde_parab_1d_coll`.

Finally the procedure `nag_pde_parab_1d_cntrl_init` assigns default values to all the components of a structure of the derived type `nag_pde_parab_1d_cntrl_wp`, which may be used to supply optional parameters to `nag_pde_interp_1d_fd` and `nag_pde_interp_1d_coll`.

Procedure: nag_pde_parab_1d_fd

1 Description

nag_pde_parab_1d_fd integrates a system of n linear or nonlinear parabolic partial differential equations (PDEs), and optionally coupled ODEs, in one space variable:

$$\sum_{j=1}^{n} P_{i,j}\frac{\partial U_j}{\partial t} + Q_i = x^{-m}\frac{\partial}{\partial x}(x^m R_i),\ i = 1,2,...,n,\ a \le x \le b,\ t \ge t_0, \tag{1}$$

$$F = G - A\dot{V} - B\begin{pmatrix} U_t^* \\ U_{xt}^* \end{pmatrix}, \tag{2}$$

where (1) defines the PDE part which is discretised over a set of n_η mesh points, and (2) generalizes the coupled system of p ODEs which is discretised over a set of n_ξ 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.

In (1), $P_{i,j}$ and R_i depend on x, t, U, $\dfrac{\partial U}{\partial x}$, and V; Q_i depends on x, t, U, $\dfrac{\partial U}{\partial x}$, V and linearly on $\dot{V}$. The vector U is the set of PDE solution values

$$U(x,t) = [U_1(x,t), \ldots, U_n(x,t)]^T,$$

and the vector $\dfrac{\partial U}{\partial x}$ is the partial derivative with respect to x. The vector V is the set of ODE solution values

$$V(t) = [V_1(t), \ldots, V_p(t)]^T,$$

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

In (2) $F = [F_1, \ldots, F_p]^T$, G is a vector of length p, A is a p by p matrix, B is a p by $(n_\xi \times n)$ matrix, and the entries in G, A and B may depend on t, ξ, U^*, $\dfrac{\partial U^*}{\partial x}$, and V. ξ represents the vector of the n_ξ spatial coupling points and U^*, $\dfrac{\partial U^*}{\partial x}$, R^*, $\dfrac{\partial U^*}{\partial t}$, and $\dfrac{\partial^2 U^*}{\partial x \partial t}$ are the functions U, $\dfrac{\partial U}{\partial x}$, R, $\dfrac{\partial U}{\partial t}$, and $\dfrac{\partial^2 U}{\partial x \partial t}$ evaluated at these coupling points. (See the Module Introduction for more details.)

In practice the user only needs to supply a vector of information to define the ODEs and not the matrices A and B. (See Section 3.2 for the specification of the optional user-supplied procedure ode_coef).

The spatial discretisation is performed using finite differences, optionally with automatic adaptive spatial remeshing; 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). The integration in time is from t_0 to t_{out}, over the space interval $a \le x \le b$, where $a = x_1$ and $b = x_{n_\eta}$ are the leftmost and rightmost points of a mesh $x_1, x_2, \ldots, x_{n_\eta}$ defined initially by the user and (possibly) adapted automatically during the integration according to user-specified criteria. The co-ordinate system in space is defined by the following values of m; $m = 0$ for Cartesian co-ordinates, $m = 1$ for cylindrical polar co-ordinates and $m = 2$ for spherical polar co-ordinates.

The PDE system which is defined by the functions $P_{i,j}$, Q_i and R_i must be specified in the user-supplied procedure pde_coef. The initial ($t = t_0$) values of the functions $U(x,t)$ and $V(t)$ must be specified in a procedure init_value supplied by the user. Note that init_value will be called again following any initial remeshing, and so $U(x,t_0)$ should be specified for all values of x in the interval $a \le x \le b$, and not just the initial mesh points. The functions R_i which may be thought of as fluxes, are also used in the definition of the boundary conditions. The boundary conditions must be specified in a procedure bound_cond provided by the user and must have the form:

$$\beta_i(x,t)R_i(x,t,U,U_x,V) = \gamma_i(x,t,U,U_x,V,\dot{V}), \quad i = 1,2,...,n, \tag{3}$$

where $x = a$ or $x = b$. The function γ_i may depend linearly on $\dot{V}$. The algebraic-differential equation system which is defined by the functions F_i must be specified in the user-supplied procedure `ode_coef` (See Section 3.2 and the Further Comments sections for more details).

Several options are available for controlling the operation of this procedure, for example the level of printed output and algorithmic parameters, such as integration method and order, etc.. These options are grouped together in the optional argument `control`, which is a structure of the derived type `nag_pde_parab_1d_cntrl_wp`.

The problem as defined by (1), (2) and (3) is subject to some additional constraints. For details see the Further Comments section.

2 Usage

```
USE nag_pde_parab_1d

CALL nag_pde_parab_1d_fd(pde_coef, bound_cond, init_value, t_start, t_end, x_pde, &
                         u  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '`x`(n)' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$n_\eta \geq 3$ — (= `SIZE(x_pde)`) the number of PDE mesh points in the interval $[a, b]$;
$n_\xi \geq 0$ — (0 or `SIZE(x_ode)` if present) the number of ODE/PDE coupling points, if $p = 0$ then $n_\xi = 0$ and if $p > 0$ then $n_\xi \geq 0$;
$0 \leq n_f \leq n_\eta - 2$ — (0 or `SIZE(x_fix)` if present) the number of fixed PDE mesh points;
$p \geq 0$ — the number of coupled ODE (see the optional argument `num_ode`);
$n \geq 1$ — the number of PDEs in the system (`SIZE(u)` $= n \times n_\eta + p$).

3.1 Mandatory Arguments

pde_coef — subroutine

The procedure `pde_coef`, supplied by the user, must compute the functions $P_{i,j}$, Q_i and R_i which define the system of PDEs as in equation (1). The functions may depend on x, t, U, $\dfrac{\partial U}{\partial x}$ and V. Q_i may depend linearly on $\dot{V}$. The procedure `pde_coef` is called approximatively midway between each pair of mesh points in turn by `nag_pde_parab_1d_fd`.

Its specification is:

```
subroutine pde_coef(t, x, u, du_dx, p, q, r, finish, v, vdot, i_comm, r_comm)

real(kind=wp), intent(in) ::  t
```
Input: the current value of the independent variable t.

```
real(kind=wp), intent(in) ::  x
```
Input: the current value of the spatial variable x.

```
real(kind=wp), intent(in) ::  u(:)
```
Shape: `u` has shape (n).
Input: `u`(i) contains the value of the component $U_i(x, t)$, for $i = 1, 2, \ldots, n$.

`real(kind=wp), intent(in) :: du_dx(:)`

Shape: `du_dx` has shape (n).

Input: `du_dx`(i) contains the value of the component $\dfrac{\partial U_i}{\partial x}(x,t)$, for $i = 1,2,\ldots,n$.

`real(kind=wp), intent(out) :: p(:,:)`

Shape: `p` has shape (n,n).

Output: `p`(i,j) must be set to the values of $P_{i,j}(x,t,U,\dfrac{\partial U}{\partial x},V)$, for $i,j = 1,2,\ldots,n$.

`real(kind=wp), intent(out) :: q(:)`

Shape: `q` has shape (n).

Output: `q`(i) must be set to the values of $Q_i(x,t,U,\dfrac{\partial U}{\partial x},V,\dot{V})$, for $i = 1,2,\ldots,n$.

`real(kind=wp), intent(out) :: r(:)`

Shape: `r` has shape (n).

Output: `r`(i) must be set to the values of $R_i(x,t,U,\dfrac{\partial U}{\partial x},V)$, for $i = 1,2,\ldots,n$.

`integer, intent(out) :: finish`

Output: the user may use `finish` to control the integration procedure as follows:

if `finish` $= 1$ or -1, indicates normal exit from the user supplied procedure;

if `finish` $= 2$, indicates to the integrator that control should be passed back immediately to the calling (sub)program with the error indicator set to `error%code` $= 205$;

if `finish` $= 3$, 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 `finish` $= 3$ when a physically meaningless input or output value has been generated. If the user consecutively sets `finish` $= 3$, then `nag_pde_parab_1d_fd` returns to the calling (sub)program with the error indicator set to `error%code` $= 203$;

if `finish` = any value other than 1, -1, 2 or 3, indicates an abnormal exit from the user supplied procedure, and returns to the calling (sub)program with the error indicator set to `error%code` $= 207$.

`real(kind=wp), intent(in), optional :: v(:)`

Shape: `v` has shape (p).

Input: `v`(i) contains the value of the component $V_i(t)$, for $i = 1,2,\ldots,p$.

`real(kind=wp), intent(in), optional :: vdot(:)`

Shape: `vdot` has shape (p).

Input: `vdot`(i) contains the value of the component $\dot{V}_i(t)$, for $i = 1,2,\ldots,p$.

Note: $\dot{V}_i(t)$ for $i = 1,2,\ldots,p$ may only appear linearly in Q_j, for $j = 1,2,\ldots,n$.

`integer, intent(in), optional :: i_comm(:)`

`real(kind=wp), intent(in), optional :: r_comm(:)`

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

bound_cond — subroutine

The procedure `bound_cond`, supplied by the user, must compute the functions β_i and γ_i which define the boundary conditions as in equation (3).

Its specification is:

```
subroutine bound_cond(t, u, du_dx, bound, beta, gamma, finish, v, vdot, &
                      i_comm, r_comm)
```

`real(kind=wp), intent(in) :: t`

Input: the current value of the independent variable t.

`real(kind=wp), intent(in) :: u(:)`

Shape: `u` has shape (n).

Input: `u`(i) contains the value of the component $U_i(x,t)$ at the boundary specified by `bound`, for $i = 1, 2, \ldots, n$.

`real(kind=wp), intent(in) :: du_dx(:)`

Shape: `du_dx` has shape (n).

Input: `du_dx`(i) contains the value of the component $\dfrac{\partial U_i}{\partial x}(x,t)$ at the boundary specified by `bound`, for $i = 1, 2, \ldots, n$.

`integer, intent(in) :: bound`

Input: determines the position of the boundary conditions. If `bound` $= 0$, then the procedure `bound_cond` must set up the coefficients of the left-hand boundary $x = a$. Any other value of `bound` indicates that the procedure `bound_cond` must set up the coefficients of the right-hand boundary $x = b$.

`real(kind=wp), intent(out) :: beta(:)`

Shape: `beta` has shape (n).

Output: `beta`(i) must be set to the values of $\beta_i(x,t)$ at the boundary specified by `bound`, for $i = 1, 2, \ldots, n$.

`real(kind=wp), intent(out) :: gamma(:)`

Shape: `gamma` has shape (n).

Output: `gamma`(i) must be set to the values of $\gamma_i(x, t, U, \dfrac{\partial U}{\partial x}, V, \dot{V})$ at the boundary specified by `bound`, for $i = 1, 2, \ldots, n$.

`integer, intent(out) :: finish`

Output: the user may use `finish` to control the integration procedure as follows:

if `finish` $= 1$ or -1, indicates normal exit from the user supplied procedure;

if `finish` $= 2$, indicates to the integrator that control should be passed back immediately to the calling (sub)program with the error indicator set to `error%code` $= 205$;

if `finish` $= 3$, 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 `finish` $= 3$ when a physically meaningless input or output value has been generated. If the user consecutively sets `finish` $= 3$, then `nag_pde_parab_1d_fd` returns to the calling (sub)program with the error indicator set to `error%code` $= 203$;

if `finish` $=$ any value other than 1, -1, 2 or 3, indicates an abnormal exit from the user supplied procedure, and returns to the calling (sub)program with the error indicator set to `error%code` $= 207$.

`real(kind=wp), intent(in) :: v(:)`

Shape: `v` has shape (p).

Input: `v`(i) contains the value of the component $V_i(t)$, for $i = 1, 2, \ldots, p$.

`real(kind=wp), intent(in) :: vdot(:)`

Shape: `vdot` has shape (p).

Input: `vdot`(i) contains the value of the component $\dot{V}_i$, for $i = 1, 2, \ldots, p$.

Note: $\dot{V}_i(t)$ for $i = 1, 2, \ldots, p$ may only appear linearly in γ_j, for $j = 1, 2, \ldots, n$.

`integer, intent(in), optional :: i_comm(:)`

`real(kind=wp), intent(in), optional :: r_comm(:)`

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

init_value — subroutine

The procedure `init_value`, supplied by the user, must supply the initial ($t = t_0$) values of $U(x, t)$ and $V(t)$ for all values of x in the interval $a \leq x \leq b$.

Its specification is:

`subroutine init_value(x_pde, u, x_ode, v, i_comm, r_comm)`

`real(kind=wp), intent(in) :: x_pde(:)`

Shape: `x_pde` has shape (n_η).

Input: the current mesh. `x_pde`(i) contains the value of the x_i for $i = 1, 2, \ldots, n_\eta$.

`real(kind=wp), intent(out) :: u(:,:)`

Shape: `x_pde` has shape (n, n_η).

Output: `u`(i, j) must contain the value of component $U_i(x_j, t_0)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\eta$.

`real(kind=wp), intent(in), optional :: x_ode(:)`

Shape: `x_ode` has shape (n_ξ).

Input: `x_ode`(i) contains the value of the ODE/PDE coupling point, ξ_i, for $i = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(out), optional :: v(:)`

Shape: `v` has shape (p).

Output: `v`(i) must contain the value of component $V_i(t_0)$, for $i = 1, 2, \ldots, p$.

`integer, intent(in), optional :: i_comm(:)`

`real(kind=wp), intent(in), optional :: r_comm(:)`

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

t_start — real(kind=*wp*), intent(inout)

Input: the initial value t_0 of the independent variable t.

Output: the value of t corresponding to the solution value U. Normally `t_start` = `t_end` on exit.

Constraints: `t_start` < `t_end`.

t_end — real(kind=*wp*), intent(in)

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

x_pde(n_η) — real(kind=*wp*), intent(inout)

Input: the initial mesh points in the spatial direction. `x_pde`(1) must specify the left-hand boundary, a, and `x_pde`(n_η) must specify the right-hand boundary, b.

Output: the final values of the mesh points.

Constraints: `x_pde`(1) $<$ `x_pde`(2) $< \cdots <$ `x_pde`(n_η).

u($n \times n_\eta + p$) — real(kind=*wp*), intent(inout)

Input: `u`($n \times (j-1) + i$) contains the computed solution $U_i(x_j, t)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\eta$, and `u`($n \times n_\eta + k$) contains $V_k(t)$, for $k = 1, 2, \ldots, p$; evaluated on a previous call to the procedure. The user does not need to initialialise `u` at the very first call of the procedure.

Output: `u`($n \times (j-1) + i$) contains the computed solution $U_i(x_j, t)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\eta$, while `u`($n \times n_\eta + k$) contains $V_k(t)$, for $k = 1, 2, \ldots, p$; evaluated at the output value of $t =$ `t_start`.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

first_call — logical, intent(in), optional

Input: indicates whether the time integration is a continuation of a previous step or not:

if `first_call` = .`true`., starts or restarts the integration in time;

if `first_call` = .`false`., continues the integration after an earlier exit from the procedure. In this case, only the parameter `t_end` and the remeshing optional arguments `n_remesh`, `dx_mesh`, `t_remesh`, `ratio_mesh` and `bound_int` should be reset between calls to `nag_pde_parab_1d_fd`.

Default: `first_call` = .`false`..

ode_coef — subroutine, optional

The procedure `ode_coef`, supplied by the user, must evaluate the functions F_i, which define the system of ODEs, as given in (2). If p is set to 0, `ode_coef` should not be present.

Its specification is:

```
subroutine ode_coef(t, v, vdot, x_ode, ucp, ducp_dx, rcp, ducp_dt, &
                    d2ucp_dtdx, f, finish, g_in_f, i_comm, r_comm)

real(kind=wp), intent(in) ::  t
```

Input: the current value of the independent variable t.

```
real(kind=wp), intent(in) ::  v(:)
```

Shape: `v` has shape (p).

Input: `v`(i) contains the value of component $V_i(t)$, for $i = 1, 2, \ldots, p$.

