

AFOSR - TR - 75 - 1530

FINAL TECHNICAL REPORT

AFOSR-74-2732

SOFTWARE TOOLS FOR CLIMATE SIMULATION

By

John M. Gary
Professor

Department of Computer Science
University of Colorado
Boulder, Colorado 80302
Telephone No. 492-8842

Sponsored By

Advanced Research Projects Agency
ARPA Order No. 2792
Program Code 4P10

Duration: 6/1/74 - 8/31/75

Contract Amount: \$37,595

Approved for public release;
distribution unlimited.

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is
approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

ADA017664

DDC
RECEIVED
NOV 26 1975
RECEIVED

12

409496

ACCESSION BY	
NTIS	DATE
DOO	TIME
CHARACTER	
POSTAL ADDRESS	
BY	
DATE	
A	

Unclassified


SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

18. REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER AFOSR-TR-75-1534		2. GOVT ACCESSION NO.	
3. TITLE (and Subtitle) 6. Software Tools for Climate Simulation.		3. RECIPIENT'S CATALOG NUMBER	
4. AUTHOR(s) 10. John Gary M. Gary		5. TYPE OF REPORT AND PERIOD COVERED 9. Scientific - Final Rept.	
7. AUTHORING ORGANIZATION NAME AND ADDRESS University of Colorado Computer Science Department Boulder, Colorado 80302		6. PERFORMING ORG. REPORT NUMBER	
8. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency/NMR 1400 Wilson Boulevard Arlington, VA 22209		7. CONTRACT OR GRANT NUMBER(s) 15. AF-ARPA Order-2792	
9. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research/NP 1400 Wilson Boulevard Arlington, VA 22209		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62701E AO 2792	
11. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		12. REPORT DATE October 1975	
13. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		13. NUMBER OF PAGES 84	
14. SUPPLEMENTARY NOTES TECH, OTHER		14. SECURITY CLASS. (of this report) Unclassified	
15. KEY WORDS (Continue on reverse side if necessary and identify by block number) MACRO PREPROCESSOR FORTRAN		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The macro preprocessor provides a standard type of macro replacement with arguments for FORTRAN programs. Conditional macro expansion and macrotime arithmetic computation is included. Structured control statements (IF...THEN..., etc.) are added to FORTRAN. A preprocessor for FORTRAN containing macro capability, vector arithmetic, and finite difference operators was designed, but only partially			

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

implemented. A first version of a package for the solution of hyperbolic-elliptic equations was implemented, but is not yet documented.



Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FINAL REPORT

Software Tools for Climate Simulation

AFOSR grant 74-2732

Principle investigator: John Gary
Computer Science Department
University of Colorado
Boulder, Colorado 80302

Our objective was the design and implementation of software tools to aid in the programming of atmospheric simulations based on partial differential equations. In the course of this project we designed three such software tools.

- 1) A macro preprocessor.
- 2) A vector extension to Fortran intended either for serial or parallel computers.
- 3) A finite difference extension to Fortran.

These are described in three CU technical notes which are included with this report.

We did produce these initial designs, but we failed to implement much of it. A lexical scanner of about 700 lines was written and mostly debugged. A skeleton of the finite difference operator extension was written (around 2000 lines of Fortran), however it is not complete. It became clear that we needed to improve the methods we were using to implement these designs. Our intention is to drop the code we did produce and start over using a syntax-directed compiler generator. There are several possibilities for such a generator, and we have not yet selected one. It will take us some time to redesign our software to use the compiler generator and to become familiar with the generator. Therefore we will probably not resume the implementation until the summer or Fall of 1976.

This summer I designed and programmed a software package (HYPPACK) for the solution of a class of partial differential equations common in atmospheric simulation. I have not given up on the three software tools mentioned above. Within the next two or three years I hope to at least complete a first version of the macro preprocessor and the vector extension. The HYPPACK package will give us a good program package to which these tools can be applied and permit comparisons to be made. Also we wanted to have some tangible output during the rather lengthy period required to learn how to apply the compiler generator to our problem.

This software package is designed to solve a system of prognostic equations coupled with a single elliptic equation. Examples of such systems are the Navier-Stokes equations, an anelastic cloud model, or a vorticity-stream function model. The code allows a second or fourth order finite difference approximation to be used in space. A second order leapfrog or fourth order Runge-Kutta-Fehlberg ODE solver can be used for the time discretization. We use a code written by John Adams at NCAR, which is based on deferred corrections, to solve the elliptic equation. This is in turn based on the package of Sweet and Swartztrauber for the direct solution of elliptic equations. The user of this package need only program some simple subroutines which define the coefficients in the differential equations in order to solve these equations. One of the main problems with this code is the proper specification of the boundary conditions for the prognostic equations. We have enclosed a fourth CU technical note which deals with this question. Another problem is the computational efficiency of the code which I suspect is now not too good due to the generality of the code. We plan to attempt to solve this problem by the use of a macro-preprocessor,

initially IFTRAN or MORTRAN2, eventually our own. We also plan to use the macro-processor to generate two versions from a single program, one using LCM variables and another using variables contained in central memory. One, two and three dimensional versions could also be generated from this same basic program. Although there are several interesting programming questions connected with the development of this package, the most serious problems will probably be of a numerical nature. The code is currently running although we have not checked out the elliptic equation solver and the graphics output is not yet included. Also considerable cleanup of the code is required and the users manual must be greatly improved. The code I wrote is about 1800 lines currently, and the elliptic equation solver from NCAR is around 2000 lines. Both are written in Fortran and should be reasonably portable.

A Macro Preprocessor for a
FORTRAN Dialect

John Gary

Department of Computer Science
University of Colorado
Boulder, Colorado
80302

August 1975

This is a revision of Report #CU-CS-054-74, August 1974.

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

0. Introduction. Our objective is to provide a macro preprocessor for a language similar to FORTRAN. This is a "mesh operator" language (PDELAN) intended for the construction of finite difference codes for partial differential equations. It is described in a companion report [8]. The PDELAN compiler generates a FORTRAN object program. These ideas have not been implemented at the time of this writing.

The macro preprocessor could be used with a modified version of FORTRAN in which blanks are delimiters and certain "keywords" such as IF, FORMAT, DO, etc., are also reserved words which cannot be used as names by the user. The syntax of the macro preprocessor is intended to be natural to a FORTRAN programmer. Its most frequent application, such as the propagation of COMMON declarations throughout subroutines, should be easy to remember and use. It would place the error messages from the PDELAN compiler in the original source code. PDELAN contains structured control statements such as the following [13]:

```
IF ... THEN ... ELSE ... ENDIF
```

```
REPEAT ... UNTIL ... ENDREPEAT
```

The macro preprocessor permits long names (up to 29 characters) which are shortened to 6 characters on output (with name conflicts avoided). It also permits, through conditional macro expansion, the generation of code which is more machine independent. It is not intended to allow user defined language extension, except trivial extensions. It seems to us to be too difficult to include good error diagnostics in a macro extension. Also the macro extensions are probably too slow. Thus we place the preprocessor for the mesh language in a compiler which follows the macro preprocessor. This compiler for the PDELAN language is described elsewhere [8].

This is a continuation of work started at NCAR in 1970. A first version of PDELAN for solving partial differential equations was developed by Gary and Helgason [1]. A set of graphics commands was added to PDELAN by Löcs and Gary [2]. This code is in light use at NCAR (fewer than 10 people use it). However Helgason wrote an improved version of the macro preprocessor called FRED which has been further improved by Dave Kennison [5]. FRED contains macro capability, subscript bounds checking, a "TIDY" feature to renumber and indent a deck, and other features. However, it does not contain difference operators or graphics commands. Here we propose a macro facility somewhat different from FRED. It will not have the subscript bounds checking or the TIDY feature. It will have the capability to modify and generate tokens, perform conditional macro expansion, and execute macro-time expressions. It will be based on the PDELAN language in which blanks are delimiters and the keywords are reserved. PDELAN contains many FORTRAN features such as COMMON, SUBROUTINE, the same type of data structures (or lack of them), and the same I/O structure. However, it does have structured conditional and repetitive statements such as the PASCAL language or the preprocessors RATFOR, FLECS, MORTRAN, IFTRAN [14].

One of our main objectives is to make the macro syntax appear natural to an experienced FORTRAN programmer. Thus the macro definitions are similar to subroutines and the macro calls are similar to a FORTRAN function call. The macro-time statements and conditional compilation are similar to FORTRAN although we did use the IF ... THEN ... ELSE

construction. We feel our macro preprocessor is easier for a FORTRAN programmer to learn and remember than the macro preprocessor for FORTRAN called MP/1 by Macleod [6] or the preprocessor imbedded in the LLL FORTRAN [3]. MP/1 is more powerful since it is a pattern matching macro and it has a good set of macro-time commands. The LLL macros do not have much macro-time capability. The macro processors designed for FORTRAN (FLECS, MORTRAN, IFTRAN, RATFOR) that we are familiar with do not have enough macro-time commands or sufficiently flexible macros.

1. The syntax. In order to simplify the macro preprocessor a modified FORTRAN syntax is used in which blanks are delimiters and the keywords are reserved. Tokens can be recognized by a lexical scan ahead of the macro expansion and thus also ahead of the syntactical analysis. Macro calls are recognized by the appearance of a macro name with no special delimiting character or pattern matching required. The syntax of a macro definition is similar to that of a FORTRAN subroutine or function. The macro definitions can be stored as a string of tokens rather than a character string which should permit a more compact internal representation. The token types include integer and real constants, identifiers, and the operators or delimiters + - * / ** .. : () =. The delimiters \$ # and ' are also used. The first two are used in conjunction with the macros, the last is used to delimit character strings. The last three characters have a different form on the keypunch than the teletype form given here. Various period delimited operators are also used (.LE. .LT. .GT. .NE. .EQ. .AND. .OR. .NOT. .NULL.). Boolean constants (10B) and Hollerith constants ("ABCD" 4HABCD) are also used. Format specifications must be included as tokens (for example, 2E10.3, 2P, 2PE10.3). A nonblank character in column six denotes a continue card, which is almost the standard FORTRAN convention. Columns 1 thru 5 can be used only for statement labels.

In order to use a compiler writing system such as that described by Cohen [15], we might allow only identifiers, integers and character delimiters as tokens. These tokens can be recognized by a lexical scanner built into the compiler writing system.

The statements are almost "free-form". They may start anywhere on the card after column six. Column six is used to mark continuation cards. Only statement labels may appear in Columns 2 thru 5. The statement termination ";" can be used to place more than one statement on a card. The added statements can be labeled. If the first token of the statement is an integer constant, then this constant is a statement label. In order to produce better error diagnostics, it was decided not to attempt to make the macro preprocessor independent of the language syntax. It is also an advantage if the macro preprocessor can recognize statement labels. Eventually we will even partially parse the input ahead of the macro expansion in order to indent the input source program according to the nesting level of the source statements.

2. The macros and macro-time statements. All the "macro-time" statements are preceded by a name which starts with the delimiter "%" or by the delimiter "%" itself. The character used for this delimiter is system dependent, it does not exist on the KRONOS timeshare system. These delimiters can be easily altered. There are two types of macro, a statement macro and an expression macro. The body of the statement macro consists of one or more statements. Such a macro name can be used as a statement in the PDELAN language. The macro name is an abbreviation for the macro body which replaces it prior to compilation of the PDELAN program. For example, consider the following abbreviation of a subprogram header block.

```
%MACRO COMMONBLKA
COMMON A,E,C,D,U
COMMON /B/ H,DLX,DLY
INTEGER HX,UX
%ENDMACRO
```

The body of the second type of macro is an expression. Both types of macro may have arguments. The following is an example of an expression macro definition and usage.

The definition is

```
%EXPMACRO FN(F,X) = EXP(F(X))
```

An example of the usable is

```
T = FN(SIN,W+1.)
```

The second statement would expand into

```
T = EXP(SIN(W+1.))
```

The definition of this type of macro is a single statement and therefore does not require the %ENDMACRO statement. More than one expression macro may be defined in a single statement. For example

```
%EXPMACRO UT = U(N1), U2 = U(N2)
```

Macro calls may be nested, that is a macro definition may contain a call to a macro. Recursion is allowed, that is a macro may call itself. However, a macro definition may not contain another macro definition. A statement number may be placed on a statement macro call, for example

```
100 INITIALIZE
```

In this case a CONTINUE statement is generated ahead of the body of the macro INITIALIZE.

The scope of the macro definitions. When a macro definition

is encountered the name of the macro is placed in the symbol table and its body is placed on a "definition stack." To avoid overflow of this stack it is possible to limit the scope of a macro definition.

The command %MACROBLOCK indicates the start of a "block." All

macros defined within the block are regarded as local to the block. When the command %ENDMACROBLOCK is encountered all these local macros will be removed from the symbol table and their definitions popped from the definition stack. Macros not within such a block are regarded as global and cannot be removed from the definition stack. In addition, macros defined within a subprogram are available only within that subprogram. The %MACROBLOCK and %ENDMACROBLOCK commands cannot occur within a subprogram.

Macro formal parameters are local to the definition and can be used as names elsewhere in the source deck. A macro definition must appear ahead of its first usage or call.

3. Macro-time statements. These are declarations, expressions and the conditional control of macro expansion. These statements are executed at "macro-time," that is, when the expansion is performed. In this first version only integer variables can be declared at macro-time, and these variables must be global, they cannot be declared within a macro definition or a macro block. An example of a macro-time declaration of an integer variable is the following

```
%INTEGER N,M
```

There must not be a blank following the %, it is part of the identifier.

Expressions involving the operators + - * / and integer variables are allowed. For example

```
% M = (N+1)/2
```

The % alone denotes a macro-time replacement statement.

Conditional expansion. A Boolean expression can be evaluated at macro-time to control macro expansion. For example

```
%IF N .LT. 0 THEN
    PRINT 250, X,Y,Z
    KC = KC+1
%ENDIF
```

If the Boolean is true then these two statements will be included in the code to be compiled, otherwise they will not. Note that THEN is a reserved word. The %IF THEN %ELSE %ENDIF compound statement must not overlap a macro definition, it must be completely within or completely outside a macro definition.

The Boolean expression can involve relational operations on macro-time integer expressions. Comparisons(.EQ. or .NE.) between character strings are allowed. For example

```
%MACRO M(X)
    %IF X .EQ. A THEN CALL SUB(A): %ENDIF
    .....
%ENDMACRO
```

If the actual argument of a macro call of this macro M is the token A, then the code

```
CALL SUB(A) ;
```

will be added to the output. These conditional statements can be nested. Note that %ELSE and %ENDIF must be used and not ELSE or ENDIF.

4. Generated symbols. Statement labels distinct from any used in the source code may be generated by use of the symbols #L(3) or #L(N) where N is a macro-time integer variable. Within each macro call #L(3) represents the same statement label different from the label generated by #L(3) on other calls of this macro. The label generated by #L(N) will depend on the value of N within a given call of the macro, and

will also be different on different calls of the macro. As an example consider

```
%MACRO SUM(S,A,N)
  S=0.
  DO #L(1) K=1,N
    #L(1) S=S+A(K)
  %ENDMACRO
```

A generated name, distinct from other names in the program, which starts with "I" is obtained from #I(1) or #I(N). For names starting with "E" the symbols #E(1) or #E(N) are included. These can be used to generate "local" variables within a macro call.

In addition the macro preprocessor will shorten long identifiers to 6 characters. This will be done by truncation provided no conflicts arise, otherwise the last character or characters will be modified until a unique name is obtained. This modification is carried out for each subprogram. Therefore external names should not exceed 6 characters.

5. Commands to control expansion. These are identifiers whose first character is %. They constitute a complete statement in PDELAN. Not all of these will be implemented initially.

%NOLIST - cease listing source code

%LIST - list source code

%NEWPAGE OR

%NEWPAGE(ID) - skip source listing to new page, Print identifier ID on top of page.

%LONGNAMES - at end of preprocessor source listing print longnames and their shortened form

%NAMEMAP - list line numbers where each identifier is used

%FINUS - end of source check

%MACROMAP - list line numbers where macros are defined or used.

%MACROBLOCK - indicate start of a block of macro definitions

%ENDMACROBLOCK - indicates the end of a block of macros. When this command is encountered the macros in the block will be popped from the macro definition stack. Therefore macros within the block are not defined following this card. This permits the use of "local" macros and may prevent overflow of the macro stack.

6. Error diagnostics. When an error is detected we will print the error message with the listing of the original source. If the error is inside a nested set of macro calls, then the error message will be printed after the innermost macro call. However, the name of each macro called within the nested set will be printed along with the line number from which it was called. We will try to print the innermost line of code along with a pointer to the token at which the scanner stopped, and of course a message to indicate the type of error. The error recovery will try to resume at the next statement inside the innermost macro. We are using recursive descent to parse the PDELAN variant of FORTRAN, and we may have some difficulty achieving good error recovery.

Note that good error messages and recovery probably requires the compiler which follows the macro expansion to be designed together with the macro preprocessor. Errors which the compiler finds should probably be printed in the original source code which is input to the macro preprocessor and not in the input to the compiler which is the output from the macro preprocessor. We regard the output of good error messages in this context as a difficult problem. Note that our mesh operator variant of FORTRAN (PDELAN) produces FORTRAN code as output which must then be input to a FORTRAN compiler. The code passed to the FORTRAN compiler by the mesh language compiler should never contain any undetected errors. The user should not be required to inspect the FORTRAN output any more than a user need look at assembly language output from a FORTRAN compiler.

7. Desirable additions. We should allow array % variables and % variables of real or double precision type. For example

```
%REAL ARRAY A, B(10)
```

```
%INTEGER XM(10)
```

The usual FORTRAN functions should be available within the % statements. Also, % variables could be initialized within their declarations. For example

```
%REAL CPI = 4.*ATAN(1.)
```

```
DATA PI/CPI/
```

The DATA statement in the output program sets the value of PI to π .

Note that a % variable which appears outside a % statement is transformed to a character string in the output program. Thus, CPI becomes 3.1416... using the number of digits carried by the machine.

A macro-time repetition command. This is the

```
%REPEAT ... % UNTIL ... %ENDREPEAT
```

command. The Boolean expression following %UNTIL is the same type as that used with the %IF command. Note that we have not included labeled % statements or a %GOTO. This is probably a mistaken position from which we will have to retreat.

The & macro formal parameter marker. A formal parameter in a macro definition can be indicated by an identifier as in a FORTRAN subroutine. An integer constant or an INTEGER variable prefaced by a & could also be used. For example &(1) or &(N). This type of notation is found in many macro processors. For example suppose the macro has a variable number of arguments and is to generate a subroutine call for each argument.


```

%INTEGER N
%MACRO PLOTV
% N=1
%REPEAT
  CALL PLOT(&(N))
  % N=N+1
%UNTIL &(N) .EQ. NULL
%ENDMACRO

```

Then the macro call

```
PLOTV(U,V)
```

would generate the statements

```

CALL PLOT(U)
CALL PLOT(V)

```

Note that we have added a macro-time repetition command

```
%REPEAT ..... %UNTIL .....
```

We have also added a reserved word to represent the null token, namely NULL.

Symbol table information. Our macro preprocessor is coupled to the compiler for PDELAN. This compiler has a symbol table containing information such as arithmetic type, array dimensions, etc. This information should be available within % statements. This can be done by providing another function call for the % statements (a suggestion of Tom Wright from NCAR). For example

```

% N1 = ARITHTYPE (&(1))
% N2 = DIMEN(&(1))
% N3 = DIMEXTENT(U,2)

```

The macro actual parameter substituted for &1 must be a single token which is an identifier. If the variable U is declared

```
REAL U(20,30,10)
```

and &(1) = U, then N2 = 3 and N3 = 30.

A parenthesized list could be used as an actual argument.

For example, consider the macro call

```
MAC(X,(A,B),Y)
```

In this &(1) would be replaced by X, &(2) by A,B. The parentheses are dropped. Also &(2).1 is A and &(2).2 is B. A similar construct is used in the macro preprocessor of Macleod [6].

A second type of expression macro. This type of macro allows macro-time conditional statements to be used to select the expression which defines the macro. A new type of statement is allowed which permits a concatenation of tokens. This statement is defined by occurrence of the macro name on the left of the "=" with a string of tokens separated by blanks on the right. The macro name can also appear on the right. The macro name represents a string of tokens which is null when the macro is entered. The contents of this string can be changed by the concatenation statement as the sequence of macro-time statements within the expression macro are executed. Only macro-time statements are allowed in the body of this expression macro. The contents of the string when the %RETURN is executed define the macro. For example consider the following definition of an A format specification. Here NWORD is a global %INTEGER variable. This type of macro is distinguished from the previous %EXPMACRO by the absence of the "=" in the %EXPMACRO statement.

```
%EXPMACRO AFORM
  %IF NWORD .EQ. 4 THEN
    % AFORM = 20A4 %ENDIF
  %IF NWORD .EQ. 10 THEN
    % AFORM = 8A10 %ENDIF
  % AFORM = FORMAT (AFORM)
  %RETURN
%ENDEXPMACRO
```

The following macro may not produce the same result as the one above.

```
%EXPMACRO AFORM
  %INTEGER N
  % N = 80/NWORD
  % AFORM = FORMAT(N A NWORD)
  %RETURN
%ENDEXPMACRO
```

When a macro-time %INTEGER variable appears in the expression macro string it is replaced by the character string which represents its value. If NWORD = 4, then the token string (N A NWORD) consists of 5 tokens (20 A 4). If these tokens are converted back into a character string before the output is given to a compiler, then this %EXPMACRO should produce the same result as the first macro. If the tokens are input directly to the syntactical scan of a compiler, then these two macros might not yield the same result. They would certainly output different token strings.

Macro-time string variables might be included which could increase the power of this second type of expression macro.

A pattern matching macro. It should be possible to insert a pattern matching macro into the preprocessor ahead of the lexical analysis which produces the tokens. If no pattern macros are defined, then the pattern matching could be suppressed. Pattern matching and token generation with symbol table lookup are said to cost about the same [11]. The pattern macro could be modeled after those in MORTRAN2 [14] which is fairly easy to understand and implement and probably provides sufficient power.

REFERENCES

- [1] J. Gary and R. Helgason, "An Extension of FORTRAN Containing Finite Difference Operators", *Software-Practice and Experience*, 2, 1972, pp. 321-336.
- [2] G. Locs and J. Gary, "A FORTRAN Extension for Data Display", to appear in *IEEE Transactions on Computers*.
- [3] J. Martin, R. Zwakenberg, S. Solbeck, "LTSS Livermore Time-Sharing System", Computation Department, M-026, Lawrence Livermore Laboratory, Livermore, California.
- [4] S. Mandil, "A General Purpose 'Problem-to-Program' Translator", *Comp. Bull.* 16, 1972, pp. 492-497.
- [5] D. Kennison, "FRED, A FORTRAN Editor", NCAR Scientific Library, National Center for Atmospheric Research, Boulder, Colorado, 80302, 1973.
- [6] I. Macleod, "MP/1 - A FORTRAN Macroprocessor", *Comp. Jour.*, 1970.
- [7] H. Mills, "Topdown Programming in Large Systems", in "Debugging Techniques in Large Systems", