Computing Science Technical Report #53

Numerical Solution of Time-Varying Partial Differential Equations in One Space Variable

N. L. Schryer

April 15, 1977

Numerical Solution of Time-Varying Partial Differential Equations in One Space Variable.

N. L. Schryer

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

An algorithm is presented for the numerical solution of time-varying partial differential equations in one space dimension. The technique used is a combination of

Spatial discretization by Galerkin's method, using B-splines, and

Solution in time by a variable order, variable time-step extrapolation procedure.

The algorithm is capable of dealing with coupled systems of partial differential equations, those depending on both time and space, and ordinary differential equations, those depending only on time. Also, non-local conditions may be imposed on the solution, such as making it periodic in space, or specifying its spatial integral as a known function of time.

A preliminary 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 at the least possible cost. A user's guide to POST is given along with several examples.

September 19, 1976

Numerical Solution of Time-Varying Partial Differential Equations in One Space Variable.

N. L. Schryer

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction.

Many applications require the solution of time-varying partial differential equations (PDE's) in one space variable. Typically these equations are sufficiently complex that their solution must be carried out numerically. In the past, this effort has required the collaboration of the person(s) who formulated the PDE problem with numerical analysts and mathematicians. This interaction, while fun and interesting, is exceedingly costly.

This paper describes the preliminary release of a package of portable FORTRAN software, called POST (Partial and Ordinary differential equations in Space and Time), for solving systems of PDE's in one spatial variable and time. A subsequent paper will describe the completed package, which will hopefully benefit from user comments based on experience with the current package. Using POST, the formulator of a PDE may easily and personally solve the PDE numerically. POST 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 package also allows for ordinary differential equations (ODE's) in time to be coupled to the PDE's and the boundary conditions. Furthermore, it is possible for non-local statements to be made about the solution, such as forcing it to be periodic or making its integral, say $\int u(t,x)\,dx$, be a known function of time.

Section 2 describes the class of PDE problems treated. Section 3 discusses briefly the numerical method used to solve the PDE's (Galerkin's method in space, using B-splines, and a variable order, variable time-step extrapolated backward difference procedure in time). Section 4 discusses several simple applications of POST. Section 5 discusses the mechanism which allows for non-local statements to be made about the solution, and also allows for the coupling of ODE's in time. Section 6 discusses several examples where POST is used to solve PDE-ODE combinations and handle periodic boundary conditions.

Section 7 gives a list of error states and problems which may arise when using POST, and what the common causes of such difficulties are. Sections 8 and 9 discuss the algorithm used by POST in considerable detail. Appendix 1 gives a brief tutorial on B-splines. Appendix 2 gives a brief tutorial on extrapolation. Appendix 3 discusses improvements which could be made, and those which will be made, in POST.

Several other FORTRAN software packages have recently become available for solving PDE's in one spatial variable [45,46]. All of the these packages assume that the PDE has the form

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

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

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

or

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

While this type of formulation covers a very wide range of physically interesting problems, it

does not cover problems with $\mathbf{u}_{xt}$ or $\mathbf{u}_{xxt}$ terms [51], nor does it deal with non-local statements such as periodic boundary conditions [31]. Finally, the above formulation does not allow for the coupling of ODE's in time with the PDE's in space and time [1,39,56].

2. Statement of the PDE-BC Problem.

The general PDE-BC form that can be solved with the approach used in POST is given by the following equations, where $\mathbf{u}$ is a vector of PDE variables of length n_u. The full PDE-BC-ODE form is described in section 5. The PDE's are assumed to be in semi-linear, divergence-form

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

where $\mathbf{a}$ and $\mathbf{f}$ are vector-valued functions of their arguments, for $L \leqslant x \leqslant R$. It is required that the length of $\mathbf{a}$ and $\mathbf{f}$ be equal to n_u, the number of PDE variables. 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 component of the BC vectors $\mathbf{b}_L$ or $\mathbf{b}_R$ which is identically zero 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 [35]

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

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 both parabolic (elliptic) and hyperbolic problems. It also encompasses PDE's which have no solution, such as

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

over the real field.

3. General Method of Solution.

Let the solution $\mathbf{u}(t,x)$, for a given instant in time, be approximated by a B-spline [2,3] of **order** k on a **mesh** $X(1) \leqslant \cdots \leqslant X(NX)$, 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 $k \geqslant 2$ is any integer the user desires. If we set $h = \max_i |X(i+1) - X(i)|$, then the error $\max_x | u_i(t,x) - \hat{u}_i(t,x) |$ is $O(h^k)$ [3] for some B-spline $\hat{u}_i$. Since k may be taken to be any integer $\geqslant 2$, this gives a very powerful technique for approximating the solution $\mathbf{u}(t,x)$ in space. We can use the Rayleigh-Ritz-Galerkin (R-R-G)

method [50] to essentially find 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 [27,39,50] for the coefficients $U_{ji}(t)$ in the expansion

$$u_i(t,x) = \sum_j U_{ji}(t) B_j(x) \tag{3.1}$$

where the $B_j(x)$ are the B-spline basis functions.

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" [13,14], an implicit differencing scheme must be used to solve them. This virtually requires that the partial derivatives of the **a** and **f** in (2.1), and the $\mathbf{b}_L$ and $\mathbf{b}_R$ in (2.2), with respect to their arguments be known, either analytically or numerically.

The next step in the solution process 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 [35], or an exponentially-fitted technique [28], or even an explicit method such as Gragg's modified mid-point rule [27,28], could be used.

All of the above techniques, and many others, have the property that for a given time-step δ they produce an approximate solution which is accurate to $O(\delta^\gamma)$, where typically γ is 1 or 2. Moreover, if the equations are solved using time-steps of δ and $\delta/2$, the results of these two computations can be combined using extrapolation to the limit [5,27] to obtain a result which is accurate to $O(\delta^{2\gamma})$. This process can be repeated indefinitely, with the result that a basic process of accuracy δ^γ can be used to generate a sequence of processes of accuracy $O(\delta^\gamma)$, $O(\delta^{2\gamma}), \cdots, O(\delta^{P\gamma}), \cdots$.

A step-size and order monitor is available [40,41] for carrying out this extrapolation process and *automatically* deciding what time-step δ and order $P\gamma$ should be used, when given the accuracy desired in the solution. Thus, the user need only specify how accurately the solution in time is to be computed, and the time integration then proceeds automatically, with no need for the user to worry about choosing δ, or whether the numerical solution is accurate enough.

The algorithm 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.

4. Software for the PDE-BC Problem.

The algorithm outlined in section 3 and described in detail in sections 8 and 9, has been implemented in EFL [29], a Ratfor-[32]-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, a FORTRAN preprocessor, programs written in FORTRAN, Ratfor, EFL or other FORTRAN preprocessing languages may be used to drive and communicate with POST.

It should be noted that the current, preliminary implementation is still evolving (see Appendix 3), and user's complaints, comments and suggestions are encouraged before the package is firmed-up (petrified) for release.

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

```
Call Posts(U,nu,k,X,nx,*,*,
           tstart,tstop,dt,
           AF,B,*,*,
           errpar,
           Handle)
```

where a * represents an argument (described in Section 6) for dealing with the coupling of ODE's in time to the PDE-BC formulation of (2.1)-(2.2). The input to Posts is

U - The B-spline coefficients (3.1) for the initial s of the PDE variables **u**. U(*i*,*j*), for *i*=1, · · · , nx-k, are the coefficients u_j, *j*=1, · · · , nu.

nu - The number n_u of PDE variables **u**.

k - The B-spline order to be used. k ⩾ 2 is necessary.

X - The B-spline mesh to be used. The multiplicity of X(1) and X(nx) must be k.

nx - The length of the mesh array X.

tstart - Start integration at time tstart.

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

dt - The initial choice for the time-step. The performance of Posts is substantially independent of the initial value of dt chosen. It is sufficient that dt merely be within several orders of magnitude of being "correct". The value of dt will automatically be adjusted by Posts 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. AF must be declared External in the user's calling program. This user-supplied subprogram will be described later.

B - A subroutine for specifying the boundary conditions **b**. B must be declared External in the user's calling program. This user-supplied subprogram will be described later.

errpar - A Real vector of length 2 for determining the error desired (to be allowed) in the solution of the equations in time. For the i^{th} component of the PDE solution **u**, the error at each time-step in the time integration will be at most

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

where

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

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).

Handle - A user-supplied subroutine which will be called by Posts at the end of each time-step. Handle must be declared External in the user's calling program. This subprogram will be described later.

The output from Posts is

U - The B-spline coefficients for the PDE solution **u** at time tstop.

tstop - May be altered by user-supplied subroutine Handle. If an error state exists on return (see section 7), 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 - Final value of the "optimal" time-step.

The amount of scratch space used on the dynamic stack of the PORT library [23] is, neglecting lower order terms,

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

Real words (storage units).

The user-supplied subroutines AF and B, which define the PDE-BC problem to be solved, are now described. When Posts needs to compute **a** and **f**, it will

```
Call AF(t,Xe,nxe,U,Ux,Ut,Uxt,nu,*,*,*,
        A,AU,AUx,AUt,AUxt,*,*,
        F,FU,FUx,FUt,FUxt,*,*)
```

where a * represents an ODE argument described in section 6. The input to AF is

- 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.
- nxe - The length of Xe.
- U - The *values* of **u** at the Xe(i). U(i,j) = u_j(t,Xe(i)), i=1, · · · , nxe and j=1, · · · , nu.
- Ux - The values of $\mathbf{u}_x$ at the Xe(i), stored as above.
- Ut - The values of $\mathbf{u}_t$ at the Xe(i), stored as above.
- Uxt - The values of $\mathbf{u}_{xt}$ at the Xe(i), stored as above.
- nu - The number n_u of PDE variables **u**.
- A - An array set to zero, see below for output values.
- AU - An array set to zero, see below for output values.
- AUx - An array set to zero, see below for output values.
- AUt - An array set to zero, see below for output values.
- AUxt - An array set to zero, see below for output values.
- F - An array set to zero, see below for output values.
- FU - An array set to zero, see below for output values.
- FUx - An array set to zero, see below for output values.
- FUt - An array set to zero, see below for output values.
- FUxt - An array set to zero, see below for output values.

AF must return as output

- A - The value of **a** at the Xe(i). A(i,j) = a_j(t,Xe(i)), for i=1, · · · , nxe and j=1, · · · , nu.
- AU - The partial derivatives of **a** with respect to **u** at the Xe(i). AU(ix,i,j) = $\partial a_i/\partial u_j$ (t, Xe(ix)), for ix=1, · · · , nxe and i,j=1, · · · , nu.
- AUx - The partial derivatives of **a** with respect to $\mathbf{u}_x$ at the Xe(i), as above.
- AUt - The partial derivatives of **a** with respect to $\mathbf{u}_t$ at the Xe(i), as above.
- AUxt - The partial derivatives of **a** with respect to $\mathbf{u}_{xt}$ at the Xe(i), as above.
- F - The value of **f** at the Xe(i). F(i,j) = F_j(t,Xe(i)), for i=1, · · · , nxe and j=1, · · · , nu.
- FU - The partial derivatives of **f** with respect to **u** at the Xe(i). FU(ix,i,j) = $\partial f_i/\partial u_j$ (t, Xe(ix)), for ix=1, · · · , nxe and i,j=1, · · · , nu.
- FUx - The partial derivatives of **f** with respect to $\mathbf{u}_x$ at the Xe(i), as above.
- FUt - The partial derivatives of **f** with respect to $\mathbf{u}_t$ at the Xe(i), as above.
- FUxt - The partial derivatives of **f** with respect to $\mathbf{u}_{xt}$ at the Xe(i), as above.

When Posts needs the boundary conditions it will

```
Call B(t,L,R,U,Ux,Ut,Uxt,nu,*,*,*,
       B,BU,BUx,BUt,BUxt,*,*)
```

where a * represents an ODE argument described in section 6. The input to B is

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

Ux - $Ux(i,1)$ is the value of $\mathbf{u}_x(t,L)$, as above. $Ux(i,2)$ is the value of $\mathbf{u}_x(t,R)$, as above.

Ut - $Ut(i,1)$ is the value of $\mathbf{u}_t(t,L)$, as above. $Ut(i,2)$ is the value of $\mathbf{u}_t(t,R)$, as above.

Uxt - $Uxt(i,1)$ is the value of $\mathbf{u}_{xt}(t,L)$, as above. $Uxt(i,2)$ is the value of $\mathbf{u}_{xt}(t,R)$, as above.

nu - The number n_u of PDE variables $\mathbf{u}$.

B - An array set to zero, see below for output values.

BU - An array set to zero, see below for output values.

BUx - An array set to zero, see below for output values.

BUt - An array set to zero, see below for output values.

BUxt - An array set to zero, see below for output values.

B must return as output

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

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

The user-supplied output and control subroutine Handle is now described. At the end of each time-step, Posts will

```
Call Handle(t0,U0,*,t1,U1,*,nu,nxmk,*,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. The input to Handle is

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

U0 - PDE solution $\mathbf{u}$ at time t0 is given by B-spline coefficients U0.

t1 - Time at the end of the time-step just completed.

U1 - PDE solution $\mathbf{u}$ at time t1 is given by B-spline coefficients U1. If t0 = t1, then a restart is in progress and the values in U1 are meaningless.

nu - The number n_u of PDE variables $\mathbf{u}$.

nxmk - nxmk=nx-k is provided so that U0 and U1 may be dimensioned to be U0(nxmk,nu) and U1(nxmk,nu)

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

t1 - May be altered by the user.

U1 - May be altered by the user.

dt - May be altered by the user.

tstop - May be altered by the user.

The Double Precision version of Posts is called Dposts. The calling sequence for Dposts is precisely the same as that for Posts, with *all* floating-point arguments Double Precision, *except* errpar, which remains Real. The amount of scratch space used by Dposts on the dynamic stack of the PORT library [23] is, neglecting lower order terms,

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

Double Precision words (storage units).

The examples given below, and in section 6, 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, which invokes those capabilities of POST the intended problem will require, and keypunch it (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.

Example 1.

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

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

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

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

$$u(t,1) = e^t \tag{4.2}$$

with initial conditions

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

The following program unit, written in Ratfor [32], solves (4.1)-(4.3) using Posts, 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} relative accuracy. The main program uses several PORT [23] library subprograms. The first is the utility subprogram Umb for making uniformly spaced B-spline meshes. The second is the Setr subroutine for setting an array to a given Real constant. Setr is used to provide the constant IC's (4.3) via the B-spline coefficients (3.1), taking advantage of the fact that if all the B-spline coefficients are equal to a constant c, then the B-spline itself is identically equal to that constant c (see Appendix 1). Had the IC's (4.3) been non-constant, other PORT library subprograms could be used to fit the IC's with a B-spline (for example: L2sff for continuous IC's and Dl2sf for fitting discrete IC data). 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, as described in section 6. 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)
External AF,BC,PostD,Handle

nu = 1 ; nv = 0

errpar(1) = 1.0e-2 ; errpar(2) = 1.0e-6

tstop = 1.0e0 ; dt = 1.0e-6

k = 4
ndx = 4     # The number of distinct B-spline mesh points.
Call Umb(0.0e0,1.0e0,ndx,k,Mesh,nmesh)

Call Setr(nmesh-k,1.0e0,U)     # Initial conditions for U.

Call Posts(U,nu,k,Mesh,nmesh,V,nv,
           0.0e0,tstop,dt,
           AF,BC,PostD,nv,
           errpar,
           Handle)

Stop

End
```

Since $n_u = 1$, the AF and BC subroutines listed below use particularly simple dimension statements for their arguments. Note that since the arrays A, ... , FUxt are set to zero before entry to AF, only the active **a** and **f** terms, and their derivatives, need be computed in AF. The subroutine AF for specifying the PDE (4.1) is

```
Subroutine 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)

Real t,Xe(nxe),U(nxe),Ux(nxe),Ut(nxe),Uxt(nxe),V(1),Vt(1),
     A(nxe),AU(nxe),AUx(nxe),AUt(nxe),AUxt(nxe),AV(nxe),AVt(nxe),
     F(nxe),FU(nxe),FUx(nxe),FUt(nxe),FUxt(nxe),FV(nxe),FVt(nxe)

Do i = 1 , nxe
  {
  A(i) = Ux(i) ; AUx(i) = 1
  F(i) = Ut(i)+(-Xe(i)+t**2)*Exp(Xe(i)*t)
  FUt(i) = 1
  }

Return

End
```

The subroutine BC for specifying the BC's (4.2) is

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

Real  t,L,R,U(2),Ux(2),Ut(2),Uxt(2),V(1),Vt(1),
      B(2),BU(2),BUx(2),BUt(2),BUxt(2),BV(2),BVt(2)

B(1) = U(1)-1.0e0      # u(t,0) = 1.
B(2) = U(2)-Exp(t)     # u(t,1) = Exp(t).

BU(1) = 1
BU(2) = 1

Return

End
```

The following output subroutine simply 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.

```
Subroutine Handle(t0,U0,V0,t1,U1,V1,nu,nxmk,nv,k,X,nx,dt,tstop)

Real  t0,U0(nxmk,nu),V0(nv),t1,U1(nxmk,nu),V1(nv),X(nx),dt,tstop

Real  xe(3),Ue(3)

If ( t0 == t1 ) { Return }

xe(1) = 1.0e0/3.0e0 ; xe(2) = 0.5e0 ; xe(3) = 2.0e0/3.0e0

Call Spline(k,X,nx,U1,xe,3,Ue)

Write(11mach(2),9000) t1,(Ue(i),i=1,3)
9000 Format(" U(x,",1p1e9.2," ) =",1p3e10.2)

Return

End
```

The output of the above program is

```
U(X,  1.00E-06 ) =   1.00E 00   1.00E 00   1.00E 00
U(X,  7.38E-04 ) =   1.00E 00   1.00E 00   1.00E 00
U(X,  1.58E-01 ) =   1.05E 00   1.08E 00   1.11E 00
U(X,  6.20E-01 ) =   1.23E 00   1.37E 00   1.52E 00
U(X,  1.00E 00 ) =   1.40E 00   1.65E 00   1.95E 00
```

A skeptic might observe that it is difficult to determine that the above output is in fact accurate to 1%. Well, since the exact solution of the problem is known, the program may also check the accuracy of the numerical solution using the PORT [23] library subroutine Eesff to estimate the error $||u-\hat{u}||$ in the computed solution $\hat{u}$. By changing the body of Handle to read

```
Common /Time/ tt
Real tt

Real eU,Eesff
External Uofx     # Returns u(t,x) = Exp(xt).

If ( t0 == t1 ) { Return }

tt=t1

eU = Eesff(k,X,nx,U1,Uofx)     # Get the error in U1.

Write(Ilmach(2),9000) tt,eU
9000 Format(" Error in U(x,",1ple9.2," ) =",1ple9.2)
```

where the following subroutine computes the exact solution u of (4.1)-(4.3)

```
Subroutine Uofx(x,nx,U,W)

Real x(nx),U(nx),W(nx)

Common /Time/ t
Real t

Do i = 1 , nx
   {
   U(i)=Exp(x(i)*t)
   }

Return

End
```

we may obtain the output

```
ERROR IN U(X,  1.00E-06 ) = 2.98E-08
ERROR IN U(X,  7.38E-04 ) = 1.19E-07
ERROR IN U(X,  1.58E-01 ) = 1.19E-03
ERROR IN U(X,  6.20E-01 ) = 5.96E-03
ERROR IN U(X,  1.00E 00 ) = 6.84E-03
```

and we can see that (4.1)-(4.3) has indeed been solved accurate to 1%.

Example 2.

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. The next 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 $u_2 = u_x$. The PDE (4.1) then becomes

$$u_{1t} = u_{2x} + (x - t^2)e^{xt} \tag{4.4}$$

$$u_{1x} = u_2$$

and the BC's (4.2) become

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

$$u_1(t,1) = e^t. \tag{4.5}$$

The initial conditions are

$$u_1(0,x) = 1$$

$$u_2(0,x) = 0. \tag{4.6}$$

The following program solves (4.4)-(4.6) using Posts, 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)
External AF,BC,PostD,Handle

nu = 2 ; nv = 0

errpar(1) = 0 ; errpar(2) = 1.0e-2

tstop = 1.0e0 ; dt = 1.0e-2

k = 4
ndx = 4     # The number of distinct B-spline mesh points.
Call Umb(0.0e0,1.0e0,ndx,k,Mesh,nmesh)

Call Setr(nmesh-k,1.0e0,U)     # Initial conditions for U1=1.
Call Setr(nmesh-k,0.0e0,U(nmesh-k+1))     # Initial conditions for U2=0.

Call Posts(U,nu,k,Mesh,nmesh,V,nv,
           0.0e0,tstop,dt,
           AF,BC,PostD,nv,
           errpar,
           Handle)

Stop

End
```

The dimension statements for the various arguments of the AF and BC subroutines given below are for arbitrary input, and thus will not be repeated in subsequent examples. The subroutine AF specifying the PDE (4.4) is

```
Subroutine 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)

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

Do i = 1 , nxe
  {
  A(i,1) = U(i,2) ; AU(i,1,2) = 1
  F(i,1) = Ut(i,1)+(-Xe(i)+t**2)*Exp(Xe(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
  }

Return

End
```

The subroutine BC specifying the BC's (4.5) is

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

Real  t,L,R,U(nu,2),Ux(nu,2),Ut(nu,2),Uxt(nu,2),V(nv),Vt(nv),
      B(nu,2),BU (nu,nu,2),BUx (nu,nu,2),
              BUt(nu,nu,2),BUxt(nu,nu,2),
              BV (nu,nv,2),BVt (nu,nv,2)

B(1,1) = U(1,1)-1.0e0     # U(t,0) = 1.
B(1,2) = U(1,2)-Exp(t)     # U(t,1) = Exp(t).