```
real(kind=wp), intent(in) ::  vdot(:)
```

Shape: `vdot` has shape (p).

Input: `vdot`(i) contains the value of component $\dot{V}_i$, for $i = 1, 2, \ldots, p$.

```
real(kind=wp), intent(in) ::  x_ode(:)
```

Shape: `x_ode` has shape (n_ξ).

Input: `x_ode`(i) contains the value of the ODE/PDE coupling point, ξ_i, for $i = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: ucp(:,:)`

Shape: `ucp` has shape (n, n_ξ).

Input: `ucp`(i,j) contains the value of $U_i(x,t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: ucp_dx(:,:)`

Shape: `ucp_dx` has shape (n, n_ξ).

Input: `ucp_dx`(i,j) contains the value of $\dfrac{\partial U_i}{\partial x}(x,t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: rcp(:,:)`

Shape: `rcp` has shape (n, n_ξ).

Output: `rcp`(i,j) contains the value of R_i at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: ucp_dt(:,:)`

Shape: `ucp_dt` has shape (n, n_ξ).

Output: `ucp_dt`(i,j) contains the value of $\dfrac{\partial U_i}{\partial t}(x,t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: d2ucp_dtdx(:,:)`

Shape: `d2ucp_dtdx` has shape (n, n_ξ).

Output: `d2ucp_dtdx`(i,j) contains the value of $\dfrac{\partial^2 U_i}{\partial x \partial t}(x,t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(out) :: f(:)`

Shape: `f` has shape (p).

Output: `f`(i) must contain the i^{th} component of F, for $i = 1, 2, \ldots, p$; where F is defined (see the optional argument `g_in_f`) as follows:

$$F = G - A\dot{V} - B\begin{pmatrix} U_t^* \\ U_{xt}^* \end{pmatrix},$$

or

$$F = -A\dot{V} - B\begin{pmatrix} U_t^* \\ U_{xt}^* \end{pmatrix}.$$

The definition of F is determined by the input value of the optional argument `g_in_f`.

`integer, intent(out) :: finish`

Output: the user may use `finish` to control the integration procedure as follows:

if `finish` $= 1$ or -1, indicates normal exit from the user supplied procedure;

if `finish` $= 2$, indicates to the integrator that control should be passed back immediately to the calling (sub)program with the error indicator set to `error%code` $= 205$;

if `finish` $= 3$, 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 `finish` $= 3$ when a physically meaningless input or output value has been generated. If the user consecutively sets `finish` $= 3$, then `nag_pde_parab_1d_fd` returns to the calling (sub)program with the error indicator set to `error%code` $= 203$;

if `finish` $=$ any value other than 1, -1, 2 or 3, indicates an abnormal exit from the user supplied procedure, and returns to the calling (sub)program with the error indicator set to `error%code` $= 207$.

`logical, intent(in), optional :: g_in_f`

Input: indicates which of forms of `f` is returned (see the argument description of `f`). If `g_in_f` = .true., then the first equation above must be used; and if `g_in_f` = .false., then the second equation above must be used.

Default: `g_in_f` = .false..

`integer, intent(in), optional :: i_comm(:)`

`real(kind=wp), intent(in), optional :: r_comm(:)`

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

Constraints: `ode_coef` must not be present unless `num_ode` is present with a non zero value.

num_ode — integer, intent(in), optional

Input: the number of coupled ODEs in the system, p.

Constraints: `num_ode` ≥ 0.

Default: `num_ode` $= 0$.

x_ode(n_ξ) — real(kind=*wp*), intent(in), optional

Input: `x_ode`(i), for $i = 1, 2, \ldots, n_\xi$, must be set to the ODE/PDE coupling points ξ_i.

Constraints:

If n_ξ is set to 0, `x_ode` should not be present;

`x_pde`$(1) \leq$ `x_ode`$(1) <$ `x_ode`$(2) < \cdots <$ `x_ode`$(n_\xi) \leq$ `x_pde`(n_η).

comm_ode — type(nag_pde_parab_1d_comm_*wp*), intent(inout), optional

Input: a structure containing data for the underlying ODE solver between consecutive calls to the procedure.

Constraints:

if `first_call` = .true., `comm_ode` should be present if a further start is planned;

if `first_call` = .false., this argument should be present and should be the same as returned from the previous call.

Output: a structure containing information about the underlying ODE solver required for a further call. This structure may be passed to `nag_pde_parab_1d_fd` for the continuation of the time integration.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private. The procedure allocates a certain amount of real(kind=*wp*) and integer elements of storage to the structure (those dimensions may be accessed via the optional argument `dim_struct`). If you wish to deallocate this storage when the structure is no longer required, you must call the procedure `nag_deallocate`, as illustrated in Example 1 of this module document.

coord_sys — character(len=1), intent(in), optional

Input: indicates which co-ordinate system (the index m) is being used:

if `coord_sys` = `'C'` or `'c'`, indicates Cartesian co-ordinates ($m = 0$);

if `coord_sys` = `'P'` or `'p'`, indicates cylindrical polar co-ordinates ($m = 1$);

if `coord_sys` = `'S'` or `'s'`, indicates spherical polar co-ordinates ($m = 2$).

Constraints:

`coord_sys` should be one of the following values `'C'`, `'c'`, `'P'`, `'p'`, `'S'` or `'s'`;

if `coord_sys` is one of the following values `'P'`, `'p'`, `'S'` or `'s'` (i.e., $m > 0$) then `x_pde`(1) should be ≥ 0.

Default: `coord_sys` = `'C'`.

rel_loc_tol($n \times n_\eta + p$) — real(kind=*wp*), intent(in), optional

Input: the relative local error tolerance used in the local error test. The error test to be satisfied is $\|e_i/w_i\| < 1.0$, where w_i is,

$$w_i = \alpha_i \times |U_i| + \epsilon_i, \quad i = 1, 2, \ldots, n \times n_\eta + p;$$

the e_i denotes the estimated local error for the i^{th} component of the coupled PDE/ODE system and `rel_loc_tol`$(i) = \alpha_i$, for $i = 1, 2, \ldots, n \times n_\eta + p$.

Constraints: `rel_loc_tol`$(i) \geq 0.0$, for all $i = 1, 2, \ldots, n \times n_\eta + p$.

Default: `rel_loc_tol`(i) = `EPSILON(1.0_wp)`, for all $i = 1, 2, \ldots, n \times n_\eta + p$.

abs_loc_tol($n \times n_\eta + p$) — real(kind=*wp*), intent(in), optional

Input: the absolute local error tolerance used in the local error test; `abs_loc_tol`$(i) = \epsilon_i$, for $i = 1, 2, \ldots, n \times n_\eta + p$.

Constraints: `abs_loc_tol`$(i) \geq 0.0$, for all $i = 1, 2, \ldots, n \times n_\eta + p$, and corresponding elements of `abs_loc_tol` and `rel_loc_tol` (if present) cannot both be 0.

Default: `abs_loc_tol`(i) = `EPSILON(1.0_wp)`, for all $i = 1, 2, \ldots, n \times n_\eta + p$.

remesh — logical, intent(in), optional

Input: indicates whether or not spatial remeshing should be performed.

If `remesh` = .`true`., indicates that spatial remeshing should be performed as specified.

If `remesh` = .`false`., indicates that spatial remeshing should be suppressed.

Note: `remesh` should not be changed between consecutive calls to the procedure `nag_pde_parab_1d_fd`. Remeshing can be switched off or on at specified times by using appropriate values for the optional arguments `n_remesh` and `t_remesh` at each call.

Default: `remesh` = .`false`..

x_fix(n_f) — real(kind=*wp*), intent(in), optional

Input: `x_fix`(i), for $i = 1, 2, \ldots, n_f$, must contain the value of the x coordinate at the i^{th} fixed mesh point.

Note: the end points `x_pde`(1) and `x_pde`(n_η) are fixed automatically and hence should not be specified as fixed points. the position of the fixed mesh points in the array `x_pde` remain fixed during remeshing, and so the number of mesh points between adjacent fixed points (or between fixed points and end points) does not change. The user should take this in account when choosing the initial mesh distribution.

Constraints:

if `remesh` is not present or = .`false`. (i.e., no remeshing), or $n_f = 0$, then `x_fix` should not be present;

`x_fix`$(i) <$ `x_fix`$(i + 1)$, for $i = 1, 2, \ldots, n_f - 1$;

each fixed mesh point must coincide with a user supplied initial mesh point, that is `x_fix`(i) = `x_pde`(j), for some j, $2 \leq j \leq n_\eta - 1$.

n_remesh — integer, intent(in), optional

Input: specifies the spatial remeshing frequency and criteria for the calculation and adoption of a new mesh:

if `n_remesh` < 0, indicates that a new mesh is adopted according to the optional argument `dx_mesh` below. The mesh is tested every ABS(`n_remesh`) time steps;

if `n_remesh` $= 0$, indicates that remeshing should take place just once at the end of the first time step reached when $t >$ `t_remesh` (see below);

if `n_remesh` > 0, indicates that remeshing will take place every `n_remesh` time steps, with no testing using `dx_mesh`.

Note: n_remesh may be changed between consecutive calls to nag_pde_parab_1d_fd to give greater flexibility over the times of remeshing.

Constraints: n_remesh should not be present unless remeshing is to be performed i.e., remesh is present with value .true..

Default: n_remesh = 0.

dx_mesh — real(kind=*wp*), intent(in), optional

Input: determines whether a new mesh is adopted when n_remesh is set less than zero. A possible new mesh is calculated at the end of every |n_remesh| time steps, but is adopted only if:

$$x_i^{new} > x_i^{old} + \texttt{dx\_mesh} \times (x_{i+1}^{old} - x_i^{old})$$

or

$$x_i^{new} < x_i^{old} - \texttt{dx\_mesh} \times (x_i^{old} - x_{i-1}^{old})$$

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

Constraints:

dx_mesh should not be present unless remeshing is to be performed i.e., remesh is present with value .true.,

dx_mesh ≥ 0.0.

Default: dx_mesh = 0.0.

t_remesh — real(kind=*wp*), intent(in), optional

Input: specifies when remeshing will take place if n_remesh is set to zero. Remeshing will occur just once at the end of the first time step reached when t is greater than t_remesh.

Note: t_remesh may be changed between consecutive calls to nag_pde_parab_1d_fd to force remeshing at several specified times.

Constraints: t_remesh should not be present unless remeshing is to be performed i.e., remesh is present with value .true..

Default: t_remesh = t_end.

ratio_mesh — real(kind=*wp*), intent(in), optional

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

$$(x_i - x_{i-1})/\texttt{ratio\_mesh} < x_{i+1} - x_i < \texttt{ratio\_mesh} \times (x_i - x_{i-1}).$$

Constraints:

if remesh is not present or = .false., i.e., no remeshing, then ratio_mesh should not be present,

ratio_mesh > 1.0.

Default: ratio_mesh = 1.5.

bound_int — real(kind=*wp*), intent(in), optional

Input: an input bound on the sub-integral of the monitor function $F^{mon}(x)$ over each space step. The remeshing procedures will attempt to ensure that,

$$\int_{x_i}^{x_{i+1}} F^{mon}(x)dx \leq \texttt{bound\_int} \int_{x_1}^{x_{n_\eta}} F^{mon}(x)dx,$$

(see Furzeland [10]). bound_int gives the user more control over the mesh distribution, e.g., decreasing bound_int allows more clustering. A typical value is $2.0/(n_\eta - 1)$, but the user is encouraged to experiment with different values. Its value is not critical and the mesh should be qualitatively correct for all values in the range given below.

Constraints:

if remesh is not present or = .false., i.e., no remeshing, then bound_int should not be present, $0.1/(n_\eta - 1) \leq$ bound_int $\leq 10.0/(n_\eta - 1)$.

Default: bound_int $= 2.0/(n_\eta - 1)$.

remesh_fun — subroutine, optional

The procedure remesh_fun, supplied by the user, must supply and evaluate a remesh monitor function to indicate the solution behaviour of interest. If the optional argument remesh is not present or if the user specifies remesh = .false., i.e., no remeshing, then remesh_fun should not be present.

Its specification is:

```
subroutine remesh_fun(t, x_pde, u, r, fmon, i_comm, r_comm)

real(kind=wp), intent(in) :: t
```

Input: the current value of the independent variable t.

```
real(kind=wp), intent(in) :: x_pde(:)
```

Shape: x_pde has shape (n_η).

Input: the current mesh. x_pde(i) contains the value of the x_i for $i = 1, 2, \ldots, n_\eta$.

```
real(kind=wp), intent(in) :: u(:,:)
```

Shape: u has shape (n, n_η).

Input: u(i, j) contains the value of $U_i(x, t)$ at $x =$ x_pde(j), for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\eta$.

```
real(kind=wp), intent(in) :: r(:,:)
```

Shape: r has shape (n, n_η).

Input: r(i, j) contains the value of $R_i(x, t, U, \dfrac{\partial U}{\partial x}, V)$ at $x =$ x_pde(j), for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\eta$.

```
real(kind=wp), intent(out) :: fmon(:)
```

Shape: fmon has shape (n_η).

Output: fmon(i) must contain the value of the monitor function F^{mon} at mesh point $x =$ x_pde(i), for $i = 1, 2, \ldots, n_\eta$.

Constraints: fmon$(i) \geq 0$.

```
integer, intent(in), optional :: i_comm(:)
real(kind=wp), intent(in), optional :: r_comm(:)
```

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

control — type(nag_pde_parab_1d_cntrl_*wp*), intent(in), optional

Input: a structure containing scalar components; these are used to alter the default values of those parameters which control the behaviour of the algorithm and the level of printed output. The initialization of this structure and its use is described in the procedure document for nag_pde_parab_1d_cntrl_init.

dim_struct(2) — integer, intent(out), optional

Output: the dimension of the real and integer arrays in the structure `comm_ode`, are stored in `dim_struct`(1) and `dim_struct`(2), respectively.

num_time_step — integer, intent(out), optional

Output: the number of steps taken in time.

num_resid_eval — integer, intent(out), optional

Output: the number of residual evaluations of the resulting ODE system used. One such evaluation involves the PDE functions at all mesh points, as well as one evaluation of the functions in the boundary conditions.

num_jac_eval — integer, intent(out), optional

Output: the number of Jacobian evaluations performed by the time integrator.

num_time_iter — integer, intent(out), optional

Output: 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.

i_comm(:) — integer, intent(in), optional

r_comm(:) — real(kind=*wp*), intent(in), optional

Input: these arrays are not used by this procedure, but they are passed directly from the calling (sub)program to the user supplied procedures `pde_coef`, `bound_cond`, `init_value`, `ode_coef` and/or `remesh_fun`, and hence may be used to pass information to them.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
304	Invalid presence of an optional argument.
305	Invalid absence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	The underlying ODE solver cannot make any further progress from the current point $t =$ t_start, with the given values of rel_loc_tol and abs_loc_tol. The components of U contain the computed values at the current point $t =$ t_start.
202	In the underlying ODE solver, there were repeated error or corrector convergence test failures on an attempted step, before completing the requested task. The problem may have a singularity, or the error requirement may be inappropriate. Integration was successful as far as $t =$ t_start.
203	In setting up the ODE system, the internal initialisation procedure was unable to initialise the derivative of the ODE system. This could be due to the fact that finish was **repeatedly** set to 3 in the user supplied procedures pde_coef, bound_cond or ode_coef, when the residual in the underlying ODE solver was being evaluated.
204	In solving the ODE system a singular Jacobian has been encountered. Check your problem formulation.
205	When evaluating the residual in the ODE system, finish was set to 2 in at least one of the user supplied procedures pde_coef, bound_cond or ode_coef. Integration was successful as far as $t =$ t_start.
206	The value of rel_loc_tol and abs_loc_tol are so small that the procedure is unable to start the integration in time.
207	In one of the user supplied procedures pde_coef, bound_cond or ode_coef, finish was set to an invalid value.
208	A serious error has occurred in an internal call to an integrator for stiff ODE procedure. Check problem specification and all parameters and array dimensions. Setting control%print_level_ode = 1 may provide more information. If the problem persists, contact NAG.
209	An error occurred during Jacobian formulation of the ODE system. A more detailed error description may be directed to the current advisory message unit. If using the sparse matrix algebra option, the values of control%first_piv_jac and control%rel_piv_thr may be inappropriate.
210	Some error weights w_i became zero during the time integration (see description of abs_loc_tol). Pure relative error control (abs_loc_tol(i) = 0.0) was requested on a variable (the i^{th}) which has become zero. Integration was successful as far as $t =$ t_start.
211	The flux function R_i was detected as depending on time derivatives, which is not permissible.
212	remesh has been changed between calls to the procedure nag_pde_parab_1d_fd, which is not permissible.
213	The remeshing process has produced zero or negative mesh-spacing. The user is advised to check the user-supplied monitor procedure remesh_fun and to try adjusting the values of the optional arguments dx_mesh, ratio_mesh and bound_init. Setting control%print_level_remesh = 1 may provide more information.

Warnings (error%level = 1):

error%code	Description
101	The required task has been completed, but it is estimated that a small change in `rel_loc_tol` and `abs_loc_tol` is unlikely to produce any change in the computed solution. Only applies when the user is not operating in one step mode, that is when `control%task` $\neq 2$ or 5.
102	In solving the ODE system, the maximum number of steps specified in `control%num_step_max` has been taken.
103	`control%unit` is different from its default value while no output is required (`control%print_level_remesh` and `control%print_level_ode` have their default value). `control%unit` will not be used.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 and 3 of this module document. The user may find other examples in the additional examples section (Examples 4, 6, 7 and 8).

Assume that all relevant arguments have been declared correctly as described in Section 3, and that input and input/output arguments have been appropriately initialized. The following example illustrates the use of the optional argument `control` in order to solve a parabolic PDE using a finite difference discretisation.

The value of `control%print_level_ode` = 1 generates warning messages from the PDE solver.

```
   ...
! Initialize control structure
CALL nag_pde_parab_1d_cntrl_init(control)

! Set required value
control%print_level_ode = 1

CALL nag_pde_parab_1d_fd(pde_coef, bound_cond, init_value, t_start, &
                 t_end, x_pde, u, control=control)
   ...
```

6 Further Comments

6.1 Additional Problem Constraints

The problem is subject to the following restrictions:

In (1), $\dot{V}_j(t)$, for $j = 1, 2, \ldots, p$, may only appear linearly in the functions Q_i, for $i = 1, 2, \ldots, n$, with a similar restriction for γ_i in (3);

$P_{i,j}$ and the flux R_i must not depend on any time derivatives;

$t_0 < t_{out}$, so that integration is in the forward direction;

the evaluation of the terms $P_{i,j}$, Q_i and R_i is done approximately at the mid-points of the mesh `x_pde`(i), for $i = 1, 2, \ldots, n_\eta$, by calling the procedure `pde_coef` for each mid-point in turn. Any discontinuities in these functions **must** therefore be at one or more of the fixed mesh points specified by `x_fix`;

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;

If $m > 0$ then x_1 which is the left boundary point, must be non-negative. Furthermore, if $m > 0$ and $x_1 = 0.0$, then it must be ensured that the PDE solution is bounded at this point. This can be done either by specifying the solution at $x = 0.0$ or by specifying a zero flux there, that is $\beta_i = 1.0$ and $\gamma_i = 0.0$.

6.2 Algorithmic Detail

The parabolic equations are approximated by a system of ODEs in time for the values of U_i at mesh points. For simple problems in Cartesian co-ordinates, this system is obtained by replacing the space derivatives by the usual central, three-point finite-difference formula. However, for polar and spherical problems, or problems with nonlinear coefficients, the space derivatives are replaced by a modified three-point formula which maintains second order accuracy. In total there are $n \times n_\eta + p$ ODEs in the time direction. This system is then integrated forwards in time using a Backward Differentiation Formula (BDF) or a Theta method.

The adaptive special remeshing can be used to generate meshes that automatically follow the changing time-dependent nature of the solution, generally resulting in a more efficient and accurate solution using fewer mesh points than may be necessary with a fixed uniform or non-uniform mesh. Problems with travelling wavefronts or variable-width boundary layers for example will benefit from using a moving adaptive mesh. The discrete time-step method used here (developed by Furzeland [10]) automatically creates a new mesh based on the current solution profile at certain time-steps, and the solution is then interpolated onto the new mesh and the integration continues.

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

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

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

Procedure: nag_pde_interp_1d_fd

1 Description

This procedure interpolates in the spatial co-ordinate the solution and its first spatial derivative of a system of partial differential equations on an interval $[a; b]$; at a set of user-specified points. The solution must be computed using a finite difference scheme, and this procedure will normally be used in conjunction with `nag_pde_parab_1d_fd`.

2 Usage

```
USE nag_pde_parab_1d
CALL nag_pde_interp_1d_fd(u, x_pde, x_interp, u_interp  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$n_\eta \geq 3$ — (= `SIZE(x_pde)`) the number of PDE mesh points in the interval $[a, b]$;

$n \geq 1$ — the number of PDEs in the system (`SIZE(u)` $= n \times n_\eta$);

$n_{interp} \geq 1$ — the number of interpolation points in the interval $[a, b]$;

$i_{type} = 1$ or 2 — specifies the interpolation to be performed:

if $i_{type} = 1$, the solution at the interpolation points is computed,

if $i_{type} = 2$, both the solution and its first derivative at the interpolation points are computed.

3.1 Mandatory Arguments

u($n \times n_\eta$) — real(kind=*wp*), intent(in)

Input: `u`$(n \times (j-1) + i)$ must contain the computed solution $U_i(x_j)$ to be interpolated, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\eta$.

x_pde(n_η) — real(kind=*wp*), intent(in)

Input: the initial mesh points in the spatial direction. `x_pde`(1) must specify the left-hand boundary, a, and `x_pde`(n_η) must specify the right-hand boundary, b.

Constraints: `x_pde`$(1) <$ `x_pde`$(2) < \cdots <$ `x_pde`(n_η).

x_interp(n_{interp}) — real(kind=*wp*), intent(in)

Input: must contain the spatial interpolation points.

Constraints: `x_pde`$(1) \leq$ `x_interp`$(1) <$ `x_interp`$(2) < \cdots <$ `x_interp`$(n_{interp}) \leq$ `x_pde`(n_η).

u_interp(n, n_{interp}, i_{type}) — real(kind=*wp*), intent(out)

Output:

if $i_{type} = 1$, `u_interp`$(i,j,1)$ contains the value of the solution $U_i(x_j)$ at the interpolation point $x_j =$ `x_interp`(j), for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_{interp}$;

if $i_{type} = 2$, `u_interp`$(i,j,1)$ and `u_interp`$(i,j,2)$ contain respectively the value of the solution $U_i(x_j)$ and the derivative $\dfrac{\partial U_i}{\partial x}(x_j)$ at the interpolation point $x_j =$ `x_interp`(j), for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_{interp}$.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

coord_sys — character(len=1), intent(in), optional

Input: indicates which co-ordinate system (the index m) is being used:

if `coord_sys` = `'C'` or `'c'`, indicates Cartesian co-ordinates ($m = 0$);

if `coord_sys` = `'P'` or `'p'`, indicates cylindrical polar co-ordinates ($m = 1$);

if `coord_sys` = `'S'` or `'s'`, indicates spherical polar co-ordinates ($m = 2$).

Constraints: `coord_sys` should be one of the following values `'C'`, `'c'`, `'P'`, `'p'`, `'S'` or `'s'`.

Default: `coord_sys` = `'C'`.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.

5 Examples of Usage

Complete examples of the use of this procedure appear in Example 1 of this module document. The user may find other examples in the additional examples section (Examples 4, 5 and 8).

Procedure: nag_pde_parab_1d_coll

1 Description

`nag_pde_parab_1d_coll` integrates a system of n linear or nonlinear parabolic partial differential equations (PDEs), and optionally coupled in ODEs, in one space variable:

$$\sum_{j=1}^{n} P_{i,j}\frac{\partial U_j}{\partial t} + Q_i = x^{-m}\frac{\partial}{\partial x}(x^m R_i),\ i = 1,2,...,n,\ a \le x \le b,\ t \ge t_0, \tag{4}$$

$$F = G - A\dot{V} - B\begin{pmatrix} U_t^* \\ U_{xt}^* \end{pmatrix}, \tag{5}$$

where (4) defines the PDE part which is discretised over a set of n_b collocation points (break-points), and (5) generalizes the coupled system of p ODEs which is discretised over a set of n_ξ 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 collocation points.

In (4), $P_{i,j}$ and R_i depend on x, t, U, $\dfrac{\partial U}{\partial x}$, and V; Q_i depends on x, t, U, $\dfrac{\partial U}{\partial x}$, V and linearly on $\dot{V}$. The vector U is the set of PDE solution values

$$U(x,t) = [U_1(x,t), \ldots, U_n(x,t)]^T,$$

and the vector $\dfrac{\partial U}{\partial x}$ is the partial derivative with respect to x. The vector V is the set of ODE solution values

$$V(t) = [V_1(t), \ldots, V_p(t)]^T,$$

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

In (5) $F = [F_1, \ldots, F_p]^T$, G is a vector of length p, A is a p by p matrix, B is a p by $(n_\xi \times n)$ matrix, and the entries in G, A and B may depend on t, ξ, U^*, $\dfrac{\partial U^*}{\partial x}$, and V. ξ represents a vector of n_ξ spatial coupling points and U^*, $\dfrac{\partial U^*}{\partial x}$, R^*, $\dfrac{\partial U^*}{\partial t}$, and $\dfrac{\partial^2 U^*}{\partial x \partial t}$ are the functions U, $\dfrac{\partial U}{\partial x}$, R, $\dfrac{\partial U}{\partial t}$, and $\dfrac{\partial^2 U}{\partial x \partial t}$ evaluated at these coupling points (See the Module Introduction for more details).

In practice the user only needs to supply a vector of information to define the ODEs and not the matrices A and B. (See Section 3.2 for the specification of the optional user-supplied procedure `ode_coef`).

The spatial discretisation is performed using a Chebyshev C^0 collocation method (with Chebyshev polynomial of degree d), 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 method or a Theta method. The integration in time is from t_0 to t_{out}, over the space interval $a \le x \le b$, where $a = x_1$ and $b = x_{n_b}$ are the leftmost and rightmost of a user-defined set of break-points $x_1, x_2, \ldots, x_{n_b}$. The co-ordinate system in space is defined by the value of m; $m = 0$ for Cartesian co-ordinates, $m = 1$ for cylindrical polar co-ordinates and $m = 2$ for spherical polar co-ordinates.

The PDE system which is defined by the functions $P_{i,j}$, Q_i and R_i must be specified in a procedure `pde_coef` supplied by the user. The initial values of the functions $U(x,t)$ and $V(t)$ must be given at $t = t_0$. These values are calculated in a user-supplied procedure, `init_value`. The functions R_i which may be thought of as fluxes, are also used in the definition of the boundary conditions. The boundary conditions must be specified in a procedure `bound_cond` provided by the user, and must have the form:

$$\beta_i(x,t)R_i(x,t,U,\frac{\partial U}{\partial x},V) = \gamma_i(x,t,U,\frac{\partial U}{\partial x}V,\dot{V}), \qquad i = 1,2,\ldots,n; \tag{6}$$

where $x = a$ or $x = b$. The functions γ_i may only depend linearly on $\dot{V}$. The algebraic-differential equation system which is defined by the functions F_i must be specified in a procedure `ode_coef` supplied by the user.

Several options are available for controlling the operation of this procedure, for example the level of printed output and algorithmic parameters, such as integration method and order, etc.. These options are grouped together in the optional argument `control`, which is a structure of the derived type `nag_pde_parab_1d_cntrl_wp`.

The problem as defined by (4), (5) and (6) is subject to some additional constraints. For details see the Further Comments section.

2 Usage

```
USE nag_pde_parab_1d

CALL nag_pde_parab_1d_coll(pde_coef, bound_cond, init_value, deg_poly, t_start, t_end,
&                          x_bkpts, u  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathtt{x}(n)$' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$n_b \geq 3$ — (= `SIZE(x_bkpts)`) the number of collocation points (break-points) in the interval $[a, b]$;

$n_\xi \geq 0$ — (0 or `SIZE(x_ode)` if present) the number of ODE/PDE coupling points, if $p = 0$ then $n_\xi = 0$ and if $p > 0$ then $n_\xi \geq 0$;

$p \geq 0$ — the number of coupled ODE (see the optional argument `num_ode`);

$n \geq 1$ — the number of PDEs in the system (`SIZE(u)` $= n \times ((n_b - 1) \times d + 1) + p$).

3.1 Mandatory Arguments

pde_coef — subroutine

The procedure `pde_coef`, supplied by the user, must compute the functions $P_{i,j}$, Q_i and R_i which define the system of PDEs as in equation (4). The functions may depend on x, t, U, $\dfrac{\partial U}{\partial x}$ and V; Q_i may depend linearly on $\dot{V}$. The functions must be evaluated at a set of points.

Its specification is:

```
subroutine pde_coef(t, x, u, du_dx, p, q, r, finish, v, vdot, i_comm, r_comm)

real(kind=wp), intent(in) ::  t
```

Input: the current value of the independent variable t.

```
real(kind=wp), intent(in) ::  x(:)
```

Shape: `x` has shape $(d + 1)$.

Input: contains a set of mesh points at which $P_{i,j}$, Q_i and R_i are to be evaluated. $\mathtt{x}(1)$ and $\mathtt{x}(d + 1)$ contain successive break-points from the user-supplied array `x_bkpts`, the elements of `x` will satisfy $\mathtt{x}(1) < \mathtt{x}(2) < \cdots < \mathtt{x}(d + 1)$.

```
real(kind=wp), intent(in) ::  u(:,:)
```

Shape: `u` has shape $(n, d + 1)$.

Input: $\mathtt{u}(i, j)$ contains the value of the component $U_i(x, t)$ where $x = \mathtt{x}(j)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, d + 1$.

`real(kind=wp), intent(in) :: du_dx(:,:)`

Shape: `du_dx` has shape $(n, d+1)$.

Input: `du_dx`(i,j) contains the value of the component $\dfrac{\partial U_i}{\partial x}(x,t)$ where $x =$ `x`(j), for $i = 1,2,\ldots,n$ and $j = 1,2,\ldots,d+1$.

`real(kind=wp), intent(out) :: p(:,:,:)`

Shape: `p` has shape $(n, n, d+1)$.

Output: `p`(i,j,k) must be set to the values of $P_{i,j}(x,t,U,\dfrac{\partial U}{\partial x},V)$ where $x =$ `x`(k), for $i,j = 1,2,\ldots,n$ and $k = 1,2,\ldots,d+1$.

`real(kind=wp), intent(out) :: q(:,:)`

Shape: `q` has shape $(n, d+1)$.

Output: `q`(i,j) must be set to the values of $Q_i(x,t,U,\dfrac{\partial U}{\partial x},V,\dot{V})$ where $x =$ `x`(j), for $i = 1,2,\ldots,n$ and $j = 1,2,\ldots,d+1$.

`real(kind=wp), intent(out) :: r(:,:)`

Shape: `r` has shape $(n, d+1)$.

Output: `r`(i,j) must be set to the values of $R_i(x,t,U,\dfrac{\partial U}{\partial x},V)$ where $x =$ `x`(j), for $i = 1,2,\ldots,n$ and $j = 1,2,\ldots,d+1$.

`integer, intent(out) :: finish`

Output: the user may use `finish` to control the integration procedure as follows:

if `finish` $= 1$ or -1, indicates normal exit from the user supplied procedure;

if `finish` $= 2$, indicates to the integrator that control should be passed back immediately to the calling (sub)program with the error indicator set to `error%code` $= 205$;

if `finish` $= 3$, 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 `finish` $= 3$ when a physically meaningless input or output value has been generated. If the user consecutively sets `finish` $= 3$, then `nag_pde_parab_1d_fd` returns to the calling (sub)program with the error indicator set to `error%code` $= 203$;

if `finish` = any value other than 1, -1, 2 or 3, indicates an abnormal exit from the user supplied procedure, and returns to the calling (sub)program with the error indicator set to `error%code` $= 207$.

`real(kind=wp), intent(in), optional :: v(:)`

Shape: `v` has shape (p).

Input: `v`(i) contains the value of the component $V_i(t)$, for $i = 1,2,\ldots,p$.

`real(kind=wp), intent(in), optional :: vdot(:)`

Shape: `vdot` has shape (p).

Input: `vdot`(i) contains the value of the component $\dot{V}_i(t)$ for $i = 1,2,\ldots,p$.

Note: $\dot{V}_i(t)$ for $i = 1,2,\ldots,p$ may only appear linearly in Q_j, for $j = 1,2,\ldots,n$.

`integer, intent(in), optional :: i_comm(:)`

`real(kind=wp), intent(in), optional :: r_comm(:)`

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

bound_cond — subroutine

The procedure `bound_cond`, supplied by the user, must compute the functions β_i, and γ_i which define the boundary conditions as in equation (6).

Its specification is:

```
subroutine bound_cond(t, u, du_dx, bound, beta, gamma, finish, v, vdot, &
                      i_comm, r_comm)
```

`real(kind=wp), intent(in) :: t`

Input: the current value of the independent variable t.

`real(kind=wp), intent(in) :: u(:)`

Shape: `u` has shape (n).

Input: `u`(i) contains the value of the component $U_i(x,t)$, at the boundary specified by `bound`, for $i = 1, 2, \ldots, n$.

`real(kind=wp), intent(in) :: du_dx(:)`

Shape: `du_dx` has shape (n).

Input: `du_dx`(i) contains the value of the component $\dfrac{\partial U_i}{\partial x}(x,t)$, at the boundary specified by `bound`, for $i = 1, 2, \ldots, n$.

`integer, intent(in) :: bound`

Input: determines the position of the boundary conditions. If `bound` $= 0$, then the procedure `bound_cond` must set up the coefficients of the left-hand boundary $x = a$. Any other value of `bound` indicates that the procedure `bound_cond` must set up the coefficients of the right-hand boundary $x = b$.

`real(kind=wp), intent(out) :: beta(:)`

Shape: `beta` has shape (n).

Output: `beta`(i) must be set to the values of $\beta_i(x,t)$ at the boundary specified by `bound`, for $i = 1, 2, \ldots, n$.

`real(kind=wp), intent(out) :: gamma(:)`

Shape: `gamma` has shape (n).

Output: `gamma`(i) must be set to the values of $\gamma_i(x, t, U, \dfrac{\partial U}{\partial x}, V, \dot{V})$ at the boundary specified by `bound`, for $i = 1, 2, \ldots, n$.

`integer, intent(out) :: finish`

Output: the user may use `finish` to control the integration procedure as follows:

if `finish` $= 1$ or -1, indicates normal exit from the user supplied procedure;

if `finish` $= 2$, indicates to the integrator that control should be passed back immediately to the calling (sub)program with the error indicator set to `error%code` $= 205$;

if `finish` $= 3$, 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 `finish` $= 3$ when a physically meaningless input or output value has been generated. If the user consecutively sets `finish` $= 3$, then `nag_pde_parab_1d_fd` returns to the calling (sub)program with the error indicator set to `error%code` $= 203$;

if `finish` $=$ any value other than 1, -1, 2 or 3, indicates an abnormal exit from the user supplied procedure, and returns to the calling (sub)program with the error indicator set to `error%code` $= 207$.

`real(kind=wp), intent(in), optional :: v(:)`

Shape: `v` has shape (p).

Input: `v`(i) contains the value of the component $V_i(t)$, for $i = 1, 2, \ldots, p$.

`real(kind=wp), intent(in), optional :: vdot(:)`

Shape: `vdot` has shape (p).

Input: `vdot`(i) contains the value of the component $\dot{V}_i$, for $i = 1, 2, \ldots, p$.

Note: $\dot{V}_i(t)$ for $i = 1, 2, \ldots, p$ may only appear linearly in γ_j, for $j = 1, 2, \ldots, n$.

`integer, intent(in), optional :: i_comm(:)`

`real(kind=wp), intent(in), optional :: r_comm(:)`

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

init_value — subroutine

The procedure `init_value`, supplied by the user, must compute the initial values of the PDE components $U_i(x_j, t_0)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, (n_b - 1) \times d + 1$.

Its specification is:

```
subroutine init_value(x, u, v, i_comm, r_comm)
```

`real(kind=wp), intent(in) :: x(:)`

Shape: `x` has shape $((n_b - 1) \times d + 1)$.

Input: `x`(j) contains the value of the j^{th} mesh point, for $j = 1, 2, \ldots, (n_b - 1) \times d + 1$.

`real(kind=wp), intent(out) :: u(:,:)`

Shape: `u` has shape $(n, (n_b - 1) \times d + 1)$.

Output: `u`(i, j) must be set to the initial value $U_i(x_j, t_0)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, (n_b - 1) \times d + 1$.

`real(kind=wp), intent(out), optional :: v(:)`

Shape: `v` has shape (p).

Output: `v`(i) must contain the value of component $V_i(t_0)$, for $i = 1, 2, \ldots, p$.

`integer, intent(in), optional :: i_comm(:)`

`real(kind=wp), intent(in), optional :: r_comm(:)`

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

deg_poly — integer, intent(in)

Input: the degree of the Chebyshev polynomial to be used in approximating the PDE solution between each pair of break-points, d.

Constraints: $1 \leq$ `deg_poly` ≤ 49.

t_start — real(kind=*wp*), intent(inout)

Input: the initial value t_0 of the independent variable t.

Output: the value of t corresponding to the solution value U. Normally `t_start` = `t_end` on exit.

Constraints: `t_start` < `t_end`.

t_end — real(kind=*wp*), intent(in)

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

x_bkpts(n_b) — real(kind=*wp*), intent(in)

Input: the values of the break-points in the space direction. `x_bkpts`(1) must specify the left-hand boundary, a, and `x_bkpts`(n_b) must specify the right-hand boundary, b.

Constraints: `x_bkpts`(1) < `x_bkpts`(2) < $\cdots$ < `x_bkpts`(n_b).

u($n \times ((n_b - 1) \times d + 1) + p$) — real(kind=*wp*), intent(inout)

Input`u`($n \times (j-1) + i$) contains the computed solution $U_i(x_j, t)$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, (n_b - 1) \times d + 1$, and `u`($((n_b - 1) \times d + 1) \times n + k$) contains $V_k(t)$, for $k = 1, 2, \ldots, p$; evaluated on a previous call to the procedure. The user **does not** need to initialialise `u` at the very first call of the procedure.

Output: `u`($n \times (j-1) + i$) contains the computed solution $U_i(x_j, t)$, for $i = 1, 2, \ldots, n$; $j = 1, 2, \ldots, (n_b - 1) \times d + 1$, and `u`($((n_b - 1) \times d + 1) \times n + k$) contains $V_k(t)$, for $k = 1, 2, \ldots, p$; evaluated at the output value of t = `t_start`.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

first_call — logical, intent(in), optional

Input: indicates whether the time integration is a continuation of a previous step or not:

if `first_call` = `.true.`, starts or restarts the integration in time;

if `first_call` = `.false.`, continues the integration after an earlier exit from the procedure. In this case, only the parameter `t_end` should be reset between calls to `nag_pde_parab_1d_coll`.

Default: `first_call` = `.false.`.

ode_coef — subroutine, optional

The procedure `ode_coef`, supplied by the user, must evaluate the functions F_i, which defines the system of ODEs, as given in (5). If p is set to 0, `ode_coef` should not be present.

Its specification is:

```
subroutine ode_coef(t, v, vdot, x_ode, ucp, ducp_dx, rcp, ducp_dt, &
                    d2ucp_dtdx, f, finish, g_in_f, i_comm, r_comm)

real(kind=wp), intent(in) ::  t
```

Input: the current value of the independent variable t.

```
real(kind=wp), intent(in) ::  v(:)
```

Shape: `v` has shape (p).

Input: `v`(i) contains the value of component $V_i(t)$, for $i = 1, 2, \ldots, p$.

```
real(kind=wp), intent(in) ::  vdot(:)
```

Shape: `vdot` has shape (p).

Input: `vdot`(i) contains the value of component $\dot{V}_i$, for $i = 1, 2, \ldots, p$.

`real(kind=wp), intent(in) :: x_ode(:)`

Shape: `x_ode` has shape (n_ξ).

Input: `x_ode`(i) contains the value of the ODE/PDE coupling point, ξ, for $i = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: ucp(:,:)`

Shape: `ucp` has shape (n, n_ξ).

Input: `ucp`(i, j) contains the value of $U_i(x, t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: ducp_dx(:,:)`

Shape: `ducp_dx` has shape (n, n_ξ).

Input: `ducp_dx`(i, j) contains the value of $\dfrac{\partial U_i}{\partial x}(x, t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: rcp(:,:)`

Shape: `rcp` has shape (n, n_ξ).

Input: `rcp`(i, j) contains the value of R_i at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: ducp_dt(:,:)`

Shape: `ducp_dt` has shape (n, n_ξ).

Input: `ducp_dt`(i, j) contains the value of $\dfrac{\partial U_i}{\partial t}(x, t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(in) :: d2ucp_dtdx(:,:)`

Shape: `d2ucp_dtdx` has shape (n, n_ξ).

Input: `d2ucp_dtdx`(i, j) contains the value of $\dfrac{\partial^2 U_i}{\partial x \partial t}(x, t)$ at the coupling point $x = \xi_j$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_\xi$.

`real(kind=wp), intent(out) :: f(:)`

Shape: `f` has shape (p).

Output: `f`(i) must contain the i^{th} component of F, for $i = 1, 2, \ldots, p$; where F is defined (see the optional argument `g_in_f`) as follows:

$$F = G - A\dot{V} - B \begin{pmatrix} U_t^* \\ U_{xt}^* \end{pmatrix},$$

or

$$F = -A\dot{V} - B \begin{pmatrix} U_t^* \\ U_{xt}^* \end{pmatrix}.$$

The definition of F is determined by the input value of the optional argument `g_in_f`.

`integer, intent(out) :: finish`

Output: the user may use `finish` to control the integration procedure as follows:

if `finish` = 1 or −1, indicates normal exit from the user supplied procedure;

if `finish` = 2, indicates to the integrator that control should be passed back immediately to the calling (sub)program with the error indicator set to `error%code` = 205;

if `finish` = 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 `finish` = 3 when a physically meaningless input or output value has been generated. If the user consecutively sets `finish` = 3, then `nag_pde_parab_1d_fd` returns to the calling (sub)program with the error indicator set to `error%code` = 203;

if `finish` = any value other than 1, −1, 2 or 3, indicates an abnormal exit from the user supplied procedure, and returns to the calling (sub)program with the error indicator set to `error%code` = 207.

`logical, intent(in), optional :: g_in_f`

Input: indicates which of forms of `f` is returned (see the argument description of `f`). If `g_in_f` = .`true`., then the first equation above must be used; and if `g_in_f` = .`false`., then the second equation above must be used.

Default: `g_in_f` = .`false`..

`integer, intent(in), optional :: i_comm(:)`

`real(kind=`*wp*`), intent(in), optional :: r_comm(:)`

Input: you are free to use these arrays to supply information to this procedure from the calling (sub)program.

Constraints: `ode_coef` must not be present unless `num_ode` is present with a non zero value.

num_ode — integer, intent(in), optional

Input: the number of coupled ODEs in the system, p.

Constraints: `num_ode` ≥ 0.

Default: `num_ode` = 0.

x_ode(n_ξ) — real(kind=*wp*), intent(in), optional

Input: `x_ode`(i), for $i = 1, 2, \ldots, n_\xi$, must be set to the ODE/PDE coupling points ξ_i.

Constraints:

If n_ξ is set to 0, `x_ode` should not be present;

`x_bkpts`(1) $\leq$ `x_ode`(1) $<$ `x_ode`(2) $< \cdots <$ `x_ode`(n_ξ) $\leq$ `x_bkpts`(n_b).

comm_ode — type(nag_pde_parab_1d_comm_*wp*), intent(inout), optional

Input: a structure containing data for the underlying ODE solver between consecutive calls to the procedure.

Constraints:

if `first_call` = .`true`., `comm_ode` should be present if a further start is planned;

if `first_call` = .`false`., this argument should be present and should be the same as returned from the previous call;

if an interpolation of the solution, using the procedure `nag_pde_interp_1d_coll` is planned by the user, `comm_ode` should be present.

Output: a structure containing informations about the underlying ODE solver needed for a further call. So this structure may be passed to nag_pde_parab_1d_coll for the continuation of the time integration; and to nag_pde_parab_1d_coll to interpolate the solution.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private. The procedure allocates a certain amount of real(kind=*wp*) and integer elements of storage to the structure (those dimensions may be accessed via the optional argument dim_struct). If you wish to deallocate this storage when the structure is no longer required, you must call the procedure nag_deallocate, as illustrated in Example 1 of this module document.

coord_sys — character(len=1), intent(in), optional

Input: indicates which co-ordinate system (the index m) is being used:

if coord_sys = 'C' or 'c', indicates Cartesian co-ordinates ($m = 0$);

if coord_sys = 'P' or 'p', indicates cylindrical polar co-ordinates ($m = 1$);

if coord_sys = 'S' or 's', indicates spherical polar co-ordinates ($m = 2$).

Constraints:

coord_sys should be one of the following values 'C', 'c', 'P', 'p', 'S' or 's';

if coord_sys is one of the following values 'P', 'p', 'S', 's' (i.e., $m > 0$) then x_bkpts(1) should be ≥ 0.

Default: coord_sys = 'C'.

x_mesh($(n_b - 1) \times d + 1$**)** — real(kind=*wp*), intent(out), optional

Output: the mesh points chosen by the procedure in the spatial direction. The values of x_mesh will satisfy x_mesh(1) $<$ x_mesh(2) $< \cdots <$ x_mesh($(n_b - 1) \times d + 1$).

rel_loc_tol($n \times ((n_b - 1) \times d + 1) + p$**)** — real(kind=*wp*), intent(in), optional

Input: the relative local error tolerance used in the local error test. The error test to be satisfied is $\|e_i/w_i\| < 1.0$, where w_i is,

$$w_i = \alpha_i \times |U_i| + \epsilon_i, i = 1, 2, \ldots, ; n \times ((n_b - 1) \times d + 1) + p;$$

the e_i denotes the estimated local error for the i^{th} component of the coupled PDE/ODE system and rel_loc_tol(i) = α_i, for $i = 1, 2, \ldots, n \times ((n_b - 1) \times d + 1) + p$.

Constraints: rel_loc_tol(i) ≥ 0.0, for all $i = 1, 2, \ldots, n \times ((n_b - 1) \times d + 1) + p$.

Default: rel_loc_tol(i) = EPSILON(1.0_*wp*), for all $i = 1, 2, \ldots, n \times ((n_b - 1) \times d + 1) + p$.

abs_loc_tol($n \times ((n_b - 1) \times d + 1) + p$**)** — real(kind=*wp*), intent(in), optional

Input: the absolute local error tolerance used in the local error test; abs_loc_tol(i) = ϵ_i, for $i = 1, 2, \ldots, n \times ((n_b - 1) \times d + 1) + p$.

Constraints: abs_loc_tol(i) ≥ 0.0, for all $i = 1, 2, \ldots, n \times ((n_b - 1) \times d + 1) + p$, and corresponding elements of abs_loc_tol and rel_loc_tol (if present) cannot both be 0.

Default: abs_loc_tol(i) = EPSILON(1.0_*wp*), for all $i = 1, 2, \ldots, n \times ((n_b - 1) \times d + 1) + p$.

control — type(nag_pde_parab_1d_cntrl_*wp*), intent(in), optional

Input: a structure containing scalar components; these are used to alter the default values of those parameters which control the behaviour of the algorithm and the level of printed output. The initialization of this structure and its use is described in the procedure document for nag_pde_parab_1d_cntrl_init.

dim_struct(2) — integer, intent(out), optional

Output: the dimension of the real and integer arrays in the structure comm_ode, are stored in dim_struct(1) and dim_struct(2), respectively.

num_time_step — integer, intent(out), optional

Output: num_time_step contains the number of steps taken in time.

num_resid_eval — integer, intent(out), optional

Output: num_resid_eval contains the number of residual evaluations of the resulting ODE system used. One such evaluation involves the PDE functions at all mesh points, as well as one evaluation of the functions in the boundary conditions.

num_jac_eval — integer, intent(out), optional

Output: num_jac_eval contains the number of Jacobian evaluations performed by the time integrator.

num_time_iter — integer, intent(out), optional

Output: num_time_iter 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.

i_comm(:) — integer, intent(in), optional

r_comm(:) — real(kind=*wp*), intent(in), optional

Input: these arrays are not used by this procedure, but they are passed directly from the calling (sub)program to the user supplied procedures pde_coef, bound_cond, init_value and/or ode_coef, and hence may be used to pass information to them.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
304	Invalid presence of an optional argument.
305	Invalid absence of an optional argument.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	The underlying ODE solver cannot make any further progress, from the current point t = t_start, with the given values of rel_loc_tol and abs_loc_tol. The components of U contain the computed values at the current point t = t_start.
202	In the underlying ODE solver there were repeated error or corrector convergence test failures on an attempted step, before completing the requested task.

The problem may have a singularity, or the error requirement may be inappropriate. Integration was successful as far as $t =$ t_start.

203 In setting up the ODE system, the internal initialisation procedure was unable to initialise the derivative of the ODE system.

This could be due to the fact that finish was **repeatedly** set to 3 in the user supplied procedures pde_coef, bound_cond or ode_coef, when the residual in the underlying ODE solver was being evaluated.

204 In solving the ODE system a singular Jacobian has been encountered.

Check your problem formulation.

205 When evaluating the residual in the solving the ODE system, finish was set to 2 in at least one of the user supplied procedures pde_coef, bound_cond or ode_coef.

Integration was successful as far as $t =$ t_start.

206 The value of rel_loc_tol and abs_loc_tol are so small that the procedure is unable to start the integration in time.

207 In one of the user supplied procedures pde_coef, bound_cond or ode_coef, finish was set to an invalid value.

208 A serious error has occurred in an internal call to a stiff ODE integrator. Check problem specification and all parameters and array dimensions.

Setting control%print_level_ode = 1 may provide more information. If the problem persists, contact NAG.

209 An error occurred during Jacobian formulation of the ODE system.

A more detailed error description may be directed to the current advisory message unit. If using the sparse matrix algebra option, the values of control%first_piv_jac and control%rel_piv_thr may be inappropriate.

210 Some error weights w_i became zero during the time integration (see description of abs_loc_tol).

Pure relative error control (abs_loc_tol$(i) = 0.0$) was requested on a variable (the i^{th}) which has become zero. Integration was successful as far as $t =$ t_start.

211 The flux function R_i was detected as depending on time derivatives, which is not permissible.

Warnings (error%level = 1):

error%code	Description

101 The required task has been completed, but it is estimated that a small change in rel_loc_tol and abs_loc_tol is unlikely to produce any change in the computed solution.

Only applies when the user is not operating in one step mode, that is when control%task $\neq 2$ or 5.

102 In solving the ODE system, the maximum number of steps specified in control%num_step_max has been taken.

103 control%unit is different from its default value while no output is required (control%print_level_ode has its default value).

control%unit will not be used.

5 Examples of Usage

Complete examples of the use of this procedure appear in Example 2 of this module document. The user may find other examples in the additional examples section (Examples 5 and 9).

Assume that all relevant arguments have been declared correctly as described in Section 3, and that input and input/output arguments have been appropriately initialized. The following example illustrates the use of the optional argument `control` in order to solve a parabolic PDE using a collocation method.

The value of `control%print_level_ode` = 1 generates warning messages from the PDE solver.

```
   ...
! Initialize control structure
CALL nag_pde_parab_1d_cntrl_init(control)

! Set required value
control%print_level_ode = 1

CALL nag_pde_parab_1d_coll(pde_coef, bound_cond, init_value, deg_poly, &
                           t_start, t_end, x_bkpts, u, control=control)
   ...
```

6 Further Comments

6.1 Additional Problem Constraints

The routine is designed to solve parabolic systems (possibly including elliptic equations) with second-order derivatives in space. The parameter specification allows the user to include equations with only first-order derivatives in the space direction but there is no guarantee that the method of integration will be satisfactory for such systems. The position and nature of the boundary conditions are critical in defining a stable problem.

The problem is subject to the following restrictions:

in (4), $\dot{V}_j(t)$, for $j = 1, 2, \ldots, p$, may only appear linearly in the functions Q_i, for $i = 1, 2, \ldots, n$, with similar restrictions for γ_i in (6);

$t_0 < t_{out}$, so that integration is in the forward direction;

$P_{i,j}$, Q_i and the flux R_i must not depend on any time derivatives;

the evaluation of the functions $P_{i,j}$, Q_i and R_i is done at both the break-points and internally selected points for each element in turn, that is $P_{i,j}$, Q_i and R_i are evaluated twice at each break-point. Any discontinuities in these functions **must** therefore be at one or more of the break-points $x_1, x_2, \ldots, x_{n_b}$;

at least one of the functions $P_{i,j}$ must be non-zero so that there is a time derivative present in the problem;

if $m > 0$ and $x_1 = 0.0$, which is the left boundary point, then it must be ensured that the PDE solution is bounded at this point. This can be done by either specifying the solution at $x = 0.0$ or by specifying a zero flux there, that is $\beta_i = 1.0$ and $\gamma_i = 0.0$.

6.2 Algorithmic Detail

The parabolic equations are approximated by a system of ODEs in time for the values of U_i at the mesh points. This ODE system is obtained by approximating the PDE solution between each pair of break-points by a Chebyshev polynomial of degree d. The interval between each pair of break-points is treated by `nag_pde_parab_1d_coll` as an element, and on this element, a polynomial and its space and time derivatives are made to satisfy the system of PDEs at $(d-1)$ spatial points, which are chosen internally by the code, and at the break-points. In the case of just one element, the break-points are the boundaries. The user-defined break-points and the internally selected points together define the mesh. The smallest value that d can take is one, in which case, the solution is approximated by piecewise linear

polynomials between consecutive break-points and the method is similar to an ordinary finite element method.

In total there are $(n_b - 1) \times d + 1$ mesh points in the spatial direction, and $n \times ((n_b - 1) \times d + 1)$ ODEs in the time direction; one ODE at each break-point for each PDE component and $(d - 1)$ ODEs for each PDE component between each pair of break-points. The system is then integrated forwards in time using a backward differentiation formula method.

Procedure: nag_pde_interp_1d_coll

1 Description

This procedure interpolates in the spatial co-ordinate the solution and its first spatial derivative of a system of partial differential equations on an interval $[a; b]$; at a set of user-specified points. The solution must be computed using a Chebyshev C^0 collocation method, and this procedure will normally be used in conjunction with the procedure `nag_pde_parab_1d_coll`.

2 Usage

```
USE nag_pde_parab_1d
CALL nag_pde_interp_1d_coll(u, x_bkpts, deg_poly, comm_ode, x_interp, u_interp &
                            [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '`x`(n)' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$n_b \geq 3$ — (= `SIZE(x_bkpts)`) the number of collocation points (break-points) in the interval $[a, b]$;

$n \geq 1$ — the number of PDEs in the system (`SIZE(u)` $= n \times ((n_b - 1) \times d + 1)$);

$n_{interp} \geq 1$ — the number of interpolation points in the interval $[a, b]$;

$i_{type} = 1$, or 2 — specifies the interpolation to be performed:

if $i_{type} = 1$, the solution at the interpolation points is computed,

if $i_{type} = 2$, both the solution and its first derivative at the interpolation points are computed.

3.1 Mandatory Arguments

u($n \times ((n_b - 1) \times d + 1)$**)** — real(kind=*wp*), intent(in)

Input: `u`$(n \times (j - 1) + i)$ contains the computed solution $U_i(x_j)$ to be interpolated, as returned by the procedure `nag_pde_parab_1d_coll`, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, (n_b - 1) \times d + 1$.

x_bkpts(n_b**)** — real(kind=*wp*), intent(in)

Input: the values of the break-points in the space direction. `x_bkpts`(1) must specify the left-hand boundary, a, and `x_bkpts`(n_b) must specify the right-hand boundary, b.

Constraints: `x_bkpts`$(1) <$ `x_bkpts`$(2) < \cdots <$ `x_bkpts`(n_b).

deg_poly — integer, intent(in)

Input: the degree d of the Chebyshev polynomial to be used in approximating the PDE solution between each pair of break-points.

Constraints: $1 \leq$ `deg_poly` ≤ 49.

comm_ode — type(nag_pde_parab_1d_comm_*wp*), intent(in)

Input: a structure containing data returned as an optional argument (which should therefore be present) from the procedure `nag_pde_parab_1d_coll`.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private. If you wish to deallocate this storage when the structure is no longer required, you must call the procedure `nag_deallocate`, as illustrated in Example 1 of this module document.

x_interp(n_{interp}) — real(kind=*wp*), intent(in)

Input: must contain the spatial interpolation points.

Constraints: x_bkpts(1) $\leq$ x_interp(1) $<$ x_interp(2) $< \cdots <$ x_interp(n_{interp}) $\leq$ x_bkpts(n_b). When $i_{type} = 2$, x_interp(i) $\neq$ x_bkpts(j), for any $i = 1, 2, \ldots, n_{interp}$ and $j = 1, 2, \ldots, n_b$.

u_interp(n, n_{interp}, i_{type}) — real(kind=*wp*), intent(out)

Output:

if $i_{type} = 1$, u_interp(i,j,1) contains the value of the solution $U_i(x_j)$ at the interpolation point x_j = x_interp(j), for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_{interp}$;

if $i_{type} = 2$, u_interp(i,j,1) and u_interp(i,j,2) contain respectively the value of the solution $U_i(x_j)$ and the derivative $\dfrac{\partial U_i}{\partial x}(x_j)$ at the interpolation point x_j = x_interp(j), for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_{interp}$.

3.2 Optional Argument

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document.

Procedure: nag_pde_parab_1d_cntrl_init

1 Description

nag_pde_parab_1d_cntrl_init assigns default values to the components of a structure of the derived type nag_pde_parab_1d_cntrl_*wp*.

2 Usage

```
USE nag_pde_parab_1d
CALL nag_pde_parab_1d_cntrl_init(control)
```

3 Arguments

3.1 Mandatory Argument

control — type(nag_pde_parab_1d_cntrl_*wp*), intent(out)

Output: a structure containing the default values of those parameters which control the behaviour of the algorithm and level of printed output. A description of its components is given in the document for the derived type nag_pde_parab_1d_cntrl_*wp*.

4 Error Codes

None.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1 to 3 of this module document. The user may find other examples in the additional examples section (Examples 4, 5, 6, 7, 8 and 9).

Derived Type: nag_pde_parab_1d_comm_*wp*

Note. The names of derived types containing real/complex components are precision dependent. For double precision the name of this type is `nag_pde_parab_1d_comm_dp`. For single precision the name is `nag_pde_parab_1d_comm_sp`. Please read the Users' Note for your implementation to check which precisions are available.

1 Description

The derived type `nag_pde_parab_1d_comm_wp` is used to communicate data needed by the underlying ODE solver. The procedures `nag_pde_parab_1d_fd` and `nag_pde_parab_1d_coll` returns a structure of this type suitable for passing to:

the same procedures respectively (to continue the time integration);

the interpolation procedure `nag_pde_interp_1d_coll`, only in the case of the Chebyshev C^0 collocation method.

On a first call to the procedure `nag_pde_parab_1d_fd` or `nag_pde_parab_1d_coll`, storage is dynamically allocated to the pointer components of the structure. For details of the amount of storage allocated see the description of the optional argument `comm_ode` in the procedure document for `nag_pde_parab_1d_fd` or `nag_pde_parab_1d_coll`.

If you wish to deallocate the storage when the structure is no longer required, you must call the generic deallocation procedure `nag_deallocate`, which is described in the module document `nag_lib_support` (1.1).

The procedure `nag_pde_parab_1d_fd` or `nag_pde_parab_1d_coll` checks whether the structure has already had storage allocated to it in a previous call. If it has, that storage is deallocated before allocating the storage required for the new call, only if the optional argument `first_call` is present and set to `.true.`(see the description of the optional argument `first_call` in the procedure document for `nag_pde_parab_1d_fd` or `nag_pde_parab_1d_coll`).

The components of this type are private.

2 Type Definition

```
type nag_pde_parab_1d_comm_wp
  private
  .
  .
  .
end type nag_pde_parab_1d_comm_wp
```

3 Components

In order to reduce the risk of accidental data corruption the components of this type are private and may not be accessed directly.

Derived Type: nag_pde_parab_1d_cntrl_*wp*

Note. The names of derived types containing real/complex components are precision dependent. For double precision the name of this type is `nag_pde_parab_1d_cntrl_dp`. For single precision the name is `nag_pde_parab_1d_cntrl_sp`. Please read the Users' Note for your implementation to check which precisions are available.

1 Description

A structure of type `nag_pde_parab_1d_cntrl_wp` is used to supply a number of optional parameters, for example the level of printed output, or various algorithmic parameters.

If this structure is supplied then it *must* be initialized prior to use by calling the procedure `nag_pde_parab_1d_cntrl_init`, which assigns default values to all the structure components. You may then assign required values to selected components of the structure (as appropriate).

2 Type Definition

The public components are listed below; components are grouped according to their function. A full description of the purpose of each component is given in Section 3.

```
type nag_pde_parab_1d_cntrl_wp
  !  Printing parameters
  integer ::  unit
  integer ::  print_level_ode
  integer ::  print_level_remesh
  !
  !  Algorithm choice
  logical ::  l2_norm
  character ::  matrix_type
  integer ::  task
  logical ::  method_bdf
  integer ::  bdf_max_ord
  logical ::  bdf_newton_it
  logical ::  bdf_petzold_test
  real(kind=wp) ::  theta
  logical ::  theta_newton_it
  logical ::  theta_switch
  real(kind=wp) ::  t_crit
  real(kind=wp) ::  step_size_min
  real(kind=wp) ::  step_size_max
  real(kind=wp) ::  step_size_init
  integer ::  num_step_max
  logical ::  mod_newton_it
  real(kind=wp) ::  first_piv_jac
  real(kind=wp) ::  rel_piv_thr
end type nag_pde_parab_1d_cntrl_wp
```

3 Components

3.1 Printing Parameters

unit — integer

Specifies the Fortran unit number to which all output produced by the procedures `nag_pde_parab_1d_fd` or `nag_pde_parab_1d_coll` is sent.

Default: `unit` = the default Fortran output unit number for your implementation.

Constraints: `unit` ≥ 0.

print_level_ode — integer

the level of diagnostic information required from the procedure `nag_pde_parab_1d_fd` or `nag_pde_parab_1d_coll` and the underlying ODE solver. The following output is sent to the Fortran unit number defined by the optional argument `unit`:

if `print_level_ode` ≤ -1, no ouput is generated;

if `print_level_ode` $= 0$, only warning messages from the PDE solver are generated;

if `print_level_ode` $= 1$, the output from the underlying ODE solver is generated. This output contains details of Jacobian entries, the nonlinear iteration and the time integration during the computation of the ODE system;

if `print_level_ode` ≥ 2, the similar output to that produced when `print_level_ode` $= 1$ is generated in greater detail. Inexperienced users are advised to set `print_level_ode` $= 0$.

Default: `print_level_ode` $= -1$.

print_level_remesh — integer

the level of print information regarding the adaptive remeshing. The following output is sent to the Fortran unit number defined by the optional argument `unit`:

if `print_level_remesh` $= 0$, no output is generated;

if `print_level_ode` $= 1$, brief summary of mesh characteristics;

if `print_level_ode` $= 2$, more detailed information, including old and new mesh points, mesh sizes and monitor function values.

Constraints:

`print_level_remesh` should not be different from the default value unless remeshing is to be performed i.e., `remesh` is present with value `.true.`,

$0 \leq$ `print_level_remesh` ≤ 2.

Default: `print_level_remesh` $= 0$.

3.2 Algorithm Options

l2_norm — logical

indicates the type of norm to be used for the error test to be satisfied:

if `l2_norm` = `.false.`, maximum norm,

if `l2_norm` = `.true.`, average L_2 norm.

Default: `l2_norm` = `.false.`.

Note: if U_{norm} denotes the norm of the vector U of length $n \times n_\eta + p$, then the average L_2 norm is:

$$U_{norm} = \sqrt{\frac{1}{n \times n_\eta + p} \sum_{i=1}^{n \times n_\eta + p} (U_i/w_i)^2},$$

while the maximum norm is:

$$U_{norm} = \max_i |U_i/w_i|.$$

See the description of the optional arguments `rel_loc_tol` and `abs_loc_tol` for the formulation of the weight vector $w = \{w_i\}_{i=1,2,\ldots(n \times n_\eta + p)}$.

matrix_type — character(len=1)

the type of the matrix algebra required. The possible choices are:

if `matrix_type` = `'F'` or `'f'`, full matrix procedures to be used,

if `matrix_type` = `'B'` or `'b'`, banded matrix procedures to be used,

if `matrix_type` = `'S'` or `'s'`, sparse matrix procedures to be used.

Constraints: `matrix_type` = `'F'`, `'f'`, `'B'`, `'b'`, `'S'`, or `'s'`.

Default: `matrix_type` = `'b'`.

Note: the user is recommended to use the banded option when no coupled ODEs are present (i.e., `num_ode` $= p = 0$).

task — integer

specifies the task to be performed by the underlying ODE integrator. The permitted values of `task` and their meanings are detailed below:

if `task` = 1, normal computation of output values U at t = `t_end` (by overshooting and interpolation);

if `task` = 2, one step and return;

if `task` = 3, stop at first internal integration point at or beyond t = `t_end`;

if `task` = 4, normal computation of output values U at t = `t_end` but without overshooting $t = t_{crit}$, where t_{crit} is described under the component `t_crit`;

if `task` = 5, take one step in the time direction and return, without passing t_{crit}, where t_{crit} is described under the component `t_crit`.

Constraints: $1 \leq$ `task` ≤ 5.

Default: `task` = 1.

method_bdf — logical

selects the ODE integration method to be used:

if `method_bdf` = `.true.`, a BDF method is used;

if `method_bdf` = `.false.`, a Theta method is used.

Default: `method_bdf` = `.true.`.

Note: if `method_bdf` = `.true.`, then `theta`, `theta_newton_it` and `theta_switch` are not used. If `method_bdf` = `.false.`, then `bdf_max_ord`, `bdf_newton_it` and `bdf_petzold_test` are not used.

bdf_max_ord — integer

specifies the maximum order of the BDF integration formula to be used.
Constraints: $1 \leq$ `bdf_max_ord` ≤ 5.

Default: `bdf_max_ord` = 5.

bdf_newton_it — logical

specifies what method is to be used to solve the system of non-linear equations arising on each step of the BDF method:

if `bdf_newton_it` = `.true.`, a modified Newton iteration is used;

if `bdf_newton_it` = `.false.`, a functional iteration is used.

Default: `bdf_newton_it` = `.true.`.

Note: if `method_bdf` = `.false.`, i.e., a functional iteration is selected and the integrator encounters difficulty, then there is an automatic switch to the modified Newton iteration.

bdf_petzold_test — logical

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 $j = 1, 2, \ldots, n$; for some i or when there is no $\dot{V}_i(t)$ dependence in the coupled ODE system.

If `bdf_petzold_test` = `.true.`, then the Petzold test is used;

if `bdf_petzold_test` = `.false.`, then the Petzold test is not used.

Default: `bdf_petzold_test` = `.true.`.

theta — real(kind=*wp*)

specifies the value of Theta to be used in the Theta integration method.
Constraints: $0.51 \leq$ `theta` ≤ 0.99.
Default: `theta` = 0.55.

theta_newton_it — logical

specifies what method is to be used to solve the system of non-linear equations arising on each step of the Theta method.

If `theta_newton_it` = `.true.`, then a modified Newton iteration is used;

if `theta_newton_it` = `.false.`, then a functional iteration is used.

Default: `theta_newton_it` = `.true.`.

theta_switch — logical

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 `theta_switch` = `.true.`, then switching is allowed;

if `theta_switch` = `.false.`, then switching is not allowed.

Default: `theta_switch` = `.true.`.

t_crit — real(kind=*wp*)

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 component `task`. A value of 0.0 for `t_crit`, say, should be specified even if `task` subsequently specifies that t_{crit} will not be used.
Default: `t_crit` = 0.0.

step_size_min — real(kind=*wp*)

specifies the minimum absolute step size to be allowed in the time integration.
Default: `step_size_min` = 0.0.

step_size_max — real(kind=*wp*)

specifies the maximum absolute step size to be allowed in the time integration.
Default: `step_size_max` = 0.0.

step_size_init — real(kind=*wp*)

specifies the initial step size to be attempted by the integrator. If `step_size_ini` = 0.0, then the initial step size is calculated internally.
Default: `step_size_init` = 0.0.

num_step_max — integer

specifies the maximum number of steps to be attempted by the integrator in any one call. If `num_step_max` = 0, then no limit is imposed.
Default: `num_step_max` = 0.

mod_newton_it — logical

specifies what method is to be used to solve the non-linear equations at the initial point to initialise the value of U, $\dfrac{\partial U}{\partial t}$, V, and $\dot{V}$.

If `mod_newton_it` = `.true.`, a modified Newton iteration is used;

if `mod_newton_it` = `.false.`, a functional iteration is used.

Default: `mod_newton_it` = `.true.`.

first_piv_jac — real(kind=*wp*)

governs the choice of pivots during the decomposition of the first Jacobian matrix. It should lie in the range $0.0 \leq$ `first_piv_jac`$(29) \leq 1.0$, with smaller values biasing the algorithm towards maintaining sparsity at the expense of numerical stability.
Default: `first_piv_jac` = 0.1.

Note: if `first_piv_jac` lies outside this range then default value is used. If the procedure regards the Jacobian matrix as numerically singular then increasing `first_piv_jac` towards 1.0 may help, but at the cost of increasing fill-in.

rel_piv_thr — real(kind=*wp*)

is used as a relative pivot threshold during subsequent Jacobian decompositions (See component `first_piv_jac`) below which an internal error is invoked. `rel_piv_thr` must be greater than zero, otherwise the default value is used. If `rel_piv_thr` 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 component `first_piv_jac`).
Default: `rel_piv_thr` = 0.0001.

Example 1: Elliptic-parabolic PDEs Solved Using Finite Difference Scheme and the BDF Method

The example below, given in Dew and Walsh [5], consists of an elliptic-parabolic pair of PDEs. The problem was originally derived from a single third-order in space PDE. The elliptic equation is

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r^2\frac{\partial U_1}{\partial r}\right) = 4\alpha\left(U_2 + r\frac{\partial U_2}{\partial r}\right)$$

and the parabolic equation is

$$(1-r^2)\frac{\partial U_2}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\left(\frac{\partial U_2}{\partial r} - U_2U_1\right)\right)$$

where $(r,t) \in [0,1] \times [0,1]$. The boundary conditions are given by

$$U_1 = \frac{\partial U_2}{\partial r} = 0 \text{ at } r = 0,$$

and

$$\frac{\partial}{\partial r}(rU_1) = 0 \text{ and } U_2 = 0 \text{ at } r = 1.$$

The first of these boundary conditions implies that the flux term in the second PDE, $\left(\frac{\partial U_2}{\partial r} - U_2U_1\right)$, is zero at $r = 0$.

The initial conditions at $t = 0$ are given by;

$$U_1 = 2\alpha r \text{ and } U_2 = 1.0, \text{ for } r \in [0,1].$$

The value $\alpha = 1$ was used in the problem definition. A mesh of 20 points was used with a circular mesh spacing to cluster the points towards the right-hand side of the spatial interval, $r = 1$.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE nag_pde_parab_1d_ex01_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  SUBROUTINE pde_coef(t,x,u,du_dx,p,q,r,finish,v,vdot,i_comm,r_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Scalar Arguments ..
    INTEGER, INTENT (OUT) :: finish
    REAL (wp), INTENT (IN) :: t, x
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), INTENT (IN) :: du_dx(:), u(:)
```

```
  REAL (wp), INTENT (OUT) :: p(:,:), q(:), r(:)
  REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:), v(:), vdot(:)
  ! .. Local Scalars ..
  REAL (wp) :: alpha
  ! .. Executable Statements ..
  alpha = r_comm(1)

  finish = 1

  q(1) = 4.0_wp*alpha*(u(2)+x*du_dx(2))
  q(2) = 0.0_wp
  r(1) = x*du_dx(1)
  r(2) = du_dx(2) - u(1)*u(2)
  p(1,1:2) = 0.0_wp
  p(2,1) = 0.0_wp
  p(2,2) = 1.0_wp - x*x

END SUBROUTINE pde_coef

SUBROUTINE bound_cond(t,u,du_dx,bound,beta,gamma,finish,v,vdot,i_comm, &
   r_comm)
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Scalar Arguments ..
  INTEGER, INTENT (IN) :: bound
  INTEGER, INTENT (OUT) :: finish
  REAL (wp), INTENT (IN) :: t
  ! .. Array Arguments ..
  INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
  REAL (wp), INTENT (OUT) :: beta(:), gamma(:)
  REAL (wp), INTENT (IN) :: du_dx(:), u(:)
  REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:), v(:), vdot(:)
  ! .. Executable Statements ..

  finish = 1

  IF (bound==0) THEN
    beta(1:2) = (/ 0.0_wp, 1.0_wp/)
    gamma(1) = u(1)
    gamma(2) = -u(1)*u(2)
  ELSE
    beta(1:2) = (/ 1.0_wp, 0.0_wp/)
    gamma(1:2) = -u(1:2)
  END IF

END SUBROUTINE bound_cond

SUBROUTINE init_value(x_pde,u,x_ode,v,i_comm,r_comm)
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC SIZE
  ! .. Array Arguments ..
  INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
  REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:), x_ode(:)
  REAL (wp), INTENT (OUT) :: u(:,:)
  REAL (wp), OPTIONAL, INTENT (OUT) :: v(:)
  REAL (wp), INTENT (IN) :: x_pde(:)
  ! .. Local Scalars ..
  INTEGER :: npts_pde
  REAL (wp) :: alpha
  ! .. Executable Statements ..
```

```
    alpha = r_comm(1)
    npts_pde = SIZE(x_pde)

    u(1,1:npts_pde) = 2.0_wp*alpha*x_pde(1:npts_pde)
    u(2,1:npts_pde) = 1.0_wp

  END SUBROUTINE init_value

END MODULE nag_pde_parab_1d_ex01_mod
PROGRAM nag_pde_parab_1d_ex01

  ! Example Program Text for nag_pde_parab_1d
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_math_constants, ONLY : nag_pi
  USE nag_pde_parab_1d, ONLY : nag_pde_parab_1d_comm_wp => &
   nag_pde_parab_1d_comm_dp, nag_pde_parab_1d_cntrl_wp => &
   nag_pde_parab_1d_cntrl_dp, nag_pde_parab_1d_cntrl_init, &
   nag_pde_parab_1d_fd, nag_pde_interp_1d_fd, nag_deallocate
  USE nag_pde_parab_1d_ex01_mod, ONLY : pde_coef, bound_cond, init_value, &
   wp
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC EPSILON, MAX, REAL, SIN, SQRT
  ! .. Parameters ..
  INTEGER, PARAMETER :: itype = 1, ninterp = 6, npde = 2, npts_pde = 20
  REAL (wp), PARAMETER :: x_interp(ninterp) = (/ 0.0_wp, 0.4_wp, 0.6_wp, &
   0.8_wp, 0.9_wp, 1.0_wp/)
  ! .. Local Scalars ..
  INTEGER :: i, it, num_jac_eval, num_resid_eval, num_time_iter, &
   num_time_step
  REAL (wp) :: acc, alpha, hx, pi, piby2, t_end, t_start
  LOGICAL :: first_call
  CHARACTER (1) :: coord_sys
  TYPE (nag_pde_parab_1d_comm_wp) :: comm_ode
  TYPE (nag_pde_parab_1d_cntrl_wp) :: control
  ! .. Local Arrays ..
  REAL (wp) :: abs_loc_tol(npde*npts_pde), rel_loc_tol(npde*npts_pde), &
   r_comm(1), u(npde*npts_pde), u_interp(npde,ninterp,itype), &
   x_pde(npts_pde)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) &
   ' Example Program Results for nag_pde_parab_1d_ex01 '

  pi = nag_pi(0.0_wp)
  piby2 = 0.5_wp*pi

  ! Set initial conditions

  alpha = 1.0_wp
  acc = MAX(SQRT(EPSILON(alpha)),1.0E-4_wp)
  hx = piby2/REAL((npts_pde-1),kind=wp)
  t_start = 0.0_wp
  t_end = 0.1E-4_wp
  first_call = .TRUE.
  coord_sys = 'P'

  r_comm(1) = alpha
  rel_loc_tol = acc
```

```
      abs_loc_tol = acc

      ! Set spatial mesh points

      x_pde(1) = 0.0_wp
      x_pde(npts_pde) = 1.0_wp

      DO i = 2, npts_pde - 1
        x_pde(i) = SIN(hx*REAL((i-1),kind=wp))
      END DO

      WRITE (nag_std_out,999) acc, alpha
      WRITE (nag_std_out,998) (x_interp(i),i=1,6)

      ! Initialize the structure control and set required control parameters

      CALL nag_pde_parab_1d_cntrl_init(control)

      control%l2_norm = .TRUE.

      DO it = 1, 5
        t_end = 10.0_wp*t_end

        CALL nag_pde_parab_1d_fd(pde_coef,bound_cond,init_value,t_start,t_end, &
         x_pde,u,first_call=first_call,comm_ode=comm_ode,coord_sys=coord_sys, &
         rel_loc_tol=rel_loc_tol,abs_loc_tol=abs_loc_tol,control=control, &
         num_time_step=num_time_step,num_resid_eval=num_resid_eval, &
         num_jac_eval=num_jac_eval,num_time_iter=num_time_iter,r_comm=r_comm)

        first_call = .FALSE.

        ! Interpolate at required spatial points

        CALL nag_pde_interp_1d_fd(u,x_pde,x_interp,u_interp, &
         coord_sys=coord_sys)

        WRITE (nag_std_out,996) t_end, u_interp(1,1:ninterp,1)
        WRITE (nag_std_out,995) u_interp(2,1:ninterp,1)
      END DO

      ! Free structure comm_ode  allocated by NAG fl90

      CALL nag_deallocate(comm_ode)

      ! Print integration statistics

      WRITE (nag_std_out,997) num_time_step, num_resid_eval, num_jac_eval, &
       num_time_iter

999   FORMAT (//' Accuracy requirement  = ',E12.5/' Parameter ALPHA =', &
       '       ',E12.3/)
998   FORMAT ('   T  /  X  ',6F8.4/)
997   FORMAT (' Number of integration steps in time                  ', &
       I4/' Number of residual evaluations of resulting ODE sys','tem', &
       I4/' Number of Jacobian evaluations               ','         ', &
       I4/' Number of iterations of nonlinear solve','r              ',I4/)
996   FORMAT (1X,F6.4,' U(1)',6F8.4)
995   FORMAT (8X,'U(2)',6F8.4/)

    END PROGRAM nag_pde_parab_1d_ex01
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_pde_parab_1d_ex01


Accuracy requirement  =  0.10000E-03
Parameter ALPHA =          0.100E+01

  T  /  X    0.0000  0.4000  0.6000  0.8000  0.9000  1.0000

0.0001 U(1)  0.0000  0.8008  1.1988  1.5990  1.7958  1.8483
       U(2)  0.9997  0.9995  0.9994  0.9988  0.9664 -0.0000

0.0010 U(1)  0.0000  0.7982  1.1940  1.5841  1.7179  1.6735
       U(2)  0.9969  0.9952  0.9937  0.9483  0.6389 -0.0000

0.0100 U(1)  0.0000  0.7676  1.1238  1.3548  1.3639  1.2834
       U(2)  0.9627  0.9495  0.8752  0.5545  0.2914 -0.0000

0.1000 U(1)  0.0000  0.3910  0.5011  0.5302  0.5126  0.4749
       U(2)  0.5470  0.4303  0.2999  0.1482  0.0726 -0.0000

1.0000 U(1)  0.0000  0.0005  0.0006  0.0006  0.0006  0.0005
       U(2)  0.0007  0.0005  0.0003  0.0002  0.0001 -0.0000

Number of integration steps in time                     35
Number of residual evaluations of resulting ODE system 159
Number of Jacobian evaluations                           9
Number of iterations of nonlinear solver                85
```

Example 2: Elliptic-parabolic PDEs Solved Using Collocation Scheme and the BDF Method

The problem consists of a fourth-order PDE which can be written as a pair of second-order elliptic-parabolic PDEs for $U_1(x,t)$ and $U_2(x,t)$,

$$0 = \frac{\partial^2 U_1}{\partial x^2} - U_2,$$

$$\frac{\partial U_2}{\partial t} = \frac{\partial^2 U_2}{\partial x^2} + U_2\frac{\partial U_1}{\partial x} - U_1\frac{\partial U_2}{\partial x};$$

where $-1 \leq x \leq 1$ and $t \geq 0$. The boundary conditions are given by

$$\frac{\partial U_1}{\partial x} = 0 \text{ and } U_1 = 1 \text{ at } x = -1, \text{ and}$$

$$\frac{\partial U_1}{\partial x} = 0 \text{ and } U_1 = -1 \text{ at } x = 1.$$

The initial conditions at $t = 0$ are given by

$$U_1 = -\sin\frac{\pi x}{2} \text{ and } U_2 = \frac{\pi^2}{4}\sin\frac{\pi x}{2}.$$

The absence of boundary conditions for $U_2(x,t)$ does not pose any difficulties provided that the derivative flux boundary conditions are assigned to the first PDE which has the correct flux, $\frac{\partial U_1}{\partial x}$. The conditions on $U_1(x,t)$ at the boundaries are assigned to the second PDE by setting $\beta_2 = 0.0$ in the boundary condition equation (6) (see the procedure `nag_pde_parab_1d_coll` document) and placing the Dirichlet boundary conditions on $U_1(x,t)$ in the function γ_2.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE nag_pde_parab_1d_ex02_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  SUBROUTINE pde_coef(t,x,u,du_dx,p,q,r,finish,v,vdot,i_comm,r_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIZE
    ! .. Scalar Arguments ..
    INTEGER, INTENT (OUT) :: finish
    REAL (wp), INTENT (IN) :: t
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), INTENT (IN) :: du_dx(:,:), u(:,:), x(:)
    REAL (wp), INTENT (OUT) :: p(:,:,:), q(:,:), r(:,:)
```

```
  REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:), v(:), vdot(:)
  ! .. Local Scalars ..
  INTEGER :: nptl
  ! .. Executable Statements ..
  nptl = SIZE(x)

  finish = 1

  q(1,1:nptl) = u(2,1:nptl)
  q(2,1:nptl) = u(1,1:nptl)*du_dx(2,1:nptl) - du_dx(1,1:nptl)*u(2,1:nptl &
   )
  r(1,1:nptl) = du_dx(1,1:nptl)
  r(2,1:nptl) = du_dx(2,1:nptl)
  p(1,1,1:nptl) = 0.0_wp
  p(1,2,1:nptl) = 0.0_wp
  p(2,1,1:nptl) = 0.0_wp
  p(2,2,1:nptl) = 1.0_wp

END SUBROUTINE pde_coef

SUBROUTINE bound_cond(t,u,du_dx,bound,beta,gamma,finish,v,vdot,i_comm, &
   r_comm)
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Scalar Arguments ..
  INTEGER, INTENT (IN) :: bound
  INTEGER, INTENT (OUT) :: finish
  REAL (wp), INTENT (IN) :: t
  ! .. Array Arguments ..
  INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
  REAL (wp), INTENT (OUT) :: beta(:), gamma(:)
  REAL (wp), INTENT (IN) :: du_dx(:), u(:)
  REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:), v(:), vdot(:)
  ! .. Executable Statements ..

  finish = 1
  beta(1:2) = (/ 1.0_wp, 0.0_wp/)
  gamma(1) = 0.0_wp

  IF (bound==0) THEN
    gamma(2) = u(1) - 1.0_wp
  ELSE
    gamma(2) = u(1) + 1.0_wp
  END IF

END SUBROUTINE bound_cond

SUBROUTINE init_value(x,u,v,i_comm,r_comm)
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC SIN, SIZE
  ! .. Array Arguments ..
  INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
  REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:)
  REAL (wp), INTENT (OUT) :: u(:,:)
  REAL (wp), OPTIONAL, INTENT (OUT) :: v(:)
  REAL (wp), INTENT (IN) :: x(:)
  ! .. Local Scalars ..
  INTEGER :: npts_pde
  REAL (wp) :: piby2
  ! .. Executable Statements ..
```

```
      piby2 = r_comm(1)
      npts_pde = SIZE(x)

      u(1,1:npts_pde) = -SIN(piby2*x(1:npts_pde))
      u(2,1:npts_pde) = -piby2*piby2*u(1,1:npts_pde)

    END SUBROUTINE init_value

  END MODULE nag_pde_parab_1d_ex02_mod
  PROGRAM nag_pde_parab_1d_ex02

    ! Example Program Text for nag_pde_parab_1d
    ! NAG fl90, Release 4. NAG Copyright 2000.

    ! .. Use Statements ..
    USE nag_examples_io, ONLY : nag_std_out
    USE nag_math_constants, ONLY : nag_pi
    USE nag_pde_parab_1d, ONLY : nag_pde_parab_1d_comm_wp => &
     nag_pde_parab_1d_comm_dp, nag_pde_parab_1d_cntrl_wp => &
     nag_pde_parab_1d_cntrl_dp, nag_pde_parab_1d_cntrl_init, &
     nag_pde_parab_1d_coll, nag_pde_interp_1d_coll, nag_deallocate
    USE nag_pde_parab_1d_ex02_mod, ONLY : pde_coef, bound_cond, init_value, &
     wp
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC REAL
    ! .. Parameters ..
    INTEGER, PARAMETER :: deg_poly = 3, itype = 1, ninterp = 6, npde = 2, &
     npts_bk = 10
    INTEGER, PARAMETER :: nel = npts_bk - 1
    INTEGER, PARAMETER :: npts = nel*deg_poly + 1
    REAL (wp), PARAMETER :: x_interp(ninterp) = (/ -1.0_wp, -0.6_wp, &
     -0.2_wp, 0.2_wp, 0.6_wp, 1.0_wp/)
    ! .. Local Scalars ..
    INTEGER :: i, it, num_jac_eval, num_resid_eval, num_time_iter, &
     num_time_step
    REAL (wp) :: acc, pi, piby2, t_end, t_start
    LOGICAL :: first_call
    TYPE (nag_pde_parab_1d_comm_wp) :: comm_ode
    TYPE (nag_pde_parab_1d_cntrl_wp) :: control
    ! .. Local Arrays ..
    REAL (wp) :: abs_loc_tol(npde*npts), rel_loc_tol(npde*npts), r_comm(1), &
     u(npde*npts), u_interp(npde,ninterp,itype), x_bkpts(npts_bk)
    ! .. Executable Statements ..
    WRITE (nag_std_out,*) &
     ' Example Program Results for nag_pde_parab_1d_ex02 '

    pi = nag_pi(0.0_wp)
    piby2 = 0.5_wp*pi

    ! Set initial conditions and accuracy

    t_start = 0.0_wp
    acc = 1.0E-4_wp
    t_end = 0.1E-4_wp
    first_call = .TRUE.
    r_comm(1) = piby2
    rel_loc_tol = acc
    abs_loc_tol = acc

    ! Set the break-points
```

```
      DO i = 1, npts_bk
        x_bkpts(i) = -1.0_wp + 2.0_wp*REAL((i-1),kind=wp)/REAL((npts_bk-1), &
         kind=wp)
      END DO

      WRITE (nag_std_out,999) deg_poly, nel
      WRITE (nag_std_out,998) acc, npts
      WRITE (nag_std_out,997) x_interp(1:ninterp)

      ! Initialize the structure control and set required control parameters

      CALL nag_pde_parab_1d_cntrl_init(control)

      control%l2_norm = .TRUE.

      DO it = 1, 5
        t_end = 10.0_wp*t_end

        CALL nag_pde_parab_1d_coll(pde_coef,bound_cond,init_value,deg_poly, &
         t_start,t_end,x_bkpts,u,first_call=first_call,comm_ode=comm_ode, &
         rel_loc_tol=rel_loc_tol,abs_loc_tol=abs_loc_tol,control=control, &
         num_time_step=num_time_step,num_resid_eval=num_resid_eval, &
         num_jac_eval=num_jac_eval,num_time_iter=num_time_iter,r_comm=r_comm)

        first_call = .FALSE.

        ! Interpolate at required spatial points

        CALL nag_pde_interp_1d_coll(u,x_bkpts,deg_poly,comm_ode,x_interp, &
         u_interp)

        WRITE (nag_std_out,996) t_end, u_interp(1,1:ninterp,1)
        WRITE (nag_std_out,995) u_interp(2,1:ninterp,1)
      END DO

      ! Free structure comm_ode  allocated by NAG fl90

      CALL nag_deallocate(comm_ode)

      ! Print integration statistics

      WRITE (nag_std_out,994) num_time_step, num_resid_eval, num_jac_eval, &
       num_time_iter

999   FORMAT (' Polynomial degree =',I4,'   No. of elements = ',I4)
998   FORMAT (' Accuracy requirement = ',E9.3,'  Number of points = ',I5/)
997   FORMAT ('  T /   X   ',6F8.4/)
996   FORMAT (1X,F6.4,' U(1)',6F8.4)
995   FORMAT (8X,'U(2)',6F8.4/)
994   FORMAT (' Number of integration steps in time                 ', &
       I4/' Number of residual evaluations of resulting ODE sys','tem', &
       I4/' Number of Jacobian evaluations               ',','         ', &
       I4/' Number of iterations of nonlinear solve','r              ',I4/)

    END PROGRAM nag_pde_parab_1d_ex02
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_pde_parab_1d_ex02
Polynomial degree =   3   No. of elements =    9
Accuracy requirement = 0.100E-03  Number of points =    28

 T /   X    -1.0000 -0.6000 -0.2000  0.2000  0.6000  1.0000

0.0001 U(1)  1.0000  0.8090  0.3090 -0.3090 -0.8090 -1.0000
       U(2) -2.4850 -1.9957 -0.7623  0.7623  1.9957  2.4850

0.0010 U(1)  1.0000  0.8085  0.3088 -0.3088 -0.8085 -1.0000
       U(2) -2.5597 -1.9913 -0.7606  0.7606  1.9913  2.5597

0.0100 U(1)  1.0000  0.8051  0.3068 -0.3068 -0.8051 -1.0000
       U(2) -2.6961 -1.9481 -0.7439  0.7439  1.9481  2.6961

0.1000 U(1)  1.0000  0.7951  0.2985 -0.2985 -0.7951 -1.0000
       U(2) -2.9021 -1.8339 -0.6338  0.6338  1.8339  2.9021

1.0000 U(1)  1.0000  0.7939  0.2972 -0.2972 -0.7939 -1.0000
       U(2) -2.9233 -1.8247 -0.6120  0.6120  1.8247  2.9233

Number of integration steps in time                     15
Number of residual evaluations of resulting ODE system 154
Number of Jacobian evaluations                           7
Number of iterations of nonlinear solver                40
```

Example 3: Coupled PDE/ODE System Solved Using Finite Difference Scheme and the BDF Method

This problem consists of a simple coupled system of one PDE and one ODE.

$$(V_1)^2 \frac{\partial U_1}{\partial t} - xV_1\dot{V}_1 \frac{\partial U_1}{\partial x} = \frac{\partial^2 U_1}{\partial x^2},$$

$$\dot{V}_1 = V_1 U_1 + \frac{\partial U_1}{\partial x} + 1 + t;$$

for $t \in [10^{-4}, 0.1 \times 2^i],$ for $i = 1, 2, \ldots, 5,$ $x \in [0, 1]$.

The left boundary condition at $x = 0$ is

$$\frac{\partial U_1}{\partial x} = -V_1 e^t.$$

The right boundary condition at $x = 1$ is

$$\frac{\partial U_1}{\partial x} = -V_1 \dot{V}_1$$

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

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

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

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE nag_pde_parab_1d_ex03_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Default Accessibility ..
  PUBLIC
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  SUBROUTINE pde_coef(t,x,u,du_dx,p,q,r,finish,v,vdot,i_comm,r_comm)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Scalar Arguments ..
    INTEGER, INTENT (OUT) :: finish
    REAL (wp), INTENT (IN) :: t, x
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), INTENT (IN) :: du_dx(:), u(:)
    REAL (wp), INTENT (OUT) :: p(:,:), q(:), r(:)
    REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:), v(:), vdot(:)
    ! .. Executable Statements ..

    finish = 1
    q(1) = -x*du_dx(1)*v(1)*vdot(1)
    r(1) = du_dx(1)
```

```
  p(1,1) = v(1)*v(1)

END SUBROUTINE pde_coef

SUBROUTINE bound_cond(t,u,du_dx,bound,beta,gamma,finish,v,vdot,i_comm, &
   r_comm)
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC EXP
  ! .. Scalar Arguments ..
  INTEGER, INTENT (IN) :: bound
  INTEGER, INTENT (OUT) :: finish
  REAL (wp), INTENT (IN) :: t
  ! .. Array Arguments ..
  INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
  REAL (wp), INTENT (OUT) :: beta(:), gamma(:)
  REAL (wp), INTENT (IN) :: du_dx(:), u(:)
  REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:), v(:), vdot(:)
  ! .. Executable Statements ..

  finish = 1

  beta(1) = 1.0_wp
  IF (bound==0) THEN
    gamma(1) = -v(1)*EXP(t)
  ELSE
    gamma(1) = -v(1)*vdot(1)
  END IF

END SUBROUTINE bound_cond

SUBROUTINE init_value(x_pde,u,x_ode,v,i_comm,r_comm)
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC EXP, SIZE
  ! .. Array Arguments ..
  INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
  REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:), x_ode(:)
  REAL (wp), INTENT (OUT) :: u(:,:)
  REAL (wp), OPTIONAL, INTENT (OUT) :: v(:)
  REAL (wp), INTENT (IN) :: x_pde(:)
  ! .. Local Scalars ..
  INTEGER :: npts_pde
  REAL (wp) :: t_start
  ! .. Executable Statements ..
  t_start = r_comm(1)
  npts_pde = SIZE(x_pde)

  u(1,1:npts_pde) = EXP(t_start*(1.0_wp-x_pde(1:npts_pde))) - 1.0_wp
  v(1) = t_start

END SUBROUTINE init_value

SUBROUTINE ode_coef(t,v,vdot,x_ode,ucp,ducp_dx,rcp,ducp_dt,d2ucp_dtdx,f, &
   finish,g_in_f,i_comm,r_comm)
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Scalar Arguments ..
  INTEGER, INTENT (OUT) :: finish
  REAL (wp), INTENT (IN) :: t
```

```
    LOGICAL, OPTIONAL, INTENT (IN) :: g_in_f
    ! .. Array Arguments ..
    INTEGER, OPTIONAL, INTENT (IN) :: i_comm(:)
    REAL (wp), INTENT (IN) :: d2ucp_dtdx(:,:), ducp_dt(:,:), ducp_dx(:,:), &
     rcp(:,:), ucp(:,:), v(:), vdot(:), x_ode(:)
    REAL (wp), INTENT (OUT) :: f(:)
    REAL (wp), OPTIONAL, INTENT (IN) :: r_comm(:)
    ! .. Executable Statements ..

    IF (g_in_f) THEN
      f(1) = vdot(1) - v(1)*ucp(1,1) - ducp_dx(1,1) - 1.0_wp - t
      finish = 1
    ELSE
      f(1) = vdot(1)
      finish = -1
    END IF

  END SUBROUTINE ode_coef

  SUBROUTINE exact_sol(t,x_pde,exact_u)
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC EXP, SIZE
    ! .. Scalar Arguments ..
    REAL (wp), INTENT (IN) :: t
    ! .. Array Arguments ..
    REAL (wp), INTENT (OUT) :: exact_u(:)
    REAL (wp), INTENT (IN) :: x_pde(:)
    ! .. Local Scalars ..
    INTEGER :: npts_pde
    ! .. Executable Statements ..
    npts_pde = SIZE(x_pde)

    exact_u(1:npts_pde) = EXP(t*(1.0_wp-x_pde(1:npts_pde))) - 1.0_wp

  END SUBROUTINE exact_sol

END MODULE nag_pde_parab_1d_ex03_mod
PROGRAM nag_pde_parab_1d_ex03

  ! Example Program Text for nag_pde_parab_1d
  ! NAG fl90, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_out
  USE nag_pde_parab_1d, ONLY : nag_pde_parab_1d_comm_wp => &
   nag_pde_parab_1d_comm_dp, nag_pde_parab_1d_cntrl_wp => &
   nag_pde_parab_1d_cntrl_dp, nag_pde_parab_1d_cntrl_init, &
   nag_pde_parab_1d_fd, nag_deallocate
  USE nag_pde_parab_1d_ex03_mod, ONLY : pde_coef, bound_cond, init_value, &
   ode_coef, exact_sol, wp
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC REAL
  ! .. Parameters ..
  INTEGER, PARAMETER :: npde = 1, npts_ode = 1, npts_pde = 21, num_ode = 1
  INTEGER, PARAMETER :: neqn = npde*npts_pde + num_ode
  ! .. Local Scalars ..
  INTEGER :: i, it, num_jac_eval, num_resid_eval, num_time_iter, &
   num_time_step
```

```
REAL (wp) :: acc, t_end, t_start
LOGICAL :: first_call
TYPE (nag_pde_parab_1d_comm_wp) :: comm_ode
TYPE (nag_pde_parab_1d_cntrl_wp) :: control
! .. Local Arrays ..
REAL (wp) :: abs_loc_tol(neqn), exact_u(npde*npts_pde), &
 rel_loc_tol(neqn), r_comm(1), u(neqn), x_ode(npts_ode), x_pde(npts_pde)
! .. Executable Statements ..
WRITE (nag_std_out,*) &
 ' Example Program Results for nag_pde_parab_1d_ex03 '

! Set initial conditions and accuracy

t_end = 0.0_wp
first_call = .TRUE.
acc = 1.0E-4_wp
t_start = 1.0E-4_wp
r_comm(1) = t_start
rel_loc_tol = acc
abs_loc_tol = acc

WRITE (nag_std_out,997) acc, npts_pde

! Set spatial mesh points

DO i = 1, npts_pde
  x_pde(i) = REAL((i-1),kind=wp)/REAL((npts_pde-1),kind=wp)
END DO

x_ode(1) = 1.0_wp

WRITE (nag_std_out,999) x_pde(1), x_pde(5), x_pde(9), x_pde(13), &
 x_pde(21)

! Initialize the structure control and set required control parameters

CALL nag_pde_parab_1d_cntrl_init(control)

control%matrix_type = 'F'

DO it = 1, 5
  t_end = 0.1_wp*(2.0_wp**it)

  CALL nag_pde_parab_1d_fd(pde_coef,bound_cond,init_value,t_start,t_end, &
   x_pde,u,first_call=first_call,ode_coef=ode_coef,num_ode=num_ode, &
   x_ode=x_ode,comm_ode=comm_ode,rel_loc_tol=rel_loc_tol, &
   abs_loc_tol=abs_loc_tol,control=control,num_time_step=num_time_step, &
   num_resid_eval=num_resid_eval,num_jac_eval=num_jac_eval, &
   num_time_iter=num_time_iter,r_comm=r_comm)

  first_call = .FALSE.

  CALL exact_sol(t_end,x_pde,exact_u)

  WRITE (nag_std_out,998) t_start
  WRITE (nag_std_out,995) u(1), u(5), u(9), u(13), u(21), u(22)
  WRITE (nag_std_out,994) exact_u(1), exact_u(5), exact_u(9), &
   exact_u(13), exact_u(21), t_start
END DO

! Free structure comm_ode  allocated by NAG fl90
```

```
      CALL nag_deallocate(comm_ode)

      ! Print integration statistics

      WRITE (nag_std_out,996) num_time_step, num_resid_eval, num_jac_eval, &
       num_time_iter

999   FORMAT (’  X        ’,5F9.3/)
998   FORMAT (’ T = ’,F6.3)
997   FORMAT (//’  Simple coupled PDE using BDF ’/’  Accuracy require’, &
       ’ment =’,E10.3,’ Number of points = ’,I4/)
996   FORMAT (’ Number of integration steps in time = ’,I6/’ Number o’, &
       ’f function evaluations = ’,I6/’ Number of Jacobian eval’,’uations =’, &
       I6/’ Number of iterations = ’,I6/)
995   FORMAT (1X,’App.  sol.  ’,F7.3,4F9.3,’  ODE sol. =’,F8.3)
994   FORMAT (1X,’Exact sol.  ’,F7.3,4F9.3,’  ODE sol. =’,F8.3/)

    END PROGRAM nag_pde_parab_1d_ex03
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_pde_parab_1d_ex03


 Simple coupled PDE using BDF
 Accuracy requirement = 0.100E-03 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    0.001  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.    0.493    0.379    0.273    0.175    0.002  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.    1.229    0.901    0.622    0.383    0.008  ODE sol. =   0.798
Exact sol.    1.226    0.896    0.616    0.377    0.000  ODE sol. =   0.800

T =  1.600
App.  sol.    3.959    2.610    1.629    0.917    0.027  ODE sol. =   1.594
Exact sol.    3.953    2.597    1.612    0.896    0.000  ODE sol. =   1.600

T =  3.200
App.  sol.   23.472   11.976    5.886    2.665    0.074  ODE sol. =   3.184
Exact sol.   23.533   11.936    5.821    2.597    0.000  ODE sol. =   3.200

 Number of integration steps in time =     19
 Number of function evaluations =    187
 Number of Jacobian evaluations =     6
 Number of iterations =     50
```

Additional Examples

Not all example programs supplied with NAG *fl*90 appear in full in this module document. The following additional examples, associated with this module, are available.

`nag_pde_parab_1d_ex04`

Parabolic equation solved using finite difference scheme and remeshing with the BDF method:

This example uses Burgers Equation, a common test problem for remeshing algorithms, given by

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

for $x \in [0,1]$ and $t \in [0,1]$, where ϵ is a small constant.

The initial and boundary conditions (of Dirichlet type) are given by the exact solution

$$U(x,t) = \frac{0.1\exp(-A) + 0.5\exp(-B) + \exp(-C)}{\exp(-A) + \exp(-B) + \exp(-C)};$$

where

$$A = \frac{50}{\epsilon}(x - 0.5 + 4.95t),$$

$$B = \frac{250}{\epsilon}(x - 0.5 + 0.75t),$$

$$C = \frac{500}{\epsilon}(x - 0.375).$$

`nag_pde_parab_1d_ex05`

Elliptic-parabolic PDEs solved using collocation scheme and the BDF method:

The problem consists of a fourth-order PDE which can be written as a pair of second-order elliptic-parabolic PDEs for $U_1(x,t)$ and $U_2(x,t)$,

$$0 = \frac{\partial^2 U_1}{\partial x^2} - U_2,$$

$$\frac{\partial U_2}{\partial t} = \frac{\partial^2 U_2}{\partial x^2} + U_2\frac{\partial U_1}{\partial x} - U_1\frac{\partial U_2}{\partial x};$$

where $-1 \leq x \leq 1$ and $t \geq 0$. The boundary conditions are given by

$$\frac{\partial U_1}{\partial x} = 0 \text{ and } U_1 = 1 \text{ at } x = -1, \text{ and}$$

$$\frac{\partial U_1}{\partial x} = 0 \text{ and } U_1 = -1 \text{ at } x = 1.$$

The initial conditions at $t = 0$ are given by

$$U_1 = -\sin\frac{\pi x}{2} \text{ and } U_2 = \frac{\pi^2}{4}\sin\frac{\pi x}{2}.$$

The absence of boundary conditions for $U_2(x,t)$ does not pose any difficulties provided that the derivative flux boundary conditions are assigned to the first PDE which has the correct flux, $\frac{\partial U_1}{\partial x}$. The conditions on $U_1(x,t)$ at the boundaries are assigned to the second PDE by setting $\beta_2 = 0.0$ in the boundary condition equation (6) (see the procedure `nag_pde_parab_1d_coll` document) and placing the Dirichlet boundary conditions on $U_1(x,t)$ in the function γ_2.

`nag_pde_parab_1d_ex06`

Coupled PDE/ODE system solved using finite difference scheme and remeshing:

This problem consists of a simple coupled system of one PDE and one ODE.

$$(V_1)^2 \frac{\partial U_1}{\partial t} - xV_1\dot{V}_1 \frac{\partial U_1}{\partial x} = \epsilon \frac{\partial^2 U_1}{\partial x^2},$$

$$\dot{V}_1 = V_1 U_1 + \epsilon \frac{\partial U_1}{\partial x} + 1 + t,$$

for $t \in [10^{-4}, 0.2],\ x \in [0, 1]$, where ϵ is a small constant.

The left boundary condition at $x = 0$ is

$$\frac{\partial U_1}{\partial x} = -\frac{V_1}{\epsilon} \exp\left(\frac{t}{\epsilon}\right).$$

The right boundary condition at $x = 1$ is

$$\epsilon \frac{\partial U_1}{\partial x} = -V_1\dot{V}_1$$

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

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

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

This example has been derived from the Example 3 by adding a small diffusive term. To capture the steep gradient of the solution in the neighbourhood of $x = 0$, as $\epsilon \longrightarrow 0$ (see Figure 1), the remeshing process has been used.

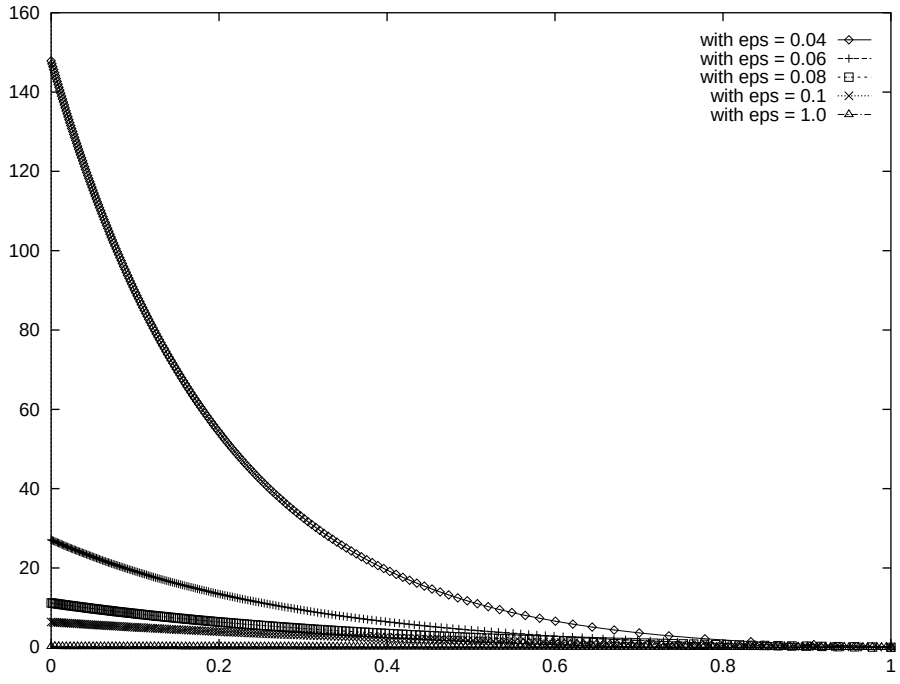

Figure 1. Solution at $t = 0.2$.

`nag_pde_parab_1d_ex06`

Figure 2 shows the mesh distribution on the interval $[0; 1]$ which becomes finer in the neighbourhood of $x = 0$, as $\epsilon \longrightarrow 0$.

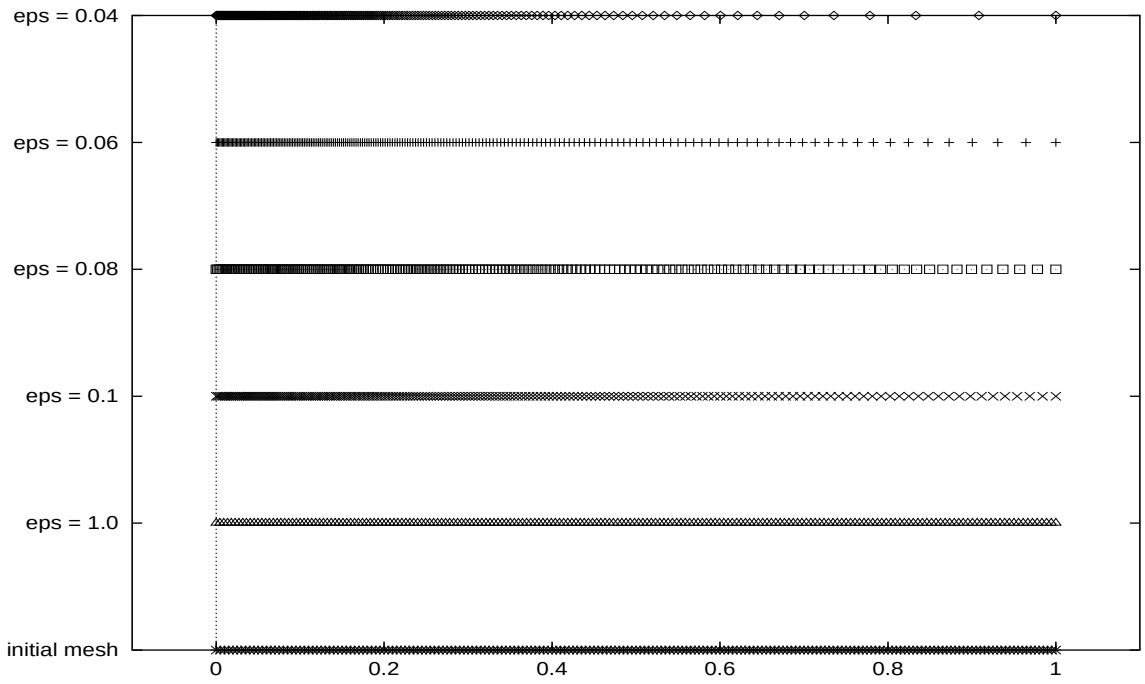

Figure 2. Distribution of mesh points after the remeshing process.

Notice that as $\epsilon \longrightarrow 0$ the problem becomes more difficult to solve and the number of integration steps in time increases quite rapidly.

`nag_pde_parab_1d_ex07`

Coupled PDE/ODE system solved using finite difference scheme and the Theta method:

This is the same example as the third example, but this one is solved using the Theta method to solve the ODE system.

`nag_pde_parab_1d_ex08`

Parabolic equation solved using finite difference scheme and remeshing with the Theta method: This is the same example as the fourth example, but this one is solved using the Theta method to solve the ODE system.

`nag_pde_parab_1d_ex09`

Elliptic-parabolic PDEs solved using collocation scheme and the Theta method:

This is the same example as the fifth example, but this one is solved using the Theta method to solve the ODE system.

References

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

[2] Mikhlin S G and Smolitsky K L (1967) *Approximate Methods for the Solution of Differential and Integral Equations* Elsevier

[3] 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

[4] Skeel R D and Berzins M (1990) A method for the spatial discretization of parabolic equations in one space variable *SIAM J. Sci. Statist. Comput.* **11 (1)** 1–32

[5] Dew P M and Walsh J (1981) A set of library routines for solving parabolic equations in one space variable *ACM Trans. Math. Software* **7** 295–314

[6] Berzins M and Dew P M (1991) Algorithm 690: Chebyshev polynomial software for elliptic-parabolic systems of PDEs *ACM Trans. Math. Software* **17** 178–206

[7] Zaturska N B, Drazin P G and Banks W H H (1988) On the flow of a viscous fluid driven along a channel by a suction at porous walls *Fluid Dynamics Research* **4**

[8] 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

[9] Berzins M, Dew P M and Furzeland R M (1988) Software tools for time-dependent equations in simulation and optimisation of large systems *Proc. IMA Conf. Simulation and Optimization* (ed A J Osiadcz) Clarendon Press, Oxford 35–50

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