# POST - A Package for Solving Partial Differential Equations in One Space Variable.

*N. L. Schryer*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

A formulation is presented for the numerical solution of partial differential equations in one space dimension. The formulation also allows ordinary differential equations in time to be coupled to the partial differential equations in space and time. This extension of the classical partial differential equation formulation allows physical phenomena such as shock-waves and steep fronts to be followed automatically and accurately. It also allows non-local operators to be dealt with, such as integro-partial-derivative operators.

An implementation of the algorithm in portable Fortran, called POST ( Partial and Ordinary differential equations in Space and Time ), is described. The package is especially easy to use since only the spatial mesh and the accuracy desired in the solution of the equations in time need to be specified. The time evolution is then automatically carried out to achieve the desired accuracy. A user's guide to POST is given along with many examples.

**The examples are available through electronic mail.**

There are 10 examples and the **fortran** for each is available in single or double precision. For example, the command

mail research!netlib
send only postx1 from port
send only dpostx10 from port
.

will cause you to receive in the mail the first example in single and the tenth example in double precision.

August 30, 1984

# POST - A Package for Solving Partial Differential Equations in One Space Variable.

*N. L. Schryer*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

**Table of Contents**

Introduction ........................................ pp 1-2
Statement of **pde-bc** Problem ........................................ p 3
Statement of **pde-bc-ode** Problem ........................................ pp 3-4
General Method of Solution ........................................ p 6
Examples
  Linear Heat Equation ........................................ p 7
  Coupled System ........................................ pp 7-8
  Simple **pde-bc-ode** Coupling ........................................ p 8
  Nonlinear Heat Equation with Periodic **bc**s ........................................ pp 8-9
  Galerkin's Method Forces Fluxes ( **a** ) to be Continuous ........................................ p 9
  Two-Phase Moving Front ........................................ p 10
  Moving Shock Wave ........................................ p 11
    Hyperbolic Caveat ........................................ p 11
  Tracking a Boundary Layer ........................................ pp 11-12
  An Integro-Differential Equation ........................................ p 12
  Conversion of a Fourth Order **pde** into Second Order System ........................................ pp 12-13
  A First Order Hyperbolic **pde** with **two bc**s ........................................ pp 13-14
  Incorrect **ic**s ........................................ p 14
  Converting Static into Transient Problems ........................................ p 14
  Automatic Mesh Refinement ........................................ pp 14-15
  Error Estimation ........................................ pp 15-16
Software
  The Software ........................................ pp 17-19
    Scratch Space Used ........................................ p 19
    Defining the **pde** ........................................ pp 19-20
    Defining the **bc**s ........................................ p 21
    Defining the **ode**s ........................................ p 21
    Output ........................................ pp 21-22
    Double Precision Version ........................................ p 22
    Trouble in the **pde**, **bc** or **ode** ........................................ pp 22-23
  Linearly Parameterized Piecewise Linear Mappings
    Mapping with All Partials ........................................ p 23
    Mapping with Only Some Partials ........................................ p 23
    Generating a Mapping ........................................ pp 23-24
    The Inverse of a Mapping ........................................ p 24
    Time-step Restriction for a Mapping ........................................ p 24
  **pde** Mapping Software
    Mapping into User System ........................................ pp 24-25
    Mapping into Internal System ........................................ pp 25-27

**bc** Mapping Software
Mapping into User System ........ pp 27-29
Mapping into Internal System ........ pp 28-29
Example ........ pp 29-30
Time Discretization
Reduction of **pde-bc-ode** to Nonlinear Equations ........ p 31
Linearization ........ pp 31-32
Solution of Linearization ........ pp 32-33
Spatial Discretization
Reduction of **bc**s to Canonical Form ........ pp 34-35
**bc** Placement ........ pp 35-37
Galerkin's Method for Linear Systems ........ pp 37-39
Computational Cost of Galerkin's Method ........ p 39
Sporting Advice
Outline of Software ........ p 40
To Override Default Procedures ........ pp 40-41
Error Control ........ pp 41-42
Initial Newton Method Iterate ........ pp 42-43
Scaling the Linear System ........ pp 43-44
Twiddling Knobs ........ pp 44-46
Table of Values ........ p 46
Examples of Different Parameter Settings ........ pp 46-48
Table Summarizing Results ........ p 47
Example of LBE Blowing it ........ p 47
Example of CN Blowing it ........ p 47
Linear Semi-implicit Caveat ........ p 48
Example of LBE Being Slow ........ p 48
Run-time Statistics ........ p 48
Other Ways into Software ........ pp 49-50
Acknowledgements ........ p 51
B-splines - Appendix 1
Extrapolation - Appendix 2
A Wish List for Future Work - Appendix 3
Accuracy Problems for Collocation-Least-Squares Techniques ........ p 2
Program Examples - Appendix 4
Linear Heat Equation ........ pp 1-6
Coupled System ........ pp 6-9
Simple **pde-bc-ode** Coupling ........ pp 9-13
Nonlinear Heat Equation with Periodic **bc**s ........ pp 13-15
Two-Phase Moving Front ........ pp 15-21
Moving Shock Wave ........ pp 21-26
Tracking a Boundary Layer ........ pp 27-32
An Integro-Differential Equation ........ pp 32-37
Automatic Mesh Refinement ........ pp 37-41
Error Estimation ........ pp 42-45
Reference Guide to Calling Sequences and Common Regions - Appendix 5
Error States and their Interpretation - last 3 pages of Appendix 5.
Cross Reference Index - Appendix 6.

## 1. Introduction.

Many physical problems require the solution of partial differential equations (**pde**'s) in one space variable. Typically these equations are sufficiently complex that their solution must be carried out numerically.

This paper describes a formulation for solving systems of **pde**'s in one spatial variable and time. The formulation allows for terms of the form $\mathbf{u}$, $\mathbf{u}_x$, $\mathbf{u}_t$, $\mathbf{u}_{xt}$, $\mathbf{u}_{xx}$, $\mathbf{u}_{xxt}$ in the **pde**'s, and $\mathbf{u}$, $\mathbf{u}_x$, $\mathbf{u}_t$, $\mathbf{u}_{xt}$ in the boundary conditions ( **bc**'s ), where $\mathbf{u}$ is a vector of **pde** variables, and $\mathbf{u}_x$ denotes $\partial\mathbf{u}/\partial x$, etc. The formulation also allows ordinary differential equations in time to be coupled to the partial differential equations in space and time; see section 2. This extension of the classical partial differential equation formulation allows physical phenomena such as shock-waves and steep fronts to be followed automatically and accurately. It also allows non-local operators to be dealt with, such as integro-partial-derivative operators. Many examples are formulated in section 4.

The numerical solution technique employs Galerkin's method in space, using B-splines, and a variable order, variable time-step extrapolated backward difference procedure in time; see section 3 for an outline, and sections 6 and 7 for a detailed description. Section 5 is a user manual for the software called `POST` for Partial and Ordinary differential equations in Space and Time. Section 8 describes ways of making `POST` run faster than the default settings allow and also describes alternative ways of entering and using the package.

Appendix 1 gives a brief tutorial on B-splines. Appendix 2 gives a brief tutorial on extrapolation. Appendix 3 discusses improvements that could be made in `POST`. Appendix 4 presents the programs used to solve the examples discussed in section 4.

Appendix 5 summarizes the basic procedures available in `POST`, along with their arguments, and gives a list of error states and problems that may arise when using `POST`, along with the common causes of such difficulties. Appendix 6 is a cross-reference index to the paper.

**Getting Started.**

If you have not read this screed before, want to solve a simple **pde** and have no interest in fancy things having nothing to do with your immediate needs, just do the following

- Read section 2, Statement of the Problem, pp 3-4.
- Read section 4, Formulation, Example 1, p 7.
- Skim section 5, Software, pp 17-22.
- Read Appendix 4, Programs, Example 1, pp 1-6.
- Copy the example program ( either from a file or the paper ).
- Run it and make sure it works as advertised.
- Alter it to solve your problem; see the start of Appendix 4 for the steps involved here.

With luck, your problem is now solved. The above scheme typically involves changing only a couple dozen lines of code in the example to get a problem solved.

**The Examples**

If the above "Getting Started" reading does not seem to address your problem, there are many examples discussed in section 4. One of them is quite likely to be of help. The examples are

- A simple **pde**, the heat equation, see Example 1, p-7.
- A coupled system of **pde**s, see Example 2, pp 7-8.
- A coupled **pde-ode** system, see Example 3, p 8.
- A **pde** with periodic **bc**s, see Example 4, pp 8-9.
- A moving front ( $\mathbf{u}_x$ discontinuous ), see Example 5, pp 9-11.
- A moving shock wave ( $\mathbf{u}$ discontinuous ), see Example 6, p 11.
- A moving boundary layer, see Example 7, pp 11-12.
- An integro-**pde**, see Example 8, p 12.
- A fourth order **pde** ( $\nabla^4$ ) reduced to two coupled second order **pde**s, see pp 12-13.
- How to handle initial conditions that conflict with the **pde**, **bc**s and/or the **ode**s, see p 14.
- Converting a steady-state problem ( no $\mathbf{u}_t$ ) into a transient problem, see p 14.

- Automatic mesh refinement to achieve a given accuracy, see Example 9, p 15.
- Error estimation for reliability, see Example 10, pp 15-16.

**A Principle.**

The guiding principle used during the design of `POST` was

> It is better a user complain the package runs slowly than
> complain the package cannot solve the problem at hand.

As a result, people wanting to solve various model equations in a hurry, so they can get on to other models and problems ( that is, people whose time costs more than machine time ), should find `POST` very useful. However, people wanting to solve the same problem many times in a production environment may find `POST` slow for their needs; see section 8 for ways to speed `POST` up considerably.

**History.**

Several other Fortran software packages are available for solving **pde**'s in one spatial variable, for example [47,48]. These packages assume that the **pde** has the form

$$\mathbf{u}_t = \mathbf{f}(\, t,\, x,\, \mathbf{u},\, \mathbf{u}_x,\, (\mathbf{D}\mathbf{u}_x)_x \,) \, , \tag{1.1}$$

where $\mathbf{D}(t,\, x, \mathbf{u})$ is a diffusion coefficient, with boundary conditions of the form

$$\mathbf{u} = \boldsymbol{\alpha}(t)$$

or

$$\boldsymbol{\alpha}(t, \mathbf{u}) + \boldsymbol{\beta}(t, \mathbf{u})\mathbf{u}_x = \boldsymbol{\gamma}(t, \mathbf{u}). \tag{1.2}$$

While this type of formulation covers a wide range of physically interesting problems, it does not cover problems with $\mathbf{u}_{xt}$ or $\mathbf{u}_{xxt}$ terms [52], nor does it deal with non-local statements such as periodic boundary conditions [34]. Finally, the formulation (1.1)-(1.2) does not allow for the coupling of **ode**'s in time with the **pde**'s in space and time [22,26,42,51,56,57].

**2. Statement of the Problem.**

The general form of equations that can be handled by `POST` will be described in two stages. The first will be a description of an essentially classical, text-book divergence-form **pde** with a general set of boundary conditions ( **bc**s ). The second stage will present an extension of the classical form that couples variables depending only upon time, rather than both time and space, to the classical **pde-bc** formulation.

**The pde-bc Formulation.**

The general **pde-bc** form that can be solved with the approach used in `POST` is given by the following equations, where **u** is a vector of **pde** variables of length $n_u$. The full **pde-bc-ode** form is described later in this section. All physical laws that are second order in space are in divergence form, so the **pde**'s are assumed to be in semi-linear, divergence-form

$$\frac{\partial}{\partial x}\, \mathbf{a}(t,\, x,\, \mathbf{u},\, \mathbf{u}_x,\, \mathbf{u}_t,\, \mathbf{u}_{xt}) = \mathbf{f}(t,\, x,\, \mathbf{u},\, \mathbf{u}_x,\, \mathbf{u}_t,\, \mathbf{u}_{xt}) \, , \tag{2.1}$$

where **a** and **f** are vector-valued functions of their arguments, for $L \le x \le R$. It is required that the length of **a** and **f** be equal to $n_u$, the number of **pde** variables, that is, the length of the vector **u**. The boundary conditions are assumed to have the form

$$\begin{aligned} \mathbf{b}_L(t,\, \mathbf{u}(t,L),\, \mathbf{u}_x(t,L),\, \mathbf{u}_t(t,L),\, \mathbf{u}_{xt}(t,L)) &= 0 \\ \mathbf{b}_R(t,\, \mathbf{u}(t,R),\, \mathbf{u}_x(t,R),\, \mathbf{u}_t(t,R),\, \mathbf{u}_{xt}(t,R)) &= 0, \end{aligned} \tag{2.2}$$

where $\mathbf{b}_L$ and $\mathbf{b}_R$ are vector-valued functions, of length $n_u$, of their arguments. Any identically zero component of the **bc** vectors $\mathbf{b}_L$ or $\mathbf{b}_R$ is treated as an inactive **bc**. If each of the **pde**'s is second order in space, then each of the **bc**'s will have to be active. If any of the **pde**'s are of order less than 2 in space, some of the **bc**'s must accordingly be inactive. Initial conditions (**ic**'s) $\mathbf{u}(0,x)$ must be supplied, but need not satisfy the **bc**'s (2.2).

A classical example of the above form ( with $n_u = 1$ ) is the heat equation [39]

$$u_t = u_{xx} \quad \text{on} \quad [0,1]$$

subject to boundary conditions

$$u(t,0) = 0 \quad \text{and} \quad u(t,1) = 1$$

with initial conditions

$$u(0,x) = 0.$$

Note that these initial conditions do not satisfy the **bc**'s. For this equation we have

$$a = u_x \quad \text{and} \quad f = u_t$$

with

$$b_L = u(t,0) \quad \text{and} \quad b_R = u(t,1) - 1.$$

Note that the form of (2.1)-(2.2) encompasses parabolic ( $u_t = u_{xx}$ ), elliptic ( $u_{xx} = 0$ ) and hyperbolic ( $u_t = u_x$ ) problems. It also encompasses **pde**'s that have no solution, such as

$$u_x^2 + u_t^2 = -1$$

over the real field.

**The pde-bc-ode Formulation.**

It is often necessary to make a non-local statement about the solution of a **pde**. For example, the solution may be periodic in space [34], or it may be defined in a coordinate system that is rotating, accelerating or being dynamically scaled [42]. Such statements may be accommodated using the ability of POST to couple **ode**'s in time to the **pde-bc** formulation of (2.1)-(2.2). **pde-ode** coupling may also arise naturally in the formulation of a problem [22,26,51,56,57].

The classical formulation (2.1)-(2.2) is not able to deal with such conditions. The formulation is purely local in nature. Thus, the user can only deal with conditions holding at every point *x* using (2.1), or at the end points $x = L$ or *R* using (2.2). This inability to deal with a specific interior point ( $L < x < R$ ) precludes the user from dealing with dynamic coordinate systems, shock waves, integro-partial differential equations and much more.

An extension of (2.1)-(2.2) is now presented that does not suffer from these handicaps. The object is to make software for solving **pde**s more widely usable in the physical sciences.

How may a variable, say $v(t)$, which depends only upon time, be coupled to a variable, say $u(t,x)$, which depends upon both space and time? Well, POST internally represents the solution **u** in terms of a set of B-spline [3] basis functions $B_j\ (x)$, with each component of **u** given by

$$u_i(t,x) = \sum_j U_{ji}(t) B_j(x) \tag{2.3}$$

Since the $B_j$ are known, all information about $\mathbf{u}\ (x,t)$ is contained in the array $\mathbf{U}\ (t)$. POST actually computes $\mathbf{U}\ (t)$ and uses (2.3) to evaluate the solution $\mathbf{u}\ (t,x)$.

The mechanism used by POST to handle **pde-ode** coupling is to say that there are **ode** variables $\mathbf{v}(t)$ which are coupled to the **pde** variables $\mathbf{u}(t,x)$ through the values of the B-spline coefficients $\mathbf{U}(\ t\ )$ of $\mathbf{u}(t,x)$, see (2.3), and their partial derivatives with respect to *t*, $\mathbf{U}_t$. Generalizing (2.1)-(2.2), the **pde** is assumed to have the form

$$\frac{\partial}{\partial x}\ \mathbf{a}(t,\ x,\ \mathbf{u},\ \mathbf{u}_x,\ \mathbf{u}_t,\ \mathbf{u}_{xt},\ \mathbf{v},\ \mathbf{v}_t) = \mathbf{f}(t,\ x,\ \mathbf{u},\ \mathbf{u}_x,\ \mathbf{u}_t,\ \mathbf{u}_{xt},\ \mathbf{v},\ \mathbf{v}_t) \tag{2.4}$$

while the **bc**'s are assumed to have the form

$$\mathbf{b}_L(t,\ \mathbf{u}(t,L),\ \mathbf{u}_x(t,L),\ \mathbf{u}_t(t,L),\ \mathbf{u}_{xt}(t,L),\ \mathbf{v},\ \mathbf{v}_t) = 0$$
$$\mathbf{b}_R(t,\ \mathbf{u}(t,R),\ \mathbf{u}_x(t,R),\ \mathbf{u}_t(t,R),\ \mathbf{u}_{xt}(t,R),\ \mathbf{v},\ \mathbf{v}_t) = 0. \tag{2.5}$$

The **ode**'s determining the **ode** variables **v** are assumed to have the form

$$\mathbf{d}(t,\ \mathbf{U},\ \mathbf{U}_t,\ \mathbf{v},\ \mathbf{v}_t) = 0\ , \tag{2.6}$$

where **d** is a vector-valued function of its arguments. The length of **d** must be the number of **ode** variables, $n_v$, that is, the length of the vector **v**.

The **pde-bc-ode** combination must have the property that if the **ode** solution vector $\mathbf{v}(t)$ were *given* rather than unknown, then the resulting **pde-bc** problem given by (2.4)-(2.6) would be well-posed. That is, the **ode** must be coupled to a **pde-bc** system that is well-posed in the sense described in section 7.

For certain applications, neither **v** nor $\mathbf{v}_t$ may be present in **d**, and (2.6) may not be a differential equation involving **v** and $\mathbf{v}_t$. In these circumstances, **d** merely represents a condition to be placed upon the **pde** solution **u**. Such a case occurs when a condition like

$$\int_0^1 u(t,x)\,dx = 1$$

is to be imposed on the solution of the **pde**, rather than a "standard" boundary condition. By writing, see Appendix 1 to note that $\int_0^1 B_i\ (x)\ dx = (\ x_{i+k} - x_i\ )/k$,

$$\int_0^1 u(t,x)\,dx \equiv \sum_i\ U_i(t)(\ x_{i+k} - x_i\ )/k = 1$$

we get a condition of the form (2.6). In general, any statement about the solution of the **pde** is equivalent to a statement about its B-spline coefficients in (2.3).

**Apologia**

For readers mathematically pure-at-heart, formulation (2.4)-(2.6) usually seems awkward, at least initially. Coupling $\mathbf{u}\ (\ \cdot\ ,\ t\ )$ to $\mathbf{v}(\ t\ )$ via $\mathbf{U}(\ t\ )$ appears to be indirect and to require too much of the user - namely, knowledge of *how* the solution is internally represented by `POST`. However, (2.4)-(2.6) requires the minimal possible knowledge of the user. To see this, consider a clean functional coupling of the form

$$\Phi\ (\ u\ (\ \cdot\ ,\ t\ )\ ,\ v\ (\ t\ )\ ) = 0.$$

Since the numerical procedure used generally requires the partial derivatives of $\Phi$ with respect to **U** and $v$,

$$\frac{\partial}{\partial\,\mathbf{U}_i}\ \Phi\ (\ \sum_j\ \mathbf{U}_j\ (\ t\ )\ B_j\ (\ \cdot\ )\ ,\ v\ (\ t\ )\ ) \equiv \frac{\partial\,\Phi}{\partial\,u}\ B_i\ (\ \cdot\ )$$

is required, where $\frac{\partial\Phi}{\partial u}$ is the Frechet derivative [38], an operator. Thus, the program must ask the user for the action of that operator on the basis splines $B_i$, one way or another. I have found that most physical scientists do not know what a Frechet derivative is, and moreover, they do not want to know. Explaining Frechet derivatives and asking users to evaluate their action on the $B_i$ is not necessary if the user is simply told to take the plain, regular, Calculus I derivative of

$$\Phi\ (\ \sum_j \mathbf{U}_j\ (\ t\ )\ B_i\ (\ \cdot\ )\ ,\ \mathbf{v}\ (\ t\ )\ )$$

with respect to the $\mathbf{U}_i$ and **v**. Since the user must know about the solution representation to give initial conditions and to evaluate the solution, the path of least resistance has been to eschew the mathematically cleaner

$$\Phi\ (\ u\ (\ \cdot\ ,\ t\ )\ ,v\ (\ t\ )\ ) = 0$$

formulation for that of (2.6), which only requires standard differentiation.

**3. General Method of Solution.**

Let the solution $\mathbf{u}(t,x)$, for a given instant in time, be approximated by a B-spline [3,4,5] of **order** $k$ on a **mesh** $X(1) \le \cdots \le X(NX)$, see Appendix 1. That is, each component of the solution will be approximated by a piecewise polynomial function of degree less than $k$, with $k-2$ continuous derivatives, where $k \ge 2$ is any integer the user desires. If we set $h = \max_i \left| X(i+1) - X(i) \right|$ then the error, $\max_x \left| u_i(t,x) - \hat{u}_i(t,x) \right|$ is $O(h^k)$, for some B-spline $\hat{u}_i$, see [5]. Since $k$ may be taken to be any integer $\ge 2$, this gives a powerful technique for approximating the solution $\mathbf{u}(t,x)$ in space. We can use the Rayleigh-Ritz-Galerkin (R-R-G) method [50,45] to find essentially the projection of the solution of the **pde** onto the space of B-splines we have selected. This reduces the **pde**'s in space and time to **ode**'s in time [27,50] for the coefficients $U_{ji}(t)$ in the expansion (2.3).

Thus, after the spatial discretization, only **ode**'s in time remain to be solved. Since these **ode**'s are known to be in general "stiff" [17,18], an implicit differencing scheme must be used to solve them. This virtually requires that the partial derivatives of the **a** and **f** in (2.4), of the $\mathbf{b}_L$ and $\mathbf{b}_R$ in (2.5), and of the **d** in (2.6) with respect to their arguments be known, either analytically or numerically.

The next step is the solution of these time-varying **ode**'s. Here we assume that some basic one-step **ode** solver is available. For example, a backwards-Euler or Crank-Nicholson scheme [39], or an exponentially-fitted technique [31], or even an explicit method such as Gragg's modified mid-point rule [30,31], could be used.

All the above techniques, and many others, have the property that for a given time-step $\delta$ they produce an approximate solution accurate to $O(\delta^\gamma)$, where typically $\gamma$ is 1 or 2. Moreover, if the equations are solved using time-steps of $\delta$ and $\delta/2$, the results of these two computations can be combined using extrapolation [8,30] to obtain a result accurate to $O(\delta^{2\gamma})$. This process can be repeated indefinitely, so a basic process of accuracy $\delta^\gamma$ can be used to generate a sequence of processes of accuracy $O(\delta^\gamma)$, $O(\delta^{2\gamma}), \cdots,$ $O(\delta^{P\gamma}), \cdots$ , see Appendix 2.

A step-size and order monitor is available [43,44] for carrying out this extrapolation process and *automatically* deciding what time-step $\delta$ and order $P\gamma$ should be used, given the accuracy desired in the solution. The user need only specify how accurately the solution in time should be computed, and the time integration then proceeds automatically, with no need for the user to worry about choosing $\delta$.

The algorithm implemented by `POST` for solving such **pde**'s then consists of 3 steps:

1) Discretize the equations in space using R-R-G with B-splines.
2) Produce a one-step method for solving the resulting **ode**'s.
3) Feed that one-step process to the extrapolation step-size and order monitor.

Sections 6 and 7 describe the above steps in considerable detail.

**4. Examples - Formulation.**

This section gives the formulation of many examples in terms of (2.4)-(2.6). See Appendix 4 for programs that solve these problems using `POST`. The first two examples only deal with **pde**-**bc** problems and the remainder deal with the full **pde**-**bc**-**ode** system.

**Example 1 - A Simple Heat Equation.**

As a simple example of the use of `POST`, consider solving the scalar heat equation

$$u_t = u_{xx} + g(t,x) \qquad \text{for} \quad x \,\varepsilon\, (0,1), \tag{4.1}$$

where the source term $g(t,x)$ is chosen ( $g = (x-t^2)e^{xt}$ ) so that the solution is a known function, $u(t,x) = e^{xt}$. The boundary conditions are then taken to be

$$\begin{aligned} u(t,0) &= 1 \\ u(t,1) &= e^t \end{aligned} \tag{4.2}$$

with initial conditions

$$u(0,x) = 1 . \tag{4.3}$$

The **pde** (4.1) is equivalent to (2.4) with

$$a = -u_x \quad \text{and} \quad f = g(t,x) - u_t$$

while the **bc**s (4.2) are equivalent to (2.5) with

$$b_L = u\,(t,0) - 1$$

$$b_R = u(t,1) - e^t .$$

See example 1 in Appendix 4 for code solving this problem.

**Example 2 - A Coupled System of pdes.**

The solution of a coupled system of **pde**'s is now illustrated. In example 1, $u_x$ may be computed from the B-spline representation for $u$. However, $u$ is accurate to $O(h^k)$, while $u_x$ is only accurate to $O(h^{k-1})$, where $h$ is the B-spline mesh spacing used, see Appendix 1. This example shows how $u_x$ may be computed accurate to $O(h^k)$ by considering (4.1)-(4.3) as a system of 2 coupled **pde**'s through setting $u_1 = u$ and $u_2 = u_x$. This is a silly way to get $u_x$ more accurately ( a better way would be to use a finer mesh ), but it provides a simple system of **pde**s to solve. This example also illustrates the process of reducing a higher order **pde** to a system of lower order **pde**s. The **pde** (4.1) then becomes

$$\begin{aligned} u_{1t} &= u_{2x} + (x - t^2)\,e^{xt} \\ & \qquad\qquad \text{for} \quad x \,\varepsilon\, (0,\ 1) \\ u_{1x} &= u_2 \end{aligned} \tag{4.4}$$

and the **bc**'s (4.2) become

$$\begin{aligned} u_1(t,0) &= 1 \\ u_1(t,1) &= e^t . \end{aligned} \tag{4.5}$$

The initial conditions are

$$\begin{aligned} u_1(0,x) &= 1 \\ u_2(0,x) &= 0. \end{aligned} \tag{4.6}$$

The **pde** (4.4) is equivalent to (2.4) with

$$a_1 = -u_{2x} \quad \text{and} \quad f_1 = -u_{2t} + (\,x - t^2\,)\,e^{xt}$$

$$a_2 = u_1 \quad \text{and} \quad f_2 = u_2$$

while the **bc**s (4.5) are equivalent to (2.5) with

$$b_L = u_1(t,0) - 1$$

$$b_R = u_1(t,1) - e^t .$$

See example 2 in Appendix 4 for code solving this problem.

**Example 3 - Simple pde-ode Coupling.**

The first example of the use of `POST` to solve a **pde**-**bc**-**ode** combination is contrived to be simple but illustrative. Consider the **pde**

$$u_t = u_{xx} + v(t) + g(t,x) \quad \text{for} \quad x \,\varepsilon\, (0,1) \tag{4.7}$$

with the coupled **ode**

$$v_t(t) = u(t,\tfrac{1}{2}) , \tag{4.8}$$

where $g(t,x)$ is chosen so that the solution is known, say $u(t,x) = \cos(xt)$ and $v(t) = 2\sin(t/2)$. The **bc**'s are then taken to be

$$u(t,0) = 1$$
$$u(t,1) = \cos(t) \tag{4.9}$$

with initial conditions

$$u(0,x) = 1$$
$$v(0) = 0\,. \tag{4.10}$$

The **pde** (4.7) is equivalent to (2.4) with

$$a = -u_x \qquad \text{and} \qquad f = v(t) + g(t,x) - u_t$$

while the **bc**s (4.9) are equivalent to (2.5) with

$$b_L = u(t,0) - 1$$

$$b_R = u(t,1) - \cos(t)\,.$$

If we choose $i$ so that $x_i \le \frac{1}{2} < x_{i+1}$, where **x** is the spatial mesh, then, see Appendix 1,

$$u(t, \tfrac{1}{2}) \equiv \sum_{j=i+1-k}^{i} U_j(t)\, B_j(\tfrac{1}{2})$$

and the **ode** (4.8) is equivalent to (2.6) with

$$d = v_t\,(t) \; - \sum_{j=i+1-k}^{i} U_j(t)\, B_j(\tfrac{1}{2})\,.$$

Not to worry about finding $i$ above, there is a Port Library [25] routine, `INTRVR`, for getting it. See example 3 in Appendix 4 for code solving this problem.

**Example 4 - Nonlinear Heat Equation with Periodic Boundary Conditions.**

The second **pde-bc-ode** example is the use of `POST` to solve a nonlinear heat equation subject to periodic boundary conditions. Consider the **pde**

$$u_t = u_{xx} - u^3 + g(t,x) \qquad \text{for} \quad x \,\varepsilon\, (-\pi,+\pi)\;, \tag{4.11}$$

subject to periodic boundary conditions

$$u(t,-\pi) = u(t,+\pi) \tag{4.12a}$$

$$u_x(t,-\pi) = u_x(t,+\pi)\;, \tag{4.12b}$$

where $g(t,x)$ is chosen to make the solution $u$ a known function, say $u(t,x) = \cos(x)\sin(t)$. We must rewrite (4.11)-(4.12) slightly to put them into the form (2.4)-(2.6). Define an **ode** variable $v(t)$, which will play the role of $u_x(t,-\pi) = u_x(t,+\pi)$. The **bc**'s used are

$$u_x(t,-\pi) = v(t)$$
$$u_x(t,+\pi) = v(t) \tag{4.13}$$

which force (4.12b) to hold. The remaining condition, (4.12a), is used to determine $v(t)$. Relation (4.12a) is an example of an **ode** of the form (2.6) having neither **v** nor $\mathbf{v}_t$ present. The **ode** used to determine $v(t)$ is then

$$u(t,-\pi) = u(t,+\pi)\;, \tag{4.14}$$

which forces (4.12a) to hold. The effect of (4.13)-(4.14) is to treat (4.11)-(4.12) as a **pde-bc** problem whose solution has floating, $v(t)$, boundary derivative values that are to be determined by (4.14). Relation (4.14) is converted to form (2.6) by noting that $u(t, -\pi) = U_1$ and $u(t, +\pi) = U_{nmesh-k}$, see Appendix 1, where **U** are the B-spline coefficients of (2.3) and *nmesh* is the length of the mesh used.

The **pde** (4.11) is equivalent to (2.4) with

$$a = -u_x \qquad \text{and} \qquad f = -u^3 + g(t,x) - u_t$$

while the **bc**s (4.13) are equivalent to (2.5) with

$$b_L = u_x(t, -\pi) - v(t)$$

$$b_R = u_x(t, +\pi) - v(t)$$

and the **ode** (4.14) is equivalent to (2.6) with

$$d = U_1 - U_{nmesh-k}.$$

See example 4 of Appendix 4 for code solving this problem.

It should be noted that the choice of (4.12b) as the **bc**s for (4.11), with (4.12a) being the **ode**, was *not* random. At first glance, (4.12a) and (4.12b) appear quite interchangable in their roles as either **bc**s or **ode**. However, it would be a *big* mistake to reverse their roles. Most any condition can be safely used in the **bc**s, but the same is not true of the **ode**s. The **bc**s are only forced to hold in the Galerkin sense, which means conditions on $\mathbf{u}_x$ are only forced to hold approximately, see section 7; they are not forced to hold exactly because that would lower the rate of convergence for the solution **u** from $O(h^k)$ to $O(h^{k-1})$. Since the **ode**s *are* forced to hold exactly, if (4.12b) were chosen to be the **ode**, the convergence rate would be $O(h^{k-1})$, or substantially less than optimal.

**More Meaty Examples**

There are many excellent schemes for spatially discretizing (2.4)-(2.6), reducing them to a system of **ode**'s, and thus solving the problem. The most elegant and most acceptable to people in the physical sciences is Galerkin's method [45,50]. That method has the property that the term $\mathbf{a}(\cdot)$ in (2.4) is forced to be continuous everywhere, see [45]. Since $\mathbf{a}(\cdot)$ is a flux, physically, and physicists expect fluxes to be continuous, this means that Galerkin's method gives the solution the physicists want. Other spatial discretization methods such as finite-differences, least-squares and collocation do not automatically make $\mathbf{a}(\cdot)$ continuous. Most of the rest of the examples in this section are stripped down versions of real problems and the use of Galerkin's method in `POST` is important in their formulation. Moving fronts, shocks and boundary layers will be dealt with, as well as integro-partial differential operators.

**Example 5 - A Two-phase, Moving Front.**

This example considers a parabolic **pde** with a moving interface and represents a simplified version of a program used to study the action of lasers in semiconductor annealing [51]

$$u_t = (c(u)\, u_x)_x + g(x, t) \qquad \text{for} \quad x \,\varepsilon\, (-1, 2), \tag{4.15}$$

where $c(u)$ is a piecewise analytic function of *u*, depending on the sign of $u - 1$. At the position $x(t)$ where *u* hits 1 there is an interface condition

$$c(u)\, u_x \Big|_{x_-(t)}^{x_+(t)} = \alpha \frac{dx}{dt} \tag{4.16}$$

which simply says that the jump in the flux across the moving interface is proportional to the velocity of the interface. For $u < 1$ the semiconductor is solid and for $u > 1$ it is liquid.

First, rewrite the **pde** (4.15) as

$$u_t = \frac{\partial}{\partial x} \left( c(u)\, u_x - H(x, x(t))\, \alpha\, x'(t) \right) + g(x, t) \tag{4.17}$$

where

$$H(\ x,\ x(t)\ )\ \equiv \begin{cases} 0 & x\ <\ x(t) \\ 1 & x\ >\ x(t). \end{cases}$$

In this latter form, when Galerkin's method is used, the interface condition (4.16) is forced to hold.

Now all we need to do is solve (4.17) in a moving coordinate system that tracks $x(t)$. This is easy to accomplish. Let $x(\ \xi;\ v_1(t)\ )\ :\ [0,\ 2]\ \rightarrow [-1,\ 2]$, with $x(\ 1;\ v_1(t)\ )\ =\ v_1(t)$, be a change of independent spatial variable $x$ to $\xi$. In such a mapping, we will want to choose $v_1(t)$ to be $x(t)$, which is unknown. Specifically, let

$$x(\ \xi;\ v_1(t)\ )\ \equiv \begin{cases} \xi\ (\ v_1(t)\ +\ 1\ )\ -\ 1 & 0 \le \xi \le 1, \\ v_1(t)\ +\ (2-v_1(t)\ )\ (\ \xi\ -1) & 1 \le \xi \le 2. \end{cases} \tag{4.18}$$

We can rewrite the **pde** (4.17) in terms of $w(\xi,\ t)\ \equiv\ u(x(\xi;\ v_1(t)),\ t)$ and $v_1(t)$. In fact, POST will do the **pde** and **bc** mapping automatically for us, see section 5. The user tells POST to solve the problem for **w** in the fixed $\xi$ coordinate system, but the **pde** and **bc** definitions can still work in the physical **u** and $x$ system by using the mapping software. The **pde** (4.17) is equivalent to (2.4) with

$$a\ =\ c(u)\ u_x\ -\ H(x,\ v_1\ (t)\ )\ \alpha\ v_{1t}\ (t)$$

$$f\ =\ u_t\ -\ g(t,x).$$

To determine $v_1(t)$ we use

$$\mathbf{d}\ =\ w(\ 1,\ t\ )\ -\ 1.$$

where, as in example 3 above, $w(\ 1,\ t\ )$ will have to be evaluated via (2.3).

Note that since the mapping (4.18) is piecewise analytic, $w(\ \xi,\ t)$ will have the same property. Thus, it is important for the spatial discretization of $w$ to allow $w_\xi$ to be discontinuous at $\xi\ =\ 1$. For example, if B-splines of order $k$ are used then the mesh point $\xi\ =\ 1$ must be of multiplicity $k-1$, see Appendix 1. See example 5 in Appendix 4 for the way to do this.

The above spatial mapping scheme has been used to study the use of lasers in annealing semiconductor substrates [51], to track boundary layers [22], and to follow the motion of magnetic domain walls [42].

Coordinate transformations of the form (4.18) are used so frequently that an automatic interface has been provided to map **pde**'s and **bc**'s from one spatial coordinate system to another, see section 5.

**Example 6 - A Moving Shock Wave.**

This example consists of a hyperbolic **pde** with a moving shock-wave

$$u_t\ =\ -\ u_x\ +\ g\ (\ x,\ t\ ) \qquad \text{for}\ \ x\ \varepsilon\ (a,\ b) \tag{4.19}$$

whose solution has a jump discontinuity at $x\ =\ x(t)$. Let $\hat{u}(\ x,\ t\ )\ \equiv\ u\ +\ v_1(t)\ H(\ x,\ x(t)\ )$, where, as before, $H(x,\ x(t)\ )$ is a Heavyside function based at $x(t)$ and $v_1(t)$ is the unknown height of the jump at $x\ =\ x(t)$. Then $\hat{u}(\ x,\ t)$ will be continuous, and we may rewrite the **pde** (4.19) as

$$\hat{u}_t\ -\ v_{1t}(t)\ H(x,x(t)\ )\ =\ -\ \hat{u}_x\ +\ g(t,x). \tag{4.20}$$

As in the previous example, we now must solve (4.20) in a moving coordinate system that tracks the unknown position $x(t)$. Let $x(\ \xi;\ v_2(t)\ )\ :\ [0,\ 2]\ \ \rightarrow\ \ [a,\ b]$ be given by

$$x(\ \xi;\ v_2(t)\ )\ \equiv\ \begin{cases} a\ +\ (\ v_2(t)\ -\ a\ )\ \xi & 0 \le \xi \le 1, \\ v_2(t)\ +\ (\ b\ -\ v_2(t)\ )\ (\ \xi - 1) & 1 \le \xi \le 2, \end{cases} \tag{4.21}$$

so that $x(1;\ v_2(t)\ )\ =\ v_2(t)$. The variables $v_1(t)$ and $v_2(t)$ represent the unknown height and position of the shock, respectively. To determine them we use the two Rankine-Hugoniot relations [15] for the problem at hand. A typical set of jump conditions would be of the form

$$\mathbf{d}\ :\ \ \begin{cases} \hat{u}(x(t)_+,\ t)\ =\ u_0(t) \quad \text{(given)} \\ x'(t)\ =\ u(x(t)_+,\ t)\ -\ u(x(t)_-,\ t). \end{cases} \tag{4.22}$$

We can then solve for $w(\xi, t) \equiv \hat{u}(x(t, \xi; v_2(t)), t)$, $v_1(t)$ and $v_2(t)$ on the fixed spatial domain $[0,2]$. If we choose $\mathbf{v}(0)$ to describe the shock correctly initially, then it will continue to describe the shock correctly for all time. The **pde** (4.20) is equivalent to (2.4) with

$$a = -\hat{u}$$

$$f = \hat{u}_t - g(x,t) - H(x,x(t))\, v_{1t}.$$

Note that in the $\xi$ coordinate system (4.22) becomes

$$\mathbf{d} = \begin{cases} w(1_+, t) = u_0(t), \\ x_{v_2}(1; v_2(t))\, v_{2t}(t) = v_1 . \end{cases} \tag{4.23}$$

The two conditions in (4.23) determine the position ( $v_2(t)$ ) and height ( $v_1(t)$ ) of the shock. See example 6 in Appendix 4 for code solving this problem.

**A Hyperbolic Caveat.**

Systems of hyperbolic **pde**s are sometimes not as nice as (4.19) to solve numerically. Specifically, the numerical scheme may be *unstable* if the **pde**s and **bc**s are not put into a suitably scaled canonical form. Users of `POST` should be aware of analysis and algorithms for performing this scaling, see [36] for more details. If present, the instability grows exponentially in time, so its presence is easily detected and remedies [36] may then be applied.

**Example 7 - Tracking a Moving Boundary Layer.**

Consider tracking a boundary layer in a **pde** like

$$u_t = u_{xx} + g(x, t) \qquad \text{for } x \,\varepsilon\, (0, 1) \tag{4.24}$$

where $g$ is chosen so that the solution is $e^{-x\,t}$. If the integration is over, say, the time interval $[\,20,\, 10^6\,]$, (4.24) becomes a formidable boundary layer tracking problem. Any attempt to solve this problem is faced with tracking a moving and very steep front. A scheme will be presented that has worked well on many problems, see [22,51]. For more complicated problems other mesh distribution schemes could be used; this problem is presented to illustrate the power of the mesh mapping facility to deal with really nasty boundary layers.

The trick is to assume that the boundary layer has an exponentially damped shape. This is physically natural, aside from being exact for the problem at hand. It doesn't really make much difference to the scheme if the front is exponentially or Gaussian damped. Under the assumption of $e^{-c\,\xi}$ behavior, it is obvious that if we use a map from $[\,0,\,2\,]$ to $[\,0,\,1\,]$, as done in (4.21) above, and force

$$u\,(\,x(1,t)/[\,\mathrm{D}\ln 10\,]\,,\,t\,) \equiv 1/e \tag{4.25}$$

to hold for some D, then $u(\,x(1,t)\,,\,t\,) \equiv 10^{-\mathrm{D}}$. Using (4.25) as the **ode** we can then solve (4.24) accurately in a moving coordinate system. That is, the interval in $\xi$, $[\,0,\,1\,]$, will be mapped into the boundary layer, keeping the mapped mesh points in the region of interest, provided that D is chosen sufficiently large, say D = 3. See example 7 in Appendix 4 for code solving this problem.

**Example 8 - An Integro-Partial Differential Equation.**

This example is an integro-differential equation

$$u_t = p\,u_{xx} + \int_0^1 c(x,y)\; u(y,t)\,dy \qquad \text{for } x \,\varepsilon\, (0, 1), \tag{4.26}$$

where $c(x,y)$ is some given kernel. This is clearly beyond the scope of formulation (1.1)-(1.2). It is trivial to solve using (2.4)-(2.6). Simply introduce **ode** variables $v_1(t), \cdots, v_N(t)$, where $N$ is the number of basis functions to be used in the spatial discretization of the problem. Let

$$\mathbf{d} = \mathbf{v} - \mathbf{U} = \mathbf{0}.$$

Then $\mathbf{v} \equiv \mathbf{U}(t)$ and we can functionally couple $u$ back into the **pde** (4.26) via $\mathbf{v}$. To see this, note that from (2.3) we have

$$\int_0^1 c(x,y)\ u(y,\ t)\ dy \equiv \sum_i U_i \int_0^1 c(x,y)\ B_i(y)\ dy.$$

If we let $c_i(x) \equiv \int_0^1 c(x,y)\ B_i(y)\ dy$, this lets us rewrite (4.26) as

$$u_t = p\ u_{xx} + \sum_{i=1}^{N} v_i(t)\ c_i(x). \tag{4.27}$$

which has the desired form (2.4)-(2.6). This is a very simple use of (2.4)-(2.6) to solve an integro-partial differential equation. If the integral operator were nonlinear, rather than linear, a suitable quadrature rule would have to be employed, rather than defining the $c_i(x)$. See example 8 of Appendix 4 for code solving such an equation.

**Even More Examples.**

The following examples are given to show some of the fine points of using `POST`. Many of these examples are not supported by code in Appendix 4. The examples will deal with the conversion of higher order systems of **pde**s to lower order systems, converting static problems into transient ones for robustness, tricking `POST` into believing a hyperbolic problem is elliptic, automatic generation of the spatial mesh for static problems, automatic mesh refinement and error estimation schemes.

**Conversion of a Fourth Order pde into `POST` Form.**

This example comes from some of the preliminary work done in [32] studying a fourth order equation in two space dimensions that models the onset of turbulent fluid flow. Consider the **pde**

$$z_t = -\ \nabla^4\ z \qquad \text{for} \quad x\ \varepsilon\ (0,1) \tag{4.28}$$

with periodic **bc**s

$$z(t,0) = 0 = z(t,1) \tag{4.29a}$$

$$z_x(t,0) = z_x(t,1). \tag{4.29b}$$

(4.29a) represents Dirichlet data for $z$ and (4.29b) represents Neumann data for $z$.

This problem could be attacked directly via (2.4)-(2.6) by simply setting $u_1 = z$ and $u_2 = \nabla^2\ z$

The **pde** would then read

$$u_{1t} = -\nabla^2\ u_2 \tag{4.30a}$$

$$u_2 = \nabla^2 u_1. \tag{4.30b}$$

However, `POST` would not be able to solve (4.28) in the above form because it is singular! This is most easily seen by noting that (4.30b) is

$$\nabla^2\ u_1 = u_2$$

which when viewed as an equation for $u_1$ given $u_2$ with **bc**s given by the Neumann data in (4.29b) has infinitely many solutions: If any given $u_1$ solves it, then so does $u_1 + c$ where $c$ is any constant.

By slightly altering the definition of $u_2$ we can nearly get a stable formulation, but not quite. Let $u_1 = z$ and $u_2 = \nabla^2\ u_1 + u_1$. Then the **pde** (4.28) becomes

$$\begin{aligned} u_{1t} &= -\nabla^2\ u_2 + u_2 - u_1 \\ \nabla^2\ u_1 &= u_2 - u_1, \end{aligned} \tag{4.31}$$

but when **bc**s (4.29) are used, `POST` would again find the problem singular. The problem is that $u_2$ appears to second order in the **pde**, but the **bc**s (4.29) give no boundary data for it. So we will have to fake up some data for it, but it will have to be consistent with the Dirichlet data in (4.29a).

If we let $v_1$ and $v_2$ be two **ode** variables, and let them play the roles of boundary data for $u_2$, then we can use the Neumann data from (4.29b) for $u_1$ and the **ode** for **v** can be the Dirichlet data from (4.29a). To be specific, the boundary conditions are taken to be

$$u_{1x}(t,0) = 0 = u_{1x}(t,1)$$

$$u_2(t,0) = v_1$$
$$u_2(t,1) = v_2$$

with the **ode**

$$u_1(t,0) = 0 = u_1(t,1).$$

The effect is to treat **pde** (4.31) as having **bc**s for $u_2$ that are unknown, but to be determined so that the Dirichlet data of (4.29a) holds. This last formulation is stable and effective.

Instead of $u_2 = \nabla^2 z + z$, we could have used $u_2 = \nabla^2 z + \varepsilon z$ where $\varepsilon$ is any number smaller than the smallest eigenvalue of $\nabla^2$ on $(0,1)$, or $\pi^2$. $\varepsilon = 1$ fits the bill nicely.

**A First Order Hyperbolic pde with Two Boundary Conditions**

Consider the **pde**

$$u_t = ( x u )_x \qquad \text{for} \quad x \,\varepsilon\, ( -1, +1 )$$

which requires boundary data at each end-point of $( -1, +1 )$. The formulation (2.4)-(2.6) doesn't allow this directly; see section 7 for the reasons. Because only $u_x$ appears on the right hand side of the **pde**, it believes boundary data can only be given at *one* boundary point. However, the formulation can be fooled quite easily. By adding a term of the form $\varepsilon\, u_{xx}$ to the right-hand side of the **pde**, we get a second order operator and two **bc**s are allowable. If $\varepsilon$ is chosen small enough, say $10^{-30}$, the extra term doesn't alter the equation or its solution at all, and we get the desired solution.

The situation is not so nice however when we consider the above **pde** with the term $x\,u$ replaced by $-\,x\,u$. Then there can be no boundary data, because the characteristics are all outgoing at the boundary. There is no swindle that can get `POST` to solve this problem, it will demand boundary data and then die on it. The package needs to handle such weird hyperbolic equations and their boundary data properly, but does not yet do so.

**Incorrect Initial Conditions**

Sometimes it is not convenient to have the initial conditions exactly obey conditions in (2.6). For example, when tracking a steep front, it may be easy to start the mesh in a reasonable position, but difficult to exactly obey the definition used in **d**. Say that you want to use $\mathbf{d} \equiv T(\mathbf{u}, \mathbf{v})$, but that $T(\mathbf{u}_0, \mathbf{v}_0)$ is not $\mathbf{0}$ for the initial conditions. That will mean that Newton's method will have to work hard on the first few time steps of the solution, and it may even fail to converge. Indeed, the solution will tend to be jumpy if **d** starts off non-zero. A simple way to fix that problem is to use something like $\mathbf{d} = T(\mathbf{u}, \mathbf{v}) - T(\mathbf{u}_0, \mathbf{v}_0)\, e^{-10^6 t}$ which has the property that $\mathbf{d} = \mathbf{0}$ and yet for $t \geq 10^{-4}$ we have $\mathbf{d} \approx T$. So long as the constant in the exponential is chosen sufficiently large, in a short time the altered **d** will be the correct **d**, and the solution will exist and be smooth for all time. There will be no jumping about at time 0.

**Converting Steady State Equations to Transients.**

Assume that you want to solve the steady state **pde**

$$0 = \nabla^2 u - e^u \tag{4.32}$$

and you have some reasonable initial approximation to the solution. You can give the problem to `POST` and, if the initial conditions are reasonably close to the solution, it will get the solution correctly.

However, if you cannot get a sufficiently close approximation to the solution, then `POST` may die because Newton's method cannot converge with such an inappropriate starting guess. One way around such problems is to convert the static problem into a transient one. For example, the above problem can be written as the equilibrium solution ( as time goes to $\infty$ ) of

$$u_t = \nabla^2 u - e^u \tag{4.33}$$

This has the great advantage that from *any* initial guess there is a solution to the transient problem. Furthermore, as time goes to $\infty$, the solution must approach that of (4.32). This makes the solution of (4.32) bomb proof and may even provide physically interesting results, see [56].

The process of conversion to a transient problem is quite general. Assume that you have a static problem, generalizing (4.32), of the form

$$0 = \nabla \cdot g( \cdots ) + h( \cdots ) \tag{4.34}$$

where $g$ and $h$ are functions of $u$, $\nabla u$, time and space. Then the equation

$$u_t = \nabla \cdot g( \cdots ) + h( \cdots ) \tag{4.35}$$

possesses a solution and that solution approaches the solution of (4.34) as time goes to $\infty$. The term $u_t$ was chosen on the assumption that $g$ has a term like $c(t,x,u)\ \nabla u$ in it, where $c > 0$. If $c < 0$ was the case, the term in (4.35) should be $-\ u_t$. Allowing the coefficients of $u_t$ and $u_{xx}$ to have opposite signs in (4.35) will guarantee instability. Make sure they have the same sign, and all will be well.

The above process can also be applied to systems, component by component.

**Example 9 - Automatic Mesh Refinement**

Assume that you have some static **pde** and some general idea where the solution is kinky, but you have no idea how fine to make the spatial mesh to get the desired accuracy, see [56]. The following pseudo-code will automatically determine the mesh for you. It starts from an initial, crude guess and keeps halving the mesh spacing until the error estimate is small enough.

```
Get an initial mesh x0 of length nx0.
Get an initial approximation u0 to the solution on the mesh x0.

error = ∞
While ( error > tol )
    {
    Get mesh x1 from mesh x0 by halving the mesh spacing.
    Get a spline fit u1 to u0 on the mesh x1.

    Solve the pde to time ∞ starting from u1, using mesh x1.

    Estimate the error in u0, given u0 and u1.

    x0 = x1; nx0 = nx1; u0 = u1
    }
```

provided that all the arrays are dimensioned large enough. See example 9 in Appendix 4 for code implementing the above mesh refinement scheme.

**Example 10 - Estimating the Error in the Computed Solution.**

Picking a spatial mesh and an accuracy requirement for the time evolution of a **pde** guarantees absolutely **nothing** about the accuracy of the computed solution. Without at least a little preliminary error estimation, the computed results may look nice and be garbage. The error in a computed solution can be easily estimated however.

There are two sources of error in the computed solution of (2.4)-(2.6). The first is the error due to the spatial discretization and the second is due to the time discretization. Crudely speaking, the error in a solution **u** is of the form

$$O(h^k) + O(e_1 \|\mathbf{u}\|_\infty + e_2) \tag{4.36}$$

where $h$ is the spatial mesh spacing and $e_1$ and $e_2$ are user settable numbers controlling the time evolution error. To make sure that $\mathbf{u}$ is as accurate as we want, some testing of the size of the two components of (4.36) is in order.

Assume that the solution $\mathbf{u}$ has been obtained on the mesh $\mathbf{x}$ using $\mathbf{e}$ for the error tolerances. Then let $\mathbf{u}_h$ be the solution of the same problem but on a mesh $\mathbf{x}_h$ whose mesh spacing is half that of $\mathbf{x}$. Also, let $\mathbf{u}_e$ be the solution of the same problem on the mesh $\mathbf{x}$ but using $\mathbf{e}/10$ for the error controls. We can then estimate the $O(h^k)$ error term in (4.36) as

$$\| \mathbf{u} - \mathbf{u}_h \|_\infty ,$$

because $\mathbf{u}_h$ is so much more accurate than $\mathbf{u}$ that it can be regarded as the true solution of the problem.

Similarly, we estimate the $O(e_1 \| \mathbf{u} \|_\infty + e_2)$ term in (4.36) as

$$\| \mathbf{u} - \mathbf{u}_e \|_\infty ,$$

because, again, $\mathbf{u}_e$ is so much more accurate than $\mathbf{u}$ that it can be regarded as the true solution of the problem.

We can assemble the above two estimates for the terms in (4.36) to obtain

$$\| \mathbf{u} - \mathbf{u}_{true} \|_\infty \le \| \mathbf{u} - \mathbf{u}_h \|_\infty + \| \mathbf{u} - \mathbf{u}_e \|_\infty \tag{4.37}$$

The estimate (4.37) is easily computable in practice since everything on the right hand side is known.

The first term is easily evaluated, see Appendix 4. The second term is a little trickier to estimate, but it has a pretty result. We have, using (2.3),

$$\| \mathbf{u} - \mathbf{u}_e \|_\infty \equiv \| \sum_i (U_i - U_{ei}) B_i \|_\infty \le \sum_i | U_i - U_{ei} | B_i \le$$
$$\sum_i \| \mathbf{U} - \mathbf{U}_e \|_\infty B_i = \| \mathbf{U} - \mathbf{U}_e \|_\infty \sum_i B_i = \| \mathbf{U} - \mathbf{U}_e \|_\infty .$$

This gives the estimate

$$\| \mathbf{u} - \mathbf{u}_e \|_\infty \le \| \mathbf{U} - \mathbf{U}_e \|_\infty . \tag{4.38}$$

See example 10 in Appendix 4 for code implementing the above estimation scheme.

## 5. Software for the pde-bc Problem.

The algorithm outlined in section 3 and described in detail in sections 6 and 7 has been implemented in EFL [33], a Ratfor-[35]-like Fortran preprocessor language of considerable elegance and power. This section is a brief user's manual for this software package called `POST` (Partial and Ordinary differential equations in Space and Time). Because `POST` is implemented in EFL programs written in Fortran, Ratfor, EFL or other Fortran preprocessing languages may be used to drive and communicate with `POST`.

Before invoking `POST` the user must

- Make a B-spline mesh.
- Make initial conditions for the B-spline coefficients $\mathbf{U}$ in (2.3).
- Write subroutines
  - `AF` - to evaluate $\mathbf{a}$ and $\mathbf{f}$ in (2.4).
  - `BC` - to evaluate $\mathbf{b}$ in (2.5).
  - `DEE` - to evaluate $\mathbf{d}$ in (2.6). If $n_v = 0$, there are no **ode** variables and `POSTD` may be used in place of a dummy `DEE` written by the user.
  - `HANDLE` - to output (print) the solution results.

Each of these preparatory steps are illustrated in Appendix 4 and will not be described here.

The outer layer of the `POST` package is called `POST` and is invoked by

```
Call POST (U,nu,k,X,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,DEE,
           errpar,
           HANDLE)
```

The input to `POST` is

`U` - The B-spline coefficients (2.3) for the initial values of the **pde** variables **u**. `U`($i,j$), for $i=1, \cdots,$ `nx-k`, are the coefficients for $u_j$, $j=1, \cdots,$ `nu`.

`nu` - The number $n_u$ of **pde** variables **u**.

`k` - The B-spline order to be used. `k` $\geq 2$ is necessary.

`X` - The B-spline mesh to be used. The multiplicity of `X`(1) and `X`(`nx`) must be `k`, see Appendix 1. The Port Library routines for making meshes, `UMB` and `LUMB`, guarantee the first and last mesh points have multiplicity `k`.

`nx` - The length of the mesh array `X`.

`V` - The initial conditions for the **ode** variables **v**. `V`($i$) = $v_i$, $i=1, \cdots,$ `nv`. If $n_v$ is 0, `V` may be any array, and need not be initialized, since the contents of `V` are neither interrogated nor altered in any way in this case.

`nv` - The number $n_v$ of **ode** variables **v**.

`tstart` - Start integration at time `tstart`.

`tstop` - Stop integration at time `tstop`. `tstop` should be a variable, *not* a constant, in the program calling `POST`; see output description below.

`dt` - The initial choice for the time-step. The performance of `POST` is substantially independent of the initial value of `dt` chosen. It is sufficient that `dt` be within several orders of magnitude of being "correct." The value of `dt` will automatically be adjusted by `POST` to obtain the solution to the desired accuracy at the least possible cost. Thus, `dt` should be a variable, *not* a constant, in the user's calling program.

`AF` - A subroutine for specifying the **a** and **f** terms in the **pde** (2.4). `AF` must be declared External in the user's calling program. This user-supplied subprogram will be described later. If there is no **pde**, that is $n_u = 0$, then the dummy subroutine `POSTA` may be used in place of `AF`.

`BC` - A subroutine for specifying the boundary conditions **b** in (2.5). `BC` must be declared External in the user's calling program. This user-supplied subprogram will be described later. If there are no **bc**s or $n_u = 0$, then the dummy subroutine `POSTB` may be used in place of `BC`.

`DEE` - A subroutine for specifying the **ode** (2.6). `DEE` must be declared External in the user's calling program. This user-supplied subprogram will be described later. If there is no **ode**, that is $n_v = 0$, then the dummy subroutine `POSTD` may be used in place of `DEE`.

`errpar` - A Real vector of length 2 for determining the error desired (to be allowed) in the solution of the equations in time. The two components govern, roughly, the relative and absolute error in the computed solution. For the $i^{th}$ component of the **pde** solution **u**, the error at each time-step in the time integration will be at most

$$\texttt{errpar}(1) * \|u_i\| + \texttt{errpar}(2)$$

where

$$\|u_i\| = \underset{(L,R)}{Max} \; | \, u_i(t,x) \, | .$$

For each component of the **ode** solution **v**, the error at each time-step in the time integration will be at most

$$\texttt{errpar}(1) * |v_i| + \texttt{errpar}(2).$$

Thus, `errpar`(1)=0 gives the solution accurate to an absolute error of `errpar`(2), and `errpar`(2)=0 gives the solution accurate to a relative error of `errpar`(1). The choice of `errpar`(1) and `errpar`(2) is highly problem dependent. A sound technique is the following: If the scale of the problem is such that *S* is the smallest value for which a prescribed relative error tolerance is desired, then the choice

$$\texttt{errpar}(1) = 10^{-2} \; ; \qquad \texttt{errpar}(2) = 10^{-2} \times \mathrm{S}$$

will essentially give 1% relative accuracy in all values down to around S in size. Values below S will have absolute error smaller than $10^{-2} \times$ S. Users should be very careful to avoid setting `errpar`(2) = 0 when the solution is zero at any point in either space or time, or the integration will die for the obvious reasons.

`HANDLE` - A user-supplied subroutine that will be called by `POST` at the end of each time-step. `HANDLE` must be declared External in the user's calling program. This subprogram will be described later. If no output is desired, then the dummy subroutine `POSTH` may be used in place of `HANDLE`.

The output from `POST` is

- `U` - The B-spline coefficients for the **pde** solution **u** at time `tstop`.
- `V` - The value of **v**(`tstop`).
- `tstop` - The time at which integration stopped. `tstop` may be altered by user-supplied subroutine `HANDLE`. If an error state exists on return (see Appendix 5), `tstop` is set to the last instant in time when the solution was known accurately. Thus, `tstop` should be a variable, *not* a constant, in the user's call to `POST`.
- `dt` - The final value of the "optimal" time-step.

**Scratch Space Used.**

The amount of scratch space used on the dynamic stack of the Port Library [25] is, neglecting lower order terms,

$$n_u \, (n_x - k) \left[ 3k \, n_u + n_v + 18 \right]$$

Real words (storage units), where $n_u$ is the number of **pde**s, $n_x$ is the number of points in the spatial mesh, *k* is the B-spline order and $n_v$ is the number of **ode** variables.

All scratch space for `POST` is taken from the stack. Solving **pde**s, even in one space variable, is a non-trivial process, requiring substantial work space. For virtually all problems solved by `POST` the user will have to declare and initialize the PORT stack to a size larger than the default size of 1000 Real words. This process will be illustrated in the first example of Appendix 4.

**`AF`, `BC` and `DEE` Descriptions.**

The user-supplied subroutines `AF`, `BC` and `DEE`, which define the **pde**-**bc**-**ode** problem to be solved, are now described. When `POST` needs to compute **a** and **f**, it will

```
Call AF(t,Xe,nxe,U,Ux,Ut,Uxt,nu,V,Vt,Nv,
        A,AU,AUx,AUt,AUxt,AV,AVt,
        F,FU,FUx,FUt,FUxt,FV,FVt)
```

Before `POST` calls `AF`, it sets to **0** the 14 arrays `A` through `FVt` and provides the input values

- `t` - The current value of time.
- `Xe` - A list of points where **a** and **f** are to be evaluated. This `Xe` is *not* the B-spline mesh X. The points `Xe` at which **a** and **f** are desired are determined by the quadrature rule used by `POST` to implement Galerkin's method; see section 7 for more detail on this process.

`nxe` - The length of Xe.

`U` - The *values* of $\mathbf{u}$ at the `Xe`($ix$), **not** the B-spline coefficients **U** of (2.3). `U`($ix,j$) = $u_j$(t,`Xe`($ix$)), $ix = 1, \cdots,$ `nxe` and $j = 1, \cdots,$ `nu`.

`Ux` - The values of $\mathbf{u}_x$ at the `Xe`($ix$), stored as above.

`Ut` - The values of $\mathbf{u}_t$ at the `Xe`($ix$), stored as above.

`Uxt` - The values of $\mathbf{u}_{xt}$ at the `Xe`($ix$), stored as above.

`nu` - The number $n_u$ of **pde** variables **u**.

`V` - The values of $\mathbf{v}(t)$. `V`($i$) = $v_i(t)$, $i = 1, \cdots,$ `nv`.

`Vt` - The value of $\mathbf{v}_t(t)$. `Vt`($i$) = $v_{it}\ (t)$, $i = 1, \cdots,$ `nv`.

`nv` - The number $n_v$ of **ode** variables **v**.

`AF` must return as output

`A` - The value of **a** at the `Xe`($ix$). `A`($ix,j$) = $a_j$(t,`Xe`($ix$)), for $ix = 1, \cdots,$ `nxe` and $j = 1, \cdots,$ `nu`.

`AU` - The partial derivatives of **a** with respect to **u** at the `Xe`($ix$). `AU`($ix,i,j$) = $\partial a_i / \partial u_j$ (t, `Xe`($ix$)), for $ix = 1, \cdots,$ `nxe` and $i,j = 1, \cdots,$ `nu`.

`AUx` - The partial derivatives of **a** with respect to $\mathbf{u}_x$ at the `Xe`($ix$), as above.

`AUt` - The partial derivatives of **a** with respect to $\mathbf{u}_t$ at the `Xe`($ix$), as above.

`AUxt` - The partial derivatives of **a** with respect to $\mathbf{u}_{xt}$ at the `Xe`($ix$), as above.

`AV` - The partial derivatives of **a** with respect to **v** at the `Xe`($ix$). `AV`($ix,i,j$) = $\partial a_i / \partial v_j$($t$,`Xe`($ix$)), for $ix = 1, \cdots,$ `nxe`, $i = 1, \cdots,$ `nu`, and $j = 1, \cdots,$ `nv`.

`AVt` - The partial derivatives of **a** with respect to $\mathbf{v}_t$ at the `Xe`($ix$), as above.

`F` - The value of **f** at the `Xe`($ix$). `F`($ix,j$) = $f_j$(t,`Xe`($ix$)), for $ix = 1, \cdots,$ `nxe` and $j = 1, \cdots,$ `nu`.

`FU` - The partial derivatives of **f** with respect to **u** at the `Xe`($ix$). `FU`($ix,i,j$) = $\partial f_i / \partial u_j$ (t, `Xe`($ix$)), for $ix = 1, \cdots,$ `nxe` and $i,j = 1, \cdots,$ `nu`.

`FUx` - The partial derivatives of **f** with respect to $\mathbf{u}_x$ at the `Xe`($ix$), as above.

`FUt` - The partial derivatives of **f** with respect to $\mathbf{u}_t$ at the `Xe`($ix$), as above.

`FUxt` - The partial derivatives of **f** with respect to $\mathbf{u}_{xt}$ at the `Xe`($ix$), as above.

`FV` - The partial derivatives of **f** with respect to **v** at the `Xe`($ix$). `FV`($ix,i,j$) = $\partial f_i / \partial v_j$($t$,`Xe`($ix$)), for $ix = 1, \cdots,$ `nxe`, $i = 1, \cdots,$ `nu`, and $j = 1, \cdots,$ `nv`.

`FVt` - The partial derivatives of **f** with respect to $\mathbf{v}_t$ at the `Xe`($ix$), as above.

When `POST` needs the boundary conditions it will

```
Call BC(t,L,R,U,Ux,Ut,Uxt,nu,V,Vt,nv,
        B,BU,BUx,BUt,BUxt,BV,BVt)
```

Before `POST` calls `BC`, it sets to **0** the 7 arrays `B` through `BVt`, and provides the input

`t` - The current value of time.

`L` - The left-hand end-point of the spatial domain.

`R` - The right-hand end-point of the spatial domain.

`U` - `U`($i$,1) = $u_i$(t,L), `U`($i$,2) = $u_i$(t,R) for $i = 1, \cdots,$ `nu`.

`Ux` - `Ux`(.,1) = $\mathbf{u}_x(t,L)$, `Ux`(.,2) = $\mathbf{u}_x(t,R)$, as above.

`Ut` - `Ut`(.,1) = $\mathbf{u}_t(t,L)$, `Ut`(.,2) = $\mathbf{u}_t(t,R)$, as above.

Uxt - Uxt(.,1) = $\mathbf{u}_{xt}(t,L)$, Uxt(.,2) = $\mathbf{u}_{xt}(t,R)$, as above.

nu - The number $n_u$ of **pde** variables **u**.

V - The value of $\mathbf{v}(t)$, V($i$) = $v_i(t)$, $i=1, \cdots$, nv.

Vt - The value of $\mathbf{v}_t(t)$, Vt($i$) = $v_{it}(t)$, $i=1, \cdots$, nv.

nv - The number $n_v$ of **ode** variables **v**.

BC must return as output

B - B($i$,1) = $b_{Li}$ and B($i$,2) = $b_{Ri}$, $i=1, \cdots$, nu, see (2.2).

BU - BU($i,j$,1) = $\partial b_{Li}/\partial u_j$(t,L) and
BU($i,j$,2) = $\partial b_{Ri}/\partial u_j$(t,R), $i,j=1, \cdots$, nu.

BUx - BUx($i,j$,1) = $\partial b_{Li}/\partial u_{jx}$(t,L) and
BUx($i,j$,2) = $\partial b_{Ri}/\partial u_{jx}$(t,R), $i,j=1, \cdots$, nu.

BUt - BUt($i,j$,1) = $\partial b_{Li}/\partial u_{jt}$(t,L) and
BUt($i,j$,2) = $\partial b_{Ri}/\partial u_{jt}$(t,R), $i,j=1, \cdots$, nu.

BUxt - BUxt($i,j$,1) = $\partial b_{Li}/\partial u_{jtx}$(t,L) and
BUxt($i,j$,2) = $\partial b_{Ri}/\partial u_{jtx}$(t,R), $i,j=1, \cdots$, nu.

BV - BV($i,j$,1) = $\partial b_{Li}/\partial v_j$ and
BV($i,j$,2) = $\partial b_{Ri}/\partial v_j$, for $i,j=1, \cdots$, nv.

BVt - BVt($i,j$,1) = $\partial b_{Li}/\partial v_{jt}$ and
BVt($i,j$,2) = $\partial b_{Ri}/\partial v_{jt}$, for $i,j=1, \cdots$, nv.

DEE **Description.**

When POST needs the value of **d** it will

```
Call DEE(t,k,X,nx,
         U,Ut,nu,nxmk,V,Vt,nv,
         D,DU,DUt,DV,DVt)
```

Before POST calls DEE, it sets to **0** the 5 arrays D through DVt, and provides the input values

t - The current value of time.

k - The B-spline order.

X - The B-spline mesh.

nx - The length of the mesh X.

U - The B-spline coefficients for the **pde** solution **u**, **U**.

Ut - The B-spline coefficients for $\mathbf{u}_t$, $\mathbf{U}_t$.

nu - The number of **pde** variables **u**.

nxmk - nxmk=nx-k is provided so that U and Ut may be dimensioned to be U(nxmk,nu) and Ut(nxmk,nu).

V - The value of $\mathbf{v}(t)$.

Vt - The value of $\mathbf{v}_t(t)$.

nv - The number $n_v$ of **ode** variables **v**.

The output from DEE should be

D - D($i$) = $d_i$, for $i=1, \cdots$, nv.

DU - The value of the partials of **d** with respect to **U**. DU($i$, $j$, $l$) = $\partial d_i/\partial \mathbf{U}_{j\,l}$, $i=1, \cdots$, nv, $j=1, \cdots$, nx-k and $l = 1, \cdots$, nu.

DUt - The value of the partials of **d** with respect to $\mathbf{U}_t$, stored as above.

DV - The value of the partials of **d** with respect to $v_i$. DV($i,j$) = $\partial d_i / \partial v_j$, for $i,j = 1, \cdots$, nv.

DVt - The value of the partials of **d** with respect to $v_{it}$.

HANDLE **Description.**

The user-supplied output and control subroutine HANDLE is now described. The numerical solution at an instant in time is obtained only after some rather lengthy and complex calculations involving trying several small sub-steps in time; see Appendix 2 for more detail. When POST has finally come up with a solution as accurate as requested by the user, it just has to tell the user the good news. At the end of each time-step, POST will

```
Call HANDLE(t0,U0,V0,t,U,V,nu,nxmk,nv,k,X,nx,dt,tstop)
```

so that the user may look at, print out, plot, fondle, or do whatever is desired with the solution. If the output at the end of each time-step is not desired, and only the solution at time tstop is needed, the "Return-End" HANDLE subroutine POSTH may be used.

POST also invokes HANDLE whenever it tries to take a time step and fails to obtain the user desired accuracy. This may be caused by the time step dt being too large. Or it may mean that there is something "funny" going on near time t. In either case, the user may want to know that POST failed at time t. Such things are called "restarts" and are typically expensive and worth knowing about.

The input provided by POST to HANDLE is

t0 - Time at the beginning of the time-step just completed.

U0 - **pde** solution **u** at time t0 is given by B-spline coefficients U0.

V0 - The **ode** solution $\mathbf{v}(t0)$ is given by V0.

t - Time at the end of the time-step just completed. The "current" value of time.

U - **pde** solution **u** at time t is given by B-spline coefficients U. If t0 = t, then a restart is in progress and the values in U are meaningless.

V - The **ode** solution $\mathbf{v}(t)$ is given by V. If t0 = t, then a restart is in progress and the values in V are meaningless.

nu - The number $n_u$ of **pde** variables **u**.

nxmk - nxmk=nx-k is provided so that U0 and U may be dimensioned to be U0(nxmk,nu) and U(nxmk,nu)

nv - The number $n_v$ of **ode** variables **v**.

k - The B-spline order.

X - The B-spline mesh.

nx - The length of the mesh X.

dt - The current "optimal" value of dt.

tstop - The current final value for time.

The output from HANDLE is

t - May be altered by the user. Not too many people want to do this, but it is allowed.

U - May be altered by the user. For example, the user may want to force the solution to be non-negative or monotone.

V - May be altered by the user.

dt - May be altered by the user. The user may want to choose dt so that some particular value of time is achieved on the next time-step.

`tstop` - May be altered by the user. For example, the user may only want to integrate until $\mathbf{u}_t$ is "small enough" and then stop.

**Other Ways to Use and Speed-Up `POST`.**

There are many other "knobs" in `POST` than those described above; section 8 describes their use.

**Double Precision Version.**

The Double Precision version of `POST` is called `DPOST`. The calling sequence for `DPOST` is precisely the same as that for `POST`, with *all* floating-point arguments Double Precision, *except* `errpar`, which remains Real. The amount of scratch space used by `DPOST` on the dynamic stack of the Port Library [25] is, neglecting lower order terms,

$$n_u \, (n_x - k) \left[ 3k \, n_u + n_v + 16 \right]$$

Double Precision words (storage units).

**Trouble in `AF`, `BC` or `DEE`.**

If, for one reason or another, the user cannot evaluate the appropriate functions when any of `AF`, `BC` or `DEE` is called, this fact can be communicated to `POST` through the named Common region

```
Common / POSTF / Failed ; Logical Failed .
```

Before `POST` calls `AF`, `BC` and `DEE`, it sets `Failed` = `.False.` . Thus, if the user doesn't use or even know of the existence of `POSTF`, `POST` assumes that everything has been correctly computed on return from those subroutines. If, however, the user has a problem, uses `POSTF`, and sets `Failed` = `.True.`, `POST` will automatically lower the time-step in an attempt to obtain a more accurate, and hence more reasonable, numerical solution so that `AF`, `BC` and `DEE` can do their job.

The Double Precision version of `POSTF` is `DPOSTF`.

**Linearly Parameterized Piecewise Linear Mappings**

Mappings like (4.18) are very useful and they have been generalized and implemented for use in `POST`. To generalize the piecewise linear map (4.18), we consider a mapping $x\,(\,\xi, \mathbf{v}\,)$ on the interval $[\,0,$ `Nv`$-1\,]$, such that $x(\,i-1, \mathbf{v}\,) \equiv v_i$, and $x\,(\,\xi, \mathbf{v}\,)$ is linear in $\xi$ on each $[\,i-1, i\,]$, for $i = 1, \ldots,$ `Nv-1`. Obviously, on any interval $[\,i-1,\, i\,]$ we have

$$x\,(\,\xi, \mathbf{v}\,) \equiv v_i + (\,\xi - i + 1\,)\,(\,v_{i+1} - v_i\,).$$

Such Linearly parameterized Piecewise Linear Maps (`LPLM`) have been "canned" for use in `POST`. They will be used in many of the examples in Appendix 4. The `LPLM` software is now described.

The

```
Call LPLM(xi,nx,V,Nv, x,xxi,xxiv,xv,xvv,xt,xtv)
```

will evaluate an `LPLM` mapping and its partial derivatives as needed to convert the **pde** and **bc**s from $x$ to $\xi$ coordinates. The input to this procedure is

- `xi` - A list of $\xi$ values for which $x(\,\xi, \mathbf{v}\,)$ is desired.
- `nx` - The length of `xi`.
- `V` - The mapping parameters `V`.
- `Nv` - The number of mapping parameters `V`.

The output of `LPLM` is

- `x` - A Real array of length `nx` with `x`$(i) = x\,(\,\xi_i, t, \mathbf{v}\,(\,t\,)\,)$, for $i = 1, \cdots,$ `nx`.

`xxi` - A Real array of length `nx` with `xxi`($i$) = $x_{\xi}\,(\,\xi_i\,,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\,=\,1,\,\cdots\,,$ `nx`.

`xxiv` - Real `xxiv(nx,Nvm)`, with `xxiv`($i$,$j$) $=\,x_{\xi\,v_j}\,(\,\xi_i\,,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\,=\,1,\,\cdots\,,$ `nx` and $j\,=\,1,\,\cdots\,,$ `Nvm`.

`xv` - Real `xv(nx,Nv)`, with `xv`($i$,$j$) $=\,x_{v_j}\,(\,\xi_i\,,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\,=\,1,\,\cdots\,,$ `nx` and $j\,=\,1,\,\cdots\,,$ `Nv`.

`xvv` - Real `xvv(nx,Nvm,Nvm)`, with `xvv`($i$,$p$,$q$) = $x_{v_p\,v_q}\,(\,v(i)\,,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\,=\,1,\,\cdots\,,$ `nx` and $p,\,q\,=\,1,\,\cdots\,,$ `Nvm`. For these maps `xvv` $\equiv\,0$.

`xt` - A Real array of length `nx` with `xt`($i$) = $x_t\,(\,\xi_i\,,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\,=\,1,\,\cdots\,,$ `nx`. For these maps `xt` $\equiv\,0$.

`xtv` - Real `xtv(nx,Nvm)`, with `xtv`($i$,$j$) $=\,x_{t\,v_j}\,(\,\xi_i\,,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\,=\,1,\,\cdots\,,$ `nx` and $j\,=\,1,\,\cdots\,,$ `Nvm`. For these maps `xtv` $\equiv\,0$.

Example 5 of Appendix 4 shows `LPLM` at work.

It is frequently only necessary to evaluate $x$ and $x_{\mathbf{v}}$ in a map. The following

```
Call LPLMX(xi,nx,V,Nv,x,xv)
```

does such a mapping. The arguments to `LPLMX` are as in `LPLM`.

When the user wants to construct initial conditions for the **v** mapping variables the

```
Call LPLMG(Nv,x,V)
```

will do the trick. The input to `LPLMG` is

`Nv` - The number of mapping parameters to be used.

`x` - Real `x(Nv)`, the $x$ points the user wishes $x\,(\,\xi\,,\,\mathbf{v}\,)$ to take on at $\xi\,=\,0,\,\cdots\,,$nv$-\,1$.

The output of `LPLMG` is

`V` - The **v** values that make the mapping hit the values in x. In fact, $\mathbf{v}\equiv$ x, so the basic function of the procedure is trivial. However, `LPLMG` also does error checking, for `Nv` $<\,0$ and x points being equal to one another. The user is urged to use `LPLMG` to get **v**.

On occasion, the inverse map $\xi\,(\,x,\,\mathbf{v}\,)$ is needed. The

```
Call LPLMI(Nv,V,x,nx,xi)
```

will do the evaluation. The input to `LPLMI` (Inverse) is

`Nv` - The length of `V`.

`V` - Real `V(Nv)`, the mapping parameters.

`x` - Real `x(nx)`, the input `x` points.

`nx` - The length of the `x` array.

The output of `LPLMI` is

`xi` - Real `xi(nx)`, the values of $\xi$ so that $x(\xi_i\,,\,\mathbf{v}\,)\,\equiv\,x_i$, for $i\,=\,1,\,\cdots\,,$ `nx`.

When solving a transient problem, it is necessary that the mapping coordinates **v** do not collide with or cross one another. If the **v** are not monotonically increasing, the mapping they represent is not physically or mathematically well-defined. The results can be garbage in such cases. If they do collide, it can cause `POST` to restart for that time step, an expensive happening. To help prevent such things, the statement

```
dt = LPLMT(t0,V0,Nv,t,V,f,dt)
```

can be used inside the user's `HANDLE` routine. The object of `LPLMT` is to keep the time-step `dt` small enough that the previous history of $\mathbf{v}\,(\,t\,)$, namely (t0,V0) and (t,V), predict that the minimum distance between the **v**'s will fall by at most a factor of `f` over the next time-step. The user **must** declare `LPLMT` Real, because its Fortran default type is Integer. The input to `LPLMT` is `t0` - The input argument to `HANDLE` of the same name.

`V0` - The input argument to `HANDLE` of the same name.

`Nv` - The length of `V` for mapping variables.

`t` - The input argument to `HANDLE` of the same name.

`V` - The input argument to `HANDLE` of the same name.

`f` - The fractional amount by which the minimum distance between consecutive **v**'s may fall over the next time step. f = $10^{-1}$ is typical.

`dt` - The input value of the time step.

The output from `LPLMT` is

`dt` - The time-step that will satisfy the "f" criterion above and not exceed the input value of `dt`.

The Double Precision versions of all the `LPLM` routines are `DLPLM`, with all Real arguments typed Double Precision.

Example 7 of Appendix 4 shows `LPLMT` at work.

**pde Mapping Software.**

The mapping (4.18) is a specific example of a general mapping $x(\xi, t, \mathbf{v})$ which can be used to transform a **pde** for solution by `POST`. `POST` has general purpose mapping transformation procedures for taking the results of a map $x(\xi, t, \mathbf{v})$ and converting the **pde**s and **bc**s from $\mathbf{u}(x, t)$ to $\mathbf{w}(\xi, t) \equiv \mathbf{u}(x(\xi, t, \mathbf{v}(t)), t)$ coordinates.

The user tells `POST` to solve for **w**, but the **pde** and **bc** definition can be made in the physical **u** system using the software to be described shortly.

The **pde** mapping proceeds as follows. We have

$$\mathbf{w}(\xi, t) \equiv \mathbf{u}(x(\xi, \mathbf{v}(t), t), t)$$

and thus

$$\mathbf{w}_t \equiv \mathbf{u}_x \frac{dx}{dt} + \mathbf{u}_t \quad \text{and} \quad \mathbf{w}_\xi \equiv \mathbf{u}_x x_\xi .$$

This implies that **pde**s which began in the form

$$\frac{\partial}{\partial x}\mathbf{a}(t, x, \mathbf{u}, \mathbf{u}_x, \mathbf{u}_t, \mathbf{u}_{xt}, \mathbf{v}, \mathbf{v}_t) = \mathbf{f}(t, x, \mathbf{u}, \mathbf{u}_x, \mathbf{u}_t, \mathbf{u}_{xt}, \mathbf{v}, \mathbf{v}_t)$$

now have the form

$$\frac{\partial}{\partial \xi}\mathbf{a}(t, x(t,\xi, \mathbf{v}), \mathbf{w}, \mathbf{w}_x/x_\xi, \mathbf{w}_t - \mathbf{w}_\xi(x_t + x_\mathbf{v}\cdot\mathbf{v}_t)/x_\xi, --, \mathbf{v}, \mathbf{v}_t) =$$

$$x_\xi\,\mathbf{f}(t, x(t,\xi, \mathbf{v}), \mathbf{w}, \mathbf{w}_x/x_\xi, \mathbf{w}_t - \mathbf{w}_\xi(x_t + x_\mathbf{v}\cdot\mathbf{v}_t)/x_\xi, --, \mathbf{v}, \mathbf{v}_t)$$

where $--$ refers to a second order term that is not allowed.

At the beginning of the user's `AF` procedure it is necessary to convert from the internal coordinate system of `POST`, namely **w** and $\xi$, to the user's physical coordinate system, **u** and $x$. To accomplish this the user must calculate the value of the map $x(\xi, t, \mathbf{v})$ at the $\xi$ input to `AF` and its partials with respect to time, $\xi$, **v**, etc., putting the results in arrays say x, `xt`, `xxi`, etc., respectively. That is, the "x" in the calling sequence to `AF` should be called "xi", and the values of "x(xi)" should be computed by the user inside `AF`. Then

```
Call POSTU(xi,x,xt,xxi,xv,Vt,nx,Nv, Ux,Ut,Nu, Ax,Fx)
```

will do the mapping from internal ( **w** ) to User ( **u** ) coordinates. Example 5 of Appendix 4 illustrates the use of `POSTU`. The arguments to `POSTU` are now described. The input to `POSTU` is xi - The input "x" variable from AF, to be interpreted as $\xi$ values. Real of length nx.

x - A Real array of length nx with x($i$) = $x(\xi_i, t, \mathbf{v}(t))$, for $i = 1, \cdots,$ nx.

xt - A Real array of length nx with xt($i$) = $x_t(\xi_i, t, \mathbf{v}(t))$, for $i = 1, \cdots,$ nx.

xxi - A Real array of length nx with xxi($i$) = $x_\xi(\xi_i, t, \mathbf{v}(t))$, for $i = 1, \cdots,$ nx.

xv - Real xv(nx,Nv), with xv($i$,$j$) = $x_{v_j}(\xi_i, t, \mathbf{v}(t))$, for $i = 1, \cdots,$ nx and $j = 1, \cdots,$ Nv.

Vt - The part of the Vt array from the input AF argument of the same name that is devoted to mapping, see the POSTI description below.

nx - The AF argument of the same name.

Nv - The length of Vt.

Ux - The AF argument of the same name. The input values of this array are the derivatives $\mathbf{w}_\xi$.

Ut - The AF argument of the same name. The input values of this array are the derivatives $\mathbf{w}_t$.

nu - The AF argument of the same name.

The output of POSTU is

Ux - The derivatives $\mathbf{u}_x$.

Ut - The derivatives $\mathbf{u}_t$.

Ax - Real Ax(nx,Nu), set to $\mathbf{0}$.

Fx - Real Fx(nx,Nu), set to $\mathbf{0}$.

With the map of **w** into user coordinates **u** done using POSTU above, the user then continues with AF as if there were no mapping being used at all. That is, the user can then think only in terms of the original variables, **u** and $x$.

However, before leaving AF the user must map the physical coordinates **u** back into the internal system POST is using, **w**. This is done by

```
Call POSTI(xi,x,xt,xxi,xv,xtv,xxiv,xvv,nx,
           Ux,Ut,Nu,
           V,Vt,Nv,iV,Nvm,
           A,Ax,AU,AUx,AUt,AUtx,AV,AVt,
           F,Fx,FU,FUx,FUt,FUtx,FV,FVt)
```

just before leaving AF.

It is not assumed that all the components of **v** in AF are involved in the spatial mapping. POSTI allows for subsets of V to be used in the mapping by giving a pointer ( iVm ) to the block of variables in V that are involved in the mapping and the blocks length ( Nvm ).

Also, note that the user must compute the partial derivatives of **a** and **f** with respect to $x$ when using POSTU and POSTI, see the description of the arrays Ax and Fx below.

The input arguments to POSTI are as in POSTU above, with the addition of

xtv - Real xtv(nx,Nvm), with xtv($i$,$j$) = $x_{t\,v_j}(\xi_i, t, \mathbf{v}(t))$, for $i = 1, \cdots,$ nx and $j = 1, \cdots,$ Nvm.

xxiv - Real xxiv(nx,Nvm), with xxiv($i$,$j$) = $x_{\xi\,v_j}(\xi_i, t, \mathbf{v}(t))$, for $i = 1, \cdots,$ nx and $j = 1, \cdots,$ Nvm.

xvv - Real xvv(nx,Nvm,Nvm), with xvv($i$,$p$,$q$) = $x_{v_p\,v_q}(\xi_i, t, \mathbf{v}(t))$, for $i = 1, \cdots,$ nx and $p, q = 1, \cdots,$ Nvm.

V - The input AF variable of the same name.

Vt - The input AF variable of the same name.

`Nv` - The length of `V`.

`iV` - The index of the first variable in the input `AF` array `V` that is involved in the mapping function. This allows subsets of `V` to be used for mapping.

`Nvm` - The number of mapping parameters starting with `V(iV)`.

`A` - The input `AF` argument of the same name.

`Ax` - Real `Ax(nx,Nu)`, with `Ax`$(i,j) = \dfrac{\partial a_j}{\partial x}(x(i), t, \cdots)$, for $i = 1, \cdots,$ `nx` and $j = 1, \cdots,$ `Nu`.

`AU` - The input `AF` argument of the same name.

`AUx` - The input `AF` argument of the same name.

`AUt` - The input `AF` argument of the same name.

`AUtx` - The input `AF` argument of the same name.

`AV` - The input `AF` argument of the same name.

`AVt` - The input `AF` argument of the same name.

`F` - The input `AF` argument of the same name.

`Fx` - Real `Fx(nx,Nu)`, with `Fx`$(i,j) = \dfrac{\partial f_j}{\partial x}(x(i), t, \cdots)$, for $i = 1, \cdots,$ `nx` and $j = 1, \cdots,$ `Nu`.

`FU` - The input `AF` argument of the same name.

`FUx` - The input `AF` argument of the same name.

`FUt` - The input `AF` argument of the same name.

`FUtx` - The input `AF` argument of the same name.

`FV` - The input `AF` argument of the same name.

`FVt` - The input `AF` argument of the same name.

The output of `POSTI` is

`A` - $\mathbf{a}$ for the $w$ and $\xi$ coordinate system.

`AU` - The partials of $\mathbf{a}$ with respect to $\mathbf{w}$.

`AUx` - The partials of $\mathbf{a}$ with respect to $\mathbf{w}_\xi$.

`AUt` - The partials of $\mathbf{a}$ with respect to $\mathbf{w}_t$.

`AUtx` - The partials of $\mathbf{a}$ with respect to $\mathbf{w}_{t\xi}$.

`AV` - The partials of $\mathbf{a}$ with respect to $\mathbf{v}$.

`AVt` - The partials of $\mathbf{a}$ with respect to $\mathbf{v}_t$.

`F` - $\mathbf{f}$ for the $\mathbf{w}$ and $\xi$ system.

`FU` - The partials of $\mathbf{f}$ with respect to $\mathbf{w}_{t\xi}$.

`FUx` - The partials of $\mathbf{f}$ with respect to $\mathbf{w}_\xi$.

`FUt` - The partials of $\mathbf{f}$ with respect to $\mathbf{w}_{t\xi}$.

`FUtx` - The partials of $\mathbf{f}$ with respect to $\mathbf{w}_{t\xi}$.

`FV` - The partials of $\mathbf{f}$ with respect to $\mathbf{v}$.

`FVt` - The partials of $\mathbf{f}$ with respect to $\mathbf{v}_t$.

The Double Precision versions of `POSTI` and `POSTU` are called `DPOSTI` and `DPOSTU`, respectively, with all Real arguments typed Double Precision.

Example 5 of Appendix 4 shows `POSTU` and `POSTI` at work.

**Mapping Boundary Conditions**

The **bc** mapping paradigm generally follows that for the **pde** mapping software described above. At the beginning of the user's `BC` procedure it is necessary to convert from the internal coordinate system of `POST`, namely **w** and $\xi$, to the user's physical coordinate system, **u** and $x$. To accomplish this the user must calculate the value of the map $x\,(\,\xi,\,t,\,\mathbf{v}\,)$ at the $\xi$ input to `BC` and its partials with respect to time, $\xi$, **v**, etc., putting the results in arrays say x, `xt`, `xxi`, etc., respectively. That is, the "L" in the calling sequence to `BC` should be called "xi", and the values of "x(xi)" should be computed by the user inside `BC`. As a convenience, the "L" in `BC` can be declared as `Real L(2)`, its first component is "L" and its second is "R." Then `POSTV` should be invoked, the **bc**s entered in terms of **u** and $x$, and then `POSTJ` invoked before returning from `BC`.

The **bc** mapping goes as follows. We have

$$\mathbf{w}(\,\xi,\,t\,) \equiv \mathbf{u}(\,x(\,\xi,\mathbf{v}(t),\,t\,),t\,)$$

and thus

$$\mathbf{w}_t \equiv \mathbf{u}_x\,\frac{dx}{dt} + \mathbf{u}_t \quad \text{and} \quad \mathbf{w}_\xi \equiv \mathbf{u}_x\,x_\xi\,.$$

This implies that **bc**s which began in the form

$$\mathbf{b}\,(\,t,\,L,\,\mathbf{u},\,\mathbf{u}_x,\,\mathbf{u}_t,\,\mathbf{u}_{xt},\,\mathbf{v},\,\mathbf{v}_t\,) = 0$$

now have the form

$$\mathbf{b}\,(\,t,\,L\,(\,t,\mathbf{v}\,),\,\mathbf{w},\,\mathbf{w}_x/x_\xi,\,\mathbf{w}_t - \mathbf{w}_\xi\,(\,x_t + x_{\mathbf{v}}\cdot\mathbf{v}_t\,)/x_\xi,\,--,\,\mathbf{v},\,\mathbf{v}_t\,) = 0$$

where $--$ refers to a second order term that is not allowed. A similar statement holds for the **bc**s at $x = R$.

At the start of `BC` the user calls a mapping procedure to do the map from internal `POST` coordinates $\xi$, the `BC` input argument "xi", to `x` coordinates. Then the

```
Call POSTV(xi,x,xt,xxi,xv,Vt,Nv, Ux,Ut,Nu, Bx)
```

will do the mapping from internal ( **w** ) to user ( **u** ) coordinates. The input to `POSTV` is

| | | |
|---|---|---|
| `xi` | - | The input "L" variable from `BC`. Real of length 2. |
| `x` | - | A Real array of length 2 with x($i$) = $x\,(\,\xi_i,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i = 1,\,2$. |
| `xt` | - | A Real array of length 2 with `xt`($i$) = $x_t\,(\,\xi_i,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i = 1,\,2$. |
| `xxi` | - | A Real array of length 2 with `xxi`($i$) = $x_\xi\,(\,\xi_i,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i = 1,\,2$. |
| `xv` | - | Real `xv`(2,`Nv`), with `xv`($i$,$j$) = $x_{v_j}\,(\,\xi_i,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i = 1,\,2$ and $j = 1,\,\cdots\,,$ `Nv`. |
| `Vt` | - | The part of the `Vt` array from the input `BC` argument of the same name that is devoted to mapping, see the `POSTJ` description below. |
| `Nv` | - | The length of `Vt`. |
| `Ux` | - | The `BC` argument of the same name. The input values of this array are the derivatives $\mathbf{w}_\xi$. |
| `Ut` | - | The `BC` argument of the same name. The input values of this array are the derivatives $\mathbf{w}_t$. |
| `Nu` | - | The `BC` argument of the same name. |

The output of `POSTV` is

| | | |
|---|---|---|
| `Ux` | - | The derivatives $\mathbf{u}_x$. |
| `Ut` | - | The derivatives $\mathbf{u}_t$. |
| `Bx` | - | Real `Bx(Nu,2)`, set to **0**. |

With the map of $w$ into user coordinates $u$ done using `POSTV` above, the user continues with `BC` as if there were no mapping being used. That is, the user can then think only in terms of the original variables, **u** and $x$.

However, before leaving `BC` the user must map the physical coordinates **u** back into the internal system `POST` is using, **w**. This is done by

```
Call POSTJ(xi,x,xt,xxi,xv,xtv,xxiv,xvv,
           Ux,Ut,Nu,
           V,Vt,Nv,iV,Nvm,
           B,Bx,BU,BUx,BUt,BUtx,BV,BVt)
```

just before leaving `BC`.

Note that the user must compute the partials of **b** with respect to $L$ and $R$ when using `POSTV` and `POSTJ`, see the description of the array `Bx` below.

The input arguments to `POSTJ` are as in `POSTV` above, with the addition of

- `xtv` - Real `xtv`(2,`Nvm`), with `xtv`($i,j$) = $x_{t\,v_j}\,(\,\xi_i,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\;=\;1,\,2$ and $j\;=\;1,\;\cdots\;,$ `Nvm`.
- `xxiv` - Real `xxiv`(2,`Nvm`), with `xxiv`($i,j$) = $x_{\xi\,v_j}\,(\,\xi_i,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\;=\;1,\,2$ and $j\;=\;1,\;\cdots\;,$ `Nvm`.
- `xvv` - Real `xvv`(2,`Nvm`,`Nvm`), with `xvv`($i,p,q$) $=\,x_{v_p\,v_q}\,(\,\xi_i,\,t,\mathbf{v}\,(\,t\,)\,)$, for $i\;=\;1,\,2$ and $p,\,q\;=\;1,\;\cdots\;,$ `Nvm`.
- `V` - The input `BC` variable of the same name.
- `Vt` - The input `BC` variable of the same name.
- `Nv` - The length of `V`.
- `iV` - The index of the first variable in the input `BC` array `V` that is involved in the mapping function. This allows subsets of `V` to be used for mapping.
- `Nvm` - The number of mapping parameters starting with `V(iVm)`.
- `B` - The input `BC` argument of the same name.
- `Bx` - `Bx`$(i,1) = \dfrac{\partial b_i}{\partial L}$ and `Bx`$(i,2) = \dfrac{\partial b_i}{\partial R}$, for $i\;=\;1,\;\cdots\;,$ `Nu`.
- `BU` - The input `BC` argument of the same name.
- `BUx` - The input `BC` argument of the same name.
- `BUt` - The input `BC` argument of the same name.
- `BUtx` - The input `BC` argument of the same name.
- `BV` - The input `BC` argument of the same name.
- `BVt` - The input `BC` argument of the same name.

The output of `POSTJ` is

- `B` - **b** for the $w$ coordinate system.
- `BU` - The partials of **b** with respect to **w**.
- `BUx` - The partials of **b** with respect to $\mathbf{w}_\xi$.
- `BUt` - The partials of **b** with respect to $\mathbf{w}_t$.
- `BUtx` - The partials of **b** with respect to $\mathbf{w}_{t\,\xi}$.
- `BV` - The partials of **b** with respect to **v**.
- `BVt` - The partials of **b** with respect to $\mathbf{v}_t$.

The Double Precision versions of `POSTJ` and `POSTV` are called `DPOSTJ` and `DPOSTV`, respectively, with all Real arguments typed Double Precision.

Suppose you were solving a **pde** and the **bc**s were

$$L\;+\;u\;-\;u_x\;+\;u_t\;=\;0$$

and

$$R\;+\;u\;+\;u_x\;+\;u_t\;=\;0$$

Then, under the assumption that there are 2 **ode** variables doing the mesh mapping and that `LPLM` is the mapper, the **bc** subroutine would be, in Fortran,

```
      subroutine bc(t,lxi,rxi,u,ux,ut,utx,nu,v,vt,nv,
     *              b,bu,bux,but,butx,bv,bvt)
      integer nu, nv
      real t,lxi(2),rxi,u(nu,2),ux(nu,2),ut(nu,2),utx(nu,2),
     *     v(nv),vt(nv)
      real b(nu,2),bu(nu,nu,2),bux(nu,nu,2),but(nu,nu,2),butx(nu,nu,2),
     *     bv(nu,nv,2),bvt(nu,nv,2)
      real xxi(100), xtv(100), xvv(100), l, r, x(100)
      real xxiv(100), bx(100), xt(100), xv(100)
      call lplm(lxi, 2, v, nv, x, xxi, xxiv, xv, xvv, xt, xtv)
      call postv(lxi, x, xt, xxi, xv, vt, nv, ux, ut, nu, bx)
      b(1, 1) = x(1)+u(1, 1)-ux(1, 1)+ut(1, 1)
      b(1, 2) = x(2)+u(1, 2)+ux(1, 2)+ut(1, 2)
      bx(1) = 1
      bu(1, 1, 1) = 1
      bux(1, 1, 1) = -1
      but(1, 1, 1) = 1
      bx(2) = 1
      bu(1, 1, 2) = 1
      bux(1, 1, 2) = 1
      but(1, 1, 2) = 1
      call postj(lxi,x,xt,xxi,xv,xtv,xxiv,xvv,ux,ut,nu,v,vt,nv,1,nv,
     *           b,bx,bu,bux,but,butx,bv,bvt)
      return
      end
```

**6. Time Discretization.**

A one-step, implicit finite difference method is employed for the time discretization of (2.4)-(2.6). The resulting nonlinear **ode**'s in space are then linearized, as are the **bc**'s and the original **ode**'s. The solution of the resulting system of linear **ode**'s in space, subject to linear **bc**'s, with coupled linear algebraic equations, is accomplished by Galerkin's method, using B-splines, as described in section 7.

An extrapolation scheme (see Appendix 2) is applied to the results of this one-step finite difference method in time. This allows an extrapolation step-size and order monitor [44] to be employed that dynamically changes both the local step-size and order of the time-integration scheme to satisfy the user's error request in a reasonable fashion.

Although this outline of the solution process appears to be inconsistent with that given in section 3, both outlines produce the *same* numerical solution. Spatial discretization followed by discretization of the resulting **ode**'s in time gives the same mathematical formulation as discretization in time followed by spatial discretization. For overview, the former outline is conceptually simpler; however, for the derivation and implementation of the equations the latter is preferable.

The time discretization is parameterized by a number $\theta$ obeying $0 \le \theta \le 1$. For $\theta = 1$ this gives the very stable backward-Euler scheme, $\theta = \frac{1}{2}$ gives the Crank-Nicholson scheme [39], and $\theta = 0$ gives the unstable forward-Euler scheme. The description of a canonical time-step follows. Let $\mathbf{u}^o$, $\mathbf{v}^o$ denote the old, known solution at time $t^o$. Assume that we want to find the solution $\mathbf{u}$, $\mathbf{v}$ at time $t$, and let $\delta = t - t^o$ be the time-step. All time derivatives are then replaced by time differences, as with $\mathbf{u}_t \approx \dfrac{\mathbf{u}-\mathbf{u}^o}{\delta}$, $\mathbf{u}_{xt} \approx \dfrac{\mathbf{u}_x-\mathbf{u}_x^o}{\delta}$ and $\mathbf{v}_t \approx \dfrac{\mathbf{v}-\mathbf{v}^o}{\delta}$. This reduces equations (2.4)-(2.6) to the following system of nonlinear **ode**'s in space, subject to nonlinear **bc**'s, coupled to a nonlinear system of equations.

$$\frac{\partial}{\partial x}\mathbf{a}\Big(\ \theta t+(1-\theta)t^o,\ x,\ \theta\mathbf{u}+(1-\theta)\mathbf{u}^o,\ \theta\mathbf{u}_x+(1-\theta)\mathbf{u}_x^o,\ \frac{\mathbf{u}-\mathbf{u}^o}{\delta},\ \frac{\mathbf{u}_x-\mathbf{u}_x^o}{\delta},\ \theta\mathbf{v}+(1-\theta)\mathbf{v}^o,\ \frac{\mathbf{v}-\mathbf{v}^o}{\delta}\ \Big) =$$

$$\frac{\partial}{\partial x}\mathbf{a}\left( \theta t+(1-\theta)t^o,\ x,\ \theta\mathbf{u}+(1-\theta)\mathbf{u}^o,\ \theta\mathbf{u}_x+(1-\theta)\mathbf{u}_x^o,\ \frac{\mathbf{u}-\mathbf{u}^o}{\delta},\ \frac{\mathbf{u}_x-\mathbf{u}_x^o}{\delta},\ \theta\mathbf{v}+(1-\theta)\mathbf{v}^o,\ \frac{\mathbf{v}-\mathbf{v}^o}{\delta} \right) = \tag{6.1a}$$

$$\mathbf{f}\left( \theta t+(1-\theta)t^o,\ x,\ \theta\mathbf{u}+(1-\theta)\mathbf{u}^o,\ \theta\mathbf{u}_x+(1-\theta)\mathbf{u}_x^o,\ \frac{\mathbf{u}-\mathbf{u}^o}{\delta},\ \frac{\mathbf{u}_x-\mathbf{u}_x^o}{\delta},\ \theta\mathbf{v}+(1-\theta)\mathbf{v}^o,\ \frac{\mathbf{v}-\mathbf{v}^o}{\delta} \right)$$

$$\mathbf{b}_L(\ \theta t+(1-\theta)t^o,\ \theta\mathbf{u}+(1-\theta)\mathbf{u}^o(t^o,L),\ \theta\mathbf{u}_x+(1-\theta)\mathbf{u}_x^o(t^o,L),$$

$$\mathbf{b}_L(\ \theta t+(1-\theta)t^o,\ \theta\mathbf{u}+(1-\theta)\mathbf{u}^o(t^o,L),\ \theta\mathbf{u}_x+(1-\theta)\mathbf{u}_x^o(t^o,L), \tag{6.1b}$$

$$\frac{\mathbf{u}(t,L)-\mathbf{u}^o(t^o,L)}{\delta},\ \frac{\mathbf{u}_x(t,L)-\mathbf{u}_x^o(t^o,L)}{\delta},\ \theta\mathbf{v}+(1-\theta)\mathbf{v}^o,\ \frac{\mathbf{v}-\mathbf{v}^o}{\delta}\ ) = 0$$

$$\mathbf{b}_R(\ \theta t+(1-\theta)t^o,\ \theta\mathbf{u}+(1-\theta)\mathbf{u}^o(t^o,R),\ \theta\mathbf{u}_x+(1-\theta)\mathbf{u}_x^o(t^o,R),$$

$$\mathbf{b}_R(\ \theta t+(1-\theta)t^o,\ \theta\mathbf{u}+(1-\theta)\mathbf{u}^o(t^o,R),\ \theta\mathbf{u}_x+(1-\theta)\mathbf{u}_x^o(t^o,R), \tag{6.1c}$$

$$\frac{\mathbf{u}(t,R)-\mathbf{u}^o(t^o,R)}{\delta},\ \frac{\mathbf{u}_x(t,R)-\mathbf{u}_x^o(t^o,R)}{\delta},\ \theta\mathbf{v}+(1-\theta)\mathbf{v}^o,\ \frac{\mathbf{v}-\mathbf{v}^o}{\delta}\ ) = 0$$

$$\mathbf{d}(\ \theta t+(1-\theta)t^o,\ \theta\mathbf{U}(t)+(1-\theta)\mathbf{U}^o(t^o),\ \frac{\mathbf{U}(\ t\ )\ -\ \mathbf{U}^o(t^o)}{\delta},\ \theta\mathbf{v}+(1-\theta)\mathbf{v}^o,\ \frac{\mathbf{v}-\mathbf{v}^o}{\delta}\ ) = 0\ , \tag{6.1d}$$

where $\mathbf{U}(t)$ is the vector of B-spline coefficients for the solution $\mathbf{u}$.

An iterative, quasi-Newton method is used to solve (6.1). Let $\mathbf{u}^J$, $\mathbf{v}^J$ denote the point of linearization, with $\mathbf{u}^J = \mathbf{u}(t^J, x)$, $\mathbf{v}^J = \mathbf{v}(t^J)$, etc. Finally, let $\mathbf{u}^n$, $\mathbf{v}^n$ be the current iterative best estimates of the solution $\mathbf{u}$, $\mathbf{v}$. The linear, quasi-Newton equations for the corrections, $\mathbf{w}(x)$ and $\mathbf{z}$, to $\mathbf{u}^n$ and $\mathbf{v}^n$, are then

$$a_{ix}^n + \sum_{j=1}^{n_u}\left[\theta a^J_{iu_j} w_j + \theta a^J_{iu_{jx}} w_{jx} + a^J_{iu_{jt}}\frac{w_j}{\delta} + a^J_{iu_{jxt}}\frac{w_{jx}}{\delta}\right]_x + \sum_{j=1}^{n_v}\left[\theta a^J_{iv_j} z_j + a^J_{iv_{jt}}\frac{z_j}{\delta}\right]_x =$$

$$a_{ix}^n + \sum_{j=1}^{n_u}\left[\theta a^J_{iu_j} w_j + \theta a^J_{iu_{jx}} w_{jx} + a^J_{iu_{jt}}\frac{w_j}{\delta} + a^J_{iu_{jxt}}\frac{w_{jx}}{\delta}\right]_x + \sum_{j=1}^{n_v}\left[\theta a^J_{iv_j} z_j + a^J_{iv_{jt}}\frac{z_j}{\delta}\right]_x = \tag{6.2a}$$

$$f_i^n + \sum_{j=1}^{n_u}\left[\theta f^J_{iu_j} w_j + \theta f^J_{iu_{jx}} w_{jx} + f^J_{iu_{jt}}\frac{w_j}{\delta} + f^J_{iu_{jxt}}\frac{w_{jx}}{\delta}\right] + \sum_{j=1}^{n_v}\left[\theta f^J_{iv_j} z_j + f^J_{iv_{jt}}\frac{z_j}{\delta}\right]$$

$$b_{Li}^n + \sum_{j=1}^{n_u}\left[\theta b^J_{Liu_j} w_j + \theta b^J_{Liu_{jx}} w_{jx} + b^J_{Liu_{jt}}\frac{w_j}{\delta} + b^J_{Liu_{jxt}}\frac{w_{jx}}{\delta}\right] + \sum_{j=1}^{n_v}\left[\theta b^J_{Liv_j} z_j + b^J_{Liv_{jt}}\frac{z_j}{\delta}\right] = 0 \tag{6.2b}$$

$$b_{Ri}^n + \sum_{j=1}^{n_u}\left[\theta b^J_{Riu_j} w_j + \theta b^J_{Riu_{jx}} w_{jx} + b^J_{Riu_{jt}}\frac{w_j}{\delta} + b^J_{Riu_{jxt}}\frac{w_{jx}}{\delta}\right] + \sum_{j=1}^{n_v}\left[\theta b^J_{Riv_j} z_j + b^J_{Riv_{jt}}\frac{z_j}{\delta}\right] = 0 \tag{6.2c}$$

$$d_i^n + \sum_{j=1}^{n_u}\left[\sum_{l=1}^{n_x-k}\left(\theta d^J_{iW_{jl}} \mathbf{w}_{jl} + d^J_{iW_{jlt}}\frac{\mathbf{w}_{jl}}{\delta}\right)\right] + \sum_{j=1}^{n_v}\left[\theta d^J_{iv_j} z_j + d^J_{iv_{jt}}\frac{z_j}{\delta}\right] = 0\ . \tag{6.2d}$$

The solution of (6.2) gives the corrections $\mathbf{w}$ and $\mathbf{z}$ to $\mathbf{u}^n$ and $\mathbf{v}^n$ which produce the next quasi-Newton iteration $\mathbf{u}^{n+1}$ and $\mathbf{v}^{n+1}$. Thus, (6.2) constitutes the iterative scheme for solving (6.1). Equations (6.2) have the form

$$\sum_{j=1}^{n_u}\left[a_{ij}^{(1)} w_{jx} + a_{ij}^{(2)} w_j\right]_x = f_{ix}^{(1)} + f_i^{(2)} + \sum_{j=1}^{n_u}\left[a_{ij}^{(3)} w_{jx} + a_{ij}^{(4)} w_j\right] + \sum_{j=1}^{n_v}\left[g_{ijx}^{(1)} + g_{ij}^{(2)}\right] z_j \tag{6.3a}$$

$$\sum_{j=1}^{n_u}\left[\alpha_{ij}^{(L)} w_j(L) + \beta_{ij}^{(L)} w_{jx}(L)\right] = \gamma_i^{(L)} + \sum_{j=1}^{n_v}\sigma_{ij}^{(L)} z_j \tag{6.3b}$$

$$\sum_{j=1}^{n_u}\left[\alpha_{ij}^{(R)} w_j(R) + \beta_{ij}^{(R)} w_{jx}(R)\right] = \gamma_i^{(R)} + \sum_{j=1}^{n_v}\sigma_{ij}^{(R)} z_j \tag{6.3c}$$

$$\sum_{j=1}^{n_u}\sum_{l=1}^{n_x-k} d_{ijl}^{(1)} \mathbf{w}_{jl} = d_i^{(2)} + \sum_{j=1}^{n_v} d_{ij}^{(3)} z_j \ . \tag{6.3d}$$

These equations may be simplified by setting

$$w_j = w_{0j} + \sum_{p=1}^{n_v} z_p w_{pj} \tag{6.4}$$

where $\mathbf{w}_0$ solves

$$(\mathbf{a}^{(1)} \mathbf{w}_{0x} + \mathbf{a}^{(2)} \mathbf{w}_0)_x = \mathbf{f}_x^{(1)} + \mathbf{f}^{(2)} + \mathbf{a}^{(3)} \mathbf{w}_{0x} + \mathbf{a}^{(4)} \mathbf{w}_0 \tag{6.5}$$

subject to **bc**'s

$$\begin{aligned}
\boldsymbol{\alpha}^{(L)} \mathbf{w}_0(L) + \boldsymbol{\beta}^{(L)} \mathbf{w}_{0x}(L) &= \boldsymbol{\gamma}^{(L)} \\
\boldsymbol{\alpha}^{(L)} \mathbf{w}_0(L) + \boldsymbol{\beta}^{(L)} \mathbf{w}_{0x}(L) &= \boldsymbol{\gamma}^{(L)} \\
\boldsymbol{\alpha}^{(R)} \mathbf{w}_0(R) + \boldsymbol{\beta}^{(R)} \mathbf{w}_{0x}(R) &= \boldsymbol{\gamma}^{(R)}
\end{aligned} \tag{6.6}$$

and $\mathbf{w}_p$ solves

$$(\mathbf{a}^{(1)} \mathbf{w}_{px} + \mathbf{a}^{(2)} \mathbf{w}_p)_x = \mathbf{g}_{px}^{(1)} + \mathbf{g}_p^{(2)} + \mathbf{a}^{(3)} \mathbf{w}_{px} + \mathbf{a}^{(4)} \mathbf{w}_p \tag{6.7}$$

subject to **bc**'s

$$\begin{aligned}
\boldsymbol{\alpha}^{(L)} \mathbf{w}_p(L) + \boldsymbol{\beta}^{(L)} \mathbf{w}_{px}(L) &= \boldsymbol{\sigma}_p^{(L)} \\
\boldsymbol{\alpha}^{(L)} \mathbf{w}_p(L) + \boldsymbol{\beta}^{(L)} \mathbf{w}_{px}(L) &= \boldsymbol{\sigma}_p^{(L)} \\
\boldsymbol{\alpha}^{(R)} \mathbf{w}_p(R) + \boldsymbol{\beta}^{(R)} \mathbf{w}_{px}(R) &= \boldsymbol{\sigma}_p^{(R)} \ .
\end{aligned} \tag{6.8}$$

Linear systems of **ode**'s, such as (6.5) and (6.7), subject to linear **bc**'s, such as (6.6) and (6.8), may be solved using Galerkin's method, as outlined in section 7.

When the $w_{pj}$ have been obtained from (6.5)-(6.6) and (6.7)-(6.8), the **z** may be computed from the linearized **ode** (6.3d), which is the system of linear algebraic equations

$$\sum_{p=1}^{n_v}\left[d_{ip}^{(3)} - \sum_{j=1}^{n_u}\sum_{l=1}^{n_x-k} d_{ijl}^{(1)} \mathbf{w}_{pjl}\right] z_p = -d_i^{(2)} + \sum_{j=1}^{n_u}\sum_{l=1}^{n_x-k} d_{ijl}^{(1)} \mathbf{w}_{0jl} \tag{6.9}$$

for **z**. When (6.9) has been solved for **z**, **w** may be obtained from (6.4).

The computation of **w** and **z** from (6.4)-(6.9) results in a single iteration in the solution of the nonlinear **ode**'s (6.1), which itself constitutes a single time-step.

## 7. Spatial Discretization.

This section discusses the implementation of Galerkin's method [11,12,13,46,50] using B-splines, for solving systems of linear **ode**'s, subject to linear **bc**'s. First the linear **bc**'s are put into a canonical form suitable for use by Galerkin's method. Next, each **bc** is paired with one, and only one, of the **ode**'s, so that the Galerkin equations may be formulated. As a by-product of these first two phases, a number of error conditions are detected. For example, too many or too few **bc**'s may be detected. So may things like a **bc** on $u'(L)$ when the **ode** for $u$ is only of first order. Thus, the extra processing that Galerkin's method requires also gives a great deal of useful debugging (error) information. Finally, the Galerkin equations are formulated.

The **ode**'s (6.5) and (6.7) have the form

$$(\mathbf{a}^{(1)}\mathbf{u}_x + \mathbf{a}^{(2)}\mathbf{u})_x = \mathbf{f}_x^{(1)} + \mathbf{f}^{(2)} + \mathbf{a}^{(3)}\mathbf{u}_x + \mathbf{a}^{(4)}\mathbf{u} \tag{7.1}$$

subject to **bc**'s

$$\begin{aligned} \boldsymbol{\alpha}^{(L)}\mathbf{u}(L) + \boldsymbol{\beta}^{(L)}\mathbf{u}_x(L) &= \boldsymbol{\gamma}^{(L)} \\ \boldsymbol{\alpha}^{(L)}\mathbf{u}(L) + \boldsymbol{\beta}^{(L)}\mathbf{u}_x(L) &= \boldsymbol{\gamma}^{(L)} \\ \boldsymbol{\alpha}^{(R)}\mathbf{u}(R) + \boldsymbol{\beta}^{(R)}\mathbf{u}_x(R) &= \boldsymbol{\gamma}^{(R)} . \end{aligned} \tag{7.2}$$

**Reduction of bc's to Canonical Form.**

This section shows how to reduce the arbitrary linear **bc**'s (7.2) to a canonical form suitable for use in Galerkin's method. The linear **bc**'s have the form, at $x = L$ and $x = R$,

$$\boldsymbol{\alpha}\mathbf{u} + \boldsymbol{\beta}\mathbf{u}_x = \boldsymbol{\gamma} . \tag{7.3}$$

For a scalar equation ($n_u = 1$) these **bc**'s have the form

$$\alpha u + \beta u_x = \gamma$$

which, depending on whether $\beta = 0$ or not, may be written as either

$$\begin{aligned} &u = \gamma_D \qquad ( \beta = 0 ) \\ &\qquad \text{or} \\ &u_x = Au + \gamma_M \quad ( \beta \neq 0 ) , \end{aligned} \tag{7.4}$$

where $\gamma_D = \dfrac{\gamma}{\alpha}$, $A = -\dfrac{\alpha}{\beta}$ and $\gamma_M = \dfrac{\gamma}{\beta}$.

A generalization of this canonical form for $n_u = 1$ to systems with $n_u > 1$ is now presented. This reduction assumes that each **bc** is for the purpose of determining one of either $u$ or $u_x$, but not both. For example, the "**bc**'s"

$$u(0) + u_x(0) = 0$$

$$u(0) - u_x(0) = 0,$$

which is a sloppy way to say $u(0) = 0 = u_x(0)$, cannot be dealt with by the following scheme, even though it is of the form (7.3).

Let $\boldsymbol{\alpha}_D$ and $\boldsymbol{\gamma}_D$ be the Dirichlet **bc**'s, that is, those rows of $\boldsymbol{\alpha}$ and $\boldsymbol{\gamma}$ which have all $\beta_{ij} = 0$. Let $\boldsymbol{\alpha}_M$, $\boldsymbol{\beta}_M$ and $\boldsymbol{\gamma}_M$ be the complementary , mixed, equations, that is, those rows of $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ which have some $\beta_{ij} \neq 0$.

Then $\boldsymbol{\alpha}_D$ will be an $n_D \times n_u$ matrix, where $1 \le n_D \le n_u$. The Dirichlet **bc**'s have the form

$$\boldsymbol{\alpha}_D \, \mathbf{u} = \boldsymbol{\gamma}_D .$$

Let $n_{u_D}$ be the number of $u_j$ which have $\alpha_{Dij} \neq 0$ for some $i$. Then we must have $n_{u_D} \ge n_D$ , for otherwise the Dirichlet **bc**'s will be overdetermined.

We may obtain a QR factorization [55] of the matrix $\boldsymbol{\alpha}_D$ the form

$$\mathbf{Q}\, \boldsymbol{\alpha}_D\, \mathbf{P} = ( \, \mathbf{R} \mid \mathbf{C} \, )$$

where $\mathbf{Q}$ is an $n_D \times n_D$ orthogonal matrix, $\mathbf{P}$ is a $n_u \times n_u$ permutation matrix, $\mathbf{R}$ is an $n_D \times n_D$ upper triangular matrix, and $\mathbf{C}$ is an $n_D \times (n_u - n_D)$ matrix. Let $\mathbf{u}_D$ be the first $n_D$ elements of $\mathbf{P}^t\, \mathbf{u}$, and $\mathbf{u}_D^C$ be the rest of $\mathbf{P}^t\, \mathbf{u}$. We then have

$$( \, \mathbf{R} \mid \mathbf{C} \, ) \mathbf{P}^t\, \mathbf{u} = \mathbf{Q}\, \boldsymbol{\gamma}_D$$

and thus

$$\mathbf{u}_D = \mathbf{R}^{-1}\, \mathbf{Q}\, \boldsymbol{\gamma}_D - \mathbf{R}^{-1}\, \mathbf{C}\, \mathbf{u}_D^C . \tag{7.5}$$

This is the canonical form that will be assumed for the Dirichlet **bc**'s.

The mixed **bc**'s have the form

$$\boldsymbol{\alpha}_M \mathbf{u} + \boldsymbol{\beta}_M \mathbf{u}_x = \boldsymbol{\gamma}_M$$

where $\boldsymbol{\alpha}_M$ and $\boldsymbol{\beta}_M$ are $n_M \times n_u$ matrices, with $1 \le n_M \le n_u$. Let $n_{u_M}$ be the number of $u_j$ that have $\beta_{Mij} \ne 0$ for some $i$. Then we must have $n_{u_M} \ge n_M$, for otherwise the mixed **bc**'s would be overdetermined. Precisely as in the Dirichlet case, we may obtain a QR factorization of the form

$$\mathbf{Q}\,\boldsymbol{\beta}_M\,\mathbf{P} = (\ \mathbf{R} \mid \mathbf{C}\ ).$$

Let $\mathbf{u}_M$ be the first $n_M$ elements of $\mathbf{P}^t\ \mathbf{u}$ and let $\mathbf{u}_M^C$ be the rest of $\mathbf{P}^t\ \mathbf{u}$. Then

$$\mathbf{u}_{Mx} = \mathbf{R}^{-1}\,\mathbf{Q}\,\boldsymbol{\gamma}_M - \mathbf{R}^{-1}\,\mathbf{Q}\,\boldsymbol{\alpha}_M\,\mathbf{u} - \mathbf{R}^{-1}\,\mathbf{C}\,\mathbf{u}_{Mx}^C\ . \tag{7.6}$$

This is the canonical form that will be assumed for the mixed **bc**'s.

The above **bc** forms, (7.5) and (7.6), have the form

$$\begin{aligned} \mathbf{u}_D &= \boldsymbol{\gamma}_D + \mathbf{C}_D\ \mathbf{u} \\ \mathbf{u}_D &= \boldsymbol{\gamma}_D + \mathbf{C}_D\ \mathbf{u} \\ \mathbf{u}_{Mx} &= \mathbf{A}\mathbf{u} + \boldsymbol{\gamma}_M + \mathbf{C}_M\ \mathbf{u}_x \end{aligned} \tag{7.7}$$

where the appropriate columns of $\mathbf{C}_D$, $\mathbf{C}_M$ and $\mathbf{A}$ are zero.

The above form (7.7) generalizes the standard form for linear **bc**'s when $n_u = 1$ to the case where $n_u > 1$. It may not represent the most general generalization, but it is sufficient for all problems of which the author is aware.

**bc Placement.**

Galerkin's method reduces (7.1)-(7.2) to a system of linear algebraic equations in the B-spline coefficients for the solution **u**, by making the error in (7.1) orthogonal to each of the B-spline basis functions [45]. Certain of these orthogonality relations are then replaced, or modified, by **bc** equations. A **bc** on $u_i$ at $x = L$ affects the first orthogonality relation for the $i^{th}$ **ode**. Similarly, a **bc** on $u_i$ at $x = R$ affects the last, $N-k^{th}$, relation for the $i^{th}$ **ode**, where $N$ is the length of the B-spline mesh being used. Galerkin's method requires that certain **bc**'s be associated with (applied to) certain kinds of differential equations. The restrictions imposed on the relation between the structure of the **ode**'s and **bc**'s are listed below.

7.1) Any Dirichlet **bc** must be applied to an equation with some $u_{ix}$ present. This rule simply requires that the **ode** to which a Dirichlet **bc** is applied be of order at least one. For example, if the **ode** is

$$\left.\begin{aligned} u_{1x} &= 0 \\ u_2 &= 0 \end{aligned}\right\} \qquad \text{for} \quad x\ \varepsilon\ (0,1)\ ,$$

with the **bc** $u_1(0) = 1$, and in the Galerkin equations one of the equations for $u_2$ is replaced by that **bc**, the resulting Galerkin matrix will be singular (since there will be only $N-k-1$ equations for the $N-k$ unknown B-spline coefficients for $u_2$).

7.2) Any mixed **bc** on $u_{jx}$ must be applied to an **ode** with $u_{jxx}$ present. This is *required* by the Galerkin procedure, and appears to be common sense as well.

7.3) It makes no difference where non-existent (inactive) **bc**'s are "applied" (associated), eg. $u_2$ above.

The above rules may be turned into a nice mathematical problem by letting $O_{ij}^{(L)}$ be the order of $u_j$ in **ode** $i$ at $x = L$, and similarly for $O_{ij}^{(R)}$ at $x = R$. To be precise, $O_{ij}^{(L)}$ is the value of

$$\lim_{x \downarrow L} \left\{ \text{The order of } u_j \text{ in } \textbf{ode}\ i\ \text{ at } x \right\},$$

where the order of $u_j$ in **ode** $i$ means the order of the highest order derivative of $u_j$ appearing in **ode** $i$ at $x$, so that **ode**'s like

$$xu_x = \sin(x) \qquad \text{for} \quad x \,\varepsilon\, (0,1)$$

have the proper order, namely 1, assigned at $x = 0$, instead of zero order. If $u_j$ doesn't appear in **ode** $i$, we simply set $O_{ij}^{(L)} = -1$. Let $\mathbf{T}^{(L)}$ be an $n_u \times 2$ matrix with elements

$$\mathbf{T}_{i,1}^{(L)} = \begin{cases} 0 & \text{if there is a Dirichlet } \mathbf{bc} \text{ for } u_i \text{ at } x = L. \\ -2 & \text{if there is no Dirichlet } \mathbf{bc} \text{ for } u_i \text{ at } x = L. \end{cases}$$

$$\mathbf{T}_{i,2}^{(L)} = \begin{cases} 1 & \text{if there is a mixed } \mathbf{bc} \text{ for } u_j \text{ at } x = L. \\ -2 & \text{if there is no } \mathbf{bc} \text{ for } u_j \text{ at } x = L. \end{cases}$$

with $\mathbf{T}^{(R)}$ defined similarly for $x = R$. The previous restrictions 7.1)-7.3) then become the problem of finding permutation arrays $\mathbf{E}_D^{(L)}$, $\mathbf{E}_M^{(L)}$, $\mathbf{E}_D^{(R)}$ and $\mathbf{E}_M^{(R)}$, each of length $n_u$, so that

7.4) $\underset{j}{Max}\ O_{E_D^{(L)}(i),j}^{(L)} > T_{i,1}^{(L)}$. Similarly for $x = R$.

7.5) $O_{E_M^{(L)}(i),j}^{(L)} > T_{i,2}^{(L)}$. Similarly for $x = R$.

7.6) $\mathbf{E}_D^{(L)} \cap \mathbf{E}_M^{(L)} = \Phi = \mathbf{E}_D^{(R)} \cap \mathbf{E}_M^{(R)}$ .

Restriction 7.6 makes sure that different **bc**'s aren't applied to the same **ode**, at the same point.

Another condition is also added

7.7) For each $i = 1, \cdots, n_u$, the number of times $i$ appears in the arrays $\mathbf{E}_D^{(L)}$, $\mathbf{E}_M^{(L)}$, $\mathbf{E}_D^{(R)}$ and $\mathbf{E}_M^{(R)}$, with one of $\mathbf{T}_{i,1}^{(L)}$, $\mathbf{T}_{i,2}^{(L)}$, $\mathbf{T}_{i,1}^{(R)}$, or $\mathbf{T}_{i,2}^{(R)}$ not equal to $-2$, should be less than or equal to $\underset{l}{Max}\ (O_{il}^{(L)}, O_{il}^{(R)})$.

Restriction 7.7 makes sure that the number of active **bc**'s applied to each **ode** equals the order of that **ode**, the number of degrees of freedom introduced by that **ode**.

The permutation arrays $\mathbf{E}_D^{(L)}$, $\mathbf{E}_M^{(L)}$, $\mathbf{E}_D^{(R)}$, and $\mathbf{E}_M^{(R)}$ can easily be found using a tree-back-tracking algorithm [28]. Once found, these arrays tell us which **ode** has which **bc**'s associated with it, and vice-versa.

An additional restriction is that

$$\sum_j \underset{i}{Max}\ (O_{ij}^{(L)}, O_{ij}^{(R)}) = \sum_j (\text{ The order of } u_j\ ) =$$

$$\sum_i \underset{j}{Max}\ (O_{ij}^{(L)}, O_{ij}^{(R)}) = \sum_i (\text{ The order of } \mathbf{ode}\ i\ ) =$$

$$\sum_{L,R} \text{ The number of active } \mathbf{bc}'\text{s},$$

which simply says that the number of degrees of freedom in the **ode** system, whether counted by **ode** or by **ode** variable, must be the same as the number of active **bc**'s at $x = L$ and $x = R$.

These restrictions mean that there should be no transformation of the **ode** system that would lower the order of any $u_j$ in the **ode** without introducing new **ode** variables. For example, the **ode** system

$$u_{1xx} = 0$$

$$u_{1xx} + u_{1x} = u_2$$

has $\sum$ order $u_j = 2 \neq 4 = \sum$ order $\mathbf{ode}_i$, which violates the above restriction. The **ode** may easily be altered, however, to read,

$$u_{1xx} = 0$$

$$u_{1x} = u_2$$

which has $\sum$ order $u_j = 2 \neq 3 = \sum$ order $\mathbf{ode}_i$, which is better. Finally, it can be altered to read

$$u_{2x} = 0$$

$$u_{1x} = u_2$$

which has $\sum$ order $u_j = 2 = \sum$ order $\mathbf{ode}_i$.

**Galerkin's Method, using B-splines, for Linear Systems of ode**'s.

This section shows how Galerkin's method, using B-splines, may be used to solve linear systems of **ode**'s, of the form (6.5) and (6.7), subject to linear **bc**'s of the form (7.3).

Let the B-spline mesh $\pi$ be given, with $n$ elements, as described in Appendix 1. We wish to solve linear **ode** systems of the form

$$(\mathbf{a}^{(1)}\mathbf{u}_x + \mathbf{a}^{(2)}\mathbf{u})_x = (\mathbf{a}^{(3)}\mathbf{u}_x + \mathbf{a}^{(4)}\mathbf{u}) + \mathbf{f}_x^{(1)} + \mathbf{f}^{(2)} \tag{7.8}$$

subject to linear **bc**'s of the form (7.3) at $x = L$ and $R$. Let $(\ f,\ g\ ) \equiv \int_L^R f(x)g(x)\ dx$ be the standard $L_2$ inner-product. The Galerkin equations for (7.8) are, after the usual integration by parts [45],

$$\begin{aligned} &-\sum_{j=1}^{n_u}\left[(\ a_{ij}^{(1)}u_{jx} + a_{ij}^{(2)}u_j,\ B_{px}\ ) + (\ a_{ij}^{(3)}u_{jx} + a_{ij}^{(4)}u_j,\ B_p\ )\right] + \\ &-\sum_{j=1}^{n_u}\left[(\ a_{ij}^{(1)}u_{jx} + a_{ij}^{(2)}u_j,\ B_{px}\ ) + (\ a_{ij}^{(3)}u_{jx} + a_{ij}^{(4)}u_j,\ B_p\ )\right] + \\ &(\ a_{ij}^{(1)}u_{jx} + a_{ij}^{(2)}u_j\ )B_p\ \Big|_L^R = -(\ f_i^{(1)},B_{px}\ ) + f_i^{(1)}B_p\ \Big|_L^R + (\ f_i^{(2)},\ B_p\ )\ . \end{aligned} \tag{7.9}$$

These functional equations are now transformed into linear algebraic equations. Let

$$u_j = \sum_{q=1}^{n-k} y_{j+(q-1)n_u}B_q(x) \qquad j=1,\cdots,n_u \tag{7.10}$$

where the $B_q$ are the B-spline basis functions (see Appendix 1). This ordering of the B-spline coefficients $\mathbf{y}$ results in an *interleaved* structure, *not* a block structure. This ordering is chosen to give a system of linear algebraic equations that is banded, rather than block banded.

Aside from boundary terms (those terms involving only $x = L$ or $R$) the Galerkin matrix $\mathbf{G}$ is given by the equations:

$$\begin{aligned} &\sum_{j=1}^{n_u}\sum_{q=1}^{n-k}\left[(\ a_{ij}^{(1)}B_{qx} + a_{ij}^{(2)}B_q,\ B_{px}\ ) + (\ a_{ij}^{(3)}B_{qx} + a_{ij}^{(4)}B_q,\ B_p\ )\right] y_{j+(q-1)n_u} \equiv \\ &\sum_{j=1}^{n_u}\sum_{q=1}^{n-k}\left[(\ a_{ij}^{(1)}B_{qx} + a_{ij}^{(2)}B_q,\ B_{px}\ ) + (\ a_{ij}^{(3)}B_{qx} + a_{ij}^{(4)}B_q,\ B_p\ )\right] y_{j+(q-1)n_u} \equiv \\ &\sum_{j=1}^{n_u}\sum_{q=1}^{n-k} G_{i+(p-1)n_u,j+(q-1)n_u}\ y_{j+(q-1)n_u}\ ,\quad i=1,\cdots,n_u,\ p=1,\cdots,n-k. \end{aligned} \tag{7.11}$$

Since $|p-q| < k$ and $|i-j| < n_u$, we see that the half-bandwidth of $\mathbf{G}$ is $n_u - 1 + (k-1)n_u + 1 = kn_u$ and the bandwidth of $\mathbf{G}$ is

$$2kn_u - 1\ . \tag{7.12}$$

This banded storage scheme for $\mathbf{G}$ is quite efficient. This is best seen by noting that the total number of non-zero elements in a row of $\mathbf{G}$ is $(2k-1)n_u$. Thus, the relative overhead, over just storing the non-zero elements of $\mathbf{G}$, is $\dfrac{n_u - 1}{(2k-1)n_u}$ which is 0 for $n_u = 1$ and less than $\dfrac{1}{2k-1}$ for all $n_u$.

The right-hand side of the Galerkin equations is $\mathbf{b}$, which, aside from boundary terms, is given by

$$b_{i+(p-1)n_u} = -(\ f_i^{(2)},\ B_{px}\ ) + (\ f_i^{(1)},\ B_p\ ),\quad i=1,\cdots,n_u,\ p=1,\cdots,n-k. \tag{7.13}$$

We now need the boundary terms in (7.9). These come in two flavors, mixed and unmixed. Let $M_L(j)$, $j=1,\cdots,N_M^L$, be the indices of $\mathbf{u}_M^{(L)}$, similarly for $\mathbf{M}_R$. Also, let $\mathbf{C}(\mathbf{M}_L)$ be the complement of $\mathbf{M}_L$, that is, those $i=1,\cdots,n_u$ which are not in the array $\mathbf{M}_L$, similarly for $\mathbf{C}(\mathbf{M}_R)$. Finally, let $r(i,j) \equiv i+(j-1)n_u$. The boundary terms $\delta\mathbf{G}$ for the Galerkin matrix $\mathbf{G}$ are obtained from the equations

$$\delta_{p,1}\left[\sum_{j\varepsilon\mathbf{C}(\mathbf{M}_L)} (\ a_{ij}^{(1)}(L)\ (\ y_jB_{1x}(L) + y_{j+n_u}B_{2x}(L)\ ) + a_{ij}^{(2)}(L)y_j\ ) + \right.$$

$$\sum_{j=1}^{N_M^L} ( a^{(1)}_{iM_L(j)}(L) \; ( \sum_{l=1}^{n_u} ( A^{(L)}_{jl} y_l + C^{(L)}_{Mjl} \; ( y_l B_{1x}(L) + y_{l+n_u} B_{2x}(L) \; ) \; ) \; ) +$$

$$\left. a^{(2)}_{iM_L(j)}(L) y_{M_L(j)} \; ) \right] -$$

$$\delta_{p,n-k} \left[ \sum_{j \varepsilon \mathbf{C}(\mathbf{M}_R)} ( a^{(1)}_{ij}(R) \; ( y_{r(j,p-1)} B_{p-1x}(R) + y_{r(j,p)} B_{px}(R) \; ) + a^{(2)}_{ij}(R) y_{r(j,p)} \; ) + \right.$$

$$\sum_{j=1}^{N_M^R} ( a^{(1)}_{iM_R(j)}(R) \; ( \sum_{l=1}^{n_u} ( A^{(R)}_{il} y_{r(l,p)} + C^{(R)}_{Mjl} \; ( y_{r(l,p-1)} B_{p-1x}(R) + y_{r(l,p)} B_{px}(R) \; ) \; ) \; ) +$$

$$\left. a^{(2)}_{iM_R(j)}(R) y_{r(M_R(j),p)} \right] \equiv$$

$$\left. a^{(2)}_{iM_R(j)}(R) y_{r(M_R(j),p)} \right] \equiv \tag{7.14}$$

$$\delta_{p,1} \left[ \sum_{j \varepsilon \mathbf{C}(\mathbf{M}_L)} ( \delta G_{ij} y_j + \delta G_{i,j+n_u} y_{j+n_u} \; ) + \sum_{j=1}^{N_M^L} ( \sum_{l=1}^{n_u} ( \delta G_{il} y_l + \delta G_{i,l+n_u} y_{l+n_u} \; ) + \delta G_{i,M_L(j)} y_{M_L(j)} \; ) \right] -$$

$$\delta_{p,n-k} \left[ \sum_{j \varepsilon \mathbf{C}(\mathbf{M}_R)} ( \delta G_{r(i,p),r(j,p-1)} y_{r(j,p-1)} + \delta G_{r(i,p),r(j,p)} y_{r(j,p)} \; ) + \right.$$

$$\sum_{j=1}^{N_M^R} ( \sum_{l=1}^{n_u} ( \delta G_{r(i,p),r(l,p)} y_{r(l,p)} + \delta G_{r(i,p),r(l,p-1)} y_{r(l,p-1)} \; ) +$$

$$\left. \delta G_{r(i,p),r(M_R(j),p)} y_{r(i,p)} \; ) \right]$$

where $\delta_{ij}$ is the Kronecker delta function. The boundary terms $\delta \mathbf{b}$ for the right-hand-side $\mathbf{b}$ are obtained from the equations

$$\delta_{p,n-k} \left[ f^{(1)}_i(R) - \sum_{j=1}^{N_M^R} a^{(1)}_{iM_R(j)}(R) \gamma^{(R)}_{Mj} \right] - \delta_{p,1} \left[ f^{(1)}_i(L) - \sum_{j=1}^{N_M^L} a^{(1)}_{iM_L(j)}(L) \gamma^{(L)}_{Mj} \right] \equiv$$

$$\delta_{p,n-k} \left[ f^{(1)}_i(R) - \sum_{j=1}^{N_M^R} a^{(1)}_{iM_R(j)}(R) \gamma^{(R)}_{Mj} \right] - \delta_{p,1} \left[ f^{(1)}_i(L) - \sum_{j=1}^{N_M^L} a^{(1)}_{iM_L(j)}(L) \gamma^{(L)}_{Mj} \right] \equiv \tag{7.15}$$

$$\delta_{p,n-k} \delta b_{i+(p-1)n_u} - \delta_{p,1} \delta b_i \; .$$

With the above boundary terms (7.14) and (7.15) added on, the Galerkin matrix $\mathbf{G}$ and right-hand-side $\mathbf{b}$ are ready to do a fully mixed problem, that is, one with no Dirichlet **bc**'s. Let $D^{(L)}(i)$, $i = 1, \cdots, n^L_{u_D}$, be the indices of $\mathbf{u}^{(L)}_D$, similarly for $D^{(R)}$. Then the $E^{(L)}_D(D^{(L)}(i))^{th}$ Galerkin equation is replaced by the Dirichlet **bc** equation for $u_{D^{(L)}(i)}(L)$, $i = 1, \cdots, n^L_{u_D}$. The $E^{(R)}_D(D^{(R)}(i)) + (n-k-1) n_u{}^{th}$ Galerkin equation is replaced by the Dirichlet **bc** equation for $u_{D^{(R)}(i)}(R)$, $i = 1, \cdots, n^{(R)}_{u_D}$. This completes the formulation of the Galerkin equations. The result is a system of linear algebraic equations of the form

$$\mathbf{G}\mathbf{y} = \mathbf{b} \tag{7.16}$$

for the B-spline coefficients $\mathbf{y}$ (7.10) of the solution $\mathbf{u}$ of (7.8).

**Computational Complexity**

The matrix **G** is $n_u(N-k) \times n_u(N-k)$ with a band-width of $(2kn_u - 1)$, see (7.12). Thus, the system (7.16) may be solved in roughly $n_u\ (N-k)\ (2\ k\ n_u - 1)\ k\ n_u$ arithmetic operations [55], and the B-spline Galerkin solution can be obtained quite efficiently.

The Galerkin system (7.16) is based on the computation of many integrals. These integrals are computed with Gauss-Legendre quadrature [45]. If we let $O\ =\ \min_i\ \max_j\ O_{ij}$ be the minimum order of all $u_i$ in the **ode** system (7.8), then we wish to compute

$$\int_L^R B_p^{(O)}(x)\, B_q(x)\ dx$$

exactly [45] with an $m_q$-point Gauss-Legendre quadrature rule. This is best accomplished for

$$m_q\ =\ \begin{cases} k & \text{if}\quad O\ =\ 0 \\ k-1 & \text{if}\quad O\ >\ 0. \end{cases} \tag{7.17}$$

Using such a quadrature rule, the Galerkin system (7.16) may be formed in roughly

$$4\ (N-k)\, k^2\, n_u^2\, m_q \tag{7.18}$$

operations, while the right-hand-side **b** alone may be computed in roughly

$$2\ (N-k)\, k n_u\, m_q \tag{7.19}$$

operations. If the LU factorization of **G** and the right-hand-side **b** are known, then the solution **y** may be computed in roughly

$$(N-k)\ n_u\ (3\ k\ n_u - 2) \tag{7.20}$$

operations.

## 8. Advice for the Sporting User

This section describes, in gory detail, the various "knobs" and ways of overriding the default options and subprograms used in `POST`. Several examples of knob twiddling that can speed `POST` up considerably are also given.

An exceedingly brief outline of the organization of `POST` is, in pseudo-English,

```
t0 = tstart
While ( t0 != tstop )        # Time-step loop.
    {
    Do m = 1, ..., mmax        # Loop to build extrapolation tableau.
        {
        Do istep = 1, ..., N(m)        # Sub-steps loop.
            {
            Get_Initial_Newton_Guess at time t0+istep*dt ( INMI )
            Do iter = 1, ..., Maxit        # Newton loop.
                {
                Solve linearized Galerkin Equations; Update solution
                Check ERROR for "convergence"; Check Convergence Rate
                }
            }
        Extrapolate and check the ERROR.
        }
    Get_Optimal_dt for next time-step.
    Output the results for this time-step ( HANDLE )
    t0 = t0 + dt
    }
```

where the capitalized items in parentheses refer to procedures names, and the linearized Galerkin solution is obtained by

Get integrals for matrix ( AF ), mesh interval by mesh interval,
  left to right, using either `k-1` or `k` Gaussian quadrature points per interval.
Get the **bc**s ( BC ).
Get **pde** ( AF ) at x = L and R (two calls, one for each side).
Scale and solve the **pde** system (multiple right-hand-sides) ( SCALE ).
Get the **ode** ( DEE )
Scale and solve the **ode** system (multiple right-hand-sides) ( SCALE ).
Use (6.4) to sew the **pde** and **ode** solutions together.

Both POST and POSTR ( described immediately below ) will try to use a $k-1$ point Gaussian quadrature rule. If that rule fails, it will try a $k$ point rule and give up if that fails, because theoretically it must work, see section 7.

This section describes the many variables and procedures that define the above algorithm. These include the maximum number of levels of extrapolation to allow ( `mmax` ), the sequence of sub-steps to take ( N(1), N(2), $\cdots$ ), and the maximum number of Newton iterations to allow ( `maxit` ).

**Twiddling Procedure Knobs.**

Control over the procedures ERROR, INMI and SCALE is given by POSTR. This routine allows the user to override some of the default routines of POST. POSTR is invoked by

```
Call POSTR(U,Nu,k,x,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,D,
           ERROR,errpar,
           INMI,SCALE,
           HANDLE)
```

The extra arguments (ERROR, INMI and SCALE) in this subroutine provide direct user control over the accuracy of the integration process (ERROR), the ability to choose the initial Newton's method guess for each sub-step (INMI) and to scale the Jacobian of the system (SCALE).

The default routines, used by POST, are POSTE for ERROR, POSTN for INMI and POSTP for

SCALE.

**Error Options**

There are several possible options available for error specification. First, the user, via the subprogram ERROR, may specify literally any accuracy requirement desired for the solution. Second, there are several popular methods of error control which are controlled by the switch erputs, and implemented by the subprogram POSTE.

The error control provided in the subroutine POSTE is based on the local value of the variables. That is, the error acceptable in X(I) is

errpar(1) * | X(I) | +errpar(2)

which depends only upon the current value of X(I), where X is either a block of B-spline coefficients **U** or a single $\mathbf{v}_i$.

The error control provided in the subroutine POSTE is an *error per time-step* criterion. This can be rather bad if the time-steps taken during the solution process get very small - many time-steps will be taken and the errors may pile up in unacceptable amounts. Another error option is to use an *error per unit-time-step* criterion. By making the error tolerance in X(I) look like

| dt | * ( errpar(1) * | X(I) | +errpar(2) ) ,

when the time-step gets small, so does the error requirement. However, when using the error per unit-time-step criterion, the reverse argument holds - when dt is large, so is the error tolerance. The error per time-step versus unit-time-step option is controlled by the switch erputs as described below.

The inputs for POSTR are as previously described in POST, with the following additions:

**ERROR**

This is a subprogram for computing accuracy requirements for the solution process. It must be declared External in the subprogram calling POSTR. The default ERROR subroutine used by POST is POSTE. POSTR uses ERROR in determining when the Newton and extrapolation processes have "converged." When POST has computed a tentative solution vector, it also has an estimate available for the error in that vector. POST uses the function subprogram ERROR to ask the user if the computed solution vector is sufficiently accurate. ERROR must have the form

```
valu = ERROR(U,Nu,nxmk,k,x,nx,V,Nv,t,dt,
             errpar,
             erputs,
             eU,eV)
```

where valu is Logical, U=U(t) is the **pde** solution vector of length (nx-k)*Nu and V=V(t) is the solution vector of length nv for which error tolerances are to be supplied by ERROR. The inputs to ERROR are

| | | |
|---|---|---|
| U | - | The array of B-spline coefficients for the **pde** solution **u**. |
| Nu | - | The number of **pde** variables. |
| nxmk | - | The number of mesh points in the mesh x minus k, the B-spline order; present so that Real U(nxmk,Nu) can be dimensioned. |
| k | - | The B-spline order, as passed to POST. |
| x | - | The B-spline mesh, as passed to POST. |
| nx | - | The length of the B-spline mesh, as passed to POST. |
| V | - | The vector of **ode** solutions at time t. |
| Nv | - | The length of the array V. |

- `t` - The time t at which the solution is evaluated.
- `dt` - The time-step used to obtain V=V(t).
- `errpar` - A Real vector of length two, as passed to `POSTR`, or as modified by a previous call to `ERROR`. (See below.)
- `erputs` - This Logical variable has the same value as the input variable `erputs` passed to `POSTR`. (See the `POSTW` description below.)
- `eU` - This Real vector gives the absolute accuracy of the solution `U`, as computed by `POST`. The absolute error in `U(I,J)` is `eU(I,J)`, for `J`=1, $\cdots$ ,`Nu` and `I`=1, $\cdots$ ,`nxmk`, for the single current time-step.
- `eV` - This Real vector gives the absolute accuracy of the solution `V`, as computed by `POST`. The absolute error in `V(I)` is `eV(I)`, for `I`=1, $\cdots$ ,`nv` for the single current time-step.

The value returned by `ERROR` to `POSTR` is

`.True.` if the tentative solution `V` is satisfactory to the user; otherwise `.False.` .

The output from `ERROR` is

- `errpar` - This vector may be altered, if desired.
- `eU` - The vector of Real absolute errors the user will tolerate in the proposed solution vector `U`. `eU(I,J)` is the acceptable absolute error in `U(I,J)`, for `I`=1, $\cdots$ ,`nxmk` and `J`=1, $\cdots$ ,`Nu`. All of the returned `eU(I,J)` must be positive.
- `eV` - The vector of Real absolute errors the user will tolerate in the proposed solution vector `V`. `eV(I)` is the acceptable absolute error in `V(I)`, for `I`=1, $\cdots$ ,`Nv`. All of the returned `eV(I)` must be positive.

The default `ERROR` procedure used by `POST` is `POSTE`.

**INMI - Initial Newton Method Iterate.**

This procedure initializes the Newton iteration for each sub-step. `INMI` should be declared External in the procedure calling `POSTR`. The default `INMI` procedure used by `POST` is `POSTN`, which uses the solution at the last sub-step as the initial guess. Before starting the Newton iteration for a sub-step in the extrapolation process, `POST` invokes `INMI` to generate an approximation to the solution at the current value of time.

```
Call INMI(Nu,Nv,nxmk,k,x,nx,
          t,dt,
          Uold,Vold,
          U,Ut,V,Vt)
```

The input to `INMI` is

- `Nu` - The number of **pde** variables, as input to `POST`.
- `Nv` - The number of **ode** variables, as input to `POST`.
- `nxmk` - The number of B-spline coefficients, `nx-k`. Present so that `Uold`, `U` and `Ut` may be dimensioned (`nxmk`,`Nu`).
- `k` - The B-spline order, as input to `POST`.
- `x` - The B-spline mesh, as input to `POST`.
- `nx` - The length of the B-spline mesh, as input to `POST`.
- `t` - The current value of time, t.
- `dt` - The current value of the time-step.

Uold - The previous value of U, at time t-dt.

Vold - The previous value of V, at time t-dt.

U - The same as Uold, the default best guess for the value of U.

Ut - An array, set to all 0. The best guess for Ut, given the above best guess for U.

V - The same as Vold, the default best guess for the value of V.

Vt - An array, set to all 0. The best guess for Vt, given the above best guess for V.

The output of INMI is

U - The estimate for the solution at time t.

Ut - The estimate for the time derivative of the solution at time t, Ut = ( U - Uold )/dt.

V - The estimate for the solution at time t.

Vt - The estimate for the time derivative of the solution at time t, Vt = ( V - Vold )/dt.

**SCALE**

This procedure provides for scaling of the Jacobian of the system before numerical solution is attempted. It must be declared External in the subprogram calling POSTR. The default SCALE subroutine used by POST is POSTP. Whenever a Jacobian is to be factored, SCALE is first invoked by

```
Call POSTP(i,j,A,m,n)
```

The input to SCALE is

i - An Integer indicating what equation is to be scaled. i = 1 indicates that the **pde** system is to be scaled. i = 4 indicates that the **ode** system is to be scaled. Don't worry about i = 2 or 3.

j - An Integer indicating what is to be scaled. j = 1 indicates that the Jacobian is to be scaled. j = 2 indicates that the right-hand-side is to be scaled. j = 3 indicates that the solution is to be scaled.

A - The item to be scaled, either the Jacobian, the right-hand-side or the solution. It is dimensioned to be A(m,n).

m - The first dimension of A.

n - The second dimension of A.

The output of SCALE is

A - The scaled item, whatever it is.

SCALE typically needs some scratch space to carry out its mission. This also provides a way for POST to see if the user's SCALE routine is alive in any sense. The common region

```
Common / A9OSTS / imem(4)
```

contains 4 Integer variables, which are initially set to 0 by POST. If in SCALE imem(i) ≡ 0, then SCALE should create whatever scratch space is needed for that problem ( **pde** or **ode** ), signal POST that it is alive by setting imem(i) to some non-zero value, and then simply return. In the default SCALE routine POSTP this is done by putting the stack pointer to the scratch space into imem(i). POSTP uses row and column scaling, very carefully done, to avoid destructive underflow and overflow.

The double precision version of POSTN is DPOSTN, while that for A9OSTS is D9OSTS.

The output from POSTR is the same as that for POST with the additional possibility that the user may alter errpar through his subprogram ERROR.

The double precision version of POSTR is DPOSTR, with all Real arguments typed double precision, except errpar, which remains Real. Similarly for POSTP and DPOSTP.

**Twiddling non-Procedure Knobs.**

The main knob-twiddling routine is `POSTW`. When the user wants to tinker some internal variable of `POST` the

```
Call POSTW(j,f,r,i,l)
```

will do the trick. The input to `POSTW` is an index ( `j` ) identifying the knob to be turned, and its value ( one of `f`, `r`, `i` or `l` ). For any value of `j`, only **one** of `f`, `r`, i or `l` is used to set the knob, the other variables may be set to anything, like `0d0`, `0e0`, `0` or `.true.`. Any number of variables can be set by calling `POSTW` any number of times. The input to `POSTW` is

- `j` - The index of the variable to be set. If `j` = 0, then **all** variables are set to their default values. You should not have to use `j = 0` unless you have already set a few parameters using `POSTW` and you now want to reset the default values.
- `f` - If the variable to be set is a working-precision item, `f` is to be its new value. Working precision is Real for `POST` and Double Precision for `DPOST`.
- `r` - If the variable to be set is a Real item ( `hfract`, `egive` ), `r` is to be its new value.
- `i` - If the variable to be set is an Integer item, `i` is to be its new value.
- `l` - If the variable to be set is a Logical item, `l` is to be its new value.

The output of `POSTW` is the internal knowledge of the new value of the variable that has been set.

The following list gives the items that can be set using `POSTW`.

- `theta` - The time-discretization parameter, see section 6. When `theta` = 1 the extremely stable, first order accurate Backwards-Euler formula is used. For `theta` = 1/2, the second-order Crank-Nicholson scheme is used. If `theta` ≠ 1/2, then `N` = { 1, 2, 3, 4, 6, $\cdots$ } and `gamma` = 1. If `theta` = $\frac{1}{2}$, then `N` = { 2, 4, 6, $\cdots$ } and `gamma` = 2. 0 <= `theta` <= 1 is required. Default: `theta` = 1. `j` = 1. `theta` = `f`.
- `beta` - The error in the discretization scheme is proportional to (`t1-t0`)**`beta`. Default: `beta` = 1. `j` = 2. `beta` = `f`.
- `gamma` - The error in the discretization scheme is proportional to `dt`**`gamma`. Default: `gamma` = 1. `j` = 3. `gamma` = `f`.
- `delta` - The error request is proportional to `dt`**`delta`. Default: `delta` = 0. `j` = 4. `delta` = `f`.
- `hfract` - A Real variable indicating how small a time-step the user will take relative to `dt`. Default: `hfract` = 1. `j` = 1001. `hfract` = `r`.
- `egive` - A Real variable controlling how accurately Newton's method will try to solve the nonlinear equations, relative to the user's accuracy request for the solution to the **pde**. Specifically, it will try to solve the nonlinear equations to a tolerance of ( the user's error request to `POST` ) / `egive`. `egive` should thus be greater than 1. Default: `egive` = 1e+2. `j` = 1002. `egive` = `r`.
- `kj` - A variable that controls the frequency with which the Jacobian is re-computed. Such evaluations can be very costly and `kj` (for "KeepJacobian") controls them in the following way:

0 - New Jacobian every Newton iteration.
Very safe and stable, expensive.
1 - New Jacobian every time sub-step.
Less safe, stable and expensive.
2 - New Jacobian for each time-step.
Not very safe, stable or cheap, except for nearly linear problems.
3 - New Jacobian whenever there is a re-start.
Mostly used for linear or nearly linear problems.
Cheap but flaky.
4 - New Jacobian whenever Newton iteration fails to converge.
Only updates Jacobian if it appears out-of-date.
Ditto rest of discussion of 3 above.
5 - Only computes the Jacobian ONCE.
Use only for nearly linear problems.
Exceedingly cheap, when it works.

Default: `kj` = 0. `j` = 2001. `kj` = `i`.

`minit` - The minimum number of Newton iterations to go before checking that the convergence rate is reasonable. Default: `minit` = 10. `j` = 2002. `minit` = `i`.

`maxit` - The maximum number of Newton iterations to use. Default: `maxit` = 50. `j` = 2003. `maxit` = `i`.

`kmax` - The maximum number of columns allowed in the extrapolation tableau. The maximal order that `POST` can achieve is then 2*`kmax` if `theta` = 0.5e0, or `kmax` if `theta` ≠ 0.5e0. Default: `kmax` = 10. `j` = 2004.

`kinit` - The initial level of extrapolation to use for the first time-step. This can allow `POST` to use a higher-order scheme from the start. Default: kinit = 4. `j` = 2005. `kinit` = `i`.

`mmax` - The maximum number of levels of extrapolation permitted. `mmax` >= `kmax` + 2 is required and `mmax` >= `kmax` + 4 is a good idea. Default: `mmax = 15`. `j` = 2006. `mmax` = `i`.

`mgq` - The number of Gaussian quadrature points to be used to compute the Galerkin integrals, see section 6. `mgq` >= `k-1` is required. Default: `mgq` = `k-1`. `j` = 2007. `mgq` = `i`.

`saveb` - If `saveb` is $<0$, space will *not* be allocated for saving the B-spline basis function values. Re-evaluation of the spline basis can be expensive, and if `saveb` is $>0$, then space is allocated for saving the basis values. If `saveb` is $=0$, then the values will be saved when $n_u > 0$ and `KJ` $> 1$. Default: `saveb` = 0. `j` = 2008. `saveb` = `i`.

`xpoly` - A Logical variable indicating whether polynomial (True) or rational (False) extrapolation is to be used. Default: `xpoly` = False. `j` = 3001. `xpoly` = `l`.

`erputs` - Logical variable controlling the use of error-per-unit-time-step. If True, then use per-unit-time-step error criterion. Otherwise, use the per-step criterion. Default: `erputs` = False. `j` = 3002. `erputs` = `l`.

`N` - An Integer array of length `mmax` giving the number of sub-steps to be used in the extrapolation. `N` must be strictly monotone increasing and positive. `N`(i) is set by `j` = 4000+i. If `N`(i) is set, then `N`(i+1) is set to 0 by default; so be sure to set `N` in increasing order of i. For any `N`(i) which is 0, a default value is computed. The rule is that if only `N`(1) is set, then the user wants `N`(i) ≡ $(\sqrt{2})^{i-1}$ `N`(1). If only `N`(1) and `N`(2) are set, then `N`(i) = ( `N`(2)/`N`(1) ) `N`(i-1) for any `N`(i) = 0. If `N`(3) is also set, then `N`(i) = 2 * `N`(i-2), for any `N`(i) = 0. See the `theta` description above for some default `N` values. `j` = 4000 + i. `N`(i) = `i`.

The following table summarizes the values that can be set by `POSTW`

| Name | j | Default | Set to |
|---|---|---|---|
| `theta` | 1 | 1 | `f` |
| `beta` | 2 | 1 | `f` |
| `gamma` | 3 | 1 | `f` |
| `delta` | 4 | 0 | `f` |
| `hfract` | 1001 | 1 | `r` |
| `egive` | 1002 | 100 | `r` |
| `kj` | 2001 | 0 | `i` |
| `minit` | 2002 | 10 | `i` |
| `maxit` | 2003 | 50 | `i` |
| `kmax` | 2004 | 10 | `i` |
| `kinit` | 2005 | 4 | `i` |
| `mmax` | 2006 | 15 | `i` |
| `mgq` | 2007 | `k-1` | `i` |
| `saveb` | 2008 | 0 | `i` |
| `xpoly` | 3001 | `False` | `l` |
| `erputs` | 3002 | `False` | `l` |
| N(i) | 4000+i | – | `i` |

The Double precision version of `POSTW` is `DPOSTW`, with all Real arguments typed Double precision, except `hfract` and `egive` which remain Real.

**Some Knob Twiddling**

We now show how twiddling some of the knobs in `POSTW` can affect the run-time for each of the examples in section 4. The knobs considered are `theta`, `maxit` and `xpoly`. These variables control the order of accuracy in the time-stepping scheme, the amount of iteration done in solving the nonlinear equations and whether rational or polynomial extrapolation is used.

The default call to `POST` uses $\theta = 1$, a full Newton iteration scheme to solve the nonlinear equations ( `maxit` = 50 ) and rational extrapolation in the time evolution ( `xpoly = .false.` ). This will be referred to as NBE for Nonlinear Backwards Euler. The run-times reported in Appendix 4 for the examples are for the default NBE scheme.

The next scheme considered is the Crank-Nicholson method [39], which will be denoted by CN. That scheme uses $\theta = 1/2$, with all other settings the same as with NBE. The advantage of CN over NBE is that CN is second order accurate in the time-step, while NBE is only first order accurate. We shall see if the order advantage gives any run-time advantage. The way in which the user tells `POST` that `theta` $= 1/2$ is to be used is by inserting the

```
call postw(1,5e-1,0e0,1,.true.)
```

just before the call to `POST`.

The third scheme considered is the same as NBE but with `maxit` set to 1. This may seem strange, in that it does only *one* Newton iteration to solve each nonlinear equation. However, it works quite well on most problems and it possesses the asymptotic expansion needed for extrapolation to work, see Appendix 2. This scheme will be denoted by LBE, for Linearized Backwards Euler. The user tells `POST` that `maxit` is to be 1 by inserting the

```
call postw(+2003,0d0,0e0,1,.true.)
```

just before the call to `POST`.

The fourth scheme will be the LBE scheme above with `xpoly = .true.`, so that polynomial extrapolation is used, rather than rational. This scheme will be denoted by LBE.p for LBE with polynomial extrapolation. The user tells `POST` that `xpoly` is to be `.true.`, as well as setting `maxit` as above, by inserting

```
call postw(+2003,0e0,0e0,1,.true.)
call postw(+3001,0e0,0e0,0,.true.)
```

just before the call to `POST`.

The last scheme will be the linear, semi-implicit Euler scheme of [1] in which `kj = 2` and `maxit = 1`. This scheme will be denoted by SIE, for Semi-implicit Euler. The user tells `POST` that `kj` is to be `2`, as well as setting `maxit = 1` as above, by inserting

```
call postw(+2001,0e0,0e0,2,.true.)
call postw(+2003,0e0,0e0,1,.true.)
```

just before the call to `POST`.

The results for the five methods described above are summarized in the table:

| Example Run-time in Seconds | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Method | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| NBE | 8.3 | 27.2 | 11.3 | 89.8 | 31.7 | 152.8 | 313.1 | 47.2 | 185.6 | 39.0 |
| CN | 15.6 | 46.8 | 15.2 | 69.7 | 55.5 | 114.3 | 936.6 | 79.9 | 653.9 | 70.2 |
| LBE | 4.8 | 15.4 | 6.3 | 46.9 | 20.9 | 120.1 | 144.1 | 25.4 | 162.1 | 21.9 |
| LBE.p | 4.7 | 17.3 | 6.5 | 53.7 | 21.4 | 126.5 | 160.3 | 25.6 | 161.1 | 22.1 |
| SIE | 4.2 | 11.1 | 5.3 | 34.1 | 21.5 | 88.8 | 170.0 | 21.8 | 144.9 | 19.8 |

Note: CN died on examples 7 and 9, and SIE gave the *wrong* answer to example 6!

CN is faster than NBE only on examples 4 and 6 of section 4. However, CN failed to solve example 7 of section 4 correctly - it died with `dt` $= 0$ after just a few time-steps. That problem is the boundary-layer tracking example, which involves an algebraic condition on the solution **u**. CN is generically incapable of solving such problems, as the following example shows. Consider the **ode**

$$y = 0$$

with a slightly incorrect initial value guess of $y = \varepsilon$. The CN approximation to the solution will be alternately $+\varepsilon$ and $-\varepsilon$. Hence, unless $\varepsilon \equiv 0$ is chosen for the initial conditions ( **ic**s ), CN will fail to converge to the solution of even the linear **ode** $y = 0$!

CN also failed on example 9, which resulted in an infinite loop trying to make the mesh fine enough to achieve the desired accuracy. The problem was that CN does not make the error decay as time approaches $\infty$, hence all the error estimates were large, no matter what the mesh.

The table shows that CN can save run-time in a few cases, but is in general unsafe and slow.

LBE is a notable improvement over both NBE and CN. It is uniformly faster, in the above results, and doesn't fail like CN. Usually, LBE results in roughly a factor of two speed-up over NBE. That is reasonable, given that it only uses one Newton iteration, while NBE usually needs two or three iterations. However, even LBE requires a couple of caveats. First, there are strange equations for which LBE can speedily give results that are dead wrong! This is most easily seen by considering the **ode**

$$( y' )^2 + y' + 1/4 = 0$$

with **ic**s $y(0) = 0$. The solution is $y(t) \equiv -t/2$. However, LBE will treat the $( y' )^2$ term as 0 and obtain the solution to

$$y' + 1/4 = 0$$

that is, $y(t) = -t/4$, the wrong answer. This kind of thing will happen whenever the time derivative term in the **ode** or **pde** appears nonlinearly. Such equations occur very rarely in practice and NBE should be used to solve them. So long as the time derivatives appear linearly, LBE is prefectly safe.

The second caveat about LBE is that it can be exceedingly slow in certain circumstances. An obvious way to break LBE would be to give it not an **ode**, but a nonlinear equation to solve with **ic**s too far from the solution for *one* Newton iteration to achieve the desired accuracy. LBE will die with `dt` $= 0$ in such a case. Clearly, the way to make an **ode** that will gag LBE is to construct an **ode** which is nearly such a nonlinear equation. Dahlquist proposed just such a problem ( for other reasons )

$$\varepsilon y' = (1 - t)\, y - y^2 \qquad \text{on} \quad [0, 2],$$

with $y(0) = 1$. For small ε, $y$ is nearly two straight line segments

$$y(t) = \begin{cases} 1 - t & t < 1 \\ 0 & t > 1. \end{cases}$$

Also, for ε small, it is nearly a nonlinear equation rather than an **ode**. NBE fairly consumes this problem, more or less independently of ε using only a few time-steps to solve the problem. LBE can be made to run arbitrarily slowly as $\varepsilon \to 0$. Consider a time $t < 1$; then the Newton correction $w$ from time $t - dt$ to time $t$ using LBE is

$$w = -dt \frac{1 - t + dt}{1 - t + dt + \dfrac{\varepsilon}{dt} + dt}.$$

Since $w = -dt$ would give the exact answer, we clearly want $\dfrac{\varepsilon}{dt} + dt = 0$, which cannot be done. We can, however, minimize $\dfrac{\varepsilon}{dt} + dt$ by choosing $dt = \sqrt{\varepsilon}$. That is precisely what happens in practice; LBE will take $\dfrac{1}{\sqrt{\varepsilon}}$ time-steps in going from time 0 to time 1, and LBE can be made to run arbitrarily slowly as $\varepsilon \to 0$. However, such situations are rare in practice; LBE is generally much faster than NBE.

Despite the caveats issued above about LBE, LBE is second only to NBE in robustness and is usually substantially faster.

Comparing LBE and LBE.p ( polynomial extrapolation ) shows that polynomial, in the examples at least, rarely beats rational, and then only by a bit, and is frequently quite a bit slower than rational.

SIE is sometimes an improvement over LBE, but it failed on example 6, getting the wrong answer. Linear semi-implicit schemes should *only* be used when the $\mathbf{u}_t$ and $\mathbf{v}_t$ terms appear linearly and with coefficients depending only on $x$, not on $\mathbf{u}$, $\mathbf{v}$ or $t$. Note that mesh mapping, as used in example 6, destroys this property even though the original problem had constant coefficients for $\mathbf{u}_t$.

**The Right Stuff.**

The above table and its discussion shows why NBE was chosen as the default method for `POST`. It is very robust and reasonably fast. Additional speed can be achieved, but only by sacrificing robustness.

If the user wishes to speed `POST` up, the simplest and safest thing to do is try LBE. It usually works and is faster than NBE. The user should make sure to compare at least a few LBE runs with NBE, to make sure it is correct and running efficiently. Once the utility of LBE for the class of problems at hand has been ensured by a few comparisons, the production runs can be done with LBE.

**Run-time Statistics.**

A subroutine is provided to print run-time statistics for `POST`. The statement

```
Call POSTX
```

will print a line of the form:

```
post(j,f,ts,ss,nit,nd,nf,r) =  130  130   15   76  130    0    0    0
```

The fields of this line refer to

`j` - The number of Jacobian evaluations.

`f` - The number of factorizations of the Jacobian.

`ts` - The number of time-steps.

`ss` - The number of sub-steps.

`nit` - The number of Newton iterations.

`nd` - The number of predicted Newton failures ( error increasing ).

`nf` - The number of Newton failures ( more than maxit iterations ).

`r` - The number of restarts.

If `POSTX` is invoked by the user while inside `POST`, the statistics reported will be the current values. If invoked outside `POST`, the statistics will be those of the last call to `POST`.

**Another way Into `POST`.**

The following subprograms are described for historical reasons. Their use is strongly discouraged, but possible.

For those who do not wish ( or like ) to use `POST`/`POSTW`/`POSTR`, there is another way of entering `POST`

```
Call POSTS(U,Nu,k,x,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,D,
           errpar,
           HANDLE)
```

with further control over the error of the solution given by

```
Call POST1(U,Nu,k,x,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,D,
           ERROR,errpar,erputs,
           HANDLE)
```

where the default method for both of the above is LBE. This is to keep them upwards compatible with their previous versions. Note that NBE is the default for `POST`, *not* LBE. Thus, the default method for `POSTS` and `POST1` is different from that for `POST`

Even more control over the integration process is given by

```
Call POST2(U,Nu,k,x,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,D,
           Equil,
           kmax,xpoly,kinit,
           ERROR,errpar,erputs,
           HANDLE)
```

where `Equil` is a `Logical` variable determining if LBE ( `.true.` ) or CN is to be used ( `.false.` ); its default value is `.true.` when invoked via `POSTS` and `POST1`.

There is further control in

```
Call POST3(U,Nu,k,x,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,D,
           theta,KeepJac,Minit,Maxit,
           mgq,
           kmax,xpoly,kinit,
           ERROR,errpar,erputs,
           INMI,SCALE,
           HANDLE)
```

and finally all 35 arguments are available for concurrent twiddling in the

```
Call POST4(U,Nu,k,x,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,D,
           theta,KeepJac,Minit,Maxit,
           mgq,
           beta,gamma,delta,N,mmax,hfract,egive,saveb,
           kmax,xpoly,kinit,
           ERROR,errpar,erputs,
           INMI,SCALE,
           HANDLE)
```

**Acknowledgements**

Many thanks are due A. V. Aho for suggesting that a tree-back-tracking scheme be used in the boundary condition placement algorithm of section 7. Special thanks are also due the brave and patient L. O. Wilson, who was among the first to use the package. She has endured evolving calling sequences, as well as the process of discovery and elimination of a bug in the package that I tried very hard to explain away as a problem with her **pde** formulation. L. A. Farrow, G. A. Nikolakopoulou, J. R. MacDonald and A. L. Simons were most creative in their use of `POST` and the package has benefited greatly from their comments and inventiveness.

**Bibliography**


[1] G. Bader and P. Deuflhard, "A Semi-implicit Mid-Point Rule for Stiff Systems of Differential Equations," Universitat Heidelberg, Institute fur Angewandte Mathematik, Report # 114, April 1981.

[2] R. E. Bank and D. J. Rose, "Global Approximate Newton Methods," *Numer. Math.* **37,** 279-295(1981).

[3] C. deBoor, **A Practical Guide to Splines,** Springer, New York, Applied Math. Sciences 27, 1978.

[4] C. de Boor, "On Uniform Approximation by Splines," *J. Approx. Th.* **1,** 219-235(1968).

[5] C. de Boor, "On Calculating with B-splines," *J. Approx. Th.* **6,** 50-62(1972).

[6] C. deBoor and B. Swartz, "Collocation at Gaussian Points," *SIAM J. Numer. Anal.* **10,** 582-606(1973).

[7] R. Bulirsch and J. Stoer, "Fehlerabschatzungen und Extrapolation mit rationalen Funktionen bei Verfahren vom Richardson-Typus," *Numer. Math.* **6,** 413-427(1964).

[8] R. Bulirsch and J. Stoer, "Numerical Treatment of Ordinary Differential Equations by Extrapolation Methods," *Numer. Math.* **8,** 1-13(1966).

[9] R. Bulirsch and J. Stoer, "Asymptotic Upper and Lower Bounds for Results of Extrapolation Methods," *Numer. Math.* **8,** 93-104(1966).

[10] J.H. Cerutti and S.V. Parter, "Collocation Methods for Parabolic Partial Differential Equations in One Space Dimension," *Numer. Math.* **26,** 227-254(1976).

[11] P. G. Ciarlet, M.H. Schultz and R.S. Varga, "Numerical Methods of High-Order Accuracy for Nonlinear Boundary Value Problems I. One Dimensional Problem.," *Numer. Math.* **9,** 394-430(1967).

[12] P. G. Ciarlet, M.H. Schultz and R.S. Varga, "Numerical Methods of High-Order Accuracy for Nonlinear Boundary Value Problems II. Nonlinear Boundary Conditions," *Numer. Math.* **11,** 331-345(1968).

[13] P.G. Ciarlet, M.H. Schultz and R.S. Varga, "Numerical Methods of Higher-Order Accuracy for Nonlinear Boundary Value Problems III. Eigenvalue Problems," *Numer. Math.* **12,** 120-133(1968).

[14] R. Courant and D. Hilbert, **Methods of Mathematical Physics,** Vol. 1, Interscience, New York, 1966.

[15] R. Courant and K. O. Friedrichs, **Supersonic Flow and Shock Waves,** Interscience, New York, 1948, pp 124-126.

[16] H.B. Curry and I.J. Schoenberg, "On Polya Frequency Functions IV: The Fundamental Spline Functions and their Limits," *J. of Anal. and Math.* **17,** 71-107(1966).

[17] G. Dahlquist, "A Special Stability Problem for Linear Multistep Methods," *BIT* **3,** 27-43(1963).

[18] G. Dahlquist, "Stability Questions for Some Numerical Methods for Ordinary Differential Equations," *Proc. Symp. for Applied Math.* **15,** 147-158(1963).

[19] J. Douglas and T. DuPont, "A Finite Element Collocation Method for Quasilinear Parabolic Equations," *Math. Comp.* **27,** 17-28(1973).

[20] J. Douglas and T. Dupont, "Galerkin Approximations for the Two-point Boundary Value Problem Using Continuous, Piecewise Polynomial Spaces," *Numer. Math.* **22,** 99-109(1974).

[21] D. Edelson and N. L. Schryer, "Modeling Chemically Reacting Flow Systems - I: A Comparison of Finite Difference and Finite Element Methods for One-Dimensional Reactive Diffusion," *Computers and Chemistry* **2,** 71-74(1978).

[22] D. Edelson, G. N. Nikoulakoupoulou and N. L. Schryer, "Modeling Chemically Reacting Flow Systems - II: An Adaptive Spatial Mesh Technique for Problems with Discontinuities and Steep Fronts," *Computers and Chemistry,* **6,** 93-99(1982).

[23] W.H. Enright, T.E. Hull and B. Lindberg, "Comparing Numerical Methods for Stiff Systems of Ordinary Differential Equations," *BIT* **15,** 10-48(1975).

[24] G. Forsythe and C. Moler, **Computer Solution of Linear Algebraic Systems,** Prentice-Hall, New York, 1967.

[25] P.A. Fox, A.D. Hall and N.L. Schryer, "The PORT Mathematical Subroutine Library," *TOMS*, **4,** 104-126(1978).

[26] D. R. Franceschetti and J. R. MacDonald, "Numerical Analysis of Electrical Response: Statics and Dynamics of Space-Charge Regions at Blocking Electrodes," *J. Appl. Phys.*, **50,** 291-302(1979).

[27] C.W. Gear, "The Automatic Integration of Ordinary Differential Equations," *Comm. ACM* **14,** 176-179(1971).

[28] S.W. Golomb and L.D. Baumert, "Backtrack Programming," *J. ACM* **12,** 516-524(1965).

[29] W.B. Gragg, "Repeated Extrapolation to the Limit in the Numerical Solution of Ordinary Differential Equations," Thesis, UCLA (1963).

[30] W.B. Gragg, "On Extrapolation Algorithms for Ordinary Initial Value Problems" *SIAM J. Num. Anal.* **2,** 384-403(1965).

[31] W.B. Gragg, "Lecture Notes on Extrapolation Methods," presented at the SIAM National Meeting, Washington, June 1971, and at the Conference on Ordinary Differential Equations, Dundee, Scotland, August, 1971.

[32] H. S. Greenside, W. M. Coughran and N. L. Schryer, "Nonlinear Pattern Formation Near the Onset of Rayleigh-Bernard Convection," *Physical Review Letters*, **48,** 726-729(1982).

[33] A.D. Hall and S.I. Feldman, "EFL, an Extended FORTRAN Language," in preparation.

[34] H. Ikezi, K.F. Schwarzenegger, A. L. Simons, Yukiharu Ohsawa, T.S. Kamimura, "Nonlinear Self-Modulation of Ion-Acoustic Waves", *Physics of Fluids*, **21,** 239-248(1978).

[35] B.W. Kernighan, "RATFOR - A Preprocessor for a Rational Fortran," *Software-Practice and Experience* **5,** 395-406(1975).

[36] W. J. Layton, "Stable and Unstable Numerical Boundary Conditions for Galerkin Approximations to Hyperbolic Systems," *Comp. and Maths. with Applics.*, **9,** 559-566(1983).

[37] J. Locker and P. M. Prenter, "Optimal $L^2$ and $L^\infty$ Error Estimates for Continuous and Discrete Least Squares Methods for Boundary Value Problems," *SIAM J. Numer. Anal.* **15,** 1151-1160(1978).

[38] L. B. Rall, **Computational Solution of Nonlinear Operator Equations,** Wiley, New York, 1969.

[39] R.D. Richtmeyer and K.W. Morton, **Difference Methods for Initial Value Problems,** Interscience, New York, 1967.

[40] B.G. Ryder and A.D. Hall, "The PFORT Verifier," Bell Laboratories Computer Science Technical Report #12, 1975.

[41] I.W. Sandberg and H. Shichman, "Numerical Integration of Systems of Stiff Nonlinear Differential Equations," *BSTJ* **47,** 511-528(1968).

[42] N.L. Schryer and L.R. Walker, "The Motion of $180^o$ Domain Walls in Uniform dc Magnetic Fields," *J. Appl. Physics* **45,** No. 12, 5406-5421(1974).

[43] N.L. Schryer, "An Extrapolation Step-Size and Order Monitor for use in Solving Differential Equations," Proceedings ACM National Meeting, San Diego, 1974.

[44] N.L. Schryer, "An Extrapolation Step-Size and Order Monitor for use in Solving Differential Equations," in preparation.

[45] N.L. Schryer, "A Tutorial on Galerkin's Method, using B-splines, for Solving Differential Equations," Bell Laboratories Computing Science Technical Report #52, 1976.

[46] M.H. Schultz, "The Galerkin Method for Non-Self-Adjoint Differential Equations," *J. Math. Anal. and Appl.* **28,** 647-651(1969).

[47] R.F. Sincovec and N.K. Madsen, "Software for Nonlinear Partial Differential Equations," *ACM*

*Trans. on Math. Software* **1,** 232-260(1975).

[48] R.F. Sincovec and N.K. Madsen, "PDEONE, Solutions of Systems of Partial Differential Equations," *ACM Trans. on Math. Software* **1,** 261-263(1975).

[49] H.J. Stetter, "Asymptotic Expansions for the Error of Discretization Algorithms for Non-Linear Functional Equations," *Numer. Math.* **7,** 18-31(1965).

[50] G. Strang and G. Fix, **An Analysis of the Finite Element Method,** Prentice-Hall, New York, 1973.

[51] C. M. Surko, A. L. Simons, D. H. Auston, J. A. Golovchenko and R. E. Slusher, "Calculation of the Dynamics of Surface Melting during Laser Annealing," *Appl. Phys. Lett.* **34,** (10), 635-637(1979).

[52] Lars Wahlbin, "Error Estimates for a Galerkin Method for a Class of Model Equations for Long Waves," *Numer. Math.* **23,** 289-303(1975).

[53] D.D. Warner, "A Partial Derivative Generator," Bell Laboratories Computing Science Technical Report #28, April, 1975.

[54] J.H. Wilkinson, **Rounding Errors in Algebraic Processes,** Prentice-Hall, New York, 1963.

[55] J.H. Wilkinson and C. Reinsch, **Handbook for Automatic Computation (II): Linear Algebra** Springer-Verlag, New York, 1971.

[56] L.O. Wilson and N.L. Schryer, "Flow Between a Stationary and a Rotating Disk with Suction," *J. Fluid Mech.,* **85,** 479-496(1978).

[57] L.O. Wilson, "Numerical Simulation of Gate Oxide Thinning in MOS Devices," *J. Electrochem. Soc.*, **129,** 831-837(1982).

# Appendix 1

## B-splines

The way in which the approximate numerical solution is to be represented is a very important decision - it affects the entire solution process. Specifically, we would like to choose a space of functions, from which to obtain the element closest to the solution. This space should have several nice properties, including being (1) easy to work with and (2) capable of approximating the solution accurately.

Such a representation exists - expansion in B-splines of order $k$ [3,4,5,16]. This is a method for representing functions by piecewise polynomials, that is, polynomials of degree $k-1$ or less over each subinterval of a mesh or grid. Here the integer $k$ is any number $k \geq 2$ the user desires. The piecewise polynomial representation is required to satisfy certain continuity restrictions at the end points of each mesh subinterval. Specifically, let $\pi = \{x_1, \cdots, x_N\}$, where $L = x_1 \leq x_2 \leq \cdots \leq x_N = R$, be a **grid** on the interval (L,R). Let $m_i$ be the **multiplicity** of $x_i$, or the number of times $x_i$ appears in the list $\pi$. The space of B-splines of **order** $k$ on the mesh $\pi$ is defined to be the collection of all functions $f$

(A1.1) that are polynomials of degree $< k$ on each interval $(x_i, x_{i+1})$ for $i = 1, \ldots, N-1$,

(A1.2) for which $d^{k-1-m_i} f(x_i) / dx^{k-1-m_i}$ exists and is continuous at each $x_i$, for $i = 1, \ldots, N$, when viewed as a function defined only on $[L,R]$, and

(A1.3) that have $f \equiv 0$ outside $[L,R]$.

The multiplicity $m_i$ of a point $x_i$ is restricted to be in the range $1 \leq m_i \leq k$. For $m_i = 1$ we have $d^{k-2} f / dx^{k-2}$ continuous at $x_i$. This is the most continuity that can be imposed at $x_i$ without making $f$ a polynomial of degree $k-1$ on $(x_{i-1}, x_{i+1})$. For $m_i = k$ the condition that $d^{-1} f / dx^{-1}$ be continuous is interpreted to mean that $f$ is continuous from the right (but not necessarily from the left) at $x = x_i$, for $x_i < R$, and continuous from the left if $x_i = R$. This means that B-splines are continuous at the end points of the mesh when viewed as functions defined only on $[L,R]$. This collection of functions is denoted by $B_{\pi,k}$. These $B_{\pi,k}$ spaces have nice approximation properties, as summed up by deBoor [3,4], in the case when $m_1 = k = m_N$:

**Theorem 1**

Let $f$ be any function with $f^{(0)}$ through $f^{(k)}$ continuous on $[L,R]$, where $f^{(j)}$ denotes the $j^{th}$ derivative of $f$. Let $h = \underset{i=1,\ldots,N-1}{Max} | x_{i+1} - x_i |$ be the largest mesh interval length. Then there is an element $g$ of $B_{\pi,k}$ so that

$$\| f^{(j)}(x) - g^{(j)}(x) \| \leq C(k,f)\ h^{k-j}$$

for $0 \leq j \leq k$, where $C(k,f)$ represents a constant that depends only upon $k$ and $f$, but not $h$.

That is, as $h \rightarrow 0$, the error in the best B-spline approximation to $f$ goes to zero like $h^k$; the error in its derivative behaves like $h^{k-1}$; etc.

Note that this theorem makes no assumption about the relative spacing of the mesh points of $\pi$ to get $h^k$ error. In many problems, the ability to grade the mesh with B-splines and still get $h^k$ error is a decided advantage.

In practice, $k$ is usually taken to be 4, 6, 8 or even 10, depending on what the function $f$ looks like and how much accuracy is desired. $k$ is usually, but not always, taken to be even due to the natural way in which such splines arise and their smoothing properties when used to approximate functions described by discrete data [4]. Typically, the more accuracy desired, the larger the value of $k$ should be. For example, if $k = 8$ and the mesh length $h$ is halved, then Theorem 1 indicates that the error should decrease by a factor of $2^8 = 256$. However, the work needed to solve a problem using `POST` is $O(Nk^3)$. Thus, a $k = 8$ solution will cost 8 times as much as a $k = 4$ solution for the *same* mesh. Hence, the optimal $k$ results from minimizing $O(N_k k^3)$, where $N_k$ is the number of mesh points needed to solve the problem to the desired accuracy using a $k$ order B-spline. This optimization is highly problem dependent.

A computationally convenient basis exists for the spaces $B_{\pi,k}$. The dimension of $B_{\pi,k}$ is $N-k$ and the basis consists of elements $B_i(x)$, $i=1,\ldots,N-k$. A complete description of the $B_i$ is given in [16] and [5]. Briefly, when the multiplicities of the first and last mesh points are both *k*, so that

$$x_1 = \cdots = x_k$$

and

$$x_{N-k+1} = \cdots = x_N,$$

the main properties of the $B_i(x)$ follow

(A1.4) Each $B_i$ is non-zero only on $[x_i, x_{i+k}]$ and is identically zero elsewhere, as well as at $x_1,\ldots,x_{i-1}$ and $x_{i+k+1},\ldots,x_N$, even if they are in $[x_i, x_{i+1}]$.

(A1.5) The sum $B_1(x)+\cdots+B_{N-k}(x)$ is identically one.

(A1.6) Each $B_i$ obeys $0 \le B_i(x) \le 1$ everywhere and possesses only one maximum.

The convergence result of Theorem 1 is independent of the multiplicities $m_i$ of the interior points $x_i$ ( $k < i \le N-k$ ) of the mesh. Usually, for smooth functions *f*, $m_i=1$ is taken for all these interior ( that is, strictly between L and R ) mesh points.

The end points of the mesh typically have multiplicity *k* since the function *f* usually has $f(L) \ne 0$ and $f(R) \ne 0$, and the elements of $B_{\pi,k}$ cannot be non-zero at *L* and *R*, unless $m_1 = k = m_N$ because of (A1.2) and (A1.3). Indeed, relations (A1.2)-(A1.5) show that the only $B_i$ that are not zero at *L* and *R* are $B_1$ and $B_{N-k}$, and these values are simply $B_1(L) = 1$ and $B_{N-k}(R) = 1$.

If the function *f* has a discontinuity in its $j^{th}$ derivative, but not its $(j-1)^{th}$, at $x_i$, then $m_i = k-j$ is chosen because this allows the elements of $B_{\pi,k}$ to have the same behavior. If a smaller multiplicity were chosen, the $j^{th}$ derivative of all the elements of $B_{\pi,k}$ would be continuous at $x_i$, and the best fit to *f* from $B_{\pi,k}$ would not be very good at $x_i$.

Another important property of B-splines is their numerical stability or *condition.* Since any B-spline *f* is of the form $f = \sum_{i=1}^{N-k} a_i B_i$ and each $B_i$ obeys $0 \le B_i \le 1$, we see that if $\|f\|$ is small compared with $\|\mathbf{a}\|$, then many significant digits are lost when computing *f* from $a_1, \cdots, a_{N-k}$ in floating-point arithmetic [54]. Specifically,

$$d \le Log_{10}(\|\mathbf{a}\| / \|\sum_{i=1}^{N-k} a_i B_i\|) \tag{A1.7}$$

decimal digits are lost, due to cancellation, in evaluating *f*. In [4] de Boor shows that

$$\|\sum_{i=1}^{N-k} a_i B_i\| \ge C_k \|\mathbf{a}\| \tag{A1.8}$$

where $C_k$ is a constant depending *only* upon *k*, and therefore

$$d \le Log_{10}(C_k^{-1}).$$

In particular, he shows for a uniform mesh, one where all the mesh intervals have the same length, that

$$C_k \approx 10^{-k/5} . \tag{A1.9}$$

Thus, when evaluating a B-spline defined on a uniform, or nearly uniform, mesh, we would expect to lose no more than about *k*/5 decimal digits. This is a very satisfactory result since it shows that, at least for uniform meshes, the conditioning of the B-spline basis is independent of the size of the mesh.

The interesting spline facts are, in summary,

$$B_i \ge 0$$

$$B_1 + \cdots + B_{N-k} \equiv 1$$

$$B_1 ( x_1 ) = 1 = B_{N-k} ( x_N )$$

$$\int_{x_1}^{x_N} B_i \, dx = \frac{x_{i+k} - x_i}{k}.$$

Also, if we let *i* be such that *x* is in $[x_i , x_{i+1})$, or $i = N-k$ if $x = x_N$, then

$$\sum_{j=1}^{N-k} a_j B_j (x) \equiv \sum_{j=i+1-k}^{i} a_j B_j (x)$$

If we let

$$f \equiv \sum_{j=1}^{N-k} a_j B_j (x)$$

then we have

$$f'(L) = \frac{k-1}{x_{k+1} - x_1} ( a_2 - a_1 )$$

and

$$f'(R) = \frac{k-1}{x_{nx} - x_{nx-k}} ( a_{nx-k} - a_{nx-k-1} )$$

# Appendix 2

## Extrapolation.

The problem treated in [8] and [29,30] is the numerical solution of the canonical form ODE initial value problem

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(t,\ \mathbf{x}) \qquad a \le t$$

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(t,\ \mathbf{x}) \qquad a \le t \tag{A2.1}$$

$$\mathbf{x}(a) = \mathbf{x}_a$$

where $\mathbf{f}(t,\ \mathbf{x})$ is some smooth vector-valued function of $t$ and $\mathbf{x}$. A brief outline of the ideas developed in those papers follows.

There are many basic differencing schemes for solving (A2.1), such as Gragg's modified mid-point rule [29,30], backwards-Euler methods [41] and Crank-Nicholson [23,39]. Most of these methods have the property [49] that if they take N time-steps $h = (t_1 - t_0)/N$ to go from $t_0$ to $t_1$ and result in an approximation to $\mathbf{x}(t_1)$ which we shall denote by $\mathbf{T}(h)$, then

$$\mathbf{T}(h) = \mathbf{x}(t_1) + \sum_{j=1}^{\infty} \mathbf{T}_j h^{j\gamma} \tag{A2.2}$$

where $\gamma$ is a positive constant depending on the basic difference scheme used, and the $\mathbf{T}_j$ are unknown constant vectors independent of $h$. For Gragg's modified mid-point rule or Crank-Nicholson $\gamma = 2$ and for backwards-Euler methods $\gamma = 1$.

Let a sequence of $h$'s be defined by

$$h_i = h_0/N_i, \qquad i = 1,\ 2,\ 3,\ \cdots . \tag{A2.3}$$

where $h_0 = t_1 - t_0$ and the $N_i$ form a monotone increasing sequence of positive integers. Bulirsch and Stoer showed in [7] that given an operator $\mathbf{T}(h)$ satisfying (A2.2), and such a sequence $h_i$, then the value at $h=0$ of the polynomial of degree $m$ that interpolates $\mathbf{T}(h_i)$ for $i=0,\ \cdots,m$, is given by $\mathbf{T}_m^0$ determined from the recursion

$$\mathbf{T}_0^i = \mathbf{T}(h_i) \qquad \text{for} \quad 0 \le i \le m$$

$$\mathbf{T}_0^i = \mathbf{T}(h_i) \qquad \text{for} \quad 0 \le i \le m \tag{A2.4}$$

$$\mathbf{T}_k^i = \mathbf{T}_{k-1}^{i+1} + (\mathbf{T}_{k-1}^{i+1} - \mathbf{T}_{k-1}^i)/\left\{(h_i/h_{i+k})^\gamma - 1\right\}$$

for $0 \le i \le m-k$ and $1 \le k \le m$. If the $T_k^i$ are organized into a **tableau** of the form

$$\begin{array}{llllllll}
\mathbf{T}(h_0) & = & \mathbf{T}_0^0 & & & & & \\
 & & & \mathbf{T}_1^0 & & & & \\
\mathbf{T}(h_1) & = & \mathbf{T}_0^1 & & \mathbf{T}_2^0 & & & \\
 & & & \mathbf{T}_1^1 & & \mathbf{T}_3^0 & & \\
\mathbf{T}(h_2) & = & \mathbf{T}_0^2 & & \mathbf{T}_2^1 & & \mathbf{T}_4^0 & \\
 & & & \mathbf{T}_1^2 & & \mathbf{T}_3^1 & & \mathbf{T}_5^0 \\
\mathbf{T}(h_3) & = & \mathbf{T}_0^3 & & \mathbf{T}_2^2 & & \mathbf{T}_4^1 & \\
 & & & \mathbf{T}_1^3 & & \mathbf{T}_3^2 & & \\
\mathbf{T}(h_4) & = & \mathbf{T}_0^4 & & \mathbf{T}_2^3 & & & \\
 & & & \mathbf{T}_1^4 & & & & \\
\mathbf{T}(h_5) & = & \mathbf{T}_0^5 & & & & &
\end{array}$$

then (A2.4) expresses each element of the $k^{th}$ column ($k>0$) in terms of its two neighbors in column $k-1$. A similar result is established for interpolation by rational functions [7].

It is also possible to estimate the error in each element of the tableau [9]. For sufficiently small $h_0$,

[9] shows that

$$|\mathbf{T}_k^i - \mathbf{T}| \approx \left[1 + \frac{1}{(h_i/h_{i+k+1})^\gamma - 1}\right] |\mathbf{T}_k^{i+1} - \mathbf{T}_k^i| \qquad \text{(A2.5)}$$

and we can estimate the error $\varepsilon_k^i = |\mathbf{T}_k^i - \mathbf{T}|$in $\mathbf{T}_k^i$. We also know from [9] that for sufficiently small $h_0$,

$$\varepsilon_k^i = h_0^\beta \mathbf{d}_k (h_i \cdots h_{i+k})^\gamma \qquad \text{(A2.6)}$$

where the $\mathbf{d}_k$ are constant vectors, $\gamma$ is the order of the basic process being extrapolated and $\beta$ is a positive constant. When extrapolating Gragg's modified mid-point rule or Crank-Nicholson, $\gamma = 2$ and $\beta = 1$. When extrapolating a Backwards-Euler time differencing process, $\gamma = 1 = \beta$. Thus, we can both estimate the accuracy of each element in the tableau and tell how rapidly each column in the tableau is converging.

In (A2.4) $m$ is called the **level** of extrapolation, while from (A2.6) we see that the **order** in column $k$ is $(k+1)\gamma$. Thus, by extrapolating the results of a basic ordinary differential equation solver, a process of arbitrarily high order can be obtained. The value $h_0 = t_1 - t_0$ is referred to as the **time-step** while the $h_i$ are called **sub-steps.** Extrapolation approximates the $\mathbf{x}(t_1)$ values accurately, but does not accurately approximate $\mathbf{x}(t+nh_i)$ for $0<n<N$.

# Appendix 3

## Wish-List

This section describes several improvements that could be made in `POST`. The improvements range from better human engineering (easier use) to making the algorithm more efficient and extending it to solve more general problems.

### Better Nonlinear Equation Solver

The first major improvement that will be made is to use a better nonlinear equation solver. The current version uses a standard quasi-Newton scheme to solve the nonlinear equations. That procedure is quite effective, but is not particularly robust. It could be made much more so by the addition of a damping process, such as described in [2].

### Numerical Jacobians

Another possible improvement would be the automatic numerical computation of the Jacobian matrix from the values of **a** and **f**. This is not a high-priority item for three reasons. The first is that it is usually an easy matter for the user to compute, by hand, the necessary partial derivatives. The second reason is that the partial derivative generator `PDGEN` [53] may be used to automatically compute the Jacobian when it *is* too difficult to do by hand. The third reason is that numerical differentiation is a "black-art" rather than a science, and its use could compromise the robustness of `POST`. Anyone feeling that numerical Jacobians are a necessity should contact me, but be ready for an argument.

### More General bc's

Problems where the **bc**'s involve giving both $\mathbf{u}$ and $\mathbf{u}_x$ at either $L$ or $R$ cannot be handled currently, except by either making $\mathbf{u}_x(t, x)$ a new **pde** variable (which increases the cost a lot), or by making $\mathbf{u}(t, L)$ (or $\mathbf{u}(t, R)$) an **ode** variable (which increases the cost a little). Such **bc**'s may arise occasionally and it would be wise to handle them cleanly and efficiently. However, most problems of this type are ill-posed [39], so implementation of this feature is not a high-priority item.

### Counting bcs in Hyperbolic Equations

As noted in section 4, hyperbolic systems are difficult to deal with in that they don't always require one **bc** per component. They may require more or less, depending on the structure of the **pde** near the boundary, see [36]. It would be good to include this analysis in `POST`.

### Other Spatial Discretization Techniques

B-splines are used in the current implementation because of their excellent approximation properties (see Appendix 1) and because there is a *local* basis for them that yields *banded* Jacobians. This choice is made for general robustness, reliability and efficiency. However, for particular problems there may be other (non-local) approximation spaces that, although they yield dense matrices, are efficient because very few basis functions are needed to accurately approximate the solution. It is conceivable that a package allowing the user to specify the basis functions, and the way to compute the Galerkin integrals, could be created and even made efficient. Such a project awaits sufficient user demand.

### Sparse pde Systems

There are situations where the **pde**'s themselves are sparse, that is, $a_i$ and $f_i$ only depend on $u_j$ for some $j$, see [21]. Advantage can be taken of this structure to decrease the amount of storage, and run-time, needed by `POST`. Work on this is currently underway.

**More Structure in the pde**

An equation of the form [52]

$$P(u)_t = Q(u_x)_x$$

is of the form (2.1)-(2.2) since it is equivalent to

$$P'(u)u_t = Q(u_x)_x \ .$$

However, it may be awkward to find $P'$ and $P''$ for use in `POST`. The form of the **pde** could be taken to be

$$a + b_x + c_t + d_{tx} = 0$$

to accommodate such problems. But, the calling sequence to the new "`AF`" would be horrendous, having 14 more arguments! Such a grotesquerie will be produced only under great pressure.

**Collocation Methods**

Collocation-Least-Squares methods [10,19], using B-splines, could be used to solve a **pde** system of the form

$$g(t,\ x,\ \mathbf{u},\ \mathbf{u}_x, \mathbf{u}_t,\ \mathbf{u}_{xt},\ \mathbf{u}_{xx},\ \mathbf{u}_{xxt},\ \mathbf{v},\ \mathbf{v}_t) = 0.$$

Such techniques approximately force the **pde** to hold at a finite number of points in each B-spline mesh interval. There are several advantages to such techniques. First, they would permit solution of more general problems than that allowed by the divergence form (2.1). The code would also be simpler and easier to use, due to the reduced number of arguments. Finally, for a given B-spline mesh, collocation methods are about twice as fast as Galerkin's method.

However, there are two problems with collocation-least-squares methods. The first is that they do not force the **a** term in (2.1) to be continuous. Since **a** corresponds, physically, to a flux, physical scientists are often and rudely surprised when the numerical solution gives them the "wrong" answer to their problem. This is unacceptable. Collocation-least-squares only solves the mathematical problem as posed. Galerkin "knows" more physics because of its derivation, see [45]. If collocation-least-squares methods were used, each user with interface conditions would have to alter the *source* of `POST` for each such problem to add that condition to the discrete equations.

The second problem for collocation and least-squares methods is their accuracy for problems with steep fronts. Each of the collocation, Galerkin and least-squares methods converge at a rate $O\ (\ h^k\ )$, where $h$ is the mesh spacing. However, the constants $O$ differ substantially in their dependence on the solution. For example, for Galerkin the $L^\infty$ error is

$$O\ (\ \|\ u\ \|_{H^k}\ )\ h^k$$

where $H^k$ is a Sobolev space of order $k$ [20]. The error for collocation is

$$O\ (\ \|\ u\ \|_{H^{2(k-1)}}\ )\ h^k,$$

see [6]. The error for least-squares is

$$O\ (\ \|\ u\ \|_{H^{k+2}}\ )\ h^k,$$

see [37]. While the above error estimates are only theoretical upper-bounds, they indicate that the error for collocation-least-squares techniques may depend on higher order derivatives of the solution than does Galerkin.

And, indeed, for some second order problems where the spatial derivatives $u^{(i)}$ of the solution grow like $\alpha^i$ for some constant $\alpha$, the error for collocation-least-squares is a factor of $\alpha$ greater than that for Galerkin. This happens because the divergence-form allows one level of spatial differentiation to be done analytically, using integration by parts (see section 7). Thus, for problems where the solution is "kinky", collocation may be much more inaccurate than Galerkin for the same B-spline mesh.

This would seriously compromise the robustness of `POST`. Should demand arise for such a collocation package (should someone find an interesting **pde** that is not in divergence-form), it could be produced easily from the code for the existing `POST` package.

## Appendix 4

## Examples - Programs

The program examples given below are intended to both illustrate the use of POST and provide prototypes for a prospective user. Anyone contemplating using POST would be well advised to pick an example program that invokes those capabilities of POST the intended problem will require, and type it in (or obtain a copy of the example code from the author). After running the example and confirming the correctness of the program, the AF and BC subroutines specifying the **pde**-**bc** may simply be altered to solve the user's problem. This progression makes it much more likely that the user will easily produce a correct program unit for the problem at hand.

The examples are taken in sequence from section 4 where they were formulated and analyzed. This section is only intended to show how to program the solution of the formulations given in section 4. The user must have read section 5 describing the POST software before proceeding in this appendix, otherwise the reading will be dark and obscure.

Before invoking POST the user must

- Make a B-spline mesh.
- Make initial conditions for the B-spline coefficients **U** in (2.3).
- Write subroutines
  - AF - to evaluate **a** and **f** in (2.4).
  - BC - to evaluate **b** in (2.5).
  - DEE - to evaluate **d** in (2.6). If $n_v = 0$, there are no **ode** variables, and POSTD may be used in place of a dummy DEE written by the user.
  - HANDLE - to output (print) the solution results.

The subroutine writing will be amply illustrated later in this section. The creation of a mesh and initial conditions ( **ic**s ) for **u** are now briefly described.

### Mesh Making

The PORT Library [25] has several B-spline mesh generation subroutines available. There is UMB for generating uniform B-spline meshes on a given interval. For creating B-spline meshes that are the union of uniform meshes over basic contiguous intervals there are LUMB and PUMB. LUMB uses the same number of mesh points in each basic interval and PUMB allows that number to vary by interval. Example 5 below illustrates the use of LUMB.

If you find yourself using more than 50 to 100 mesh points, think carefully about using LUMB to make the mesh more carefully tailored to the solution, or use mesh mapping. For example, example 7 below could be solved on a uniform mesh, but it would require roughly **6 million** mesh points to do it! Using mesh mapping and LUMB or PUMB it only requires a dozen mesh points. Remember that the run-time and memory requirements of POST are proportional to the number of mesh points used. A little thought given to mesh construction can save a lot of computer run-time and memory.

### Initial Conditions for u.

There are **ic**s for **u** and these must be converted into **ic**s for **U**, the B-spline coefficients for **u**, see (2.3). The simplest case is when the **ic**s for **u** are a constant. By simply setting all the spline coefficients to that constant, the spline **is** the constant and we are done.

If the **ic**s for $u$ are not constant, then there are two PORT Library routines available. First, L2SFF fits a given function, like $e^x + \sin(x)$, on a given mesh. This is done in example 5 below. Second, DL2SF fits discrete data pairs, $(x_1, y_1)$, $\cdots$, $(x_N, y_N)$, on a given mesh. DL2SF is often used when experimental (discrete) data is available for **ic**s.

All examples reported here were run on a VAX 11/750, equipped with a floating-point accelerator, using single-precision arithmetic, under the UNIX® operating system, Research Eighth Edition.

**Example 1 - A Simple Heat Equation.**

As a simple example of the use of `POST`, consider solving the scalar heat equation

$$a = -u_x \qquad \text{and} \qquad f = g(t,x) - u_t \qquad \text{for} \quad x \in (0,1) \tag{A4.1}$$

where the source term $g(t,x)$ is chosen ( $g = (x - t^2) e^{xt}$ ) so that the solution is a known function, $u(t,x) = e^{xt}$. The boundary conditions are

$$\begin{aligned} b_L &= u(t,0) - 1 \\ b_R &= u(t,1) - e^t . \end{aligned} \tag{A4.2}$$

The initial conditions are taken to be

$$u(0,x) = 1 . \tag{A4.3}$$

The following program unit, written in Ratfor [35], solves (A4.1)-(A4.3) using `POST`, with a cubic B-spline ($k = 4$) over a spatial mesh consisting of 4 equally spaced, distinct points on $(0,1)$, with the time evolution carried out to $10^{-2}$ absolute accuracy.

Ratfor stands for "Rational Fortran" and provides modern control structures over Fortran. Specifically, it allows multiple statements on a line to be separated by semicolons ( ; ), a # begins a comment ( hence a comment and an executable statement can be together on the same line ), and curly braces ( { and } ) are used to delimit blocks of code. This last feature removes the need for statement labels on Do loops, for example. There is "syntactic sugar" in the form of "<" for ".lt.", "&" for ".and." and " | " for ".or.". Finally, if a line ends in one of ", + - * / ( = < > & |", the next line is considered a continuation of it.

The main program uses several PORT [25] library subprograms.

- `ISTKIN` initializes the PORT Library stack. In this case, it initializes the stack to 1000 double precision items, consistent with the declaration for `Ds` in the double precision alias of the stack in the common region `CSTAK`.
- `UMB` makes uniformly spaced B-spline meshes.
- `SETR` sets an array to a given Real constant. `SETR` provides the constant **ic**'s (A4.3) via the B-spline coefficients (2.3), since if all the B-spline coefficients are equal to a constant, then the B-spline itself is identically equal to that constant (see Appendix 1).
- `WRAPUP` checks that a run has terminated without errors and prints out the stack space used.

At the end of each time-step the solution is printed out at $x = \frac{1}{3}, \frac{1}{2}$ and $\frac{2}{3}$. The "Return-End" subroutine `POSTD` is used as a dummy place-holder for a subroutine needed to deal with coupled **ode**'s. Note that the **ode** place-holder `V` need not be initialized, since it is never interrogated by `POST`. The main program is

```
Real tstop,V(1),dt,mesh(100),U(100)
Real errpar(2)
Integer k,nmesh,ndx,Nu,Nv
External AF,BC, POSTD,HANDLE

Common / CSTAK / Ds(1000); Double Precision Ds    # The PORT Stack.

Call ISTKIN(1000,4)    # Initialize the PORT Library stack length.

Nu = 1; Nv = 0

errpar(1) = 0; errpar(2) = 1e-2    # Absolute error.

tstop = 1; dt = 1e-2

k = 4; ndx = 4    # ndx uniform mesh points on (0,1).
Call UMB(0e0,1e0,ndx,k,mesh,nmesh)

Call SETR(nmesh-k,1e0,U)    # Initial conditions for U.

Call POST (U,Nu,k,mesh,nmesh,V,Nv,
           0e0,tstop,dt,
           AF,BC, POSTD,
           errpar,
           HANDLE)

Call WRAPUP    # Check for errors and stack usage statistics.

Stop

End
```

The dimension statements for the various arguments of the AF, BC and HANDLE subroutines given below are for $n_v = 0$, and thus will not be repeated in subsequent **pde-bc** examples.

Note that since the arrays A, ... , FVt are set to zero by POST before it calls AF, only the active **a** and **f** terms and their derivatives need be computed in AF. The subroutine AF for specifying the **pde** (A4.1) is

```
Subroutine AF(t,x,nx,U,Ux,Ut,Utx,Nu,V,Vt,Nv,
              A,AU,AUx,AUt,AUtx,AV,AVt,
              f,fU,fUx,fUt,fUtx,fV,fVt)

Real t,x(nx),U(nx,Nu),Ux(nx,Nu),Ut(nx,Nu),Utx(nx,Nu),V(1),Vt(1),
     A(nx,Nu),AU (nx,Nu,Nu),AUx (nx,Nu,Nu),
              AUt(nx,Nu,Nu),AUtx(nx,Nu,Nu), AV(1),AVt(1),
     f(nx,Nu),fU (nx,Nu,Nu),fUx (nx,Nu,Nu),
              fUt(nx,Nu,Nu),fUtx(nx,Nu,Nu), fV(1),fVt(1)
Integer Nu,Nv,nx

Integer i

Do i = 1, nx
  {
  A(i,1) = -Ux(i,1); AUx(i,1,1) = -1
  f(i,1) = -Ut(i,1)+(x(i)-t**2)*Exp(x(i)*t)
  fUt(i,1,1) = -1
  }

Return

End
```

Note that since the arrays `B`, ... , `BVt` are set to zero by `POST` before it calls `BC`, only the active **b** terms and their derivatives need be computed in `BC`. The subroutine `BC` for specifying the **bc**'s (A4.2) is

```
Subroutine BC(t,L,R,U,Ux,Ut,Utx,Nu,V,Vt,Nv,
              B,BU,BUx,BUt,BUtx,BV,BVt)

Real t,L,R,U(Nu,2),Ux(Nu,2),Ut(Nu,2),Utx(Nu,2),V(1),Vt(1),
     B(Nu,2),BU (Nu,Nu,2),BUx (Nu,Nu,2),
             BUt(Nu,Nu,2),BUtx(Nu,Nu,2), BV(1),BVt(1)
Integer Nu,Nv

B(1,1) = U(1,1)-1
B(1,2) = U(1,2)-Exp(t)

BU(1,1,1) = 1
BU(1,1,2) = 1

Return

End
```

The following output subroutine simply evaluates and prints $u(t,x)$, for $x = \frac{1}{3}, \frac{1}{2}$, and $\frac{2}{3}$, at the end of each successful time-step. The dimension statement for the various arguments is for arbitrary input and thus will not be repeated in subsequent examples.

Two PORT Library routines are used

- `I1MACH` determines the standard output unit number, `I1MACH`(2).
- `SPLNE` evaluates a spline, given the mesh and the coefficients.

```
Subroutine HANDLE(t0,U0,V0,t,U,V,nu,nxmk,nv,k,X,nx,dt,tstop)

Real t0,U0(nxmk,nu),V0(nv),t,U(nxmk,nu),V(nv),X(nx),dt,tstop

Real xe(3),Ue(3)

If ( t0 == t ) { Return }

xe(1) = 1e0/3e0; xe(2) = 0.5e0; xe(3) = 2e0/3e0

Call SPLNE(k,X,nx,U,xe,3,Ue)    # Evaluate spline at xe, results in Ue.

Write(I1mach(2),9000) t,(Ue(i),i=1,3)
9000 Format(" U(x,",1p1e9.2," ) =",1p3e10.2)

Return

End
```

The output of the above program unit is

```
U(x, 1.00e-02 ) =  1.00e+00  1.01e+00  1.01e+00
U(x, 2.28e-01 ) =  1.08e+00  1.12e+00  1.17e+00
U(x, 6.97e-01 ) =  1.26e+00  1.42e+00  1.59e+00
U(x, 1.00e+00 ) =  1.40e+00  1.65e+00  1.95e+00
used      618 /     2000 of the stack allowed.
```

A skeptic might observe that it is difficult to determine that the above output is in fact accurate to $10^{-2}$. Well, since the exact solution of the problem is known, the program may also check the accuracy of the numerical solution using the PORT [25] Library subroutine `EESFF` to estimate the error $\|u - \hat{u}\|$ in the computed solution $\hat{u}$. The procedure `UOFX` below evaluates the exact solution at any position in time and space. The procedure `EESFF` computes the difference between a function specified by a procedure like `UOFX` and a spline. By changing the body of `HANDLE` to read

```
 Common / TIME / tt; Real tt

 Real eU, EESFF
 Integer I1MACH
 External UOFX

 If ( t0 == t ) { Return }

 tt = t    # UOFX needs time.
 eU = EESFF(k,x,nx,U,UOFX)

 iwunit = I1MACH(2)
 Write(iwunit,9000) t,eU
9000 Format(" Error in U(x,",1p1e10.2," ) =",1p4e10.2)
```

where the following subroutine computes the exact solution $u$ of (A4.1)-(A4.3)

```
Subroutine UOFX(x,nx,U,W)

Real x(nx),U(nx),W(nx)
Integer nx

Common / TIME / t; Real t

Integer i

Do i = 1, nx  { U(i) = Exp(x(i)*t) }

Return

End
```

we may obtain the output

```
Error in U(x,  1.00e-02 ) =  1.47e-05
Error in U(x,  2.28e-01 ) =  1.89e-03
Error in U(x,  6.97e-01 ) =  4.92e-04
Error in U(x,  1.00e+00 ) =  4.91e-03
used      618 /     2000 of the stack allowed.
```

and we can see that (A4.1)-(A4.3) has indeed been solved accurate to $10^{-2}$. The run-time for the above example was 7.8 seconds.

**Example 2 - A Coupled System of pdes.**

The solution of a coupled system of **pde**'s is now illustrated. The **pde** is

$$a_1 = -u_{2x} \quad \text{and} \quad f_1 = -u_{2t} + (x - t^2)\, e^{xt}$$
$$\text{for } x \,\varepsilon\, (0,1) \qquad \text{(A4.4)}$$
$$a_2 = u_1 \quad \text{and} \quad f_2 = u_2$$

and the **bc**'s are

$$b_L = u_1(t,0) - 1$$
$$\text{(A4.5)}$$
$$b_R = u_1(t,1) - e^t$$

The initial conditions are

$$u_1(0,x) = 1$$
$$\text{(A4.6)}$$
$$u_2(0,x) = 0.$$

The following program solves (A4.4)-(A4.6) using `POST`, with a cubic B-spline, over a spatial mesh consisting of 4 equally spaced, distinct points on $(0,1)$, with the time-evolution carried out to $10^{-2}$ absolute accuracy. The error at each time-step is printed out to confirm the accuracy of the computed solution. The main program is

```
Real tstop,V(1),dt,mesh(100),U(200)
Real errpar(2)
Integer k,nmesh,ndx,Nu,Nv
External AF,BC, POSTD,HANDLE

Common / CSTAK / Ds(1100); Double Precision Ds    # The PORT Stack.

Call ISTKIN(1100,4)    # Initialize the PORT Library stack length.

Nu = 2; Nv = 0

errpar(1) = 0; errpar(2) = 1e-2    # Absolute error.

tstop = 1; dt = 1e-2

k = 4; ndx = 4    # ndx uniform mesh points on (0,1).
Call UMB(0e0,1e0,ndx,k,mesh,nmesh)

Call SETR(nmesh-k,1e0,U)    # Initial conditions for U1.
Call SETR(nmesh-k,0e0,U(nmesh-k+1))    # Initial conditions for U2.

Call POST (U,Nu,k,mesh,nmesh,V,Nv,
           0e0,tstop,dt,
           AF,BC, POSTD,
           errpar,
           HANDLE)

Call WRAPUP    # Check for errors and stack usage statistics.

Stop

End
```

The only change in the subroutine AF of the last example is the code for specifying the **pde**

```
Do i = 1, nx
  {
  A(i,1) = -U(i,2); AU(i,1,2) = -1
  f(i,1) = -Ut(i,1)+(x(i)-t**2)*Exp(x(i)*t)
  fUt(i,1,1) = -1

  A(i,2) = U(i,1); AU(i,2,1) = 1
  f(i,2) = U(i,2); fU(i,2,2) = 1
  }
```

The only change in the subroutine BC of the previous example is the code for specifying the **bc**s

```
B(1,1) = U(1,1)-1
B(1,2) = U(1,2)-Exp(t)

BU(1,1,1) = 1
BU(1,1,2) = 1
```

The body of the HANDLE subroutine simply checks the accuracy of the computed solution, using EESFF,

```
Common / TIME / tt; Real tt

Real eU(2), EESFF
Integer I1MACH
External U1OFX,U2OFX

If ( t0 == t ) { Return }

tt = t    # U1OFX and U2OFX need time.
eU(1) = EESFF(k,x,nx,U,U1OFX)
eU(2) = EESFF(k,x,nx,U(1,2),U2OFX)

iwunit = I1MACH(2)
Write(iwunit,9000) t,eU
9000 Format(" Error in U(x,",1p1e10.2," ) =",1p4e10.2)
```

where the following subroutine computes the exact value of the first component of the solution, $u_1$,

```
Subroutine U1OFX(x,nx,U,W)

Real x(nx),U(nx),W(nx)
Integer nx

Common / TIME / t; Real t

Integer i

Do i = 1, nx  { U(i) = Exp(x(i)*t) }

Return

End
```

and the following subroutine computes the exact value of the second component of the solution, $u_2$,

```
Subroutine U2OFX(x,nx,U,W)

Real x(nx),U(nx),W(nx)
Integer nx

Common / TIME / t; Real t

Integer i

Do i = 1, nx  { U(i) = t*Exp(x(i)*t) }

Return

End
```

The output of the above program unit is

```
Error in U(x,  1.00e-02 ) =  1.36e-05  2.28e-04
Error in U(x,  6.98e-02 ) =  2.87e-04  2.59e-03
Error in U(x,  1.83e-01 ) =  1.43e-04  4.95e-04
Error in U(x,  5.44e-01 ) =  3.53e-04  1.23e-03
Error in U(x,  1.00e+00 ) =  6.20e-04  2.83e-03
used     1081 /     2200 of the stack allowed.
```

The run-time for the above example was 26.0 seconds.

One reason for treating (A4.1)-(A4.3) as the system (A4.4)-(A4.6) is that the spatial derivative $u_2$ of $u = u_1$ is obtained accurate to $O(h^k)$, instead of $O(h^{k-1})$. Indeed, for this example, the error in $u_x$, as computed by differentiating the solution of (A4.1)-(A4.3), was `3.81e-02` and that computed from (A4.4)-(A4.6) was `2.83e-03`

**Example 3 - Simple pde-ode Coupling.**

The first example of the use of `POST` to solve a **pde**-**bc**-**ode** combination is contrived to be simple, but illustrative. Consider the **pde**

$$a = -u_x \qquad \text{and} \qquad f = v(t) + g(t,x) - u_t \qquad \text{for} \quad x \,\varepsilon\, (0,1) \tag{A4.7}$$

with the coupled **ode**

$$d = v_t\,(t) - u(t, \frac{1}{2}) \;\equiv\; v_t\,(t) \;-\sum_{j=i+1-k}^{i} U_j(t)\; B_j(\frac{1}{2}), \tag{A4.8}$$

where $i$ is chosen so that $x_i \le \frac{1}{2} < x_{i+1}$, and **x** is the spatial mesh. We choose $g(t,x)$ so that the solution is known, say $u(t,x) = \cos(xt)$ and $v(t) = 2\sin(t/2)$. The **bc**'s are

$$\begin{aligned} b_L &= u(t,0) - 1 \\ b_R &= u(t,1) - cos(t) \end{aligned} \tag{A4.9}$$

with initial conditions

$$\begin{aligned} u(0,x) &= 1 \\ v(0) &= 0\,. \end{aligned} \tag{A4.10}$$

The following program solves (A4.7)-(A4.10) using `POST`, with a cubic B-spline ($k = 4$) over a spatial mesh on $(0,1)$ consisting of 4 equally spaced, distinct points, with the time evolution carried out to roughly $10^{-2}$ relative accuracy. The error at each time-step is printed out to confirm the accuracy of the numerical solution. The main program is

```
Real tstop,V(1),dt,mesh(100),U(100)
Real errpar(2)
Integer k,nmesh,ndx,Nu,Nv
External AF,BC,DEE,HANDLE

Common / CSTAK / Ds(2000); Double Precision Ds    # The PORT Stack.

Call ISTKIN(2000,4)    # Initialize the PORT Library stack length.

Nu = 1; Nv = 1

errpar(1) = 1e-2; errpar(2) = 1e-6    # Essentially relative error.

tstop = 1; dt = 1e-6

k = 4; ndx = 4
Call UMB(0e0,1e0,ndx,k,mesh,nmesh)    # ndx uniform mesh points on (0,1).

Call SETR(nmesh-k,1e0,U)    # Initial conditions for U.
V(1) = 0    # Initial value for V.

Call POST (U,Nu,k,mesh,nmesh,V,Nv,
           0e0,tstop,dt,
           AF,BC,DEE,
           errpar,
           HANDLE)

Call WRAPUP    # Check for errors and stack usage statistics.

Stop

End
```

The AF subroutine for specifying the **pde** (A4.7) is

```
Subroutine AF(t,x,nx,U,Ux,Ut,Utx,Nu,V,Vt,Nv,
              A,AU,AUx,AUt,AUtx,AV,AVt,
              f,fU,fUx,fUt,fUtx,fV,fVt)

Real t,x(nx),U(nx,Nu),Ux(nx,Nu),Ut(nx,Nu),Utx(nx,Nu),V(Nv),Vt(Nv),
     A(nx,Nu),AU (nx,Nu,Nu),AUx (nx,Nu,Nu),
              AUt(nx,Nu,Nu),AUtx(nx,Nu,Nu),
              AV (nx,Nu,Nv),AVt (nx,Nu,Nv),
     f(nx,Nu),fU (nx,Nu,Nu),fUx (nx,Nu,Nu),
              fUt(nx,Nu,Nu),fUtx(nx,Nu,Nu),
              fV (nx,Nu,Nv),fVt (nx,Nu,Nv)
Integer Nu,Nv,nx

Integer i

Do i = 1, nx
  {
  A(i,1) = -Ux(i,1); AUx(i,1,1) = -1
  f(i,1) = V(1)-Ut(i,1)-
            x(i)*Sin(x(i)*t)+t**2*Cos(x(i)*t)-2*Sin(t/2)
  fUt(i,1,1) = -1
  fV(i,1,1) = 1
  }

Return

End
```

The BC subroutine for specifying the **bc**s (A4.9) is

```
Subroutine BC(t,L,R,U,Ux,Ut,Utx,Nu,V,Vt,Nv,
              B,BU,BUx,BUt,BUtx,BV,BVt)

Real t,L,R,U(Nu,2),Ux(Nu,2),Ut(Nu,2),Utx(Nu,2),V(Nv),Vt(Nv),
     B(Nu,2),BU (Nu,Nu,2),BUx (Nu,Nu,2),
             BUt(Nu,Nu,2),BUtx(Nu,Nu,2),
             BV (Nu,Nv,2),BVt (Nu,Nv,2)
Integer Nu,Nv

B(1,1) = U(1,1)-1
B(1,2) = U(1,2)-Cos(t)

BU(1,1,1) = 1
BU(1,1,2) = 1

Return

End
```

The dimension statements for the various arguments of the AF and BC subroutines given above are for the case of $n_v > 0$, and thus will not be repeated in subsequent **pde-bc-ode** examples.

The following subroutine DEE specifies the **d** of (A4.8). The dimension statement for the various arguments is for arbitrary input, and thus will not be repeated in subsequent examples.

Three PORT Library routines are used

- `INTRVR` locates 0.5 in the mesh ( that is, determines the *i* in (A4.8) )
- `BSPLN` evaluates the basis B-splines at 0.5, see Appendix 1. The value of $u$ ( 0.5, $t$ ) is the sum of those basis values times the appropriate spline coefficients.
- `SETERR` sets an error state if $k$ requires more space than reserved for the basis splines ( `Basis` ).

Note that since the arrays `D`, ... , `DVt` are set to zero by `POST` before it calls `DEE`, only the active **d** terms and their derivatives need be computed in `DEE`.

```
Subroutine DEE(t,k,x,nx,
               U,Ut,Nu,nxmk,V,Vt,Nv,
               D,DU,DUt,DV,DVt)

Real t,x(nx),U(nxmk,Nu),Ut(nxmk,Nu),V(Nv),Vt(Nv),
     D(Nv),DU(Nv,nxmk,Nu),DUt(Nv,nxmk,Nu),
           DV(Nv,Nv),DVt(Nv,Nv)
Integer k,nx,Nu,nxmk,Nv

Real Xi(1),Basis(10)
Integer ileft,INTRVR,i

Xi(1) = 0.5e0; ileft = INTRVR(nx,x,Xi(1))    # Find 0.5 in mesh.

If ( k > 10 )
  { Call SETERR("DEE - k .gt. 10, need more space in Basis",41,1,2) }

Call BSPLN(k,x,nx,Xi,1,ileft,Basis)    # B-spline basis at Xi(1).

D(1) = Vt(1); DVt(1,1) = 1
Do i = 1, k    # vt(1) - u(0.5,t) = 0.
  {
  D(1) = D(1) - U(ileft+i-k,1)*Basis(i)
  DU(1,ileft+i-k,1) = DU(1,ileft+i-k,1) - Basis(i)
  }

Return

End
```

The only change in the `HANDLE` subroutine of example 1 is in the code for computing and printing the error in the computed solution.

```
         Common / TIME / tt; Real tt

         Real eU,eV, EESFF
         Integer I1MACH
         External UOFX

         If ( t0 == t ) { Return }

         tt = t    # UOFX needs time.
         eU = EESFF(k,x,nx,U,UOFX); eV = Abs(V(1)-2*Sin(t/2))

         iwunit = I1MACH(2)
         Write(iwunit,9000) t,eU,eV
        9000 Format(" Error in U(x,",1p1e10.2," ) =",1p1e10.2,"   V =",1p4e10.2)
```

The only change in the subroutine `UOFX`, for computing $u$, of example 1 is the code for computing $u$

```
         Do i = 1, nx  { U(i) = Cos(x(i)*t) }
```

The output from this program is

```
        Error in U(x,  1.00e-06 ) =  2.38e-07   V =  4.17e-20
        Error in U(x,  2.62e-04 ) =  2.38e-07   V =  7.52e-13
        Error in U(x,  6.86e-02 ) =  3.35e-04   V =  2.54e-05
        Error in U(x,  3.45e-01 ) =  1.89e-04   V =  5.44e-05
        Error in U(x,  1.00e+00 ) =  1.28e-04   V =  1.76e-03
        used      691 /     4000 of the stack allowed.
```

and we see that (A4.7)-(A4.10) has indeed been solved accurately. The run-time for the above example was 12.7 seconds.

**Example 4 - Nonlinear Heat Equation with Periodic Boundary Conditions.**

The second example is the use of `POST` to solve a nonlinear heat equation subject to periodic boundary conditions. Consider the **pde**

$$a = -u_x \qquad \text{and} \qquad f = -u^3 + g(t,x) - u_t \qquad \text{for} \quad x\,\varepsilon\,(-\pi, +\pi) \tag{A4.11}$$

where $g(t,x)$ is chosen to make the solution $u$ a known function, say $u(t,x) = \cos(x)\sin(t)$. The **bc**'s are

$$\begin{aligned} b_L &= u_x\,(t, -\pi\,) - v(t) \\ b_R &= u_x\,(t, +\pi\,) - v(t). \end{aligned} \tag{A4.12}$$

The **ode** is

$$d = \mathbf{u} - \mathbf{u}. \tag{A4.13}$$

where *nmesh* is the length of the B-spline mesh being used.

The following program solves the **pde-bc-ode** combination (A4.11)-(A4.12)-(A4.13), using cubic B-splines over a mesh consisting of 7 equally spaced, distinct points on $(-\pi, +\pi)$, with the time-evolution carried out to $10^{-2}$ absolute accuracy. The error at each time-step is printed out to confirm the accuracy of the computed solution. The main program is

```
    Real tstop,V(1),dt,mesh(100),U(100)
    Real errpar(2)
    Integer k,nmesh,ndx,Nu,Nv
    External AF,BC,DEE,HANDLE

    Common / CSTAK / Ds(2000); Double Precision Ds    # The PORT Stack.

    Call ISTKIN(2000,4)    # Initialize the PORT Library stack length.

    Nu = 1; Nv = 1

    errpar(1) = 0; errpar(2) = 1e-2    # Absolute error.

    tstop = 8* ATAN(1e0); dt = 0.4

  # Make a mesh of ndx uniform points on (-pi,+pi).

    k = 4; ndx = 7
    Call UMB(-4* ATAN(1e0),+4* ATAN(1e0),ndx,k,mesh,nmesh)

    Call SETR(nmesh-k,0e0,U)    # Initial conditions for U.

    V(1) = 0    # Initial conditions for V.

    Call POST (U,Nu,k,mesh,nmesh,V,Nv,
               0e0,tstop,dt,
               AF,BC,DEE,
               errpar,
               HANDLE)

    Call WRAPUP    # Check for errors and stack usage statistics.

    Stop

    End
```

The body of the subroutine AF for (A4.11) is

```
    Do i = 1, nx
      {
      A(i,1) = -Ux(i,1); AUx(i,1,1) = -1
      f(i,1) = -Ut(i,1)-U(i,1)**3+
                 Cos(x(i))*(Cos(t)+Sin(t)+Cos(x(i))**2*Sin(t)**3)
      fUt(i,1,1) = -1; fU(i,1,1) = -3*U(i,1)**2
      }
```

while the body of the subroutine BC for (A4.12) is

```
    B(1,1) = Ux(1,1)-V(1)
    B(1,2) = Ux(1,2)-V(1)

    BUx(1,1,1) = 1; BV(1,1,1) = -1
    BUx(1,1,2) = 1; BV(1,1,2) = -1
```

and the body of the subroutine DEE for (A4.13) is

```
D(1) = U(1,1)-U(nx-k,1)    # u(-pi,t) - u(+pi,t) = 0.
DU(1,1,1) = 1; DU(1,nx-k,1) = -1
```

The body of the subroutine `HANDLE` for computing and printing the error is

```
 Common / TIME / tt; Real tt

 Real eU, EESFF,eV
 Integer I1MACH
 External UOFX

 If ( t0 == t ) { Return }

 tt = t    # UOFX needs time.
 eU = EESFF(k,x,nx,U,UOFX); eV = V(1)

 iwunit = I1MACH(2)
 Write(iwunit,9000) t,eU,eV
9000 Format(" Error in U(x,",1p1e10.2," ) =",1p1e10.2,"   V =",1p4e10.2)
```

The only change in the subroutine `UOFX` for computing $u$ in the previous example is the code for computing $u$.

```
Do i = 1, nx  { U(i) = Cos(x(i))*Sin(t) }
```

The output of this program is

```
Error in U(x,  4.00e-01 ) =  1.00e-03   V = -2.91e-08
Error in U(x,  1.96e+00 ) =  2.86e-03   V =  6.28e-08
Error in U(x,  3.41e+00 ) =  8.48e-03   V = -1.28e-07
Error in U(x,  5.41e+00 ) =  2.08e-03   V = -2.73e-08
Error in U(x,  6.28e+00 ) =  1.52e-04   V = -3.36e-08
used      792 /     4000 of the stack allowed.
```

The run-time for the above example was 84.3 seconds.

**Example 5 - A Two-phase, Moving Front.**

This example considers a parabolic **pde** with a moving interface and represents a simplified version of a program used to study the action of lasers in semiconductor annealing [51]. The **pde** is

$$a = c(u)\,u_x - H(x, v_2(t))\,\alpha\,v_{2t}(t)$$

$$\text{for}\quad x\,\varepsilon\,(-1,+2) \qquad \text{(A4.14)}$$

$$f = u_t - g(t,x).$$

with

$$c(u) \equiv \begin{cases} 1 & \text{for } x < x(t) \\ 2 & \text{for } x > x(t), \end{cases}$$

$\alpha = -3$, **bc**'s and the inhomogeneous term $g(t, x)$ chosen so that the solution is known to be

$$u(x, t) \equiv \begin{cases} e^{x-x(t)} & \text{for } x < x(t) \\ e^{x(t)-x} & \text{for } x > x(t) \end{cases}$$

with $x(t) \equiv t$.

**ode** variables $v_1(t)$, $v_2(t)$ and $v_3(t)$ are used in `LPLM` ( see section 5 ) to map the $\xi$ interval $[0, 2]$ into the $x$ interval $[-1, +2]$, with $x(\xi = 1, t) \equiv v_2(t)$ being the position where $u(t,x) = 1$. The **ode** is

$$
\begin{aligned}
d_1 &= v_1 + 1 \\
d_2 &= w(1, t) - 1 \\
d_3 &= v_3 - 2
\end{aligned}
\tag{A4.15}
$$

The following program unit solves (A4.14)-(A4.15) accurate to $10^{-2}$. Several PORT Library routines are used

- `MOVEFR` copies one array into another.
- `ISTKGT` allocates storage on the Port Library stack. In the example below, `iU = ISTKGT(nmesh-k,3)` sets the pointer `iU` so that locations `Ws(iU), ..., Ws(iU+nmesh-k-1)` are available for the B-spline coefficients of the solution.
- `ENTER` and `LEAVE` bracket blocks of code in which stack allocations are done. The effect is that all allocations made after the `ENTER` but before the `LEAVE` are released by the `LEAVE`.
- `ILUMB` makes a B-spline mesh by putting a uniform mesh on each of a set of contiguous intervals, in this case 6 points each on $[\,0,\,1\,]$ and $[\,1,\,2\,]$. The mesh is put on the Port stack and a pointer to it is returned.
- `IMMM` makes $\xi = 1$ a mesh point of multiplicity `k-1`, so that $w(\xi, t)$ can have its partials with respect to $\xi$ discontinuous at $\xi = 1$, see appendix 1. $k-2$ points must be added to the mesh to accomplish this. Since it is dangerous to assume that a fixed dimension array can be added to, the mesh is put on the stack by `ILUMB` where it may be safely lengthened.
- `L2SFF` fits a function, in this case defined by `UOFX`, with a spline.

`LPLMG` sets initial conditions for the mapping variables $\mathbf{v}$, see section 5.

```
Real tstart,tstop,V(3),dt,xb(3)
Real errpar(2)
Integer ndx,Nu,Nv,iU, ILUMB,ISTKGT,IMMM ,k,imesh,nmesh
External AF,BC,DEE,HANDLE,UOFX

Common / PARAM / Vc(3),x(3); Real Vc,x
Common / TIME / t; Real t

Common / CSTAK / Ds(4000); Double Precision Ds    # The PORT Stack.
Real Ws(1000)    # The PORT Library stack and its aliases.
Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

Call ISTKIN(4000,4)    # Initialize the PORT Library stack length.

Call ENTER(1)

Nu = 1; Nv = 3

errpar(1) = 0; errpar(2) = 1e-2    # Absolute error.

tstart = 0; tstop = 1; dt = 0.1

k = 4; ndx = 6    # ndx uniform mesh points on each interval of xb array.
xb(1) = 0; xb(2) = 1; xb(3) = 2
imesh = ILUMB(xb,3,ndx,k,nmesh)    # Get mesh on PORT Stack.
imesh = IMMM (imesh,nmesh,1e0,k-1)    # Make 1 of multiplicity k-1.

x(1) = -1; x(2) = 0; x(3) = +2
Call LPLMG(3,x,Vc)    # Initial values for V.

iU = ISTKGT(nmesh-k,3)    # Get U on the PORT Stack.

t = tstart    # UOFX needs time.
Call MOVEFR(Nv,Vc,V)    # UOFX needs V for mapping.
Call L2SFF(UOFX,k,Ws(imesh),nmesh,Ws(iU))    # Initial conditions for U.

Call HANDLE(t-1,Ws(iU),V,t,Ws(iU),V,Nu,nmesh-k,Nv,    # Output the ICs.
            k,Ws(imesh),nmesh,dt,tstop)

Call POST (Ws(iU),Nu,k,Ws(imesh),nmesh,V,Nv,
           tstart,tstop,dt,
           AF,BC,DEE,
           errpar,
           HANDLE)

Call LEAVE

Call WRAPUP

Stop

End
```

The AF procedure uses three routines described in section 5.

- LPLM performs a Linearly parameterized Piecewise Linear Mapping from $\xi$ to $x$.
- POSTU to convert **u** from $\xi$ to $x$ coordinates.
- POSTI to convert the **pde** from $x$ to $\xi$ coordinates.

The body of the AF procedure is

```
Common /  POSTF / Failed; Logical Failed

Real kay
Real Ax(99),Fx(99),x(99),xt(99),xxi(99),xxiv(99),xv(99),xtv(99),xvv(99)
Integer i

If ( V(2) <= V(1) | V(2) >= V(3) ) { Failed = .True.; Return }

Call LPLM(xi,nx,V,3,    # Map xi into x.
          x,xxi,xxiv,xv,xvv,xt,xtv)

Call POSTU(xi,x,xt,xxi,xv,Vt,nx,3,    # Map U into x system.
           Ux,Ut,Nu,
           Ax,Fx)

Do i = 1, nx
  {
  If ( xi(i) <= 1 )  { kay = 1 }
  Else               { kay = 2 }

  A(i,1) = kay*Ux(i,1); AUx(i,1,1) = kay

  If ( xi(i) <= 1 )
    {
    A(i,1) = A(i,1) - 3*Vt(2); AVt(i,1,2) = -3
    }

  f(i,1) = Ut(i,1); fUt(i,1,1) = 1

  If ( xi(i) <= 1 )
    {
    f(i,1) = f(i,1) + 2*Exp(x(i)-t)
    Fx(i) = 2*Exp(x(i)-t)
    }
  Else
    {
    f(i,1) = f(i,1) + Exp(t-x(i))
    Fx(i) = -Exp(t-x(i))
    }
  }

Call POSTI(xi,x,xt,xxi,xv,xtv,xxiv,xvv,nx,    # Map A and F into xi system.
           Ux,Ut,Nu,
           V,Vt,Nv,1,3,
           A,Ax,AU,AUx,AUt,AUtx,AV,AVt,
           F,Fx,FU,FUx,FUt,FUtx,FV,FVt)
```

and the body of the BC procedure is

```
B(1,1) = U(1,1)-Exp(-1-t)
B(1,2) = U(1,2)-Exp(t-2)

BU(1,1,1) = 1
BU(1,1,2) = 1
```

The `DEE` procedure uses two Port Library routines

- `INTRVR` finds the mesh interval containing a given point.
- `BSPLN` evaluates the B-spline basis functions.

The body of `DEE` is

```
Real xx(1),Bx(10)
Integer i,ileft,INTRVR

D(1) = V(1)+1; DV(1,1) = 1    # x(0,V) = -1.

xx(1) = 1
ileft = INTRVR(nx,x,xx(1))    # Find 1 in the mesh.
Call BSPLN(k,x,nx,xx,1,ileft,Bx)    # Get the B-spline basis at xx.
D(2) = -1    # u(x(1,V),t) = 1.
Do i = 1, k
  {
  D(2) = D(2) + U(ileft+i-k,1)*Bx(i)
  DU(2,ileft+i-k,1) = Bx(i)
  }

D(3) = V(3)-2; DV(3,3) = 1    # x(2,V) = +2.
```

The body of the `HANDLE` routine is

```
 Common / TIME / tt; Real tt
 Common / PARAM / Vc(3),xx(3); Real Vc,xx

 Real eU, EESFF,eV(3)
 Integer I1MACH
 External UOFX

 If ( t0 == t )
   {
   iwunit = I1MACH(2)
   Write(iwunit,9000) t
9000 Format(" Restart for t =",1p4e10.2)
   Return
   }

 tt = t
 Call MOVEFR(Nv,V,Vc)    # UOFX needs V for mapping.

 eU = EESFF(k,x,nx,U,UOFX)

 eV(1) = V(1)+1
 eV(2) = V(2)-t
 eV(3) = V(3)-2

 iwunit = I1MACH(2)
 Write(iwunit,9001) t,eU,eV
9001 Format(" Error in U(x,",1p1e10.2," ) =",1p1e10.2,"   V =",1p4e10.2)
```

The `UOFX` routine is made "bomb-proof" against too large values of `nx` by using the PORT stack instead of local arrays for the mapping intermediate results. One routine from section 5 is used

- `LPLMX` does a Linearly parameterized Piecewise Linear Mapping from $\xi$ to $x$, without calculating all the derivatives.

```
Common / TIME / t; Real t
Common / PARAM / Vc(3),x(3); Real Vc,x

Common / CSTAK / Ds(500); Double Precision Ds    # The PORT Stack.
Real Ws(1000)    # The PORT Library stack and its aliases.
Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

Real xofxi
Integer i,ISTKGT,ixx,ixv

Call ENTER(1)

ixx = ISTKGT(nx,3); ixv = ISTKGT(nx*3,3)    # Space for x and xv.

Call LPLMX(xi,nx,Vc,3,Ws(ixx),Ws(ixv))    # Map into user system.

Do i = 1, nx
  {
  xofxi = Ws(ixx+i-1)
  If ( xi(i) <= 1 ) { U(i) = Exp(xofxi-t) }
  Else              { U(i) = Exp(t-xofxi) }
  }

Call LEAVE
```

The output of the above program unit is

```
Error in U(x,  0.00e+00 ) =  2.16e-05   V =  0.00e+00  0.00e+00  0.00e+00
Error in U(x,  1.00e-01 ) =  3.03e-04   V =  0.00e+00 -2.12e-05  0.00e+00
Error in U(x,  6.65e-01 ) =  4.60e-03   V =  0.00e+00 -2.15e-03  0.00e+00
Error in U(x,  1.00e+00 ) =  4.71e-03   V =  0.00e+00 -3.45e-03  0.00e+00
used     1183 /     8000 of the stack allowed.
```

which shows that the problem has indeed been solved accurately. The run-time for the above example was 30.0 seconds.

**Example 6 - A Moving Shock Wave.**

This example consists of a hyperbolic **pde** with a moving shock-wave, based on (4.20) and (4.22) with $u_0 \equiv 0$.

$$\left.\begin{aligned} a &= -\,\hat{u} \\ f &= \hat{u}_t - g(x,t) - H(x, x(t))\, v_{4t} \end{aligned}\right. \qquad \text{for} \quad x\ \varepsilon\ (-\pi,+\pi) \tag{A4.16}$$

We solve (A4.16) with $g$ chosen so that the solution is $\hat{u} = u + H(x, x(t))\, v_4$, where

$$u \equiv \begin{cases} \sin\,(\,t + x\,) & x < x(t) \\ \cos\,(\,t + t\,) & x > x(t) \end{cases}.$$

with $x(t) \equiv \dfrac{\pi}{2} - t$. We solve this problem with $v_1$, $v_2$, $v_3$ used to describe the LPLM map from $[\ 0,\ 2\ ]$ to $[\ -\pi,\ +\pi\ ]$, so that $v_2(t)$ is the position of the shock, and $v_4 \equiv$ the jump at $x\ (t)$. The **ode** is

$$\begin{aligned} d_1 &= v_1 + \pi \\ d_2 &= w(1_+,t) - v_4 \\ d_3 &= v_3 - \pi \\ d_4 &= v_{2t} + v_4 \end{aligned} \tag{A4.17}$$

The following main program solves (A4.16)-(A4.17).

```
Real tstart,tstop,V(4),dt,xb(3)
Real errpar(2)
Integer ndx,Nu,Nv,iU, ILUMB,ISTKGT,IMMM ,k,imesh,nmesh
External AF,BC,DEE,HANDLE,UOFX

Common / PARAM / Vc(4),x(3); Real Vc,x
Common / TIME / t; Real t

Common / CSTAK / Ds(4000); Double Precision Ds    # The PORT Stack.
Real Ws(1000)    # The PORT Library stack and its aliases.
Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

Call ISTKIN(4000,4)    # Initialize the PORT Library stack length.

Call ENTER(1)

Nu = 1; Nv = 4

errpar(1) = 0; errpar(2) = 1e-2    # Absolute error.

tstart = 0; tstop = 3.14; dt = 0.4

k = 4; ndx = 6    # ndx uniform mesh points on each interval of xb.
xb(1) = 0; xb(2) = 1; xb(3) = 2
imesh = ILUMB(xb,3,ndx,k,nmesh)    # Get mesh on PORT Stack.
imesh = IMMM (imesh,nmesh,1e0,k-1)    # Make 1 of multiplicity k-1.

x(1) = -3.14; x(2) = 3.14/2; x(3) = 3.14
Call LPLMG(3,x,Vc)    # Initial values for V.

iU = ISTKGT(nmesh-k,3)    # Get U on the PORT Stack.

t = tstart    # UOFX needs time.
Vc(4) = +1     # The initial height of the jump.
Call MOVEFR(Nv,Vc,V)    # UOFX needs V for mapping.
Call L2SFF(UOFX,k,Ws(imesh),nmesh,Ws(iU))    # Initial conditions for U.

Call HANDLE(t-1,Ws(iU),V,t,Ws(iU),V,Nu,nmesh-k,Nv,    # Output ICs.
            k,Ws(imesh),nmesh,dt,tstop)

Call POST (Ws(iU),Nu,k,Ws(imesh),nmesh,V,Nv,
           tstart,tstop,dt,
           AF,BC,DEE,
           errpar,
           HANDLE)

Call LEAVE

Call WRAPUP

Stop

End
```

with the body of the AF procedure being

```
Common /  POSTF / Failed; Logical Failed

Real Ax(99),Fx(99),x(99),xt(99),xxi(99),xxiv(99),xv(99),xtv(99),xvv(99)
Integer i

If ( V(2) <= V(1) | V(2) >= V(3) ) { Failed = .True.; Return }

Call LPLM(xi,nx,V,3,    # Map xi into x.
          x,xxi,xxiv,xv,xvv,xt,xtv)

Call POSTU(xi,x,xt,xxi,xv,Vt,nx,3,    # Map U into x system.
           Ux,Ut,Nu,
           Ax,Fx)

Do i = 1, nx
  {
  A(i,1) = -U(i,1); AU(i,1,1) = -1

  f(i,1) = Ut(i,1); fUt(i,1,1) = 1

  If ( xi(i) <= 1 )
    {
    f(i,1) = f(i,1) - 2*Cos(x(i)+t)
    Fx(i) = 2*Sin(x(i)+t)
    }
  Else
    {
    f(i,1) = f(i,1) - Vt(4); fVt(i,1,4) = -1

    f(i,1) = f(i,1) + 2*Sin(x(i)+t)
    Fx(i) = 2*Cos(x(i)+t)
    }
  }

Call POSTI(xi,x,xt,xxi,xv,xtv,xxiv,xvv,nx,    # Map A and F into xi system.
           Ux,Ut,Nu,
           V,Vt,Nv,1,3,
           A,Ax,AU,AUx,AUt,AUtx,AV,AVt,
           F,Fx,FU,FUx,FUt,FUtx,FV,FVt)
```

and the body of the BC procedure being

```
B(1,1) = U(1,1)-Sin(-3.14+t); BU(1,1,1) = 1    # u(-pi,t) = Sin(-pi+t).
```

The **d** procedure uses the Port Library routine

- R1MACH to define machine constants. Here, R1MACH(4) gives the rounding error of the machine.

The body of the DEE procedure is

```
Real xx(1),Bx(10),R1MACH
Integer i,ileft,INTRVR

D(1) = V(1)+3.14; DV(1,1) = 1    # x(0,V) = -pi.

xx(1) = 1+R1MACH(4)    # xx(1) = 1 + a rounding error.
ileft = INTRVR(nx,x,xx(1))
Call BSPLN(k,x,nx,xx,1,ileft,Bx)    # Get the B-spline basis at xx.
D(2) = -V(4); DV(2,4) = -1    # u(x(t)+,t) - jump = 0.
Do i = 1, k
  {
  D(2) = D(2) + U(ileft+i-k,1)*Bx(i)
  DU(2,ileft+i-k,1) = Bx(i)
  }

D(3) = V(3)-3.14; DV(3,3) = 1    # x(2,V) = +pi.

D(4) = Vt(2)+V(4)    # jump + d( x(1,V(t)) )/dt = 0.
DVt(4,2) = 1; DV(4,4) = 1
```

The body of the HANDLE procedure is

```
 Common / TIME / tt; Real tt
 Common / PARAM / Vc(4),xx(3); Real Vc,xx

 Real eU, EESFF,eV(2)
 Integer I1MACH
 External UOFX

 If ( t0 == t )
   {
   iwunit = I1MACH(2)
   Write(iwunit,9000) t,dt
9000 Format(" Restart for t =",1p1e10.2, "   dt =",1p4e10.2)
   Return
   }

 tt = t
 Call MOVEFR(Nv,V,Vc)    # UOFX needs V for mapping.

 eU = EESFF(k,x,nx,U,UOFX)

 eV(1) = V(2)-(3.14/2-t)    # Error in position of shock.
 eV(2) = V(4)-1            # Error in height of shock.

 iwunit = I1MACH(2)
 Write(iwunit,9001) t,eU,eV
9001 Format(" Error in U(x,",1p1e10.2," ) =",1p1e10.2,"   V =",1p4e10.2)
```

with the body of the UOFX routine

```
Common / TIME / t; Real t
Common / PARAM / Vc(4),x(3); Real Vc,x

Common / CSTAK / Ds(500); Double Precision Ds    # The PORT Stack.
Real Ws(1000)    # The PORT Library stack and its aliases.
Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

Real EWE
Integer i,ISTKGT,ixx,ixv

Call ENTER(1)

ixx = ISTKGT(nx,3); ixv = ISTKGT(nx*3,3)    # Space for x and xv.

Call LPLMX(xi,nx,Vc,3,Ws(ixx),Ws(ixv))    # Map into user system.

Do i = 1, nx
  {
  U(i) = EWE(t,Ws(ixx+i-1),Vc(2))
  If ( xi(i) > 1 ) { U(i) = U(i) + 1 }
  }

Call LEAVE
```

The subroutine EWE( $t$, $x$, $x_{break}$ ) computes $w(\xi, t)$ assuming that the jump discontinuity $x(t)$ is at position $x_{break}$. $x_{break}$ = V(2) is used, for otherwise the error would be large if V(2) were not exactly $x(t)$. We wish to assess the errors in the position and height of the shock separately. The subroutine EWE is

```
Real Function EWE(t,x,xbreak)

Real t,x,xbreak

If      ( x < xbreak ) { EWE = Sin(x+t); Return }
Else If ( x > xbreak ) { EWE = Cos(x+t); Return }
Else { Call SETERR("EWE - x == xbreak",17,1,2) }

Stop

End
```

The output of the above program unit is

```
Error in U(x,  0.00e+00 ) =  1.42e-03   V =  0.00e+00  0.00e+00
Error in U(x,  4.00e-01 ) =  1.10e-03   V = -4.41e-05 -2.42e-04
Error in U(x,  1.35e+00 ) =  3.65e-03   V = -2.25e-05 -2.57e-03
Error in U(x,  2.47e+00 ) =  6.56e-03   V =  4.09e-03 -3.06e-03
Error in U(x,  3.14e+00 ) =  5.70e-03   V =  2.26e-03  1.18e-03
used     1363 /     8000 of the stack allowed.
```

and we can see that the problem has been solved accurately. The run-time for the above example was 143.2 seconds.

**Example 7 - Tracking a Moving Boundary Layer.**

Consider tracking a boundary layer in a **pde** like

$$a = -u_x \qquad \text{and} \qquad f = g(x,t) - u_t \qquad \text{for} \quad x \,\varepsilon\, (0,1) \tag{A4.18}$$

where $g$ is chosen so that the solution is $e^{-xt}$, using $v_1$, $v_2$ and $v_3$ to parameterize the `LPLM` mapping and $v_4 \equiv t$ to replace $t$ in $g$. Without the replacement of the explicit appearance of $t$ in the **pde**, the solution time would be much longer (by a factor of 4 and more). This replacement simply gives Newton's method a better starting guess for each time-step; without it the nonlinear equation solver has to work much harder. The **ode** is

$$\begin{aligned} d_1 &= v_1 \\ d_2 &= w(\,\xi = \xi_0, t) - e^{-1} \\ d_3 &= v_3 - 1 \\ d_4 &= v_{4t} - 1 \end{aligned} \tag{A4.19}$$

where $\xi_0 = 1/(\text{ D ln } 10\ )$. We use $2.3$ as an approximation to ln $10$.

The following program unit solves (A4.18)-(A4.19) accurate to $10^{-2}$ from time $t = 20$ to time $t = 10^6$.

```
Real tstart,tstop,V(4),dt,xb(3),D
Real errpar(2)
Integer ndx,Nu,Nv,iU, ILUMB,ISTKGT,IMMM ,k,imesh,nmesh
External AF,BC,DEE,HANDLE,UOFX

Common / PARAM / Vc(4),x(3),xi0; Real Vc,x,xi0
Common / TIME / t; Real t

Common / CSTAK / Ds(4000); Double Precision Ds    # The PORT Stack.
Real Ws(1000)    # The PORT Library stack and its aliases.
Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

Call ISTKIN(4000,4)    # Initialize the PORT Library stack length.

Call ENTER(1)

Nu = 1; Nv = 4

errpar(1) = 1e-2; errpar(2) = 1e-2    # Mixed relative and absolute error.

D = 3

xi0 = 1/2.3/D    # w(xi0,t) = 1/e.

tstart = 20; tstop = 1e+6; dt = 1e-2

k = 4; ndx = 6    # ndx uniform mesh points on each interval of xb.
xb(1) = 0; xb(2) = 1; xb(3) = 2
imesh = ILUMB(xb,3,ndx,k,nmesh)    # Get mesh on PORT Stack.
imesh = IMMM (imesh,nmesh,1e0,k-1)    # Make 1d0 of multiplicity k-1.

x(1) = 0; x(2) = 2.3*D/tstart; x(3) = 1
Call LPLMG(3,x,Vc)    # Initial values for V.

iU = ISTKGT(nmesh-k,3)    # Get U on PORT Stack.

t = tstart    # UOFX needs time.
Vc(4) = tstart
Call MOVEFR(Nv,Vc,V)    # UOFX needs V for mapping.
Call L2SFF(UOFX,k,Ws(imesh),nmesh,Ws(iU))    # Initial conditions for U.

Call HANDLE(t-1,Ws(iU),V,t,Ws(iU),V,Nu,nmesh-k,Nv,    # Output ICs.
            k,Ws(imesh),nmesh,dt,tstop)

Call POST (Ws(iU),Nu,k,Ws(imesh),nmesh,V,Nv,
           tstart,tstop,dt,
           AF,BC,DEE,
           errpar,
           HANDLE)

Call LEAVE

Call WRAPUP
```

```
Stop

End
```

with the body of the AF routine

```
Common /  POSTF / Failed; Logical Failed

Real  EXPL
Real Ax(99),Fx(99),x(99),xt(99),xxi(99),xxiv(99),xv(99),xtv(99),xvv(99)
Integer i

If ( V(2) <= V(1) | V(2) >= V(3) ) { Failed = .True.; Return }

Call LPLM(xi,nx,V,3,     # Map xi into x.
          x,xxi,xxiv,xv,xvv,xt,xtv)

Call POSTU(xi,x,xt,xxi,xv,Vt,nx,3,    # Map U into x system.
           Ux,Ut,Nu,
           Ax,Fx)

Do i = 1, nx
  {
  A(i,1) = -Ux(i,1); AUx(i,1,1) = -1
  f(i,1) = -Ut(i,1)- EXPL(-x(i)*V(4))*(x(i)+V(4)**2)
  fUt(i,1,1) = -1
  fV(i,1,4) = - EXPL(-x(i)*V(4))*(2*V(4)+(x(i)+V(4)**2)*(-x(i)))
  Fx(i) = - EXPL(-x(i)*V(4))*(1-V(4)*x(i)-V(4)**3)
  }

Call POSTI(xi,x,xt,xxi,xv,xtv,xxiv,xvv,nx,    # Map A and F into xi system.
           Ux,Ut,Nu,
           V,Vt,Nv,1,3,
           A,Ax,AU,AUx,AUt,AUtx,AV,AVt,
           F,Fx,FU,FUx,FUt,FUtx,FV,FVt)
```

where EXPL($x$) is a routine for computing $e^x$ for large negative $x$ without aborting because of exponent underflow. This is necessary on many machines because things like $e^{-10^6}$ are being computed, and many systems gag on such input. EXPL is provided as part of POST, so the examples will run.

The **bc**s are chosen to reflect the solution, $e^{-tx}$, and time $t$ is replaced by $v_4$ in them as well. The body of the BC routine is

```
B(1,1) = U(1,1)-1                # u(0,t) = 1
B(1,2) = U(1,2)- EXPL(-V(4))     # u(1,t) = Exp(-t)

BU(1,1,1) = 1
BU(1,1,2) = 1
BV(1,4,2) = EXPL(-V(4))
```

the body of the DEE routine is

```
Common / PARAM / Vc(4),xc(3),xi0; Real Vc,xc,xi0

Real xx(1),Bx(10)
Integer i,ileft,INTRVR

D(1) = V(1); DV(1,1) = 1    # x(0,V) = 0.

xx(1) = xi0
ileft = INTRVR(nx,x,xx(1))
Call BSPLN(k,x,nx,xx,1,ileft,Bx)    # Get the B-spline basis at xx.
D(2) = - Exp(-1e0)
Do i = 1, k    # D(2) = w(xi0,t) - Exp(-1).
  {
  D(2) = D(2) + U(ileft+i-k,1)*Bx(i)
  DU(2,ileft+i-k,1) = Bx(i)
  }

D(3) = V(3)-1; DV(3,3) = 1    # x(2,V) = 1.

D(4) = Vt(4)-1; DVt(4,4) = 1
```

The output routine uses a routine from section 5

- `LPLMT` restrains the time-step `dt` so that mapped mesh points are not expected to collide with one another.

The body of the `HANDLE` routine is

```
Common / TIME / tt; Real tt
Common / PARAM / Vc(4),xx(3),xi0; Real Vc,xx,xi0

Real eU, EESFF,eV, LPLMT
Integer I1MACH
External UOFX

If ( t0 == t )
  {
  iwunit = I1MACH(2)
  Write(iwunit,9000) t,dt
 9000 Format(" Restart for t =",1p1e10.2,"   dt =",1p4e10.2)
  Return
  }

# Let dt carry V(2) down by no more than a factor of 10.

dt = LPLMT(t,V,Nv,t0,V0,1e-1,dt)

tt = t
Call MOVEFR(Nv,V,Vc)    # UOFX needs V for mapping.
eU = EESFF(k,x,nx,U,UOFX)

eV = V(2)-1/xi0/t    # Error in position of boundary layer.

iwunit = I1MACH(2)
Write(iwunit,9001) t,eU,eV
 9001 Format(" Error in U(x,",1p1e10.2," ) =",1p1e10.2,"   V =",1p4e10.2)
```

and the body of the UOFX routine is

```
Common / TIME / t; Real t
Common / PARAM / Vc(4),x(3),xi0; Real Vc,x,xi0

Common / CSTAK / Ds(500); Double Precision Ds    # The PORT Stack.
Real Ws(1000)    # The PORT Library stack and its aliases.
Real Rs(1000) ; Integer Is(1000) ; Complex Cs(500) ; Logical Ls(1000)
Equivalence ( Ds(1),Cs(1),Ws(1),Rs(1),Is(1),Ls(1) )

Real  EXPL
Integer i,ISTKGT,ixx,ixv

Call ENTER(1)

ixx = ISTKGT(nx,3); ixv = ISTKGT(nx*3,3)    # Space for x and xv.

Call LPLMX(xi,nx,Vc,3,Ws(ixx),Ws(ixv))    # Map into user system.

Do i = 1, nx  { U(i) = EXPL(-Ws(ixx+i-1)*t) }

Call LEAVE
```

The output of the above program unit is

```
Error in U(x,  2.00e+01 ) =  2.50e-03   V =  0.00e+00
Error in U(x,  2.00e+01 ) =  1.85e-03   V = -1.22e-03
Error in U(x,  2.02e+01 ) =  1.86e-03   V = -1.22e-03
Error in U(x,  3.84e+01 ) =  1.85e-03   V = -6.36e-04
Error in U(x,  7.29e+01 ) =  1.73e-03   V = -2.99e-04
Error in U(x,  1.39e+02 ) =  4.57e-03   V = -9.76e-05
Error in U(x,  2.64e+02 ) =  7.28e-03   V = -2.78e-05
Error in U(x,  5.01e+02 ) =  7.79e-03   V = -1.26e-05
Error in U(x,  9.52e+02 ) =  7.88e-03   V = -6.47e-06
Error in U(x,  1.81e+03 ) =  7.92e-03   V = -3.36e-06
Error in U(x,  3.44e+03 ) =  7.94e-03   V = -1.76e-06
Error in U(x,  6.53e+03 ) =  7.97e-03   V = -9.23e-07
Error in U(x,  1.24e+04 ) =  7.97e-03   V = -4.85e-07
Error in U(x,  2.36e+04 ) =  7.98e-03   V = -2.55e-07
Error in U(x,  4.48e+04 ) =  7.98e-03   V = -1.34e-07
Error in U(x,  8.51e+04 ) =  7.98e-03   V = -7.06e-08
Error in U(x,  1.62e+05 ) =  7.99e-03   V = -3.71e-08
Error in U(x,  3.07e+05 ) =  7.99e-03   V = -1.96e-08
Error in U(x,  5.84e+05 ) =  7.99e-03   V = -1.03e-08
Error in U(x,  1.00e+06 ) =  7.99e-03   V = -6.01e-09
used     1306 /     8000 of the stack allowed.
```

and we can see that the problem has been solved accurately. The run-time for the above problem was 295.9 seconds.

**Example 8 - An Integro-Partial Differential Equation.**

This example is an integro-differential equation

$$a = -p\, u_x \qquad \text{and} \qquad f = -u_t + \int_0^1 c(x,y,\sum_i v_i\,(t)\, B_i\,(y)\,)\,dy \qquad \text{for } x\ \varepsilon\ (0,\,1), \quad \text{(A4.20)}$$

where $c(x,y,u)$ is some given kernel. The **ode** is

$$\mathbf{d} = \mathbf{v} - \mathbf{U} = \mathbf{0}. \qquad \text{(A4.21)}$$

The following program unit solves (A4.20)-(A4.21) with $p = 2$ and $c\,(\,x,\,y,\,u\,) \equiv -e^{x-y}\,u$, using code suitable for solving a general nonlinear kernel term $c\,(\,x,\,y,\,u(y)\,)$. The main program is

```
Real tstop,V(100),dt,U(100)
Real errpar(2)
Integer ndx,Nu,Nv,i
External AF,BC,DEE,HANDLE,UOFX

Common / CMESH / mesh(100); Real mesh
Common / KMESH / k,nmesh; Integer k,nmesh
Common / TIME / t; Real t

Common / CSTAK / Ds(5000); Double Precision Ds    # The PORT Stack.

Call ISTKIN(5000,4)    # Initialize the PORT Library stack length.

Nu = 1

errpar(1) = 0; errpar(2) = 1e-2    # Absolute error.

tstop = 1; dt = 1e-2

k = 4; ndx = 7    # ndx uniform mesh points on (0,1).
Call UMB(0e0,1e0,ndx,k,mesh,nmesh)
Nv = nmesh-k

t = 0    # UOFX needs t.
Call L2SFF(UOFX,k,mesh,nmesh,U)    # ICs for U.
Do i = 1, nmesh-k { V(i) = U(i) }    # ICs for V.

Call POST (U,Nu,k,mesh,nmesh,V,Nv,
           0e0,tstop,dt,
           AF,BC,DEE,
           errpar,
           HANDLE)

Call WRAPUP

Stop

End
```

with the body of the AF procedure

```
Common / CMESH / mesh(100); Real mesh
Common / KMESH / k,nmesh; Integer k,nmesh

Integer i

Do i = 1, nx
  {
  A(i,1) = 2*Ux(i,1); AUx(i,1,1) = 2
  f(i,1) = Ut(i,1); fUt(i,1,1) = +1
  }

Call INTGRL(k,mesh,nmesh,V,x,nx,f,fV)    # Get the Integral.
```

where `INTGRL` will be described shortly. The body of the `BC` procedure is

```
B(1,1) = U(1,1)-Exp(t); B(1,2) = U(1,2)-Exp(t+1)
BU(1,1,1) = 1;          BU(1,1,2) = 1
```

the body of the DEE procedure is

```
Integer i

Do i = 1, nxmk
  {
  D(i) = U(i,1)-V(i)
  DU(i,i,1) = 1; DV(i,i) = -1
  }
```

and the body of the HANDLE procedure is

```
 Common / TIME / tt; Real tt

 Real eU, EESFF
 Integer I1MACH
 External UOFX

 If ( t0 == t )
   {
   iwunit = I1MACH(2)
   Write(iwunit,9000) t0,dt
9000 Format(" Restart for t =",1p1e10.2,"   dt =",1p4e10.2)
   Return
   }

 tt = t

 eU = EESFF(k,x,nx,U,UOFX)

 iwunit = I1MACH(2)
 Write(iwunit,9001) t,eU
9001 Format(" Error in U(x,",1p1e10.2," ) =",1p4e10.2)
```

The body of the UOFX procedure is

```
Common / TIME / t; Real t

Integer i

Do i = 1, nx  { U(i) = Exp(t+x(i)) }
```

The procedure INTGRL for doing the integral is based on Gaussian quadrature. The first step in evaluating $\int_0^1 c(\,x, y,\ \sum_l v_l(t)\ B_l(y)\,)\ dy$ is to note that

$$\int_0^1 c(\,x, y,\ \sum_l v_l(t)\ B_l(y)\,)\ dy\ =\ \sum_i \int_{x_i}^{x_{i+1}} c(\,x, y,\ \sum_l v_l(t)\ B_l(y)\,)\ dy \tag{A4.22}$$

where the $x_i$ are the mesh points used in the spatial mesh given to POST. Each integral on the right-hand-side of (A4.22) is of a smooth function and is of the form $\int_a^b z(y)\ dy$. Gaussian quadrature rules allow accurate evaluation of integrals of the form $\int_{-1}^{+1} z(\xi)\ d\xi$ with formulae like

$$\int_{-1}^{+1} z(\xi)\ d\xi \approx \sum_m z(\xi_m)\ w_m$$

where the abcissae $\xi_m$ and weights $w_m$ can be obtained by calling the PORT Library procedure `GQM11`. Since

$$\int_a^b z(y)\,dy = \int_{-1}^{+1} z\left( \frac{a+b}{2} + \frac{b-a}{2}\ \xi \right) \frac{b-a}{2} d\xi \tag{A4.23}$$

we see that we can use (A4.22) and (A4.23) to evaluate the integral in (A4.20). The following procedure `INTGRL` does just that when $k = 4$, or a fourth order spline, is used in the spatial discretization for `POST`.

```
  Subroutine INTGRL(k,mesh,nmesh,V,x,nx,f,fV)

# To compute

#    f = Integral from mesh(1) to mesh(nmesh)
#       Kernel(x,y,Sum(i=1,...,nmesh-k) V(i)*B(i,y)) dy

#  and

#    fV = D(f)/D(V).

# Assume that Call KERNEL(x,y,u,ker,keru) Returns

#     ker = kernel(x,y,u) and
#     keru = partial kernel / partial u.

  Real mesh(nmesh),V(1),x(nx),f(nx),fV(nx,1)    # V(nmesh-k),fV(nx,nmesh-k)
  Integer k,nmesh,nx

# The following declaration is specific to k = 4 splines.

  Real xgq(3),wgq(3),xx(3),B(3,4,200),ewe,ker,keru
  Integer i,j,mgq,l,ix
  Logical First
  Data First / .True. /

  If ( nmesh-k > 200 )    # Need more local space.
    { Call SETERR("INTGRL - nmesh-k .gt. 200",27,1,2) }
  If ( k != 4 )
    { Call SETERR("INTGRL - k .ne. 4",17,2,2) }

  mgq = k-1    # Use k-1 point Gaussian-quadrature rule on each interval.

  If ( First )    # Only get GQ rule once, its expensive.
    {  Call GQM11(mgq,xgq,wgq) }    # The Gaussian Quadrature rule.

  Do i = k, nmesh-k    # Do integral interval by interval.
    {
    Do j = 1, mgq    # G.Q. points on (mesh(i), mesh(i+1)).
      { xx(j) = 0.5*(mesh(i+1)+mesh(i)) + 0.5*(mesh(i+1)-mesh(i))*xgq(j) }
    If ( First )    # Only get B-spline basis once, its expensive.
      {  Call BSPLN(k,mesh,nmesh,xx,mgq,i,B(1,1,i)) }

    Do j = 1, mgq
      {
      ewe = 0    # Get Sum V()*B()(xx).
      Do l = 1, k { ewe = ewe + V(i+l-k)*B(j,l,i) }

      Do ix = 1, nx
        {
        Call KERNEL(x(ix),xx(j),ewe,ker,keru)    # Get kernel and partial.

        f(ix) = f(ix) + ker * 0.5*(mesh(i+1)-mesh(i))*wgq(j)
```

```
            Do l = 1, k
              {
              fV(ix,i+l-k) = fV(ix,i+l-k) + B(j,l,i)*keru *
                                0.5*(mesh(i+1)-mesh(i))*wgq(j)
              }
            }
          }
        }

      First = .False.

      Return
      End
```

and the `KERNEL` procedure for evaluating the kernel is

```
      Subroutine KERNEL(x,y,u,ker,keru)

    # To evaluate the kernel Exp(x-y)*u(y) and its partial wrt. u.

      Real x,y,u,ker,keru

      keru = Exp(x-y)
      ker = keru*u

      Return

      End
```

The output for the above program unit is

```
     Error in U(x,  1.00e-02 ) =  4.74e-05
     Error in U(x,  1.49e-01 ) =  3.25e-03
     Error in U(x,  3.95e-01 ) =  5.97e-04
     Error in U(x,  9.69e-01 ) =  2.56e-03
     Error in U(x,  1.00e+00 ) =  2.37e-03
     used     1716 /    10000 of the stack allowed.
```

and we can see that the problem has been solved accurately. The run-time for the above example was 42.1 seconds.

**Example 9 - Automatic Mesh Refinement**

Assume that you have the transient **pde**

$$u_t = u_{xx} + c\, u_x \qquad \text{for} \quad x \,\varepsilon\, (0,1) \tag{A4.24}$$

and you want its equilibrium solution to a given absolute accuracy, `tol`. If we choose the **bc**s so that the solution at time $t = \infty$ is $e^{-cx}$, then for $c$ large, we have a pretty nasty problem. If the constant $c$ in (A4.24) is large, the number of mesh points needed to get the desired accuracy is not obvious at all. The following pseudo-code will automatically determine the mesh for you. A somewhat more complicated version was used in [56] to automatically refine the grid in a static problem with two boundary layers. It starts from an initial, crude guess and keeps halving the mesh spacing until the error estimate is small enough.

```
Get an initial mesh x0 of length nx0.
Get an initial approximation u0 to the solution on the mesh x0.

error = ∞
While ( error > tol )
    {
    Get mesh x1 from mesh x0 by halving the mesh spacing.
    Get a spline fit u1 to u0 on the mesh x1.

    Solve the pde to time ∞ starting from u1, using mesh x1.

    Estimate the error in u0, given u0 and u1.

    x0 = x1; nx0 = nx1; u0 = u1
    }
```

assuming that all the arrays are dimensioned large enough. To halve the mesh spacing of a mesh `x0`, the PORT Library routine `LUMB` can be used:

```
Call LUMB(x0,nx0,3,k,x1,nx1).
```

The PORT Library routine `EEBSF` uses two approximations to a function by splines to estimate the error in the spline approximation:

```
error = EEBSF(k, x0,nx0,u0, x1,nx1,u1).
```

The only tricky part of the above pseudo-code is in getting a spline fit to `u0` on the mesh `x1`. The code for doing it, assuming that `xx` and `ewe` are arrays sufficiently long, uses three Port Library calls

```
Call LUMD(x1,nx1,k,xx,nxx)    # Fitting points for refinement.
Call SPLNE(k,x0,nx0,u0,xx,nxx,ewe)    # Crude u1 on xx.
Call DL2SF(xx,ewe,nxx,k,x1,nx1,u1)    # Fit u1 to u0 on x1.
```

to evaluate `u0` at $k$ equally spaced data points ( `xx` ) per mesh interval in `x1`, putting the results in `ewe`. That discrete data is then fitted by `u1` on the mesh `x1` using `DL2SF`.

The main program uses one routine from section 8

- `POSTX` for printing the run-time statistics of `POST`.

The main program to solve (A4.24) with $c = 50$ and `tol` $= 10^{-2}$ is

```
 Real tstop,V(1),dt,x(100),U(100),xc(100),Uc(100),
      xx(1000),ewe(1000),R1MACH,err,errr, EEBSF
 Real errpar(2)
 Integer k,nx,ndx,Nu,Nv,nxc,nxx,i,I1MACH
 External AF,BC, POSTD,HANDLE

 Common / PARAM / c; Real c

 Common / CSTAK / Ds(2000); Double Precision Ds    # The PORT Stack.

 Call ISTKIN(2000,4)    # Initialize the PORT Library stack length.

 c = 50

 Nu = 1; Nv = 0

 errpar(1) = 1e-1; errpar(2) = 1e-1

 k = 4; ndx = 8

 Call UMB(0e0,1e0,ndx,k,xc,nxc)

 Call SETR(nxc-k,0e0,Uc)    # Initial conditions for Uc.

 err = R1MACH(2)    # Infinity.

 While ( err > 1e-2 )
   {
    Call LUMB(xc,nxc,3,k,x,nx)    # Halve the crude x.

    Call LUMD(x,nx,k,xx,nxx)    # Fitting points for refinement.
    Call SPLNE(k,xc,nxc,Uc,xx,nxx,ewe)    # Uc on xx.
    Call DL2SF(xx,ewe,nxx,k,x,nx,U)    # Fit U to Uc on mesh.

   tstop = 1/R1MACH(4); dt = 1e-6

   i = nx-2*(k-1)
   iwunit = I1MACH(2)
   Write(iwunit,9000) i
9000 Format(" Solving for ndx =",I3)

    Call POST (U,Nu,k,x,nx,V,Nv,
               0e0,tstop,dt,
               AF,BC, POSTD,
               errpar,
               HANDLE)

    Call POSTX    # Get run-time statistics.

   err = EEBSF(k,xc,nxc,Uc, x,nx,U)    # Error estimate for Uc.

   errr = err/16    # Error estimate for U.
   iwunit = I1MACH(2)
   Write(iwunit,9001) err,errr
```

```
9001 Format(" error estimates Uc =",1p1e10.2,"  and U =",1p4e10.2)

       nxc = nx
       Do i = 1, nx { xc(i) = x(i) }
       Do i = 1, nx-k { Uc(i) = U(i) }
       }

     Stop

     End
```

with the body of the AF procedure

```
     Do i = 1, nx
       {
       A(i,1) = Ux(i,1)+c*U(i,1); AUx(i,1,1) = 1; AU(i,1,1) = c
       f(i,1) = Ut(i,1)
       fUt(i,1,1) = 1
       }
```

The body of the BC procedure is

```
     B(1,1) = U(1,1)-1
     B(1,2) = U(1,2)-Exp(-c)

     BU(1,1,1) = 1
     BU(1,1,2) = 1
```

and the body of the HANDLE procedure is

```
     Common / TIME / tt; Real tt

     Real eU, EESFF
     Integer I1MACH
     External UOFX

     If ( t0 == t )
       {
       iwunit = I1MACH(2)
       Write(iwunit,9002) t
    9002 Format(" Restart for t =",1p4e10.2)
       Return
       }

     tt = t; eU = EESFF(k,x,nx,U,UOFX)

     iwunit = I1MACH(2)
     Write(iwunit,9003) t,eU
    9003 Format(" Error in U(x, ",1p1e10.2," ) =",1p4e10.2)
```

The body of the UOFX procedure is

```
     Do i = 1, nx { U(i) = Exp(-c*x(i)) }
```

The output for the above program unit is

```
Solving for ndx = 15
Error in U(x,   1.00e-06 ) =  3.31e-01
Error in U(x,   2.62e-04 ) =  2.14e-01
Error in U(x,   1.05e-03 ) =  8.61e-02
Error in U(x,   2.93e-03 ) =  4.20e-02
Error in U(x,   1.09e-02 ) =  3.35e-02
Error in U(x,   1.15e-01 ) =  3.32e-02
Error in U(x,   2.28e+01 ) =  3.32e-02
Error in U(x,   5.96e+03 ) =  3.32e-02
Error in U(x,   1.56e+06 ) =  3.32e-02
Error in U(x,   8.39e+06 ) =  3.32e-02
 post(j,f,ts,ss,nit,nd,nf,r) =   44   44   10   30   44    0    0    0
error estimates Uc =  1.04e+00  and U =  6.51e-02
Solving for ndx = 29
Error in U(x,   1.00e-06 ) =  3.33e-02
Error in U(x,   2.51e-04 ) =  1.43e-02
Error in U(x,   1.39e-03 ) =  4.60e-03
Error in U(x,   1.02e-02 ) =  4.50e-03
Error in U(x,   3.37e-01 ) =  4.46e-03
Error in U(x,   8.56e+01 ) =  4.46e-03
Error in U(x,   2.24e+04 ) =  4.46e-03
Error in U(x,   5.85e+06 ) =  4.46e-03
Error in U(x,   8.39e+06 ) =  4.46e-03
 post(j,f,ts,ss,nit,nd,nf,r) =   35   35    9   27   35    0    0    0
error estimates Uc =  3.95e-02  and U =  2.47e-03
Solving for ndx = 57
Error in U(x,   1.00e-06 ) =  4.40e-03
Error in U(x,   1.76e-04 ) =  8.48e-04
Error in U(x,   2.39e-03 ) =  3.69e-04
Error in U(x,   1.37e-01 ) =  3.73e-04
Error in U(x,   3.54e+01 ) =  3.73e-04
Error in U(x,   9.26e+03 ) =  3.73e-04
Error in U(x,   2.42e+06 ) =  3.73e-04
Error in U(x,   8.39e+06 ) =  3.73e-04
 post(j,f,ts,ss,nit,nd,nf,r) =   27   27    8   24   27    0    0    0
error estimates Uc =  5.16e-03  and U =  3.22e-04
```

and we can see that the problem has been accurately solved. The run-time for the above example was 184.0 seconds.

The output shows that although the error in the initial approximation, $u \equiv 1$, was large, $\approx 1$, the solution rapidly approached equilibrium with an error below 10%. The two successive refinements converge even more rapidly to equilibrium and to the true solution. Note also, that the estimate for the error in the crude solution is very safe, at every stage. But the estimates for the refined solution are not so reliable. They get better as the mesh spacing gets smaller, but the middle estimate is low by a factor of two or so. In general, the crude estimates are quite trustworthy, but the refined estimates are not, see [45].

The output of `POSTX` above shows that the total number of Newton iterations for each of the refinements were 44, 35 and 27. The run-time of each solution is proportional to the product of the number of Newton iterations and the number of mesh points. Since the number of mesh points is doubling with each refinement, we see the run-times for the refinements were proportional to 44, 70 and 54.

Thus, a nice feature of the above automatic mesh selection scheme is that it is only about twice as expensive as simply getting the solution once on the "right", but unknown, grid. It could have stopped at the second to last grid, but the reliable crude error estimates couldn't tell that until it got the solution on the last grid.

**Example 10 - Estimating the Error in the Computed Solution.**

The error estimation results of (4.37)-(4.38) are now used to estimate the error in the numerical solution of example 1. The main program solves the problem using the original, crude mesh, then on a finer mesh, and finally on the crude mesh with tighter `errpar` control. The elements of the right hand side of the estimate (4.37) are then computed. The only change to the program for example 1 is in the main program

```
 Real tstop,V(1),dt,x(100),U(100),xh(100),Uh(100),Ue(100),
      err, EEBSF
 Real errpar(2)
 Integer k,nx,ndx,Nu,Nv,nxh,i,I1MACH
 External AF,BC, POSTD,HANDLE

 Common / CSTAK / Ds(2000); Double Precision Ds    # The PORT Stack.

 Call ISTKIN(2000,4)    # Initialize the PORT Library stack length.

 Nu = 1; Nv = 0

 errpar(1) = 0; errpar(2) = 1e-2

 k = 4; ndx = 4; tstop = 1; dt = 1e-2

 Call UMB(0e0,1e0,ndx,k,x,nx)    # Crude mesh.

 Call SETR(nx-k,1e0,U)    # Initial conditions for U.

 iwunit = I1MACH(2)
 Write(iwunit,9000)
9000 Format(" Solving on crude mesh using errpar.")

 Call POST (U,Nu,k,x,nx,V,Nv,
            0e0,tstop,dt,
            AF,BC, POSTD,
            errpar,
            HANDLE)

 Call POSTX    # Get run-time statistics.

 Call UMB(0e0,1e0,2*ndx-1,k,xh,nxh)    # Halve the mesh spacing.

 Call SETR(nxh-k,1e0,Uh)    # Initial conditions for Uh.

 dt = 1e-2

 iwunit = I1MACH(2)
 Write(iwunit,9001)
9001 Format(" Solving on refined mesh using errpar.")

 Call POST (Uh,Nu,k,xh,nxh,V,Nv,
            0e0,tstop,dt,
            AF,BC, POSTD,
            errpar,
            HANDLE)

 Call POSTX    # Get run-time statistics.

 err = EEBSF(k, x,nx,U, xh,nxh,Uh)    # Estimate U error.
 iwunit = 6
 Write(iwunit,9002) err
9002 Format(" U error from U and Uh =",1p4e10.2)
```

```
 Call SETR(nx-k,1e0,Ue)    # Initial conditions for Ue.

 dt = 1e-2; errpar(1) = errpar(1)/10; errpar(2) = errpar(2)/10

 iwunit = I1MACH(2)
 Write(iwunit,9003)
9003 Format(" Solving on crude mesh using errpar/10.")

 Call POST (Ue,Nu,k,x,nx,V,Nv,
            0e0,tstop,dt,
            AF,BC, POSTD,
            errpar,
            HANDLE)

 Call POSTX    # Get run-time statistics.

 err = 0; Do i = 1, nx-k { err = Max(err,Abs(U(i)-Ue(i))) }
 iwunit = 6
 Write(iwunit,9004) err
9004 Format(" U error from U and Ue =",1p4e10.2)

 Stop

 End
```

The output for the above program unit is

```
Solving on crude mesh using errpar.
Error in U(x,  1.00e-02 ) =  1.47e-05
Error in U(x,  2.28e-01 ) =  1.89e-03
Error in U(x,  6.97e-01 ) =  4.92e-04
Error in U(x,  1.00e+00 ) =  4.91e-03
 post(j,f,ts,ss,nit,nd,nf,r) =   30   30    4   15   30    0    0    0
Solving on refined mesh using errpar.
Error in U(x,  1.00e-02 ) =  1.41e-05
Error in U(x,  2.54e-01 ) =  2.22e-03
Error in U(x,  7.68e-01 ) =  5.50e-04
Error in U(x,  1.00e+00 ) =  3.62e-03
 post(j,f,ts,ss,nit,nd,nf,r) =   30   30    4   15   30    0    0    0
U error from U and Uh =  1.38e-03
Solving on crude mesh using errpar/10.
Error in U(x,  1.00e-02 ) =  1.47e-05
Error in U(x,  7.89e-02 ) =  3.57e-04
Error in U(x,  1.88e-01 ) =  1.64e-04
Error in U(x,  4.35e-01 ) =  2.34e-04
Error in U(x,  8.07e-01 ) =  2.40e-05
Error in U(x,  1.00e+00 ) =  2.69e-04
 post(j,f,ts,ss,nit,nd,nf,r) =   68   68    6   34   68    0    0    0
U error from U and Ue =  5.45e-03
```

and we can see that the problem has been accurately solved. The run-time for the above example was 37.0 seconds.

The sum of the two error terms is a remarkably good approximation to the true error in the computed crude solution. It also shows that the dominant term in the error is from the time discretization, not the spatial discretization, for this problem.

The above error estimation scheme can be applied to the value of the solution at any point in time, not just `tstop`.

The output of `POSTX` above shows that the crude solution cost 30 Newton iterations, the refined mesh solution cost 30 and the refined `errpar` solution cost 68. Since the run-time of a solution is proportional to the product of the number of Newton iterations and the number of mesh points, we see that the solution times were proportional to 30, 60 and 68.

The computation of the error estimates thus costs roughly four times what the crude solution did. The overall expense was thus five times the cost of the basic solution. It is a *really* good idea to apply the above scheme whenever working on a new problem and concern about the accuracy of the computed solution is reasonable. Once the mesh and `errpar` have been chosen appropriately and confirmed, production runs can dispense with the error estimates and their expense. But some error estimation is a good idea. The alternative is a pile of cheap numbers that may be garbage. Good piles of numbers cost more to obtain.

# Appendix 5

## Routine, Common and Error State Summary

This appendix summarizes the calling sequences for the routines of POST, the contents of the relatively public Common regions and the error states. It is terse and meant to serve as a reference guide, not as a tutorial. The top level of POST is

```
Call POST(U,Nu,k,x,nx,V,Nv,
          tstart,tstop,dt,
          AF,BC,D,
          errpar,
          HANDLE)
```

The AF procedure is of the form

```
Subroutine AF(t,x,nx,U,Ux,Ut,Utx,Nu,V,Vt,Nv,
              A,AU,AUx,AUt,AUtx,AV,AVt,
              f,fU,fUx,fUt,fUtx,fV,fVt)
```

the BC procedure is of the form

```
Subroutine BC(t,L,R,U,Ux,Ut,Utx,Nu,V,Vt,Nv,
              B,BU,BUx,BUt,BUtx,BV,BVt)
```

the DEE procedure is of the form

```
Subroutine DEE(t,k,x,nx,
               U,Ut,Nu,nxmk,V,Vt,Nv,
               D,DU,DUt,DV,DVt)
```

and the HANDLE procedure has the form

```
Subroutine HANDLE(t0,U0,V0,t1,U1,V1,Nu,nxmk,Nv,k,x,nx,dt,tstop)
```

The vacuous ( $n_u = 0$ ) AF procedure is POSTA.

The vacuous ( $n_u = 0$ ) BC procedure is POSTB.

The vacuous ( $n_v = 0$ ) DEE procedure is POSTD.

The "Return-End" HANDLE procedure is POSTH.

The statistics printing procedure is POSTX.

The basic knob twiddling routine for POST is

```
Call POSTW(j,f,r,i,l)
```

The following table summarizes the values that can be set by POSTW

| Name | j | Default | Set to |
|---|---|---|---|
| theta | 1 | 1 | f |
| beta | 2 | 1 | f |
| gamma | 3 | 1 | f |
| delta | 4 | 0 | f |
| hfract | 1001 | 1 | r |
| egive | 1002 | 100 | r |
| kj | 2001 | 0 | i |
| minit | 2002 | 10 | i |
| maxit | 2003 | 50 | i |
| kmax | 2004 | 10 | i |
| kinit | 2005 | 4 | i |
| mmax | 2006 | 15 | i |
| mgq | 2007 | k-1 | i |
| saveb | 2008 | 0 | i |
| xpoly | 3001 | False | l |
| erputs | 3002 | False | l |
| N(i) | 4000+i | – | i |

The procedural knob twiddler is

```
Call POSTR(U,Nu,k,x,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,D,
           ERROR,errpar,
           INMI,SCALE,
           HANDLE)
```

where the `ERROR` procedure has the form

```
Logical Function ERROR(U,Nu,nxmk,k,x,nx,V,Nv,t,dt,
                       errpar,
                       erputs,
                       eU,eV)
```

the `SCALE` procedure has the form

```
Subroutine SCALE(i,j,A,m,n)
```

and the `INMI` routine has the form

```
Procedure INMI(Nu,Nv,nxmk,k,x,nx,
               t,dt,
               Uold,Vold,
               U,Ut,V,Vt)
```

If the knobs all require twiddling concurrently, there is

```
Call POST4(U,Nu,k,x,nx,V,Nv,
           tstart,tstop,dt,
           AF,BC,D,
           theta,KeepJac,Minit,Maxit,
           mgq,
           beta,gamma,delta,N,mmax,hfract,egive,
           kmax,xpoly,kinit,
           ERROR,errpar,erputs,
           INMI,SCALE,
           HANDLE)
```

The LPLM maps are structured as follows

```
Subroutine LPLMX(xi,nx,V,Nv,x,xv)
Subroutine LPLM(xi,nx,V,Nv,x,xxi,xxiv,xv,xvv,xt,xtv)
Subroutine LPLMG(Nv,x,V)
Subroutine LPLMI(Nv,V,x,nx,xi)
Real Function LPLMT(t1,V1,Nv,t2,V2,f,dt)
```

The POST coordinate mapping routines for AF are

```
Call POSTU(xi,x,xt,xxi,xv,Vt,nx,Nv,
           Ux,Ut,Nu,
           Ax,Fx)
```

to map from internal to user coordinates, and

```
Call POSTI(xi,x,xt,xxi,xv,xtv,xxiv,xvv,nx,
           Ux,Ut,Nu,
           V,Vt,Nv,iV,Nvm,
           A,Ax,AU,AUx,AUt,AUtx,AV,AVt,
           F,Fx,FU,FUx,FUt,FUtx,FV,FVt)
```

to map from user to internal coordinates.

For mapping BC from internal to user coordinates there is

```
Call POSTV(xi,x,xt,xxi,xv,Vt,Nv,
           Ux,Ut,Nu,
           Bx)
```

and to map BC from user back into internal coordinates

```
Call POSTJ(xi,x,xt,xxi,xv,xtv,xxiv,xvv,
           Ux,Ut,Nu,
           V,Vt,Nv,iV,Nvm,
           B,Bx,BU,BUx,BUt,BUtx,BV,BVt)
```

**Common Regions**

When the user cannot evaluate any of AF, BC or DEE, that fact can be signaled to POST via

```
Common / POSTF / Failed; Logical Failed
```

Finally, the SCALE routine must set

```
Common / A9OSTS / imem(4); Integer imem
```

**Error States.**

This section provides a list of the error states [25] that may be encountered when using POST. Some interpretation of these error messages is made to aid the user in finding bugs (if they exist) in the user-supplied code AF, BC, DEE or HANDLE. For each level of (entry to) POST, the error message and number for a given error state is the same, always reflecting the error from the bottom layer of POST, namely POST4 (see section 8). The list of error states below, along with interpretation, is the complete set of error states for the POST package as obtained from the lowest level of POST, namely POST4.

There are many internal variables of POST that may be controlled by the subroutine POSTW. Thus, there are many more ways to call POST with bad data than just the obvious ones involving data in the calling sequence for POST. The list of error states given below is complete for POST/POSTW, and therefore involves variables not mentioned in the calling sequence for POST. If you are only using POST, and not POSTW as well, then the error states mentioning variables not in the POST call can not occur. So if you bump into an error state below that mentions an undescribed or unknown variable, simply ignore it, it cannot happen to you.

1 - POST4 - Nu.lt.0.

2 - POST4 - Nv.lt.0.

3 - POST4 - Nu=Nv=0.

4 - POST4 - k.lt.2 when Nu positive.

5 - POST4 - nx.lt.2*k when Nu positive.

6 - POST4 - Input value of dt is 0. The user-chosen value for the time-step dt is too small, that is, tstart+dt ≡ tstart.

7 - POST4 - Input value of dt has the wrong sign. dt and tstop-tstart must have the same sign.

8 - POST4 - theta not in (0,1).

9 - POST4 - KeepJac not one of (0,...,5).

10 - POST4 - Minit.lt.1.

11 - POST4 - Maxit.lt.1.

12 - POST4 - mgq.lt.1 when Nu is positive.

13 - POST4 - kmax.lt.1.

14 - POST4 - kinit.lt.1.

15 - POST4 - x(1) not of multiplicity k.

16 - POST4 - x(nx) not of multiplicity k.

17 - POST4 - x not monotone increasing.

18 - POST4 - SCALE failed to initialize Common /A9OSTS/. The subroutine SCALE should have set the Common region, but did not. The only way POST can tell that the SCALE routine is alive is by the Common region being set. Putting any non-zero Integer value there will suffice.

19 - POST4 - beta .le.0.

20 - POST4 - gamma .le.0.

21 - POST4 - delta .lt.0.

22 - POST4 - beta+gamma-delta.le.0.

23 - POST4 - mmax.lt.kmax+2.

24 - POST4 - n(1).lt.1.

25 - POST4 - N is not monotone increasing.

26 - POST4 - hfract.le.0.

27 - `POST4` - egive .lt. 0.

1000 - `POST4` - dt=0. Recoverable. The time-step has become too small. The problem may be very badly scaled, that is units like light-years and micro-grams are being used simultaneously. Another cause is too small an accuracy requirement, like errpar(2)=0 when the solution is exceedingly small.

1001 - `POST4` - dt=0 Returned by `HANDLE`. Recoverable. Handle lowered dt and it became too small.

1002 - `POST4` - dt Returned by `HANDLE` has wrong sign. Recoverable.

1003 - `POST4` - Cannot raise dt in `HANDLE` when Failed. Recoverable.

1004 - `POST4` - e(i).le.0 Returned by `ERROR`. Recoverable. The error request is too small. Having a relative error request on a variable going to 0 can cause this.

1005 - `POST4` - Dirichlet boundary conditions are overdetermined. Recoverable. There are too many Dirichlet **bc**s.

1006 - `POST4` - Mixed boundary conditions are overdetermined. Recoverable. There are too many mixed **bc**s.

1007 - `POST4` - Improper BCs. Recoverable. The **bc**s and `pde`s do not match properly, see section 7.

1008 - `POST4` - Pde system not in minimal order form. Recoverable. The `pde` can have derivatives removed from it, see section 8.

1009 - `POST4` - Too few boundary conditions. Recoverable.

1010 - `POST4` - Too many boundary conditions. Recoverable.

1011 - `POST4` - mgq=k-1 and Order(i,.)=0. Recoverable. Must have mgq=k when one of the `pde`s is of zero order.

1012 - `POST4` - Pde(i) is vacuous. Recoverable. There is no $i^{th}$ `pde`.

1013 - `POST4` - `AF` failure. Recoverable. Failed = True occurred in `AF` so often that dt went to 0.

1014 - `POST4` - `BC` failure. Recoverable. Failed = True occurred in `BC` so often that dt went to 0.

1015 - `POST4` - `D` failure. Recoverable. Failed = True occurred in `D` so often that dt went to 0.

1016 - `POST4` - Singular Dirichlet boundary conditions. Recoverable. The Dirichlet **bc**s were singular so frequently that dt went to 0.

1017 - `POST4` - Singular mixed boundary conditions. Recoverable. The mixed **bc**s were singular so frequently that dt went to 0.

1018 - `POST4` - Singular `pde` Jacobian. Recoverable. The Jacobian for the `pde` was singular so frequently that dt went to 0.

1019 - `POST4` - Singular `ODE` Jacobian. Recoverable. The Jacobian for the `ODE` was singular so frequently that dt went to 0.

1020 - `POST4` - Too many Newton iterations predicted. Recoverable. The number of Newton iterations was predicted to be too large so often that dt went to 0. Probable cause is a bad Jacobian, see next error state.

1021 - `POST4` - Too many Newton iterations needed. Recoverable. Too many Newton iterations were needed so often that dt went to 0. Probable cause is an incorrectly computed Jacobian. Another possible cause is that Minit and/or Maxit are too small. Yet another possible cause is a very badly conditioned Jacobian. The default row and column scaling provided in `POST` provides substantial protection against this, but it is not bomb-proof. Further possible causes: too stringent an error request or a mesh that is too non-uniform.

1 - `POSTW` - j out of bounds.

The following error states cannot happen if there is no over-writing in the user's code. Their detection is a useful debugging aid.

1 - `A90STA` - `GLSBP-BC` failure. The proper ordering of the `pde`s and **bc**s is only done when necessary. The `GLSBC` routine checks the ordering and has found that `GLSBP` failed. Since this is "impossible", it implies an overwriting error in the user's code.

1 - `A90STC` - Singular Jac in `BNDFB`. The stored `pde` Jacobian has proven to be singular during the solution phase of the linear system. Since this error is caught during the matrix factorization, catching such an error in the solution phase is "impossible". Typical cause of this error is the user over-writing arrays.

1 - `A90STD` - Singular DJac in `QRQTB`. Same problem as above but for the `ODE` Jacobian instead.

# Appendix 6

## Permuted Index


| | | |
|---|---|---|
| AF evaluates | **a** and **f** in the **pde**. | 19-20 |
| Common regions POSTF and | A9OSTS. | 22-23 |
| Automatic mesh refinement for | accuracy; Example 9. | 14-15 |
| POSTA: the vacuous, nu=0 | AF. | 18 |
| Trouble in | AF, BC or DEE: POSTF. | 22-23 |
| **pde**. | AF evaluates **a** and **f** in the | 19-20 |
| ISTKGT | allocates space on the Port Library stack. | A4:16 |
| bracket blocks of code so stack | allocations are local. ENTER-LEAVE | A4:16 |
| MOVEFR copies one Real array into | another. | A4:16 |
| MOVEFR copies one Real | array into another. | A4:16 |
| SETR sets a Real | array to a given constant. | A4:2 |
| | Automatic mesh refinement for accuracy; Example 9. | 15 |
| BC evaluates | **b** in the **bc**s. | 20 |
| LBE: linearized | backwards Euler. | 46-48 |
| NBE nonlinear | backwards Euler. | 46-48 |
| BSPLN evaluates the B-spline | basis functions at a set of points. | A4:12 |
| POSTB: the vacuous, nu=0 | BC. | 18 |
| | BC evaluates **b** in the **bc**s. | 20-21 |
| POSTV: | **bc** mapping, from internal to User coordinates. | 27-28 |
| POSTJ: | **bc** mapping, from user to Internal coordinates. | 28-29 |
| | **bc** mapping: POSTV and POSTJ. | 27-30 |
| Trouble in AF, | BC or DEE: POSTF. | 22-23 |
| | **bc** placement. | 35-37 |
| BC evaluates **b** in the | **bc**s. | 20-21 |
| A **pde** with periodic | **bc**s; Example 4. | 8-9 |
| Counting | **bc**s in hyperbolic equations. | 13-14 |
| initial conditions that conflict with the **pde**, | **bc**s or **ode**s. Handling | 14 |
| Reduction of | **bc**s to canonical form. | 34-35 |
| ENTER-LEAVE bracket | blocks of code so stack allocations are local. | A4:16 |
| A first order hyperbolic **pde** with two | boundary conditions. | 13-14 |
| A moving | boundary layer; Example 7. | 11-12 |
| ENTER-LEAVE | bracket blocks of code so stack allocations are local. | A4:16 |
| DL2SF fits discrete data by a | B-spline. | A4:1-2 |
| BSPLN evaluates the | B-spline basis functions at a set of points. | A4:12 |
| EESFF estimates the error in a | B-spline fit to a given function. | A4:5 |
| EEBSF estimates the error in two | B-spline fits. | A4:38 |
| SPLNE evaluates a | B-spline given the coefficients and mesh. | A4:4 |
| UMB creates a uniform | B-spline mesh. | A4:2 |
| LUMB makes a | B-spline mesh from a union of uniform meshes. | A4:16 |
| a set of points. | BSPLN evaluates the B-spline basis functions at | A4:12 |
| is same as LPLM, but with fewer derivatives | calculated. LPLMX | 23 |
| LPLMI: | Calculates the Inverse of an LPLM mapping. | 24 |
| Reduction of **bc**s to | canonical form. | 34-35 |
| A Hyperbolic | Caveat. | 11 |
| WRAPUP | checks for error states and prints stack usage. | A4:2 |
| | CN: Crank-Nicholson. | 46-48 |
| SPLNE evaluates a B-spline given the | coefficients and mesh. | A4:4 |
| restricts ***dt*** so mapped mesh points do not | collide. LPLMT | 24 |
| | Collocation methods. | A3:2-3 |
| | Common regions POSTF and A9OSTS. | 22-23 |
| system. Computational | complexity: Cost of creating and solving Jacobian | 39 |
| Jacobian system. | Computational complexity: Cost of creating and solving | 39 |
| A first order hyperbolic **pde** with two boundary | conditions. | 13-14 |

| | | |
|---|---|---|
| DL2SFF. Initial | conditions for **u**: L2SFF, SETR, | A4:1-2 |
| or **ode**s. Handling initial | conditions that conflict with the **pde**, **bc**s | 14 |
| Handling initial conditions that | conflict with the **pde**, **bc**s or **ode**s. | 14 |
| SETR sets a Real array to a given | constant. | A4:2 |
| R1MACH reports Real machine | constants. | A4:24 |
| I1MACH reports Integer machine | constants. | A4:4 |
| INTRVR finds the mesh interval | containing a given point. | A4:12 |
| Galerkins method forces fluxes to be | continuous. | 9 |
| Physicists expect fluxes to be | continuous. | 9 |
| POSTW: | control over numerical variables used by POST. | 44-46 |
| POSTR: | control over the routines used by POST. | 40-41 |
| SCALE | controls the scaling of the Jacobian. | 43-44 |
| ERROR | controls the time evolution error. | 41-42 |
| POSTI: | converts the **pde** from $x$ to $\xi$ coordinates. | 26-27 |
| POSTU | converts **u** from $\xi$ to $x$ coordinates. | 24-26 |
| POSTU converts **u** from $\xi$ to $x$ | coordinates. | 25-26 |
| POSTI: converts the **pde** from $x$ to $\xi$ | coordinates. | 25-27 |
| POSTV: **bc** mapping, from internal to User | coordinates. | 27-28 |
| POSTJ: **bc** mapping, from user to Internal | coordinates. | 28-29 |
| MOVEFR | copies one Real array into another. | A4:16 |
| Computational complexity: | Cost of creating and solving Jacobian system. | 39 |
| | Counting **bc**s in hyperbolic equations. | 13-14 |
| A | coupled **pde-ode** system; Example 3. | 8 |
| A fourth order **pde**, $\nabla^4$, reduced to two | coupled second order **pde**s. | 12-13 |
| A | coupled system of **pde**s; Example 2. | 7-8 |
| CN: | Crank-Nicholson. | 46-48 |
| UMB | creates a uniform B-spline mesh. | A4:2 |
| Computational complexity: Cost of | creating and solving Jacobian system. | 39 |
| DEE evaluates | **d** in the **ode**. | 21 |
| DL2SF fits discrete | data by a B-spline. | A4:1-2 |
| POSTD: the vacuous, nv=0 | DEE. | 18 |
| | DEE evaluates **d** in the **ode**. | 21 |
| Trouble in AF, BC or | DEE: POSTF. | 22-23 |
| POSTE: | default ERROR routine. | 41-42 |
| POSTN: | default INMI routine. | 44 |
| POSTH: | default, no output HANDLE routine. | 21 |
| POSTP: | default SCALE routine. | 39 |
| | Definition of the **pde-bc** problem. | 3 |
| | Definition of the **pde-bc-ode** problem. | 3-4 |
| LPLMX is same as LPLM, but with fewer | derivatives calculated. | 23 |
| A moving front, $\mathbf{u}_x$ | discontinuous; Example 5. | 10-11 |
| A moving shock wave, **u** | discontinuous; Example 6. | 11 |
| DL2SF fits | discrete data by a B-spline. | A4:1-2 |
| Time | discretization. | 31-33 |
| Spatial | discretization. | 34-39 |
| LUMD makes a Locally Uniform Mesh of | Distinct points. | A4:38 |
| | DL2SF fits discrete data by a B-spline. | A4:1-2 |
| conditions for **u**: L2SFF, SETR, | DL2SFF. Initial | A4:1-2 |
| LPLM | does Linearly parameterized Piecewise Linear Mapping. | 23-24 |
| | Double precision DPOST. | 22 |
| Scratch space used: POST and | DPOST. | 19 |
| Double precision | DPOST. | 22 |
| LPLMT restricts | *dt* so mapped mesh points do not collide. | 24 |
| | EEBSF estimates the error in two B-spline fits. | A4:38 |
| given function. | EESFF estimates the error in a B-spline fit to a | A4:5 |
| stack allocations are local. | ENTER-LEAVE bracket blocks of code so | A4:16 |
| A simple **pde**, the heat | equation; Example 1. | 7 |
| Counting **bc**s in hyperbolic | equations. | 13-14 |
| ERROR controls the time evolution | error. | 41-42 |

| | | |
|---|---|---|
| | ERROR controls the time evolution error. | 41-42 |
| | Error estimation for reliability; Example 10. | 15-16 |
| EESFF estimates the | error in a B-spline fit to a given function. | A4:5 |
| EEBSF estimates the | error in two B-spline fits. | A4:38 |
| | Error options: POSTE or write your own. | 41-42 |
| POSTE: default | ERROR routine. | 41-42 |
| SETERR sets an | error state for the Port Library. | A4:12 |
| | Error states. | A5:4-6 |
| WRAPUP checks for | error states and prints stack usage. | A4:2 |
| function. EESFF | estimates the error in a B-spline fit to a given | A4:5 |
| EEBSF | estimates the error in two B-spline fits. | A4:38 |
| Error | estimation for reliability; Example 10. | 15-16 |
| LBE: linearized backwards | Euler. | 46-48 |
| NBE nonlinear backwards | Euler. | 46-48 |
| SIE: Semi-implicit | Euler. | 47-48 |
| AF | evaluates **a** and **f** in the **pde**. | 19-20 |
| SPLNE | evaluates a B-spline given the coefficients and mesh. | A4:4 |
| BC | evaluates **b** in the **bc**s. | 20-21 |
| DEE | evaluates **d** in the **ode**. | 21 |
| points. BSPLN | evaluates the B-spline basis functions at a set of | A4:12 |
| ERROR controls the time | evolution error. | 41-42 |
| A simple **pde**, the heat equation; | Example 1. | 7 |
| Error estimation for reliability; | Example 10. | 15-16 |
| A coupled system of **pde**s; | Example 2. | 7-8 |
| A coupled **pde-ode** system; | Example 3. | 8 |
| A **pde** with periodic **bc**s; | Example 4. | 8-9 |
| A moving front, $\mathbf{u}_x$ discontinuous; | Example 5. | 10-11 |
| A moving shock wave, **u** discontinuous; | Example 6. | 11 |
| A moving boundary layer; | Example 7. | 11-12 |
| An integro-**pde**; | Example 8. | 12 |
| Automatic mesh refinement for accuracy; | Example 9. | 15 |
| Physicists | expect fluxes to be continuous. | 9 |
| to 0 as *x* gets very negative. | EXPL: a version of EXP(*x*) which goes | A4:29 |
| negative. EXPL: a version of | EXP(*x*) which goes to 0 as *x* gets very | A4:29 |
| LBE.p: LBE with polynomial | extrapolation. | 46-48 |
| AF evaluates **a** and | **f** in the **pde**. | 19-20 |
| conditions. A | first order hyperbolic **pde** with two boundary | 13-14 |
| L2SFF Least-squares spline | fit to a function. | A4:16 |
| EESFF estimates the error in a B-spline | fit to a given function. | A4:5 |
| EEBSF estimates the error in two B-spline | fits. | A4:38 |
| DL2SF | fits discrete data by a B-spline. | A4:1-2 |
| Galerkins method forces | fluxes to be continuous. | 9 |
| Physicists expect | fluxes to be continuous. | 9 |
| Reduction of **bc**s to canonical | form. | 34-35 |
| Ratfor: a rational | Fortran. | A4:2 |
| coupled second order **pde**s. A | fourth order **pde**, $\nabla^4$, reduced to two | 12-13 |
| A moving | front, $\mathbf{u}_x$ discontinuous; Example 5. | 10-11 |
| L2SFF Least-squares spline fit to a | function. | A4:16 |
| estimates the error in a B-spline fit to a given | function. EESFF | A4:5 |
| Heavyside | function H(x,x(t)). | 10 |
| BSPLN evaluates the B-spline basis | functions at a set of points. | A4:12 |
| | Galerkins method for linear systems of **ode**'s. | 37-39 |
| | Galerkins method forces fluxes to be continuous. | 9 |
| GQM11: | Gaussian quadrature on (-1,1). | A4:34-35 |
| LPLMG: | Generates the mapping to obtain given *x* values. | 23-24 |
| | GQM11: Gaussian quadrature on (-1,1). | 39 |
| **pde**s. A fourth order **pde**, | $\nabla^4$, reduced to two coupled second order | 12-13 |
| process. | HANDLE outputs the results of the solution | 21-22 |
| POSTH: default, no output | HANDLE routine. | 21 |

| | | |
|---:|:---|:---|
| **pde**, **bc**s or **ode**s. | Handling initial conditions that conflict with the | 14 |
| A simple **pde**, the | heat equation; Example 1. | 7 |
| | Heavyside function H(x,x(t)). | 10 |
| Heavyside function | H(x,x(t)). | 10 |
| | I1MACH reports Integer machine constants. | A4:4 |
| on the Port Library stack. | ILUMB is same as LUMB, but puts the mesh | A4:16 |
| held on the stack. | IMMM modifies the multiplicity of a mesh point | A4:16 |
| SETR, DL2SFF. | Initial conditions for **u**: L2SFF, | A4:1-2 |
| **bc**s or **ode**s. Handling | initial conditions that conflict with the **pde**, | 14 |
| INMI specifies the | initial Newton method iterate. | 42-43 |
| ISTKIN | initializes the Port Library stack length. | A4:2 |
| POSTN: default | INMI routine. | 44 |
| iterate. | INMI specifies the initial Newton method | 42-43 |
| I1MACH reports | Integer machine constants. | A4:4 |
| An | integro-**pde**; Example 8. | 12 |
| POSTJ: **bc** mapping, from user to | Internal coordinates. | 28-29 |
| POSTV: **bc** mapping, from | internal to User coordinates. | 27-28 |
| INTRVR finds the mesh | interval containing a given point. | A4:12 |
| given point. | INTRVR finds the mesh interval containing a | A4:12 |
| LPLMI: Calculates the | Inverse of an LPLM mapping. | 24 |
| stack. | ISTKGT allocates space on the Port Library | A4:16 |
| length. | ISTKIN initializes the Port Library stack | A4:2 |
| INMI specifies the initial Newton method | iterate. | 42-43 |
| SCALE controls the scaling of the | Jacobian. | 43-44 |
| Computational complexity: Cost of creating and solving | Jacobian system. | 39 |
| Twiddling procedure | knobs. | 40-44 |
| Twiddling non-procedure | knobs. | 44-46 |
| | L2SFF Least-squares spline fit to a function. | A4:16 |
| Initial conditions for **u**: | L2SFF, SETR, DL2SFF. | A4:1-2 |
| A moving boundary | layer; Example 7. | 11-12 |
| | LBE: linearized backwards Euler. | 46-48 |
| LBE.p: | LBE with polynomial extrapolation. | 46-48 |
| | LBE.p: LBE with polynomial extrapolation. | 46-48 |
| L2SFF | Least-squares spline fit to a function. | A4:16 |
| ISTKIN initializes the Port Library stack | length. | A4:2 |
| SETERR sets an error state for the Port | Library. | A4:12 |
| ISTKGT allocates space on the Port | Library stack. | A4:16 |
| is same as LUMB, but puts the mesh on the Port | Library stack. ILUMB | A4:16 |
| ISTKIN initializes the Port | Library stack length. | A4:2 |
| LPLM does Linearly parameterized Piecewise | Linear Mapping. | 23-24 |
| Galerkins method for | linear systems of **ode**'s. | 37-39 |
| LBE: | linearized backwards Euler. | 46-48 |
| LPLM does | Linearly parameterized Piecewise Linear Mapping. | 23-24 |
| bracket blocks of code so stack allocations are | local. ENTER-LEAVE | A4:16 |
| LUMD makes a | Locally Uniform Mesh of Distinct points. | A4:38 |
| LPLMX is same as | LPLM, but with fewer derivatives calculated. | 23 |
| Linear Mapping. | LPLM does Linearly parameterized Piecewise | 23-24 |
| LPLMI: Calculates the Inverse of an | LPLM mapping. | 24 |
| values. | LPLMG: Generates the mapping to obtain given *x* | 23-24 |
| mapping. | LPLMI: Calculates the Inverse of an LPLM | 24 |
| not collide. | LPLMT restricts *dt* so mapped mesh points do | 24 |
| derivatives calculated. | LPLMX is same as LPLM, but with fewer | 23 |
| Mesh making - UMB, | LUMB and PUMB. | A4:1 |
| stack. ILUMB is same as | LUMB, but puts the mesh on the Port Library | A4:16 |
| uniform meshes. | LUMB makes a B-spline mesh from a union of | A4:16 |
| points. | LUMD makes a Locally Uniform Mesh of Distinct | A4:38 |
| R1MACH reports Real | machine constants. | A4:24 |
| I1MACH reports Integer | machine constants. | A4:4 |
| Mesh | making - UMB, LUMB and PUMB. | A4:1 |

| | | |
|---|---|---|
| LPLMT restricts *dt* so | mapped mesh points do not collide. | 24 |
| LPLMI: Calculates the Inverse of an LPLM | mapping. | 24 |
| POSTV: **bc** | mapping, from internal to User coordinates. | 27-28 |
| POSTJ: **bc** | mapping, from user to Internal coordinates. | 28-29 |
| does Linearly parameterized Piecewise Linear | Mapping. LPLM | 23-24 |
| **pde** | mapping: POSTU and POSTI. | 25-27 |
| **bc** | mapping: POSTV and POSTJ. | 27-30 |
| LPLMG: Generates the | mapping to obtain given *x* values. | 23-24 |
| UMB creates a uniform B-spline | mesh. | A4:2 |
| LUMB makes a B-spline | mesh from a union of uniform meshes. | A4:16 |
| INTRVR finds the | mesh interval containing a given point. | A4:12 |
| PUMB. | Mesh making - UMB, LUMB and | A4:1 |
| LUMD makes a Locally Uniform | Mesh of Distinct points. | A4:38 |
| ILUMB is same as LUMB, but puts the | mesh on the Port Library stack. | A4:16 |
| IMMM modifies the multiplicity of a | mesh point held on the stack. | A4:16 |
| LPLMT restricts *dt* so mapped | mesh points do not collide. | 24 |
| Automatic | mesh refinement for accuracy; Example 9. | 15 |
| evaluates a B-spline given the coefficients and | mesh. SPLNE | A4:4 |
| makes a B-spline mesh from a union of uniform | meshes. LUMB | A4:16 |
| Galerkins | method for linear systems of **ode**'s. | 37-39 |
| Galerkins | method forces fluxes to be continuous. | 9 |
| INMI specifies the initial Newton | method iterate. | 42-43 |
| Collocation | methods. | A3:2-3 |
| stack. IMMM | modifies the multiplicity of a mesh point held on the | A4:16 |
| | MOVEFR copies one Real array into another. | A4:16 |
| A | moving boundary layer; Example 7. | 11-12 |
| A | moving front, $\mathbf{u}_x$ discontinuous; Example 5. | 10-11 |
| A | moving shock wave, **u** discontinuous; Example 6. | 11 |
| IMMM modifies the | multiplicity of a mesh point held on the stack. | A4:16 |
| | NBE nonlinear backwards Euler. | 46-48 |
| of EXP(*x*) which goes to 0 as *x* gets very | negative. EXPL: a version | A4:29 |
| INMI specifies the initial | Newton method iterate. | 42-43 |
| POSTH: default, | no output HANDLE routine. | 21 |
| Converting a steady-state problem, | no $\mathbf{u}_t$, into a transient. | 14 |
| NBE | nonlinear backwards Euler. | 46-48 |
| Twiddling | non-procedure knobs. | 44-46 |
| LPLMT restricts *dt* so mapped mesh points do | not collide. | 24 |
| POSTA: the vacuous, | nu=0 AF. | 18 |
| POSTB: the vacuous, | nu=0 BC. | 18 |
| POSTW: control over | numerical variables used by POST. | 44-46 |
| POSTD: the vacuous, | nv=0 DEE. | 18 |
| LPLMG: Generates the mapping to | obtain given *x* values. | 23-24 |
| DEE evaluates **d** in the | **ode**. | 21 |
| Galerkins method for linear systems of | **ode**'s. | 37-39 |
| that conflict with the **pde**, **bc**s or | **ode**s. Handling initial conditions | 14 |
| MOVEFR copies | one Real array into another. | A4:16 |
| POST solves **pde**s in | one space variable. | 17-21 |
| Error | options: POSTE or write your own. | 41-42 |
| | Other ways to use and speed-up POST. | 49-50 |
| POSTH: default, no | output HANDLE routine. | 21 |
| HANDLE | outputs the results of the solution process. | 21-22 |
| LPLM does Linearly | parameterized Piecewise Linear Mapping. | 23-24 |
| AF evaluates **a** and **f** in the | **pde**. | 19-20 |
| Handling initial conditions that conflict with the | **pde**, **bc**s or **ode**s. | 14 |
| POSTI: converts the | **pde** from *x* to $\xi$ coordinates. | 26-27 |
| order **pde**s. A fourth order | **pde**, $\nabla^4$, reduced to two coupled second | 12-13 |
| | **pde** mapping: POSTU and POSTI. | 25-27 |
| A simple | **pde**, the heat equation; Example 1. | 7 |
| A | **pde** with periodic **bc**s; Example 4. | 8-9 |

| | | |
|---|---|---|
| A first order hyperbolic | **pde** with two boundary conditions. | 13-14 |
| Definition of the | **pde-bc** problem. | 3 |
| Definition of the | **pde-bc-ode** problem. | 3-4 |
| A coupled | **pde-ode** system; Example 3. | 8 |
| $\nabla^4$, reduced to two coupled second order | **pde**s. A fourth order **pde**, | 12-13 |
| A coupled system of | **pde**s; Example 2. | 7-8 |
| POST solves | **pde**s in one space variable. | 17-21 |
| A **pde** with | periodic **bc**s; Example 4. | 8-9 |
| | Physicists expect fluxes to be continuous. | 9 |
| LPLM does Linearly parameterized | Piecewise Linear Mapping. | 23-24 |
| **bc** | placement. | 35-37 |
| IMMM modifies the multiplicity of a mesh | point held on the stack. | A4:16 |
| finds the mesh interval containing a given | point. INTRVR | A4:12 |
| LUMD makes a Locally Uniform Mesh of Distinct | points. | A4:38 |
| evaluates the B-spline basis functions at a set of | points. BSPLN | A4:12 |
| LPLMT restricts *dt* so mapped mesh | points do not collide. | 24 |
| LBE.p: LBE with | polynomial extrapolation. | 46-48 |
| SETERR sets an error state for the | Port Library. | A4:12 |
| ISTKGT allocates space on the | Port Library stack. | A4:16 |
| is same as LUMB, but puts the mesh on the | Port Library stack. ILUMB | A4:16 |
| ISTKIN initializes the | Port Library stack length. | A4:2 |
| POSTR: control over the routines used by | POST. | 40-41 |
| POSTW: control over numerical variables used by | POST. | 44-46 |
| Other ways to use and speed-up | POST. | 49-50 |
| POSTX for printing the run-time statistics of | POST. | 49 |
| Scratch space used: | POST and DPOST. | 19 |
| Another way into | POST: POSTS, ..., POST4. | 49-50 |
| | POST solves **pde**s in one space variable. | 17-21 |
| Another way into POST: POSTS, ..., | POST4. | 49-50 |
| | POSTA: the vacuous, nu=0 AF. | 18 |
| | POSTB: the vacuous, nu=0 BC. | 18 |
| | POSTD: the vacuous, nv=0 DEE. | 18 |
| | POSTE: default ERROR routine. | 41-42 |
| Error options: | POSTE or write your own. | 41-42 |
| Trouble in AF, BC or DEE: | POSTF. | 22-23 |
| Common regions | POSTF and A9OSTS. | 22-23 |
| routine. | POSTH: default, no output HANDLE | 21 |
| **pde** mapping: POSTU and | POSTI. | 25-27 |
| coordinates. | POSTI: converts the **pde** from *x* to $\xi$ | 26-27 |
| **bc** mapping: POSTV and | POSTJ. | 27-30 |
| coordinates. | POSTJ: **bc** mapping, from user to Internal | 28-29 |
| | POSTN: default INMI routine. | 42 |
| | POSTP: default SCALE routine. | 39 |
| POST. | POSTR: control over the routines used by | 40-41 |
| Another way into POST: | POSTS, ..., POST4. | 49-50 |
| **pde** mapping: | POSTU and POSTI. | 25-27 |
| coordinates. | POSTU converts **u** from $\xi$ to *x* | 25-26 |
| **bc** mapping: | POSTV and POSTJ. | 27-30 |
| coordinates. | POSTV: **bc** mapping, from internal to User | 27-28 |
| POST. | POSTW: control over numerical variables used by | 44-46 |
| POST. | POSTX for printing the run-time statistics of | 49 |
| Double | precision DPOST. | 22 |
| POSTX for | printing the run-time statistics of POST. | 49 |
| WRAPUP checks for error states and | prints stack usage. | A4:2 |
| Definition of the **pde-bc-ode** | problem. | 3-4 |
| Definition of the **pde-bc** | problem. | 3 |
| Converting a steady-state | problem, no $\mathbf{u}_t$, into a transient. | 14 |
| Twiddling | procedure knobs. | 40-44 |
| HANDLE outputs the results of the solution | process. | 21-22 |

| | | |
|---|---|---|
| Mesh making - UMB, LUMB and | PUMB. | A4:1 |
| GQM11: Gaussian | quadrature on (-1,1). | 39 |
| | R1MACH reports Real machine constants. | A4:24 |
| | Ratfor: a rational Fortran. | A4:2 |
| Ratfor: a | rational Fortran. | A4:2 |
| MOVEFR copies one | Real array into another. | A4:16 |
| SETR sets a | Real array to a given constant. | A4:2 |
| R1MACH reports | Real machine constants. | A4:24 |
| A fourth order **pde**, $\nabla^4$, | reduced to two coupled second order **pde**s. | 12-13 |
| | Reduction of **bc**s to canonical form. | 34-35 |
| Automatic mesh | refinement for accuracy; Example 9. | 15 |
| Common | regions POSTF and A9OSTS. | 22-23 |
| Error estimation for | reliability; Example 10. | 15-16 |
| I1MACH | reports Integer machine constants. | A4:4 |
| R1MACH | reports Real machine constants. | A4:24 |
| LPLMT | restricts *dt* so mapped mesh points do not collide. | 24 |
| HANDLE outputs the | results of the solution process. | 21-22 |
| POSTX for printing the | run-time statistics of POST. | 49 |
| | SCALE controls the scaling of the Jacobian. | 43-44 |
| POSTP: default | SCALE routine. | 39 |
| SCALE controls the | scaling of the Jacobian. | 43-44 |
| | Scratch space used: POST and DPOST. | 19 |
| order **pde**, $\nabla^4$, reduced to two coupled | second order **pde**s. A fourth | 12-13 |
| SIE: | Semi-implicit Euler. | 48 |
| evaluates the B-spline basis functions at a | set of points. BSPLN | A4:12 |
| | SETERR sets an error state for the Port Library. | A4:12 |
| Initial conditions for **u**: L2SFF, | SETR, DL2SFF. | A4:1-2 |
| | SETR sets a Real array to a given constant. | A4:2 |
| SETR | sets a Real array to a given constant. | A4:2 |
| SETERR | sets an error state for the Port Library. | A4:12 |
| A moving | shock wave, **u** discontinuous; Example 6. | 11 |
| | SIE: Semi-implicit Euler. | 47-48 |
| A | simple **pde**, the heat equation; Example 1. | 7 |
| HANDLE outputs the results of the | solution process. | 21-22 |
| POST | solves **pde**s in one space variable. | 17-21 |
| Computational complexity: Cost of creating and | solving Jacobian system. | 39 |
| ISTKGT allocates | space on the Port Library stack. | A4:16 |
| Scratch | space used: POST and DPOST. | 19 |
| POST solves **pde**s in one | space variable. | 17-21 |
| | Spatial discretization. | 34-39 |
| INMI | specifies the initial Newton method iterate. | 42-43 |
| Other ways to use and | speed-up POST. | 49-50 |
| L2SFF Least-squares | spline fit to a function. | A4:16 |
| coefficients and mesh. | SPLNE evaluates a B-spline given the | A4:4 |
| ISTKGT allocates space on the Port Library | stack. | A4:16 |
| ENTER-LEAVE bracket blocks of code so | stack allocations are local. | A4:16 |
| as LUMB, but puts the mesh on the Port Library | stack. ILUMB is same | A4:16 |
| modifies the multiplicity of a mesh point held on the | stack. IMMM | A4:16 |
| ISTKIN initializes the Port Library | stack length. | A4:2 |
| WRAPUP checks for error states and prints | stack usage. | A4:2 |
| SETERR sets an error | state for the Port Library. | A4:12 |
| Error | states. | A5:4-7 |
| WRAPUP checks for error | states and prints stack usage. | A4:2 |
| POSTX for printing the run-time | statistics of POST. | 48 |
| transient. Converting a | steady-state problem, no $\mathbf{u}_t$, into a | 14 |
| complexity: Cost of creating and solving Jacobian | system. Computational | 39 |
| A coupled **pde-ode** | system; Example 3. | 8 |
| A coupled | system of **pde**s; Example 2. | 7-8 |
| Galerkins method for linear | systems of **ode**'s. | 37-39 |

| | | |
|---|---|---|
| | Time discretization. | 31-33 |
| ERROR controls the | time evolution error. | 41-42 |
| a steady-state problem, no $\mathbf{u}_t$, into a | transient. Converting | 14 |
| POSTF. | Trouble in AF, BC or DEE: | 22-23 |
| | Twiddling non-procedure knobs. | 44-46 |
| | Twiddling procedure knobs. | 40-44 |
| A first order hyperbolic **pde** with | two boundary conditions. | 13-14 |
| EEBSF estimates the error in | two B-spline fits. | A4:38 |
| A fourth order **pde**, $\nabla^4$, reduced to | two coupled second order **pde**s. | 12-13 |
| A moving shock wave, | **u** discontinuous; Example 6. | 11 |
| POSTU converts | **u** from $\xi$ to $x$ coordinates. | 25-26 |
| Initial conditions for | **u**: L2SFF, SETR, DL2SFF. | A4:1-2 |
| Converting a steady-state problem, no | $\mathbf{u}_t$, into a transient. | 14 |
| A moving front, | $\mathbf{u}_x$ discontinuous; Example 5. | 10-11 |
| | UMB creates a uniform B-spline mesh. | A4:2 |
| Mesh making - | UMB, LUMB and PUMB. | A4:1 |
| UMB creates a | uniform B-spline mesh. | A4:2 |
| LUMD makes a Locally | Uniform Mesh of Distinct points. | A4:38 |
| LUMB makes a B-spline mesh from a union of | uniform meshes. | A4:16 |
| LUMB makes a B-spline mesh from a | union of uniform meshes. | A4:16 |
| WRAPUP checks for error states and prints stack | usage. | A4:2 |
| Other ways to | use and speed-up POST. | 49-50 |
| POSTR: control over the routines | used by POST. | 40-41 |
| POSTW: control over numerical variables | used by POST. | 44-46 |
| Scratch space | used: POST and DPOST. | 19 |
| POSTV: **bc** mapping, from internal to | User coordinates. | 27-28 |
| POSTJ: **bc** mapping, from | user to Internal coordinates. | 28-29 |
| POSTA: the | vacuous, nu=0 AF. | 18 |
| POSTB: the | vacuous, nu=0 BC. | 18 |
| POSTD: the | vacuous, nv=0 DEE. | 18 |
| LPLMG: Generates the mapping to obtain given $x$ | values. | 23-24 |
| POST solves **pde**s in one space | variable. | 17-21 |
| POSTW: control over numerical | variables used by POST. | 44-46 |
| of EXP($x$) which goes to 0 as $x$ gets | very negative. EXPL: a version | A4:29 |
| A moving shock | wave, **u** discontinuous; Example 6. | 11 |
| usage. | WRAPUP checks for error states and prints stack | A4:2 |
| POSTU converts **u** from $\xi$ to | $x$ coordinates. | 25-26 |
| a version of EXP($x$) which goes to 0 as | $x$ gets very negative. EXPL: | A4:29 |
| POSTI: converts the **pde** from | $x$ to $\xi$ coordinates. | 26-27 |
| LPLMG: Generates the mapping to obtain given | $x$ values. | 23-24 |
| POSTI: converts the **pde** from $x$ to | $\xi$ coordinates. | 26-27 |
| POSTU converts **u** from | $\xi$ to $x$ coordinates. | 25-26 |