BU(1,1,1) = 1
BU(1,1,2) = 1

Return

End
```

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
External U1ofx,U2ofx     # To compute u1 and u2.

If ( t0 == t1 ) { Return }

tt=t1

eU(1) = Eesff(k,X,nx,U1(1,1),U1ofx)
eU(2) = Eesff(k,X,nx,U1(1,2),U2ofx)

Write(Ilmach(2),9000) tt,eU(1),eU(2)
9000 Format(" Error in U(x,",1p1e9.2," ) =",1p2e9.2)
```

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)

Common /Time/ t
Real t

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)

Common /Time/ t
Real t

Do i = 1 , nx
   {
   U(i)=t*Exp(x(i)*t)
   }

Return

End
```

The output of this program is

```
ERROR IN U(X,  1.00E-02 ) = 1.35E-05 2.33E-04
ERROR IN U(X,  7.00E-02 ) = 2.89E-04 2.60E-03
ERROR IN U(X,  1.84E-01 ) = 1.43E-04 5.00E-04
ERROR IN U(X,  5.45E-01 ) = 3.53E-04 1.24E-03
ERROR IN U(X,  1.00E 00 ) = 6.16E-04 2.82E-03
```

One reason for treating (4.1)-(4.3) as the system (4.4)-(4.6) is that the spatial derivative u_2 of $u = u_1$ is obtained accurate to $O(h^k)$, instead of $O(h^{k-1})$. In fact, for this example, the error in u_x, as computed by differentiating the solution of (4.1)-(4.3), was $5.2*10^{-2}$ and that computed from (4.4)-(4.6) was $2.8*10^{-3}$.

5. Coupled ODE's and Non-Local Conditions.

It is often necessary to make a non-local statement about the solution of a PDE. For example, the solution may be periodic [31], or it may be defined in a coordinate system which is rotating, accelerating or being dynamically scaled [39]. Such statements may be accommodated using the ability of POST to couple ODE's in time to the PDE-BC formulation of section 2. PDE-ODE coupling may also arise naturally in the formulation of a problem [56].

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? One way would simply be to tie the value of u at a *single* point in space, say x=0, to the value of $v'(t)$ by a relation like

$$v'(t) = u(0,t).$$

The mechanism used by POST to handle such conditions 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 $\mathbf{u}(t,x)$, and its partial derivatives, at a finite number of points x. Let $\boldsymbol{\xi}(t)$, the list of coupling points $\boldsymbol{\xi}$, be a known vector of length n_ξ. Generalizing (2.1)-(2.2), the PDE is assumed to have the form

$$\mathbf{a}(t, x, \mathbf{u}, \mathbf{u}_x, \mathbf{u}_t, \mathbf{u}_{xt}, \mathbf{v}, \mathbf{v}_t)_x = \mathbf{f}(t, x, \mathbf{u}, \mathbf{u}_x, \mathbf{u}_t, \mathbf{u}_{xt}, \mathbf{v}, \mathbf{v}_t) \tag{5.1}$$

while the BC's are assumed to have the form

$$\begin{aligned} &\mathbf{b}_L(t, \mathbf{u}(t,L), \mathbf{u}_x(t,L), \mathbf{u}_t(t,L), \mathbf{u}_{xt}(t,L), \mathbf{v}, \mathbf{v}_t) = 0 \\ &\mathbf{b}_R(t, \mathbf{u}(t,R), \mathbf{u}_x(t,R), \mathbf{u}_t(t,R), \mathbf{u}_{xt}(t,R), \mathbf{v}, \mathbf{v}_t) = 0. \end{aligned} \tag{5.2}$$

The ODE's determining the ODE variables $\mathbf{v}$ are assumed to have the form

$$\mathbf{d}(t, \mathbf{u}(t,\boldsymbol{\xi}(t)), \mathbf{u}_x(t,\boldsymbol{\xi}(t)), \mathbf{u}_t(t,\boldsymbol{\xi}(t)), \mathbf{u}_{xt}(t,\boldsymbol{\xi}(t)), \mathbf{v}, \mathbf{v}_t) = 0, \tag{5.3}$$

where $\mathbf{d}$ is vector-valued function of its arguments, and the notation $\mathbf{u}(t,\boldsymbol{\xi}(t))$ represents the list

$$\mathbf{u}(t,\boldsymbol{\xi}(t)) \equiv \mathbf{u}(t, \xi_1(t)), \cdots, \mathbf{u}(t, \xi_{n_\xi}(t)). \tag{5.4}$$

The length of $\mathbf{d}$ must be n_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 (5.1)-(5.2) would be well-posed. That is, the ODE must be coupled to a PDE-BC system which is well-posed in the sense described in section 9.

For certain applications, neither $\mathbf{v}$ nor $\mathbf{v}_t$ may actually be present in $\mathbf{d}$, and (5.3) may not be a differential equation involving $\mathbf{v}$ and $\mathbf{v}_t$. In these circumstances, $\mathbf{d}$ merely represents a condition to be placed upon the PDE solution $\mathbf{u}$. 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 replacing

$$\int_{X(i)}^{X(i+1)} u(t,x)\,dx$$

over each B-spline mesh interval, by a Gaussian-quadrature rule [37] using $k/2$ points, the full integral may be computed exactly since the numerical solution u is a B-spline of order k (degree $k-1$). Thus, an integral statement about the solution of the PDE is equivalent to a statement involving only the value of $u(t,x)$ at a finite number of Gaussian-quadrature points x.

6. Software for PDE-BC-ODE Combinations.

Posts is invoked by

```
Call Posts(U,nu,k,X,nx,V,nv,
           tstart,tstop,dt,
           AF,B,D,nxi,
           errpar,
           Handle)
```

The input to Posts is precisely as described in section 4, with the exceptions and additions noted below.

V - The initial conditions for the ODE variables **v**. V(*i*) = v_i, *i*=1, · · · , 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**.

AF - A subroutine for computing the **a** and **f** terms in the PDE. AF must be declared External in the user's calling program. This user-supplied subroutine will be described later.

B - A subroutine for computing the boundary conditions. B must be declared External in the user's calling program. This user-supplied subroutine will be described later.

D - A subroutine for computing the ODE **d**. D must be declared External in the user's calling program. This user-supplied subroutine will be described later.

nxi - The maximum number of spatial PDE-ODE coupling points allowed.

errpar - A Real vector of length 2 for determining the error desired (to be allowed) in the solution of the equations in time. For each component of the ODE solution **v**, the error at each time-step in the time integration will be at most

errpar(1) * | v_i | +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).

Handle - This user-supplied output subroutine will be described later. Handle must be declared External in the user's calling program.

The output of Posts is also as described in section 4, with the addition that

V - The value of **v**(tstop).

The scratch space used on the dynamic stack of the PORT library [23] is, neglecting lower order terms,

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

Real words (storage units).

The user-supplied subroutines AF, B and D, which define the PDE-BC-ODE problem to be solved, are now described. When Posts 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 )
```

The input to AF is as described in section 4, with the addition of

- V - The values of $\mathbf{v}(t)$. V(i) = $v_i(t)$, i=1, · · · , nv.
- Vt - The value of $\mathbf{v}_t(t)$. Vt(i) = v_{it}, i=1, · · · , nv.
- nv - The number n_v of ODE variables **v**.
- AV - An array set to zero, see below for output values.
- AVt - An array set to zero, see below for output values.
- FV - An array set to zero, see below for output values.
- FVt - An array set to zero, see below for output values.

The output from AF must be as described in section 4, with the addition of

- AV - The partial derivatives of **a** with respect to **v** at the Xe(i). AV(ix,i,j) = $\partial a_i/\partial v_j(t,\mathrm{Xe}(ix))$, for ix=1, · · · , nxe, i=1, · · · , nu, and j=1, · · · , nv.
- AVt - The partial derivatives of **a** with respect to $\mathbf{v}_t$ at the Xe(i), as above.
- FV - The partial derivatives of **f** with respect to **v** at the Xe(i). FV(ix,i,j) = $\partial f_i/\partial v_j(t,\mathrm{Xe}(ix))$, for ix=1, · · · , nxe, i=1, · · · , nu, and j=1, · · · , nv.
- FVt - The partial derivatives of **f** with respect to $\mathbf{v}_t$ at the Xe(i), as above.

When Posts needs the boundary conditions it will

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

The input to B is as described in section 4, with the addition of

- V - The value of $\mathbf{v}(t)$, V(i) = $v_i(t)$, i=1, · · · , nv.
- Vt - The value of $\mathbf{v}_t(t)$, Vt(i) = $v_i(t)$, i=1, · · · , nv.
- nv - The number n_v of ODE variables **v**.
- BV - An array set to zero, see below for output values.
- BVt - An array set to zero, see below for output values.

The output from B must be as described in section 4, with the addition of

- 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, · · · , 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, · · · , nv.

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

```
Call D( t, k, X, nx,
        U, Ut, nu, nxmk, V, Vt, nv,
        Xi, lxi, nxi,
        D, DU, DUx, DUt, DUxt, DV, DVt )
```

The input to D is

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**.

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

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**.

nxi - The maximum number of spatial coupling points allowed.

D - An array set to zero, see below for output values.

DU - An array set to zero, see below for output values.

DUx - An array set to zero, see below for output values.

DUxt - An array set to zero, see below for output values.

DV - An array set to zero, see below for output values.

DVt - An array set to zero, see below for output values.

The output from D must be as described in section 4, with the addition of

Xi - The list of spatial coupling points $\boldsymbol{\xi}$.

lxi - The current active length of Xi. Must have $0 \leqslant$ lxi $\leqslant$ nxi.

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

DU - The value of the partials of **d** with respect to $\mathbf{u}(t,\xi_i)$. DU(i,j,ix) = $\partial d_i/\partial u_j(t,\text{Xi}(ix))$, i=1, $\cdots$, nv, j=1, $\cdots$, nu and ix=1, $\cdots$, lxi.

DUx - The value of the partials of **d** with respect to $\mathbf{u}_x(t,\xi_i)$.

DUt - The value of the partials of **d** with respect to $\mathbf{u}_t(t,\xi_i)$.

DUxt - The value of the partials of **d** with respect to $\mathbf{u}_{xt}(t,\xi_i)$.

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

DVt - The value of the partials of **d** with respect to v_{it}. DVt(i,j) = $\partial d_i/\partial v_j$, for $i,j = 1, \cdots, \text{nv}$.

The user-supplied output subroutine Handle is now described. At the end of each time-step, Posts will

```
Call Handle(t0,U0,V0,t1,U1,V1,nu,nxmk,nv,k,X,nx,
            dt,tstop)
```

The input to Handle is as described in section 4, with the addition of

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

V1 - The ODE solution $\mathbf{v}(t1)$ is given by V1. If t0 = t1, then a restart is in progress and the values in U1 and V1 are meaningless.

nv - The number n_v of ODE variables **v**.

The output from Handle is as described in section 4, with the addition of

V1 - May be altered by the user.

The Double Precision version of Posts is called Dposts. The calling sequence for Dposts is precisely the same as that for Posts, with *all* floating-point arguments Double Precision, *except* errpar, which remains Real. The amount of scratch space used by Dposts on the dynamic stack of the PORT library [23] is, neglecting lower order terms,

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

Double Precision words (storage units).

Example 1.

The first example of the use of Posts to solve a PDE-BC-ODE combination is contrived to be simple, but illustrative. Consider the PDE

$$u_t = u_{xx}+v(t)+g(t,x) \quad \text{on} \quad (0,1) \tag{6.1}$$

with the coupled ODE

$$v_t(t) = u(t,\tfrac{1}{2}), \tag{6.2}$$

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

$$\begin{aligned} u(t,0) &= 1 \\ u(t,1) &= \cos(t) \end{aligned} \tag{6.3}$$

with initial conditions

$$\begin{aligned} u(0,x) &= 1 \\ v(0) &= 0. \end{aligned} \tag{6.4}$$

The following program solves (6.1)-(6.4) using Posts, 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 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)
External AF,BC,Dee,Handle

nu = 1 ; nv = 1

errpar(1) = 1.0e-2 ; errpar(2) = 1.0e-6

tstop = 1.0e0 ; dt = 1.0e-6

k = 4
ndx = 4     # The number of distinct points in the B-spline mesh.
Call Umb(0.0e0,1.0e0,ndx,k,Mesh,nmesh)     # Create the mesh.

Call Setr(nmesh-k,1.0e0,U)     # Initial conditions for U.
V(1) = 0     # Initial value for V.

Call Posts(U,nu,k,Mesh,nmesh,V,nv,
           0.0e0,tstop,dt,
           AF,BC,Dee,nv,
           errpar,
           Handle)

Stop

End
```

The only change in the subroutine AF of example 2 in section 4 is in the code for specifying the PDE, (6.1),

```
Do i = 1 , nxe
   {
   A(i,1) = Ux(i,1) ; AUx(i,1,1) = 1
   F(i,1) = -V(1)+Ut(i,1)+
             Xe(i)*Sin(Xe(i)*t)-t**2*Cos(Xe(i)*t)+2.0e0*Sin(t/2.0e0)
   FUt(i,1,1) = 1
   FV(i,1,1) = -1
   }
```

The only change in the subroutine BC of example 2 in section 4 is in the code for specifying the BC's, (6.3),

```
B(1,1) = U(1,1)-1.0e0     # u(t,0) = 1.
B(1,2) = U(1,2)-Cos(t)     # u(t,1) = Cos(t).

BU(1,1,1) = 1
BU(1,1,2) = 1
```

The following subroutine specifies the **d** of (6.2). The dimension statement for the various arguments is for arbitrary input, and thus will not be repeated in subsequent examples.

```
Subroutine Dee(t,k,X,nx,
                U,Ut,nu,nxmk,V,Vt,nv,
                Xid,LXid,NXid,
                D,DU,DUx,DUt,DUxt,DV,DVt)

Real t,X(nx),U(nxmk,nu),Ut(nxmk,nu),V(nv),Vt(nv),Xid(NXid),
     D(nv),DU (nv,nu,NXid),DUx (nv,nu,NXid),
           DUt(nv,nu,NXid),DUxt(nv,nu,NXid),
           DV(nv,nv),DVt(nv,nv)

Real Ewe(1)

Xid(1) = 0.5e0 ; LXid = 1

Call Splne(k,X,nx,U,Xid,1,Ewe)     # Ewe(1) = u(t, 1/2 ).

D(1) = Vt(1)-Ewe(1) ; DU(1,1,1) = -1 ; DVt(1,1) = 1

Return

End
```

The only change in the Handle subroutine of example 2 in section 4 is in the code for computing and printing the error in the computed solution.

```
Common /Time/ tt
Real tt

Real eU,eV,Eesff
External Uofx     # To compute u(t,x).

If ( t0 == t1 ) { Return }

tt=t1

eU=Eesff(k,X,nx,U1,Uofx) ; eV=Abs(V1(1)-2.0e0*Sin(t1/2.0e0))

Write(Ilmach(2),9000) t1,eU,eV
9000 Format(" Error in U(x,",1ple9.2," ) =",1ple9.2,
            "  error in V =",1ple10.2)
```

The only change in the subroutine Uofx, for computing u, of example 1 in section 4 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 ) = 2.24E-08   ERROR IN V =  0.
ERROR IN U(X,  7.40E-04 ) = 1.19E-07   ERROR IN V =  5.82E-11
ERROR IN U(X,  1.55E-01 ) = 1.09E-03   ERROR IN V =  2.52E-04
ERROR IN U(X,  4.62E-01 ) = 2.61E-04   ERROR IN V =  2.09E-04
ERROR IN U(X,  1.00E 00 ) = 1.04E-04   ERROR IN V =  6.69E-04
```

and we see that (6.1)-(6.4) has indeed been accurately solved.

Example 2.

The second example is the use of Posts to solve a nonlinear heat equation subject to periodic boundary conditions. Consider the PDE

$$u_t = u_{xx} - u^3 + g(t,x) \quad \text{on} \quad (-\pi,+\pi), \tag{6.5}$$

subject to periodic boundary conditions

$$u(t,-\pi) = u(t,+\pi) \tag{6.6a}$$

$$u_x(t,-\pi) = u_x(t,+\pi), \tag{6.6b}$$

where $g(t,x)$ is chosen to make the solution u a known function, say $u(t,x) = \cos(x)\sin(t)$. We must re-write (6.5)-(6.6) slightly to put it into the form (5.1)-(5.3). Define an ODE variable $v(t)$, which will play the role of $u(t,-\pi) = u(t,+\pi)$. The BC's used are

$$u(t,-\pi) = v(t)$$

$$u(t,+\pi) = v(t) \tag{6.7}$$

which force (6.6a) to hold. The remaining condition, (6.6b), is used to determine $v(t)$. Relation (6.6b) is an example of an ODE of the form (5.3) which has neither $\mathbf{v}$ nor $\mathbf{v}_t$ present. The ODE used to determine $v(t)$ is then

$$u_x(t,-\pi) = u_x(t,+\pi), \tag{6.8}$$

which forces (6.6b) to hold. The effect of (6.7)-(6.8) is to treat (6.5)-(6.6) as a PDE-BC problem whose solution has floating, $v(t)$, boundary values which are to be determined by (6.8).

The following program solves the PDE-BC-ODE combination (6.5)-(6.7)-(6.8), 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} relative 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)
External AF,BC,Dee,Handle

nu = 1 ; nv = 1

errpar(1) = 0 ; errpar(2) = 1.0e-2

tstop = 8.0e0*Atan(1.0e0) ; dt = 1.0e-1

k = 4
ndx = 7     # The number of distinct B-spline mesh points.
Call Umb(-4.0e0*Atan(1.0e0),+4.0e0*Atan(1.0e0),ndx,k,Mesh,nmesh)

Call Setr(nmesh-k,0.0e0,U)     # Initial conditions for U.

V(1) = 0     # Initial conditions for V.

Call Posts(U,nu,k,Mesh,nmesh,V,nv,
           0.0e0,tstop,dt,
           AF,BC,Dee,2,
           errpar,
           Handle)

Stop

End
```

The body of the subroutine AF for (6.5) is

```
Do i = 1 , nxe
   {
   A(i,1) = Ux(i,1) ; AUx(i,1,1) = 1
   F(i,1) = Ut(i,1)+U(i,1)**3-
              Cos(Xe(i))*(Cos(t)+Sin(t)+Cos(Xe(i))**2*Sin(t)**3)
   FUt(i,1,1) = 1 ; FU(i,1,1) = 3.0e0*U(i,1)**2
   }
```

while the body of the subroutine BC for (6.7) is

```
B(1,1) = U(1,1)-V(1)
B(1,2) = U(1,2)-V(1)

BU(1,1,1) = 1 ; BV(1,1,1) = -1
BU(1,1,2) = 1 ; BV(1,1,2) = -1
```

and the body of the subroutine Dee for (6.8) is

```
Real UL(2),UR(2)

Xi(1) = X(1) ; Xi(2) = X(nx) ; LXi = 2

Call SpInd(k,X,nx,U,Xi(1),1,2,UL)
Call SpInd(k,X,nx,U,Xi(2),1,2,UR)

D(1) = UR(2)-UL(2)
DUx(1,1,2) = 1 ; DUx(1,1,1) = -1
```

The body of the subroutine Handle for computing and printing the error is

```
Common /Time/ tt
Real tt

Real eU,Eesff,eV
External Uofx     # To compute U(t,x).

If ( t0 == t1 ) { Return }

tt=t1

eU=Eesff(k,X,nx,Ul,Uofx) ; eV = Vl(1)+Sin(t1)

Write(Ilmach(2),9000) t1,eU,eV
9000 Format(" Error in U(x,",1ple9.2," ) =",1ple9.2,
            "  error in V =",1ple10.2)
```

where the body of the subroutine Uofx is

```
Do i = 1 , nx
   {
   U(i)=Cos(x(i))*Sin(t)
   }
```

The output of this program is

```
ERROR IN U(X, 1.00E-01 ) = 3.77E-04  ERROR IN V = -3.77E-04
ERROR IN U(X, 9.21E-01 ) = 6.46E-03  ERROR IN V = -6.46E-03
ERROR IN U(X, 1.33E 00 ) = 6.07E-03  ERROR IN V = -5.47E-03
ERROR IN U(X, 1.59E 00 ) = 6.93E-03  ERROR IN V = -6.93E-03
ERROR IN U(X, 2.21E 00 ) = 9.92E-03  ERROR IN V = -9.92E-03
ERROR IN U(X, 2.61E 00 ) = 9.59E-03  ERROR IN V = -2.19E-03
ERROR IN U(X, 3.09E 00 ) = 1.45E-02  ERROR IN V =  6.15E-03
ERROR IN U(X, 3.83E 00 ) = 1.15E-02  ERROR IN V = -9.65E-03
ERROR IN U(X, 4.94E 00 ) = 8.31E-03  ERROR IN V =  8.31E-03
ERROR IN U(X, 5.76E 00 ) = 8.49E-03  ERROR IN V = -6.36E-03
ERROR IN U(X, 6.10E 00 ) = 7.79E-03  ERROR IN V = -4.32E-03
ERROR IN U(X, 6.28E 00 ) = 7.23E-03  ERROR IN V = -4.16E-03
```

7. Error States.

This section provides a list of the error states [23] which 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, B, D or Handle. For each level of (entry to) POST, the error message for a given error state is the same, but the error number may vary from one level to the next. 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 PostB. The error states flagged by an * can occur *only* when POST is entered at a level below Posts, they cannot occur when Posts is invoked.

1 - nu<0.

2 - nv<0.

3 - nu=0=nv.

4 - nu>0 and k<2

5 - nu>0 and nx<2*k.

6 - tstart+dt=tstart. The user-chosen initial value for the time-step dt is too small.

7 - The input value of dt has the wrong sign. dt and tstop-tstart must have the same sign.

8 - nxid<0.

9* - theta<0 or theta>1.

10* - Keepjac not one of (0,1,2).

11* - miter<1.

12* - mgq<1.

13* - kmax<1.

14* - kinit<1.

15 - x(1) is not of multiplicity k.

16 - x(nx) is not of multiplicity k.

17 - x is not monotone increasing.

18 - Cannot have nxid>0 and nu=0.

19 - Cannot have nxid>0 and nv=0.

20 - 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.

21 - dt=0 returned by Handle. (Recoverable). Handle lowered dt and it became too small.

22 - dt returned by Handle has the wrong sign. (Recoverable).

23* - Cannot raise dt in Handle when Failed. (Recoverable).

24 - e(i)<=0 returned by Error. (Recoverable). The error request is too small.

25 - Dirichlet BC's are overdetermined. (Recoverable). There are too many Dirichlet BC's.

26 - Mixed BC's are overdetermined. (Recoverable). There are too many mixed BC's.

27 - Improper BC's. (Recoverable). The BC's and the PDE's do not match properly, see section 9.

28 - Too few BC's. (Recoverable).

29 - Too many BC's. (Recoverable).

30 - lxid<0. (Recoverable). User supplied subroutine D returned lxid<0.

31 - lxid>nxid. (Recoverable). User supplied subroutine D returned lxid>nxid.

32 - nxid altered by D.

33* - mgq=k-1 and Order(i,j)=0. (Recoverable). Must have mgq=k when one of the PDE's is of zero order.

34 - PDE(i) is vacuous. (Recoverable). There is no i^{th} PDE.

35* - AF Failure. (Recoverable). AF failures forced dt=0.

36* - B Failure. (Recoverable). B failures forced dt=0.

37* - D failure. (Recoverable). D failures forced dt=0.

38 - Singular Dirichlet BC's. (Recoverable). Ill-posed BC's forced dt=0.

39 - Singular mixed BC's. (Recoverable). Ill-posed BC's forced dt=0.

40 - Singular PDE Jacobian. (Recoverable). Ill-posed PDE forced dt=0.

41 - Singular ODE Jacobian. (Recoverable). Ill-posed ODE forced dt=0.

42 - Jacobian coefficient of u_{xx} changes sign. (Recoverable). The linearized problem has a singular solution, which forced dt=0.

43* - Too many Newton iterations required. (Recoverable). The nonlinear equations could not be solved, which forced dt=0. The user's computation of the Jacobian values in AF, B or D may be incorrect. Another cause of this problem could be insufficiently accurate computation of the Jacobian values or the values of **a** and **f**.

8. Time Discretization.

A one-step, implicit finite difference method is employed for the time discretization of (5.1)-(5.3). 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 9.

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 [41] to be employed which dynamically changes both the local step-size and order of the time-integration scheme to satisfy the user's error request in a reasonably optimal 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 θ obeying $0 \leqslant \theta \leqslant 1$. For $\theta = 1$ this gives the very stable backward-Euler scheme, $\theta = \frac{1}{2}$ gives the Crank-Nicholson scheme [35], and $\theta = 0$ gives the rather 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 $\mathbf{u}_t \approx \frac{\mathbf{u}-\mathbf{u}^o}{\delta}$, $\mathbf{u}_{xt} \approx \frac{\mathbf{u}_x-\mathbf{u}_x^o}{\delta}$ and $\mathbf{v}_t \approx \frac{\mathbf{v}-\mathbf{v}^o}{\delta}$. This reduces equations (5.1)-(5.3) to the following system of nonlinear ODE's in space, subject to nonlinear BC's, with a coupled nonlinear system of equations

$$\mathbf{a}\left(\theta t+(1-\theta)t^o,\ x,\ \theta\mathbf{u}+(1-\theta)\mathbf{u}^o,\ \theta\mathbf{u}_x+(1-\theta)\mathbf{u}_x^o,\ \frac{\mathbf{u}-\mathbf{u}^o}{\delta},\ \frac{\mathbf{u}_x-\mathbf{u}_x^o}{\delta},\ \theta\mathbf{v}+(1-\theta)\mathbf{v}^o,\ \frac{\mathbf{v}-\mathbf{v}^o}{\delta}\right)_x = \tag{8.1a}$$

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

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

(8.1b)

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

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

(8.1c)

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

$$\mathbf{d}\left(\theta t+(1-\theta)t^o,\ \theta\mathbf{u}(t,\xi)+(1-\theta)\mathbf{u}^o(t^o,\xi^o),\ \theta\mathbf{u}_x(t,\xi)+(1-\theta)\mathbf{u}_x^o(t^o,\xi^o),\right.$$

(8.1d)

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

where $\xi^\theta=\theta\xi+(1-\theta)\xi^o$.

An iterative, quasi-Newton method is used to solve (8.1). 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. Also, let $u_{jl}\equiv u_j(t,\xi_l(t))$. 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

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

(8.2a)

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

$$b_{Li}^n+\sum_{j=1}^{n_u}\left[\theta b_{Liu_j}^J w_j+\theta b_{Liu_{jx}}^J w_{jx}+b_{Liu_{jt}}^J\frac{w_j}{\delta}+b_{Liu_{jxt}}^J\frac{w_{jx}}{\delta}\right]+\sum_{j=1}^{n_v}\left[\theta b_{Liv_j}^J z_j+b_{Liv_{jt}}^J\frac{z_j}{\delta}\right]=0 \tag{8.2b}$$

$$b_{Ri}^n+\sum_{j=1}^{n_u}\left[\theta b_{Riu_j}^J w_j+\theta b_{Riu_{jx}}^J w_{jx}+b_{Riu_{jt}}^J\frac{w_j}{\delta}+b_{Riu_{jxt}}^J\frac{w_{jx}}{\delta}\right]+\sum_{j=1}^{n_v}\left[\theta b_{Riv_j}^J z_j+b_{Riv_{jt}}^J\frac{z_j}{\delta}\right]=0 \tag{8.2c}$$

$$d_i^n+\sum_{j=1}^{n_u}\left[\sum_{l=1}^{n_\xi}\left(\theta d_{iw_{jl}}^J w_j(\xi_l)+\theta d_{iw_{jlx}}^J w_{jx}(\xi_l)+d_{iw_{jlt}}^J\frac{w_j(\xi_l)}{\delta}+d_{iw_{jlxt}}^J\frac{w_{jx}(\xi_l)}{\delta}\right)\right]+$$

(8.2d)

$$\sum_{j=1}^{n_v}\left[\theta d_{iv_j}^J z_j+d_{iv_{jt}}^J\frac{z_j}{\delta}\right]=0.$$

The solution of (8.2) 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, (8.2) constitutes the iterative scheme for solving (8.1). Equations (8.2) have the form

$$\sum_{j=1}^{n_u}\left[a_{ij}^{(1)}w_{jx}+a_{ij}^{(2)}w_j\right]_x=f_{ix}^{(1)}+f_i^{(2)}+\sum_{j=1}^{n_u}\left[a_{ij}^{(3)}w_{jx}+a_{ij}^{(4)}w_j\right]+\sum_{j=1}^{n_v}\left[g_{ijx}^{(1)}+g_{ij}^{(2)}\right]z_j \tag{8.3a}$$

$$\sum_{j=1}^{n_u}\left[\alpha_{ij}^{(L)}w_j(L)+\beta_{ij}^{(L)}w_{jx}(L)\right]=\gamma_i^{(L)}+\sum_{j=1}^{n_v}\sigma_{ij}^{(L)}z_j \tag{8.3b}$$

$$\sum_{j=1}^{n_u}\left[\alpha_{ij}^{(R)}w_j(R)+\beta_{ij}^{(R)}w_{jx}(R)\right]=\gamma_i^{(R)}+\sum_{j=1}^{n_v}\sigma_{ij}^{(R)}z_j \tag{8.3c}$$

$$\sum_{j=1}^{n_u}\left[\sum_{l=1}^{n_\xi}(d_{ijl}^{(1)}w_{jx}(\xi_l)+d_{ijl}^{(2)}w_j(\xi_l))\right]=d_i^{(3)}+\sum_{j=1}^{n_v}d_{ij}^{(4)}z_j\,. \tag{8.3d}$$

These equations may be simplified by setting

$$w_j=w_{0j}+\sum_{p=1}^{n_v}z_p\,w_{jp} \tag{8.4}$$

where $\mathbf{w}_0$ solves

$$(\mathbf{a}^{(1)}\mathbf{w}_{0x}+\mathbf{a}^{(2)}\mathbf{w}_0)_x=\mathbf{f}_x^{(1)}+\mathbf{f}^{(2)}+\mathbf{a}^{(3)}\mathbf{w}_{0x}+\mathbf{a}^{(4)}\mathbf{w}_0 \tag{8.5}$$

subject to BC's

$$\begin{aligned}&\boldsymbol{\alpha}^{(L)}\mathbf{w}_0(L)+\boldsymbol{\beta}^{(L)}\mathbf{w}_{0x}(L)=\boldsymbol{\gamma}^{(L)}\\ &\\ &\boldsymbol{\alpha}^{(R)}\mathbf{w}_0(R)+\boldsymbol{\beta}^{(R)}\mathbf{w}_{0x}(R)=\boldsymbol{\gamma}^{(R)}\end{aligned} \tag{8.6}$$

and $\mathbf{w}_p$ solves

$$(\mathbf{a}^{(1)}\mathbf{w}_{px}+\mathbf{a}^{(2)}\mathbf{w}_p)_x=\mathbf{g}_{px}^{(1)}+\mathbf{g}_p^{(2)}+\mathbf{a}^{(3)}\mathbf{w}_{px}+\mathbf{a}^{(4)}\mathbf{w}_p \tag{8.7}$$

subject to BC's

$$\begin{aligned}&\boldsymbol{\alpha}^{(L)}\mathbf{w}_p(L)+\boldsymbol{\beta}^{(L)}\mathbf{w}_{px}(L)=\boldsymbol{\sigma}_p^{(L)}\\ &\\ &\boldsymbol{\alpha}^{(R)}\mathbf{w}_p(R)+\boldsymbol{\beta}^{(R)}\mathbf{w}_{px}(R)=\boldsymbol{\sigma}_p^{(R)}\,.\end{aligned} \tag{8.8}$$

Linear systems of ODE's, such as (8.5) and (8.7), subject to linear BC's, such as (8.6) and (8.8), may be solved using Galerkin's method, as outlined in section 9.

When the w_{jp} have been obtained from (8.5)-(8.6) and (8.7)-(8.8), the $\mathbf{z}$ may be computed from the linearized ODE (8.3d), which is the system of linear algebraic equations

$$\begin{aligned}&\sum_{p=1}^{z_p}\left[d_{ip}^{(4)}-\sum_{j=1}^{n_u}\sum_{l=1}^{n_\xi}(d_{ijl}^{(1)}w_{jpx}(\xi_l)+d_{ijl}^{(2)}w_{jp}(\xi_l))\right]z_p=\\ &\\ &-d_i^{(3)}+\sum_{j=1}^{n_u}\sum_{l=1}^{n_\xi}(d_{ijl}^{(1)}w_{0jx}(\xi_l)+d_{ijl}^{(2)}w_{0j}(\xi_l))\end{aligned} \tag{8.9}$$

for $\mathbf{z}$. When (8.9) has been solved for $\mathbf{z}$, $\mathbf{w}$ may be obtained from (8.4).

The computation of $\mathbf{w}$ and $\mathbf{z}$ from (8.4)-(8.9) results in a single iteration in the solution of the nonlinear ODE's (8.1), which itself constitutes a single time-step.

9. Spatial Discretization.

This section discusses the implementation of Galerkin's method, 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 which Galerkin's method requires also gives a great deal of useful debugging (error) information. Finally, the Galerkin equations are formulated.

The ODE's (8.5) and (8.7) have the form

$$(\mathbf{a}^{(1)}\mathbf{u}_x + \mathbf{a}^{(2)}\mathbf{u})_x = \mathbf{f}_x^{(1)} + \mathbf{f}^{(2)} + \mathbf{a}^{(3)}\mathbf{u}_x + \mathbf{a}^{(4)}\mathbf{u} \tag{9.1}$$

subject to BC's

$$\begin{aligned} &\boldsymbol{\alpha}^{(L)}\mathbf{u}(L) + \boldsymbol{\beta}^{(L)}\mathbf{u}_x(L) = \boldsymbol{\gamma}^{(L)} \\ &\\ &\boldsymbol{\alpha}^{(R)}\mathbf{u}(R) + \boldsymbol{\beta}^{(R)}\mathbf{u}_x(R) = \boldsymbol{\gamma}^{(R)}. \end{aligned} \tag{9.2}$$

Reduction of BC's to Canonical Form.

This section shows how to reduce the arbitrary linear BC's (9.2) to a canonical form suitable for use in Galerkin's method. The linear BC's have the form, at $x = L$ and $x = R$,

$$\boldsymbol{\alpha}\mathbf{u} + \boldsymbol{\beta}\mathbf{u}_x = \boldsymbol{\gamma}. \tag{9.3}$$

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

$$\begin{aligned} &u = \gamma_D \qquad (\beta = 0) \\ &\quad \text{or} \\ &u_x = Au + \gamma_M \quad (\beta \neq 0), \end{aligned} \tag{9.4}$$

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

$$\begin{aligned} &u(0) + u_x(0) = 0 \\ &u(0) - u_x(0) = 0, \end{aligned}$$

which is a sloppy way to say $u(0) = 0 = u_x(0)$, cannot be dealt with by the proposed generalization, even though it is of the form (9.3).

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 $1 \leqslant n_D \leqslant n_u$. 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 $\alpha_{D_{ij}} \neq 0$ for some i. Then we must have $n_{u_D} \geqslant n_D$, for otherwise the Dirichlet BC's will be overdetermined.

We may obtain a QR factorization [55] of 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 $\mathbf{P}'\mathbf{u}$, and $\mathbf{u}_D^C$ be the rest of $\mathbf{P}'\mathbf{u}$. We then have

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

and thus

$$\mathbf{u}_D = \mathbf{R}^{-1}\mathbf{Q}\boldsymbol{\gamma}_D - \mathbf{R}^{-1}\mathbf{C}\mathbf{u}_D^C. \tag{9.5}$$

This is the canonical form which 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 $1 \leqslant n_M \leqslant n_u$. Let n_{u_M} be the number of u_j which have $\beta_{Mij} \not\equiv 0$ for some i. Then we must have $n_{u_M} \geqslant n_M$, 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 $\mathbf{P}'\mathbf{u}$ and let $\mathbf{u}_M^C$ be the rest of $\mathbf{P}'\mathbf{u}$. Then

$$\mathbf{u}_{Mx} = \mathbf{R}^{-1}\mathbf{Q}\boldsymbol{\gamma}_M - \mathbf{R}^{-1}\mathbf{Q}\boldsymbol{\alpha}_M\mathbf{u} - \mathbf{R}^{-1}\mathbf{C}\mathbf{u}_{Mx}^C. \tag{9.6}$$

This is the canonical form which will be assumed for the mixed BC's.

The above BC forms, (9.5) and (9.6), have the form

$$\begin{aligned} \mathbf{u}_D &= \boldsymbol{\gamma}_D + \mathbf{C}_D\mathbf{u} \\ \mathbf{u}_{Mx} &= \mathbf{A}\mathbf{u} + \boldsymbol{\gamma}_M + \mathbf{C}_M\mathbf{u}_x \end{aligned} \tag{9.7}$$

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

The above form (9.7) 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, and then some.

BC Placement.

Galerkin's method reduces (9.1)-(9.2) to a system of linear algebraic equations in the B-spline coefficients for the solution $\mathbf{u}$, by making the error in (9.1) orthogonal to each of the B-spline basis functions [42]. 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, $N-k^{st}$, relation for the i^{th} ODE. Galerkin's method requires that certain BC's be associated with (applied to) certain kinds of differential equations. The restrictions (assumptions) made upon the relation between the structure of the ODE's and BC's are listed below.

9.1) 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

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

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).

9.2) 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.

9.3) 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

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

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

$$xu_x = \sin(x) \qquad \text{on } (0,1)$$

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

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

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

with $\mathbf{T}^{(R)}$ defined similarly for $x = R$. The previous restrictions 9.1-9.3 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

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

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

9.6) $\mathbf{E}_D^{(L)} \bigcap \mathbf{E}_M^{(L)} = \phi = \mathbf{E}_D^{(R)} \bigcap \mathbf{E}_M^{(R)}$ ^.

Restriction 9.6 makes sure that different BC's aren't applied to the same ODE, at the same point.

Another condition is also added

9.7) 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 9.7 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-backtracking algorithm [25]. Once found, these arrays tell us which ODE has which BC's associated with it, and vice-versa.

An additional restriction is that

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

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

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

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 basically that there should be no transformation of the ODE system which 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 ODE_i, which violates the above restriction. The ODE may easily be altered to read, however,

$$u_{1xx} = 0$$

$$u_{1x} = u_2$$

which has $\sum$ order $u_j = 2 \neq 3 = \sum$ order 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 ODE_i.

It is possible to have BC's which are really initial conditions, as in the problem

$$\left.\begin{aligned} u_{1xx} &= 0 \\ u_{2x} &= 0 \end{aligned}\right\} \quad \text{on} \quad (0,1)$$

with BC's

$$u_1(0) = 0 = u_{1x}(0)$$

$$u_2(1) = 0.$$

However, the problem

$$u_{1xx} = 0 \qquad \text{on} \quad (0,1)$$

subject to $u_1(0) = 0 = u_{1x}(0)$ cannot be handled directly using the current BC placement algorithm.

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 (8.5) and (8.7), subject to linear BC's of the form (9.3).

Let the B-spline mesh π be given, with n elements, as described in Appendix 1. We wish to solve linear ODE systems of the form

$$(\mathbf{a}^{(1)}\mathbf{u}_x + \mathbf{a}^{(2)}\mathbf{u})_x = (\mathbf{a}^{(3)}\mathbf{u}_x + \mathbf{a}^{(4)}\mathbf{u}) + \mathbf{f}_x^{(1)} + \mathbf{f}^{(2)} \tag{9.8}$$

subject to linear BC's of the form (9.3) 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 (9.8) are, after the usual integration by parts [42],

$$-\sum_{j=1}^{n_u}\Big[(a_{ij}^{(1)}u_{jx} + a_{ij}^{(2)}u_j, B_{px}) + (a_{ij}^{(3)}u_{jx} + a_{ij}^{(4)}u_j, B_p)\Big] + \tag{9.9}$$

$$(a_{ij}^{(1)}u_{jx} + a_{ij}^{(2)}u_j)B_p\,|_L^R = -(f_i^{(1)}, B_{px}) + f_i^{(1)}B_p\,|_L^R + (f_i^{(2)}, B_p).$$

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

$$u_j = \sum_{q=1}^{n-k} y_{j+(q-1)n_u}B_q(x) \qquad j=1, \cdots, n_u \tag{9.10}$$

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 which 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:

$$\sum_{j=1}^{n_u}\sum_{q=1}^{n-k}\Big[(a_{ij}^{(1)}B_{qx} + a_{ij}^{(2)}B_q, B_{px}) + (a_{ij}^{(3)}B_{qx} + a_{ij}^{(4)}B_q, B_p)\Big]\, y_{j+(q-1)n_u} \equiv$$

(9.11)

$$\sum_{j=1}^{n_u}\sum_{q=1}^{n-k} G_{i+(p-1)n_u,\,j+(q-1)n_u}\, y_{j+(q-1)n_u}, \quad i=1,\cdots,n_u,\ p=1,\cdots,n-k.$$

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

$$2kn_u-1. \tag{9.12}$$

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

$$b_{i+(p-1)n_u} = -(f_i^{(2)}, B_{px}) + (f_i^{(1)}, B_p), \quad i=1,\cdots,n_u,\ p=1,\cdots,n-k. \tag{9.13}$$

We now need the boundary terms in (9.9). 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 $\mathbf{C}(\mathbf{M}_L)$. 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

$$\begin{aligned}
&\delta_{p,1}\Bigg[\sum_{j\in\mathbf{C}(\mathbf{M}_L)}\left(a_{ij}^{(1)}(L)\left(y_jB_{1x}(L)+y_{j+n_u}B_{2x}(L)\right)+a_{ij}^{(2)}(L)y_j\right)+\\
&\quad\sum_{j=1}^{N_M^L}\left(a_{iM_L(j)}^{(1)}(L)\left(\sum_{l=1}^{n_u}\left(A_{jl}^{(L)}y_l+C_{Mjl}^{(L)}\left(y_lB_{1x}(L)+y_{l+n_u}B_{2x}(L)\right)\right)\right)+\right.\\
&\qquad\left.a_{iM_L(j)}^{(2)}(L)y_{M_L(j)}\right)\Bigg]-\\
&\delta_{p,n-k}\Bigg[\sum_{j\in\mathbf{C}(\mathbf{M}_R)}\left(a_{ij}^{(1)}(R)\left(y_{r(j,p-1)}B_{p-1x}(R)+y_{r(j,p)}B_{px}(R)\right)+a_{ij}^{(2)}(R)y_{r(j,p)}\right)+\\
&\quad\sum_{j=1}^{N_M^R}\left(a_{iM_R(j)}^{(1)}(R)\left(\sum_{l=1}^{n_u}\left(A_{jl}^{(R)}y_{r(l,p)}+C_{Mjl}^{(R)}\left(y_{r(l,p-1)}B_{p-1x}(R)+y_{r(l,p)}B_{px}(R)\right)\right)\right)+\right.\\
&\qquad\left.a_{iM_R(j)}^{(2)}(R)y_{r(M_R(j),p)}\right)\Bigg]\equiv
\end{aligned} \tag{9.14}$$

$$\begin{aligned}
&\delta_{p,1}\Bigg[\sum_{j\in\mathbf{C}(\mathbf{M}_L)}\left(\delta G_{ij}y_j+\delta G_{i,j+n_u}y_{j+n_u}\right)+\sum_{j=1}^{N_M^L}\left(\sum_{l=1}^{n_u}\left(\delta G_{il}y_l+\delta G_{i,l+n_u}y_{l+n_u}\right)+\delta G_{i,M_L(j)}y_{M_L(j)}\right)\Bigg]-\\
&\delta_{p,n-k}\Bigg[\sum_{j\in\mathbf{C}(\mathbf{M}_R)}\left(\delta G_{r(i,p),r(j,p-1)}y_{r(j,p-1)}+\delta G_{r(i,p),r(j,p)}y_{r(j,p)}\right)+\\
&\quad\sum_{j=1}^{N_M^R}\left(\sum_{l=1}^{n_u}\left(\delta G_{r(i,p),r(l,p)}y_{r(l,p)}+\delta G_{r(i,p),r(l,p-1)}y_{r(l,p-1)}\right)+\right.
\end{aligned}$$

$$\delta G_{r(i,p),r(M_R(j),p)} y_{r(i,p)} \Bigg]$$

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

$$\delta_{p,n-k}\left[f_i^{(1)}(R) - \sum_{j=1}^{N_M^R} a_{iM_R^{(1)}(j)}(R)\gamma_{M_j}^{(R)}\right] - \delta_{p,1}\left[f_i^{(1)}(L) - \sum_{j=1}^{N_M^L} a_{iM_L^{(1)}(j)}(L)\gamma_{M_j}^{(L)}\right] \equiv \tag{9.15}$$

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

With the above boundary terms (9.14) and (9.15) 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 $D^{(L)}(i)$, $i=1,\cdots,n_{u_D}^L$, be the indices of $\mathbf{u}_D^{(L)}$, similarly for $D^{(R)}$. Then the $E_D^{(L)}(D^{(L)}(i))^{th}$ Galerkin equation is replaced by the Dirichlet BC equation for $u_{D^{(L)}(i)}(L)$, $i=1,\cdots,n_{u_D}^L$. The $E_D^{(R)}(D^{(R)}(i))+(n-k-1)n_u{}^{th}$ Galerkin equation is replaced by the Dirichlet BC equation for $u_{D^{(R)}(i)}(R)$, $i=1,\cdots,n_{u_D}^{(R)}$. This completes the formulation of the Galerkin equations. The result is a system of linear algebraic equations of the form

$$\mathbf{G}\mathbf{y} = \mathbf{b} \tag{9.16}$$

for the B-spline coefficients $\mathbf{y}$ (9.10) of the solution $\mathbf{u}$ of (9.8). The matrix $\mathbf{G}$ is $n_u(n-k)\times n_u(n-k)$ with a band-width of $(2kn_u-1)$, see (9.12). Thus, the system (9.16) may be solved in roughly $n_u(n-k)(2kn_u-1)kn_u$ arithmetic operations [55], and the B-spline Galerkin solution can be obtained quite efficiently.

The Galerkin system (9.16) is based on the computation of many integrals. These integrals are computed with Gauss-Legendre quadrature [42]. If we let O be the minimum order of all u_i in the ODE system (9.8), then we wish to compute

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

exactly [42] with an m_q-point Gauss-Legendre quadrature rule. This is best accomplished for

$$m_q = \begin{cases} k & \text{if } O = 0 \\ k-1 & \text{if } O > 0. \end{cases} \tag{9.17}$$

Using such a quadrature rule, the Galerkin system (9.16) may be formed in roughly

$$4(n-k)k^2n_u{}^2m_q \tag{9.18}$$

operations, while the right-hand-side $\mathbf{b}$ alone may be computed in roughly

$$2(n-k)kn_u m_q \tag{9.19}$$

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

$$(n-k)n_u(3kn_u-2) \tag{9.20}$$

operations.

Bibliography

[1] D.H. Auston and N.L. Schryer, "High-Speed Photoconductive Measurements of Auger Recombination in Semiconductors", to be published.

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

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

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

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

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

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

[8] 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).

[9] 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).

[10] 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).

[11] R. Courant and D. Hilbert, **Methods of Mathematical Physics**, Vol. I, Interscience, New York, 1966.

[12] 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).

[13] G. Dahlquist, "A Special Stability Problem for Linear Multistep Methods", *BIT* **3**, 27-43(1963).

[14] G. Dahlquist, "Stability Questions for Some Numerical Methods for Ordinary Differential Equations", *Proc. Symp. for Applied Math.* **15**, 147-158(1963).

[15] J. Douglas and T. DuPont, "A Finite Element Collocation Method for Quasilinear Parabolic Equations", *Math. Comp.* **27**, 17-28(1973).

[16] J. Douglas, T. DuPont and M.F. Wheeler, "An L_∞ Estimate and a Superconvergence Result for A Galerkin Method For Elliptic Equations Based on Tensor Products of Piecewise Polynomials", *RAIRO* **8**, 61-66(1974).

[17] T. DuPont, "A Unified Theory of Superconvergence for Galerkin Methods for Two-Point Boundary Problems", *SIAM J. Numer. Anal.* **13**, 362-368(1976).

[18] W.H. Enright, T.E. Hull and B. Lindberg, "Comparing Numerical Methods for Stiff Systems of Ordinary Differential Equations", *BIT* **15**, 10-48(1975).

[19] G. Fix, "Higher-Order Rayleigh-Ritz Approximations", *J. Math. and Mech.* **18**, 645-657(1969).

[20] G. Forsythe and C. Moler, **Computer Solution of Linear Algebraic Systems**, Prentice-Hall, New York, 1967.

[21] G. Forsythe and W. Wasow, **Finite Difference Methods for Partial Differential Equations** , Wiley, New York, 1959.

[22] P.A. Fox, "A Comparative Study of Computer Programs for Integrating Differential Equations", *Comm. ACM* **15**, 941-948(1972).

[23] P.A. Fox, A.D. Hall and N.L. Schryer, "The PORT Library Mathematical Subroutine Library", Bell Laboratories Computing Science Technical Report #47, 1976.

[24] C.W. Gear, "The Automatic Integration of Ordinary Differential Equations", *Comm. ACM* **14**, 176-179(1971).

[25] S.W. Golomb and L.D. Baumert, "Backtrack Programming", *J. ACM* **12**, 516-524(1965).

[26] W.B. Gragg, "Repeated Extrapolation to the Limit in the Numerical Solution of Ordinary Differential Equations", Thesis, UCLA (1963).

[27] W.B. Gragg, "On Extrapolation Algorithms for Ordinary Initial Value Problems" *SIAM J. Num. Anal.* **2**, 384-403(1965).

[28] 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.

[29] A.D. Hall and S.I. Feldman, "EFL, an Extended FORTRAN Language", in preparation.

[30] T.E. Hull, W.H. Enright, B.M. Fellen and A.E. Sedgwick, "Comparing Numerical Methods for Ordinary Differential Equations", Technical Report 29, 1971, Department of Computer Sciences, University of Toronto.

[31] H. Ikezi, A.L. Simons, K.F. Schwarzenegger and T.S. Kamimura, "Nonlinear Self-Modulation of Ion-Accoustic Wave Packets", to be submitted to *Physics of Fluids.*

[32] B.W. Kernighan, "RATFOR - A Preprocessor for a Rational Fortran", *Software-Practice and Experience* **5**, 395-406(1975).

[33] J. McKenna and N.L. Schryer, "Analysis of Field-Aided Charge-Coupled Device Transfer", *BSTJ* **54**, 667-685(1975).

[34] P.M. Morse and H. Feshbach, **Methods of Theoretical Physics**, Mc Graw-Hill, New York, 1953.

[35] R.D. Richtmeyer and K.W. Morton, **Difference Methods for Initial Value Problems**, Interscience, New York, 1967.

[36] B.G. Ryder and A.D. Hall, "The PFORT Verifier", Bell Laboratories Computer Science Technical Report #12, 1975.

[37] R.A. Sack and A.F Donovan, "An Algorithm for Gaussian Quadrature given Modified Moments", *Numer. Math.* **18**, 465-478(1972).

[38] I.W. Sandberg and H. Shichman, "Numerical Integration of Systems of Stiff Nonlinear Differential Equations", *BSTJ* **47**, 511-528(1968).

[39] N.L. Schryer and L.R. Walker, "The Motion of 180° Domain Walls in Uniform dc Magnetic Fields", *J. Appl. Physics* **45**, No. 12, 5406-5421(1974).

[40] N.L. Schryer, "An Extrapolation Step-Size and Order Monitor for use in Solving Differential Equations", Proceedings ACM National Meeting, San Diego, 1974.

[41] N.L. Schryer, "An Extrapolation Step-Size and Order Monitor for use in Solving Differential Equations", in preparation.

[42] N.L. Schryer, "A Tutorial on Galerkin's Method, using B-splines, for Solving Differential Equations", Bell Laboratories Computing Science Technical Report #52, 1976.

[43] M.H. Schultz, "The Galerkin Method for Non-Self-Adjoint Differential Equations", *J. Math. Anal. and Appl.* **28**, 647-651(1969).

[44] M.H. Schultz, "The Condition of a Class of Rayleigh-Ritz-Galerkin Matrices", *Bull. AMS* **76,** 840-844(1970).

[45] R.F. Sincovec and N.K. Madsen, "Software for Nonlinear Partial Differential Equations", *ACM Trans. on Math. Software* **1,** 232-260(1975).

[46] R.F. Sincovec and N.K. Madsen, "PDEONE, Solutions of Systems of Partial Differential Equations", *ACM Trans. on Math. Software* **1,** 261-263(1975).

[47] R.F. Sincovec and N.K. Madsen, PDEPACK and COLPACK, Scientific Computing Consulting Services, P.O. Box 335, Manhattan, Kansas, 66502.

[48] H.J. Stetter, "Asymptotic Expansions for the Error of Discretization Algorithms for Non-Linear Functional Equations", *Numer. Math.* **7,** 18-31(1965).

[49] J. Stoer, "Extrapolation Methods for the Solution of Initial Value Problems and their Practical Realization", Conference on the Numerical Solution of Ordinary Differential Equations, University of Texas at Austin, 1972.

[50] G. Strang and G. Fix, **An Analysis of the Finite Element Method,** Prentice-Hall, New York, 1973.

[51] Lars Wahlbin, "Error Estimates for a Galerkin Method for a Class of Model Equations for Long Waves", *Numer. Math.* **23,** 289-303(1975).

[52] D.D. Warner, "A Partial Derivative Generator", Bell Laboratories Computing Science Technical Report #28, April, 1975.

[53] M.F. Wheeler, "L_∞ Estimates of Optimal Orders for Galerkin Methods for One-Dimensional Second Order Parabolic and Hyperbolic Equations", *SIAM J. Num. Anal.* **10,** 908-913(1973).

[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] T. Yu, "Comparison of Numerical Methods for Ordinary Differential Equations", Tech. Report CNA-73, 1973, University of Texas at Austin.

Appendix 1

B-splines

The way in which the approximate numerical solution is to be represented is a very important decision. The choice of representation affects the entire solution process. Specifically, we would like to choose a space of functions, out of which we will try 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 [2,3,12]. This is a method for representing functions by piecewise polynomials, that is, polynomials of degree $k-1$ or less over each sub-interval 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 sub-interval. 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 π. The space of B-splines of **order** k defined on the mesh π is defined to be the collection of all functions f

(A1.1) which 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) for which $f \equiv 0$ 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 which 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 rather nice approximation properties, as summed up by deBoor [2], 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 which 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 π in order 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 rather natural way in which such splines arise and their smoothing properties when used to approximate functions described by discrete data [2]. 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 [12] and [3]. Briefly, when the multiplicities of the first and last mesh points are both k, so that

$$x_1 = \cdots = x_k$$

and

$$x_{N-k+1} = \cdots = x_N$$

then 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 $0 \leqslant B_i(x) \leqslant 1$ 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 ($k < i \leqslant N-k$) 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 $f(L) \neq 0$ and $f(R) \neq 0$, 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). In fact, relations (A1.2)-(A1.5) show that the only B_i which 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, 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 $0 \leqslant B_i \leqslant 1$ 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,

$$d \leqslant Log_{10}(||\mathbf{a}|| / ||\sum_{i=1}^{N-k} a_i B_i||) \tag{A1.7}$$

decimal digits are lost, due to cancellation, in evaluating f. In [2] de Boor shows that

$$||\sum_{i=1}^{N-k} a_i B_i|| \geqslant C_k ||\mathbf{a}|| \tag{A1.8}$$

where C_k is a constant depending *only* upon k, and therefore

$$d \leqslant Log_{10}(C_k^{-1}).$$

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 indicates that, at least for uniform meshes, the conditioning of the B-spline basis is independent of the size of the mesh.

Appendix 2

Extrapolation.

The problem treated in [5] and [26,27] is the numerical solution of the canonical form ODE initial value problem

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x}) \qquad a \leqslant t \tag{A2.1}$$

$$\mathbf{x}(a) = \mathbf{x}_a$$

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 [26,27], backwards-Euler methods [38] and Crank-Nicholson [18,35]. Most of these methods have the property [48] that if they take N time-steps to go from t_0 to t_1 and result in an approximation to $\mathbf{x}(t_1)$ which we shall denote by $\mathbf{T}(h)$ where $h = (t_1-t_0)/N$, then

$$\mathbf{T}(h) = \mathbf{T} + \sum_{j=1}^{\infty} \mathbf{T}_j h^{j\gamma} \tag{A2.2}$$

where $\mathbf{T} = \mathbf{x}(t_1)$, γ 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 [4] 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 which interpolates $\mathbf{T}(h_i)$ for $i=0, \cdots, m$, is given by $\mathbf{T}_m^0$ which is determined from the recursion

$$\mathbf{T}_0^i = \mathbf{T}(h_i) \qquad \text{for} \quad 0 \leqslant i \leqslant m$$

$$\mathbf{T}_k^i = \mathbf{T}_{k-1}^{i+1} + (\mathbf{T}_{k-1}^{i+1} - \mathbf{T}_{k-1}^i) \Big/ \left\{ (h_i/h_{i+k})^{\gamma} - 1 \right\} \tag{A2.4}$$

for $0 \leqslant i \leqslant m-k$ and $1 \leqslant k \leqslant m$. If the T_k^i are organized into a **lozenge** of the form

$$\begin{array}{lccccccc}
\mathbf{T}(h_0) = & \mathbf{T}_0^0 & & & & & & \\
\mathbf{T}(h_1) = & \mathbf{T}_0^1 & \mathbf{T}_1^0 & \mathbf{T}_2^0 & & & & \\
\mathbf{T}(h_2) = & \mathbf{T}_0^2 & \mathbf{T}_1^1 & \mathbf{T}_2^1 & \mathbf{T}_3^0 & \mathbf{T}_4^0 & & \\
\mathbf{T}(h_3) = & \mathbf{T}_0^3 & \mathbf{T}_1^2 & \mathbf{T}_2^2 & \mathbf{T}_3^1 & \mathbf{T}_4^1 & \mathbf{T}_5^0 & \\
\mathbf{T}(h_4) = & \mathbf{T}_0^4 & \mathbf{T}_1^3 & \mathbf{T}_2^3 & \mathbf{T}_3^2 & & & \\
\mathbf{T}(h_5) = & \mathbf{T}_0^5 & \mathbf{T}_1^4 & & & & &
\end{array}$$

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 [4].

It is also possible to estimate the error in each element of the lozenge [6]. In fact, [6] shows that for sufficiently small h_0,

$$|\mathbf{T}_k^l - \mathbf{T}| \approx \left[1 + \frac{1}{(h_l/h_{l+k+1})^\gamma - 1}\right] |\mathbf{T}_k^{l+1} - \mathbf{T}_k^l| \tag{A2.5}$$

and we can estimate the error $\epsilon_k^l = |\mathbf{T}_k^l - \mathbf{T}|$ in $\mathbf{T}_k^l$. We also know from [6] that for sufficiently small h_0,

$$\epsilon_k^l = h_0^\beta \mathbf{d}_k (h_l \cdots h_{l+k})^\gamma \tag{A2.6}$$

where the $\mathbf{d}_k$ are constant vectors, γ is the order of the basic process being extrapolated and β 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 lozenge and tell how rapidly each column in the lozenge is converging.

In (A2.4) m is called the **level** of extrapolation, while from (A2.5) 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_l are called **sub-steps.** Extrapolation approximates the $\mathbf{x}(t_1)$ values accurately, but does not accurately approximate $\mathbf{x}(t+nh_l)$ for $0 < n < N$.

Appendix 3

Wish-List

This section describes several improvements which will, could or may 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 which will be made is to use a better nonlinear equation solver. The current scheme re-computes the Jacobian matrix (the partials $\partial a_i/\partial u_j$, etc.) at every time-step. This is quite unnecessary and rather expensive, as a comparison of the operation counts (9.18)-(9.20) shows.

For linear problems, the user will be able to say that the problem is linear and that the coefficients of $\mathbf{u}$, $\mathbf{u}_x$, $\mathbf{u}_t$, $\mathbf{u}_{xt}$, $\mathbf{v}$, $\mathbf{v}_t$ are all functions of x alone (no dependence on t). In this case POST should run substantially faster because the Jacobian need only be computed *once* during the entire solution process.

For nonlinear problems, a scheme will be implemented for keeping an "old" value of the Jacobian over as many time-steps as the Jacobian can provide an effective quasi-Newton iterative solution of the nonlinear equations. This scheme will use the fact that, in general, the convergence of a quasi-Newton method with an "out-of-date" (inaccurate) Jacobian is linear rather than quadratic. The possible run-time improvement using such a scheme is kn_u, the ratio of the cost of computing the Jacobian (9.18) to the cost of just computing the right-hand-side (9.19).

Numerical Jacobians

Another possible improvement would be the automatic numerical computation of the Jacobian matrix from the values of $\mathbf{a}$ and $\mathbf{f}$. 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 [52] 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 the author, but be ready for an argument.

Special Entry for PDE-BC Problems

Another entry point can be made in POST to remove the ODE variable parameters from the calls to Posts, AF, B and Handle. This would make the use of POST to solve PDE-BC problems a little easier to explain and accomplish. Once the package has firmed-up a bit, such an overlay will undoubtedly be provided.

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 [35], so implementation of this feature is not a high-priority item.

Non-local Terms in the PDE

A radical improvement in the capabilities of POST would be to allow non-local terms in the PDE (AF). This would allow integro-differential equations to be solved. Theoretically, such problems can currently be solved using the ODE facility of POST, however, the cost is prohibitive. To do it right would require that the **a** and **f** terms in the PDE also have parameters $\mathbf{u}(t, \boldsymbol{\xi}(t))$, see (5.4). The BC's and ODE's have such non-local terms and it would be mathematically pretty if the AF term could also. However, the Jacobian matrix, instead of being banded, would be full and dense in general. This would significantly increase the running time for such problems, but also allow solution of a much broader range of problems. Non-local PDE terms can be added fairly easily to POST, but their implementation awaits sufficient user demand.

Other Spatial Discretization Techniques

B-splines are used in the current implementation due to their excellent approximation properties (see Appendix 1) and the fact that there is a *local* basis for them which 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 which, although they yield dense matrices, are very 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.

Banded PDE Systems

There are situations where the PDE's themselves are banded, that is, a_i and f_i only depend on u_j for j "near" i. Advantage can be taken of this structure to decrease the amount of storage, and run-time, needed by POST. This is a low-priority item.

More Structure in the PDE

An equation of the form [51]

$$P(u)_t = Q(u_x)_x$$

is really of the form (2.1)-(2.2) since it is equivalent to

$$P'(u)u_t = Q(u_x)_x .$$

However, it may be quite 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. However, 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 [7,15], 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 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 self-adjoint 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, for some second order problems where the spatial derivatives $u^{(i)}$ of the solution grow like α^i for

some constant α, the error for collocation is a factor of α 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 9). 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 which is not in divergence-form), it could be produced quite easily from the code for the existing POST package.