PDELAN: A MESH OPERATOR VARIANT OF FORTRAN

by

John Gary

Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-049-74 August 1974

This design was performed under an ARPA Grant AFOS-74-2732

1. Introduction. The objective of this language, which we call PDELAN, is to facilitate the coding of finite difference schemes for partial differential equations. The aspect of these codes which we have emphasized is the difference equations. An operator notation is provided so that the equations can be written as the numerical analyst frequently invites them prior to translation into a program. That is

```
U2 = U1 + DLT * DXX(U1)
```

where DXX represents the operator

$$(U_{i+1} - 2U_i + U_{i-1})/\Delta x^2$$

and DLT = Δt. Implicit difference schemes can also be written in this operator notation. The set up and solution of the resulting linear systems will be handled automatically. This treatment of implicit schemes is the most powerful facility within PDELAN.

The language is a dialect of Fortran rather than an extension. The conditional and iteration commands are taken from PASCAL and thus allow a better structured programming style than Fortran. These include

```
IF ... THEN ... ELSE ... ENDIF
REPEAT ... UNTIL ... ENDREPEAT
```

The language is coupled with a macro preprocessor which allows a topdown programming style [4]. The language is implemented as a preprocessor to Fortran. This is similar to the approach taken by Gear for a PL/I like language [5].

An earlier version of PDELAN was implemented at NCAR in 1971 [1]. The finite difference language has received only light use; however, we feel this may be due to deficiencies in the earlier version which we can eliminate. Also, implicit schemes could not be treated with the previous version. In any case a language like this is intended for a specialized use and will apply

to a small percentage of jobs even in a computing center which does much continuous simulation.

Graphics and file management capability should be provided in a language for PDE problems. The earlier version does contain a sophisticated set of high level graphics commands, but no file management commands [2]. However, our effort concentrates on the difference equations and the macro preprocessor.

2. The basic language. In this section we describe the basic set of instructions available in PDELAN. The syntax is somewhat different than in FORTRAN. The declarations are nested. For example, variable declarations are placed within the scope of a COMMON block in order to declare them as COMMON variables. The conditional statements are similar to those in PASCAL. The I/O statements are similar to FORTRAN. An end-of-card is an end-of-statement unless the statement is continued. Our objective is to provide a structured base for the mesh operator constructs, but with minimal departure from FORTRAN. We proceed to a description of the features of the language.

2.1 The lexical scan. The PDELAN syntax is restricted so that a compatable macro preprocessor can operate ahead of the PDELAN translator [4]. Therefore, blanks are delimiters. Furthermore, the PDELAN keywords such as IF, DO, FORMAT, etc. are all reserved words and may not be used as variable names. Long identifiers, up to 29 characters, may be used. Two continuation modes are allowed. The first uses a column six punch as in FORTRAN. The second uses the two characters ;+ to terminate reading of one card and indicate that the statement is to be continued to the next card. Statements may be separated by ";" which is an end-of-statement marker. A statement ends in column 72 unless it is explicitly continued to the next card. We think it better not to require that every statement be terminated by ";". An occasional use of the continuation ";+" seems preferable to this hardened FORTRAN programmer who tends to forget the ";" in PL/I and PASCAL. Blocks

are all terminated by special terminators such as ENDIF, ENDCOMMON, etc. This is done for readibility and also to reduce doubt about when a ";" is required. If a statement starts with an integer constant, then the integer is a statement label. A statement, including the label can start anywhere in columns 2 through 72. Names which start in column one are instructions for the preprocessor.

Comments can be defined by a "C" in column one as in FORTRAN, or by the "brackets" */ . . . /* as in PL/I. The "*/" delimiter is an end-of-statement marker, so this type of comment cannot be imbedded within a statement. This restriction allows all comments to be conveniently output to the object FORTRAN program.

Some examples of statements are:

```
C     SAMPLE DECK
    IF X.LT.Y  THEN
        W(I) = X*A(I)
    ELSE
        W(I) = Y*A(I)
    ENDIF
    CASE K OF 2
      1:  I = 1 ; 20 A(I) = I * K ; I = I + 1
                    IF I.LE.M THEN GO TO 20 ENDIF
      2:  FOR I = 1 TO M DO A(I) = 0.  ENDFOR
    ENDCASE
    U(1) = U(2) + A(M) * (U(3) - U(2))* ;+
                (C/B(M)) * (W(3) - W(2))  %/LEFT BOUNDARY/*
```

2.2 Declarations. The declarations are nested. A COMMON block of declarations can be declared in whose scope variable declarations may be

placed. This same type of nesting can be used to declare mesh variables and to declare groupings of variables for convenient I/O. For example, consider the following declarations of blank and labeled COMMON

```
COMMON
  REAL X, Y, T
  INTEGER A, B, C
ENDCOMMON
COMMON LAB
  DOUBLE XD, YD
ENDCOMMON
```

The arithmetic modes are INTEGER, REAL, DOUBLE, COMPLEX, LOGICAL. The only variable structure is the ARRAY. Variables may be declared as having array structure in two ways

```
REAL X, Y, U(20,30), Z, V(20,31)
REAL ARRAY  T, P(20,30), W1, W2(20,31)
```

In the first statement X, Y, and Z are scaler variables, and U and V are arrays. In the second statement T and P are declared arrays of dimension 2 and extent (20,30). If more variable types were allowed, then the PASCAL declaration style would be more appropriate. The declarations would then be grouped together as follows

```
DECLAREVAR
   COMMON LAB
      X, Y, Z : REAL
      U, T, P : ARRAY(20,30) OF REAL
      V, W1, W2 : ARRAY(20,31) OF REAL
   ENDCOMMON
ENDDECLARE
```

PASCAL permits the user to declare types and assign these types names. The PASCAL record type and scaler type could be useful in finite difference codes. It would sometimes be useful to pack flags and indices into a single word. However, the CDC 6000 version of PASCAL is about a factor of two slower than FORTRAN on matrix codes, and FORTRAN is of course more common than PASCAL. Therefore we prefer to base the language on FORTRAN in spite of the superior design of PASCAL.

2.3 Statement labels. If the first token of a statement is an integer, that integer is a statement label. An optional colon can follow the label to improve appearance. For example

```
10 : X = Y ; 20 W = A(1)
IF X.LT.0. THEN GOTO 10
```

The GOTO statement is included. There are some restrictions on the GOTO. Jumps into the scope of a FOR loop from outside the loop are not allowed. A second type of statement label uses an alphanumeric label, for example

```
LOOPA : ENDFOR
```

This is discussed below.

2.4 Control structure. We have taken our control statements from PASCAL. These are

```
IF . . . THEN . . . ENDIF
IF . . . THEN . . . ELSE . . . ENDIF
REPEAT . . . UNTIL . . . ENDREPEAT
WHILE . . . DO . . . ENDWHILE
```

Some examples are

```
IF X.LT.0. THEN X = -X ENDIF

IF A.LT.B THEN

   REPEAT A = A + H UNTIL A.GE.B ENDREPEAT

ELSE

   A = B

   CALL SET

ENDIF
```

Use of matched end-of-block markers (ENDIF, etc.) provides redundancy in the language which allows improved error diagnostics. This useage may also produce more readable code.

As case statement of the following form is included

```
CASE K OF 2

  1 : X = SIN(T)

  2 : X = SINH(T)

ENDCASE
```

These statement labels are local to the CASE block. The following code will probably be allowed (hopefully, no one writes this way, and perhaps it should not be allowed)

```
CASE K OF 2

  1 : X = 1.

  2 : Y = 1. ; GOTO 1

ENDCASE
```

2.5 Iteration and more on statement labels. The iteration statement is illustrated by the following

```
FOR K = M + 1 TO NA(N)**2 + 2 DO B(K) = K ENDFOR

FOR L = 1 STEP N - 3 TO 20 DO
   B(L) = C(L)
   F(L) = L**L
ENDFOR
```

The expression following STEP can be negative. If this expression is a positive integer constant such as STEP 2, then FOR will be translated into a DO statement. Otherwise FOR becomes a loop terminated by an IF statement containing the test on the iteration parameter.

An alphanumeric label of the following form is allowed

```
LOOPA : FOR K = 1 TO 10 DO
        A(K) = 1.
        B(K) = K
LOOPA : ENDFOR
```

This permits use of the EXIT statement. A statement of the form

```
EXIT LAB
```

causes control to drop through the control block containing the EXIT LAB statement until an END statement labeled by LAB is found. Execution then starts immediately after this labeled END. It is not necessary to label the beginning of the control block. The EXIT never refers to the beginning of a control block. However, if the beginning is labeled, then the end must also be labeled with the same label. Only alphanumeric labels can be used with the EXIT. Alphanumeric labels may not be used with a GOTO. An alphanumeric label must be followed by a colon.

2.6 Subprogram headers. These are identical to those in FORTRAN. Namely,

```
PROGRAM NAM(INPUT, . . .)
SUBROUTINE NAM . . .
FUNCTION NAM . . .
BLOCKDATA . . .
ENDPROGRAM
```

```
ENDSUBROUTINE
.
.
.
```

The PROGRAM statement is a CDC variant of FORTRAN. The usual subroutine and function calls are allowed. The ENTRY and EXTERNAL statements are also included.

2.7 I/O statements. The preprocessor will allow the following five statements which are identical with FORTRAN

```
READ(nc,nf)
WRITE(nc,nf)
READ nf,
PRINT
nf   FORMAT( . . .)
```

2.8 PASSTHRU blocks. These are blocks of statements which are passed directly to the Fortran compiler which compiles the abject code produced by the preprocessor. If a statement is not placed within such a block, then the preprocessor will attempt to parse it as a statement in PDELAN and failure will produce an error diagnostic. Most such non PDELAN statements will probably be I/O commands such as BUFFERIN to do buffered I/O, or commands to handle extended core. We could require the user to handle such commands by means of a subroutine call. However, this would not allow addition of an EQUIVALENCE statement, for example. An example of a PASSTHRU block is

```
PASSTHRU
    EQUIVALENCE (A,X)
    IMPLICIT REAL*8 (A - H, O - Z)
ENDPASSTHRU
```

3. Finite difference equations. The primary motivation for this preprocessor, is the simplification of finite difference codes arising from the solution of partial differential equations. For example consider the simple heat equation

$$\frac{\delta u}{\delta t} = \frac{\delta^2 u}{\delta x} \qquad u = u(x,t) \qquad \begin{matrix} 0 \le x \le 1 \\ 0 \le t \end{matrix}$$

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

The problem is made discrete by use of a mesh in x and t, $x_j = j\Delta x$, $0 \le j \le J$, $\Delta x = 1/J$. Using the notation $U_j^n \approx u(x_j, t_n)$, then the difference scheme might be

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{\Delta x^2}$$

This can be written as a "marching" scheme which computes values U_j^{n+1} on the new time level t_{n+1} from the known values on level t_n, namely

$$U_j^{n+1} = U_j^n + \frac{\Delta t}{\Delta x^2}(U_{j+1}^n - 2U_j^n + U_{j-1}^n) \qquad 1 \le j \le J-1$$

$$U_0^{n+1} = U_J^{n+1} = 0$$

If U^{n+1} is stored in the array U2 and U^n in the array U1, then this algorithm is written in Fortran as follows (U_j^n stored in U1(j+1), JT = J+1).

```
      U2(1) = 0.
      U2(J + 1) = 0.
      DO 100 K = 1, J
100   U2(K) = U1(K) + (DLT/DLX**2)*
     X        (U1(K + 1) - 2.*U1(K) + U1(K - 1))
```

Frequently the numerical analyst writes the difference scheme in operator notation as follows

$$\underline{U}^{n+1} = \underline{U}^n + \Delta t\, D(\underline{U}^n)$$

where $D(\underline{U})_j = (U_{j+1} - 2U_j + U_{j-1})/\Delta x^2$.

PDELAN permits the same type of subscript free, operator notation. It is possible to declare meshes, variables on these meshes, and finite difference operators which map variables or expressions from one mesh to another. The above problem would be written in PDELAN as follows (assume J = 128)

```
MESH MS(128)
  REAL U1, U2
ENDMESH
OPERATOR  DXX(W)
  FROM MS TO MS(I = 2..127)
     (W(I + 1) - 2.*W(I) + W(I - 1))/(DLX**2)
ENDOPERATOR
U2(1) = 0.
U2(128) = 0.
FORMESH  MS(J = 2..127)
  U2 = U1 + DLT*DXX(U1)
ENDFORMESH
```

Note that the mesh variables need not be subscripted within the scope of a FORMESH, we write U2 instead of U2(J). Mesh operators, such as DXX, can be applied only within the scope of a FORMESH. The operators can be used in a fairly complex way. For example, if DX and AX are mesh operators, then the following expression involving mesh variables U and V might be used

```
DX(C * AX(U) * DX(U + V)).
```

An earlier version of PDELAN was implemented at NCAR in 1971 [1]. We refer to the paper and documentation describing this version for a more complete definition and explanation of these operators. The earlier version had a different syntax and was rather awkward to use. The version described here

should be a considerable improvement over the first one. Also the new version allows implicit difference schemes to be written in operator notation. This is certainly its most powerful and useful feature. An example of an implicit scheme is the Crank-Nicolson scheme for the heat equation

$$\underline{U}^{n+1} = \underline{U}^{n} + \frac{\Delta t}{2} D(\underline{U}^{n+1} + \underline{U}^{n}).$$

We regard this as an equation for the unknown vector $\underline{U}^{n+1}$. This is a tri-diagonal system of equations for the unknown components of $\underline{U}^{n+1}$.

3.1 Mesh and variable declarations. This is a nested block of statements which name a mesh and assign its extent. The block also contains declarations of variables on this mesh. These variables are arrays with the same extent as the mesh. No memory space is required for the mesh, only for variables declared on the mesh. For example,

```
MESH UVTMESH(64,32)
  REAL U, V, T
ENDMESH
```

In this case the variables U, V, T are arrays of extent (64,32). The mesh name UVTMESH is entered into the symbol table and its associated information stored with it.

A mesh variable may be in addition an array. For example,

```
MESH UVTMESH(64,32)
  REAL ARRAY U, V, T(3)
ENDMESH
```

In this case U, V, and T are arrays of extent (64,32,3). To each point in the mesh (i,j) there are 3 values assigned. Each of these arrays can be regarded as three mesh variables $U_{i,j,1}$, $U_{i,j,2}$, and $U_{i,j,3}$. We will say more about this later.

An additional type of mesh variable, a PROJECTION variable, can be declared. For example,

```
MESH UVTMESH(64,32)
  REAL ARRAY U,V,T(3)
  REAL PROJECTION CS(,*)
ENDMESH
```

In this case CS is an array of extent (32). At each point (i,j) the mesh variable CS has the value CS(j). (Here $1 \leq i \leq 64$, $1 \leq j \leq 32$). The "*" indicates the subscripts which are not removed.

3.2 The mesh operator declaration. An example of a mesh operator declaration is the following

```
MESH MUV(64)
  REAL U1,U2
ENDMESH
MESH M(63)
  REAL SG
ENDMESH
OPERATOR DX(W)
  FROM MUV TO M(I = 1..63)
    (W(I + 1) - W(I))/DLX
  FROM M TO MUV(I = 2..63)
    (W(I) - W(I - 1))/DLX
ENDOPERATOR
```

A graphic representation of the meshes is

```
·  x  ·  x  ·          ·  x  ·
1  1  2  2  3         63 63 64
```

The MUV points are "." and the M points "x". The meaning of the DX operator

is to difference values at the surrounding points on the MUV mesh to obtain an approximate derivative at a point on the M mesh. If E is an expression on the MUV mesh, then DX(E) can be thought of as an expression on the M mesh. That is, DX(E) has a value at each point j on the M mesh, namely

$$DX(E)j = (E_{j+1} - E_j)/DLX$$

For example, if E is U1 + U2, then

$$DX(U1 + U2)(I) = ((U1(I + 1)+U2(I + 1)) - (U1(I)+U2(I))/DLX$$

The expression on the right is evaluated on the MUV mesh.

3.3 The FORMESH block. This is the means by which finite difference expressions are evaluated. For example, consider the parabolic equation

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}(\sigma\frac{\partial u}{\partial x}) + f(u)$$

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

The difference scheme might be

$$\underline{U}^{n+1} = \underline{U}^n + \Delta t\delta_x(\sigma\delta_x(\underline{U}^n)) + f(\underline{U}^n)$$

where the difference operator δ_x is

$$\delta_x(U)_{j+1/2} = (U_{j+1} - U_j)/\Delta x$$

$$\delta_x(U)_j = (U_{j+1/2} - U_{j-1/2})/\Delta x$$

Use the mesh, variable, and operator declarations given above in section 3.2. Then this difference scheme is written:

```
U2(1) = 0.
U2(64) = 0.
FORMESH  MUV(I = 2.,63)
  U2 = U1 + DLT*DX(SG*DX(U1)) + F(U1)
ENDFORMESH
```

Here F is a Fortran function subprogram, U^{n+1} and U^n are stored in U2 and U1, and σ is stored in SG.

The replacement statements within the scope of a FORMESH are evaluated for each value of I in the indicated range, in this case 2 through 63. The evaluation is performed in "parallel" in order to be compatible with parallel computers such as the Texas Instruments ASC or Seymour Cray's proposed new machine. This means that the right side is evaluated for all values of I before storage into the left side. Thus the evaluation is not the same as a conventional Fortran DO loop. For example

```
FORMESH  MUV(I = 2..63)
  U2 = DX(DX(U2))
ENDFORMESH
```

is equivalent to

```
      DO 100 I = 2,63
100   T(I) = (U2(I + 1) - 2.*U2(I) + U2(I - 1))/(DLX**2)
      DO 101 I = 2,63
101   U2(I) = T(I)
```

Here T is an array used for temporary storage of intermediate results. If there are two statements within the scope of a FORMESH, the computation for the first will be completed for all values of the index before computation is started on the second statement. This is completely different than a DO loop. The first version of PDELAN uses a DOMESH instead of this FORMESH. The DOMESH scope is executed in the same manner as a DO loop. The DOMESH does not execute in parallel. Also the syntax of the DOMESH resembles the DO. It uses a statement number termination instead of the block structure.

Next consider a difference operator which does not have a uniform definition throughout the mesh. For example,

```
MESH  MUV(128)
  REAL U1,U2,U3
ENDMESH
OPERATOR  DX(W)
  FROM MUV TO MUV(I = 128)
     (W(I - 2) - 4.*W(I - 1) + 3.*W(I))/(2.*DLX)
  FROM MUV TO MUV(I = 2..127)
     (W(I + 1) - W(I - 1))/(2.*DLX)
ENDOPERATOR
U3(1) = 0.
FORMESH MUV(I = 2.
  U3 = U1 - DLT*DX(U2)
ENDFORMESH
```

This operator has a different definition at I = 128 than it does in the interior of the mesh, $2 \leq I \leq 127$. The evaluation of the FORMESH cannot use DO loops from 2 to 128, the calculation must be broken down according to the definition of the operator. Therefore the FORMESH can be translated as follows (note that U3 does not appear on the right side of the replacement statement).

```
U3(128) = U1(128) - DLT*
          (U2(126) - 4.*U2(127) + 3.*U2(128))/(2.*DLX)
DO 100 I = 2,127
100  U3(I) = U1(I) - DLT*(U2(I + 1) - U2(I - 1))/2.*DLX)
```

The previous version of PDELAN cannot handle a mesh operator unless it has a uniform definition within the range of a DOMESH. The removal of this deficiency is an important improvement.

3.4 Implicit difference schemes. This allows the user to write implicit schemes about as easily as explicit ones. This is probably the most

useful and certainly the most powerful feature of PDELAN. To illustrate the method consider an implicit scheme for the following equation:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\sigma(x)\frac{\partial u}{\partial x}\right)$$

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

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

The implicit scheme is

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \left(\sigma(x_{i+1/2})\left(\frac{U_{i+1}^{n+1} - U_i^{n+1}}{\Delta x}\right) - \sigma(x_{i-1/2})\left(\frac{U_i^{n+1} - U_{i-1}^{n+1}}{\Delta x}\right)\right)/\Delta x$$

$$1 \leq i \leq M$$

$$U_0^{n+1} = U_{M+1}^{n+1} = 0$$

$$\Delta x = 1/(M + 1)$$

This is a tridiagonal system in the unknown vector $\{U_i^{n+1}\}$. We can write this equation in operator form as follows

```
U2 = U1 + DLT*DX(SG*DX(U2))
```

Here the declarations are given in section 3.2 above. The meshes are MUV and M. The variable SG is on mesh M. DX is defined on both meshes. If U2 is regarded as a vector unknown, then this equation defines a linear system of equations for the unknown U2. Because difference schemes are frequently nonlinear we will not attempt to solve the linear system directly. Instead we will allow the user to write out a linear difference equation in an unknown W and use this system to define a Jacobian matrix. Then this Jacobian matrix is used to solve a possibly nonlinear system by iteration. In order to illustrate the definition of this Jacobian we use this same linear parabolic problem. The following block defines the Jacobian for this example

```
SETJACOB AJ(W) ON MUV(I = 2..63)
   W = U1 - DLT*DX(SG*DX(W))
ENDSETJACOB
```

The unknown vector is $\{W_i\}$ with components in the range $2 \leqslant i \leqslant 63$. The expression defines a linear system of equations for W_i of the following form

$$\sum_{\nu=1}^{B_i} c_{i,\nu} W_{i+k_\nu} + f_i = 0$$

For this example the system is

$$c_{i,1}W_{i-1} + c_{i,2}W_i + c_{i,3}W_{i+1} + f_i = 0$$

That is, $k_1 = -1$, $k_2 = 0$, $k_3 = 1$. This can be written as a matrix equation

$$\underline{AW} = \underline{f}$$

Where A is given by

$$A_{ij} = \begin{cases} 0 & j \neq i + k_\nu \\ c_{i,\nu} & j = i + k_\nu \end{cases}$$

The SETJACOB block generates code to compute the entries in the matrix A. This matrix is stored in the mesh array AJ. The user must declare the array AJ and it must be large enough to accomodate the matrix A. In this case the declaration

```
REAL ARRAY AJ(3)
```

must be added to the MUV mesh block. The SETJACOB block will also generate a subroutine call to perform the LU decomposition of the matrix A. The result will be stored in AJ and the original matrix A will be lost. If pivoting is desired, then the command SETJACOB(PIVOT) should be used. In this case a larger AJ array must be declared.

The Jacobian is used according to the following example.

```
SOLVE  JACOB AJ ON MUV (I = 2..63)
    F(U2) = U2 - U1 - DLT*DX(SG *DX(U2))
ENDSOLVE
```

The expression in the SOLVE block defines a function of U2. The SOLVE block generates code to perform a single step of a Newton iteration using the Jacobian AJ. That is, the following equation is solved for $\underline{\delta W}$

$$A\underline{\delta W} = -F(U2)$$

Here F must be a mesh variable declared on MUV by the user. The value of the expression within the SOLVE block is stored in F. Code to obtain an updated value of U2 from the solution $\underline{\delta W}$ of the Jacobian system,

$$\underline{U2} = \underline{U2} + \underline{\delta W}$$

is generated by the SOLVE command. Since the Jacobian AJ was defined for $2 \leq i \leq 63$, the vector U2 is updated over the same range.

We only allow difference schemes which are implicit in one dimension. This means that the mesh subscript list in the SETJACOB statement can have only one vector subscript. A scheme which is implicit in two dimensions is usually two expensive because the bandwidth of the Jacobian matrix is too large. However, the Jacobian could be defined on a two dimensional array. For example

```
MESH M(128,64)
  REAL U2,U1
  REAL ARRAY AJ(3)
ENDMESH
OPERATOR DXX(W)
```

```
  FROM M TO M(I = 2..127, J = *)
     (W(I + 1,J) - 2.*W(I,J) + W(I - 1,J)/(DLX**2)
ENDOPERATOR
OPERATOR DYY(W)
  FROM M TO M(I = *, J = 2..127)
     (W(I,J + 1) - 2.*W(I,J) + W(I, J - 1))/(DLY**2)
ENDOPERATOR
   .
   .
   .
SETJACOB AJ(W) ON M(I = 2. .127, J = *)
  W - U1 - DLT*(DXX(W) + DYY(U1))
ENDSETJACOB
```

The Jacobian is still a tridiagonal matrix, but it is defined over a two dimensional mesh and thus has order 127 x 64. The array AJ has extent (128,64,3). The term AJ(W)(I) indicates a scheme implicit in I.

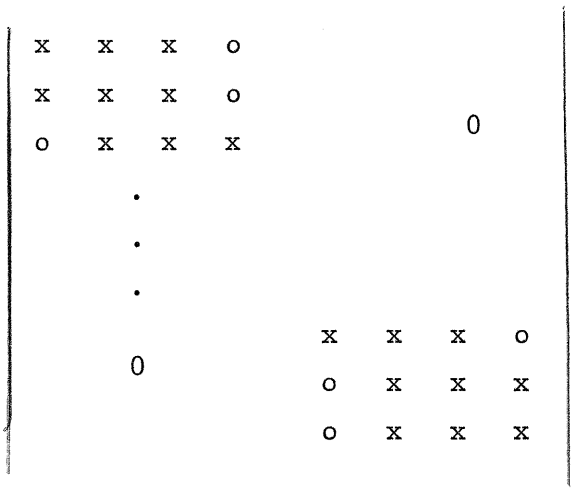
The Jacobian matrix should allow for difference schemes which have the same number of points in the stencil throughout the mesh but may be shifted near the boundary due to one sided difference approximations. For example, if the one sided approximation

$$(-3U_1 + 4U_2 - U_3)/(2\Delta x)$$

is used along with the centered formula

$$(U_{i+1} - U_{i-1})/(2\Delta x)$$

then the Jacobian matrix would have the following structure



The AJ mesh array containing the Jacobian should have extent 3 in this case (assume no pivoting).

The language should also handle implicit systems of equations. For example, consider

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\sigma(u)\frac{\partial u}{\partial x}\right) + a_1 e^{d(u+v)}$$

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\sigma(v)\frac{\partial u}{\partial x}\right) + a_2 e^{d(u+v)}$$

In this case the Jacobian might be block tridiagonal with 2x2 blocks. Here we are not using the true Jacobian because we are not using the derivative of the function $\sigma(w)$. We assume SGF is a function subprogram.

```
MESH MUV(64)
  REAL U1,U2,V1,V2, AJ(7)
ENDMESH
MESH M(63)
ENDMESH
OPERATOR DX(W)
  FROM M TO MUV(I = 2..63)
     (W(I) - W(I - 1))/DLX
```

```
  FROM MUV TO M(I = 1..63)
     (W(I + 1) - W(I))/DLX
ENDOPERATOR
OPERATOR AX(W)
  FROM MUV TO M(I = 1..63)
     (W(I + 1) + W(I))*.5
ENDOPERATOR
SET JACOB AJ(U,V) ON MUV(I = 2..63)
  U-U1-DLT*DX(SGF(AX(U1))*DX(U)) + A1*ALP*EXP(U1 + V1)*U
  V-V1-DLT*DX(SGF(AX(V1))*DX(V)) + A2*ALP*EXP(U1 + V1)*V
ENDSETJACOB
```

The SOLVE command is similar, except that the mesh function F required to hold intermediate results is an array of extent 2.

4. Extensions. These are features that we would like to add after we get the language described in the previous two sections running. For difference schemes which will not fit in fast memory, the following memory allocation scheme is useful. The data for such schemes is usually transmitted by blocks which consist of a "section" of a mesh. For example, in a three dimensional problem such a section would be all points (i, j, k) with k fixed and i and j ranging through all possible values. If the data for the scheme consists of three variables U, V, W each of dimension (50, 50, 40), then only a few sections will be in fast memory, perhaps four sections, the rest will be located in bulk storage of some kind. The bulk store should be accessed in large blocks. In this case the block would consist of one section containing 7500 words. That is U(*, *, K), V(*, *, K), W(*, *, K). Note that U(*, *, K) represents 2500 words.

$$U(I, J, K) \text{ for } 1 \leq I \leq 50,\ 1 \leq J \leq 50.$$

The Fortran dimension statement

```
DIMENSION U(50,50,4), V(50,50,4), W(50,50,4)
```

will not group this data properly. The variables in the section are not stored contiguously. The following declaration will rearrange the data allocation

```
ARRAYBLOCK NAM(4,LEN)
  REAL ARRAY U,V,W(50,50,*), ;+
             CS,SN(50,*), TH(*)
ENDARRAYBLOCK
```

The "*" is replaced by the 4. The variable LEN must be an integer variable. It will be set equal to the length of each of the 4 sections in a DATA statement in the Fortran object code. In this case the section length is 7601. The output will contain a DIMENSION statement of the form

```
DIMENSION U(200,50), V(200,50), W(200,50)
X         , CS(200), SN(200), TH(4)
```

If these variables are not within a COMMON block they will all be placed in a labeled COMMON in order to be sure that they will be stored together. That is

```
COMMON/TL0001/U,V,W,CS,SN,TH
```

Then U(I,J,K) where 1 < K < 4 is accessed by

```
U(I + K*LEN - LEN,J)
```

where LEN is replaced by its constant equivalent to yield

```
U(I + 7601*K - 7601,J)
```

A block I/O transmission can then be given in the form (on the CDC system)

```
BUFFERIN(7,1)(U(1,1,K),TH(K))
```

Data initialization. This performs the same function as the Fortran DATA statement and is implemented by means of a DATA statement in the output object code. However, the syntax is more consistent with the repetition used in FORMAT statements and avoids the use of * as a repetition indicator. This permits the use of expressions involving macro time variables [4] in the initialization. An example is

```
REAL ARRAY U(50,50) = (50(0.),50(1.),48(0.))
```

General array extent. We would prefer to have array declarations in the form

```
REAL ARRAY U(-10..10,5)
```

This is equivalent to U(-10..10,1..5). The output for a reference of the form U(I,J) would be translated to U(I + 11,J) and the dimension statement would be of the form

```
DIMENSION U(21,5)
```

Since the CDC compilers only allow three subscripts, it would be desirable to allow more than three dimensions in PDELAN and reduce to three in the output. For example if

```
COMPLEX CS(10,20,5,2)
```

then the reference CS(I,J,K,L) would become

```
CS(I,J,K + 5*L - 5).
```

We assume it is preferable to reduce to three subscripts rather than one because some compilers will not optimize the complex one dimensional subscripts as well as the three dimensional especially if the inner DO loop is over one of the first two subscripts.

Recursive procedures and dynamic storage allocation. Within a given

subprogram a set of procedures can be defined. These can contain variable declarations which are local to the procedure. These procedures allow recursive calls and are implemented by means of a stack which is simply an array in the containing Fortran subprogram. This provides dynamic storage allocation, at least within the containing subprogram. The procedures can be called only within this subprogram.

A format free I/O statement in the NAMELIST style. This would differ from NAMELIST in that the variable list would appear in the I/O command rather than in a separate declaration. A macro can be used to place the same list in several different commands. Also, the input command can work in two modes. If there are no identifiers of the form "ID=" on the input card, only a list of numbers, then these numbers are input according to the I/O list. If an identifier "ID=" appears, then the input will be governed by the identifiers. These identifiers must appear in the I/O list within the I/O command. For example

```
INLIST(7,NAM) X,Y,A(1..10,5..10),;+
      B(*,3,2.
```

The input card might appear in the form

```
$NAM 1.2,3.7,B(1,1,2) = 37. $
```

or it might have the form

```
$NAM 1,7.,21, . . . . . .
     . . . . . $
```

or

```
$NAM A(1,7) = -21. $
```

A format free output is also included. For example

```
OUTLIST(6,NAM) 'CASE21',X,Y,B(1,1..20,7)
```

The output is labeled. That is, the values are printed in the form

```
OUTLIST NAM CASE 21 X = 1.2  Y = 3.7
            B(1,1..20,7)
31.2   -21.7   8.1E + 5 . . . . . .
END OUTLIST NAM
```

A means to set the number of significant figures printed is provided. For example

```
OUTLIST(6,NAM,SIGNIF(E10.3,I6,D13.6)) . . .
```

File management and graphics. A very important aspect of a language for PDE is the I/O facilities within the language. This should include an easy way to generate graphs and contour plots from arrays. Such a facility is included in the first version of PDELAN [2]. The graphics in this earlier version should be improved in various ways. For example, the syntax of these graphics commands should be improved. Also the graphics commands should be organized into a hierarchy most of which is machine independent. It should be possible to output the graphs and plots in a form suitable for efficient transmission over phone lines and output on a variety of graphics devices [6]. However, this is a large problem in its own right, and we have decided to concentrate on the finite difference aspects of the language.

The design of a file management system and data structures suitable for PDE codes is an important problem and should be a part of the language. However, we have not put any effort into this part of the problem.

REFERENCES

1. J. Gary and R. Helgason, "An Extension of FORTRAN Containing Finite Difference Operators", Software-Practice and Experience, 2, pp. 321-336 (1972)

2. G. Locs and J. Gary, "A FORTRAN Extension for Data Display" to appear in IEEE Transactions on Computers

3. H. Mills, "Topdown Programming in Large Systems", in "Debugging Techniques in Large Systems", Rustin(ed), Prentice Hall, Inglewood Cliffs, N.J. (1971)

4. J. Gary, "A macro preprocessor for a FORTRAN variant", Computer Science Department, University of Colorado (1974)

5. W. Gear, "What do we need in programming languages", Proceed. Math Software Conference, Purdue, (1974)

6. J. Adams and J. Gary, "Compact Representation of Contour Plots for Phone Line Transmission", Comm. ACM, Vol. 17, No. 6, 333-337 (1974)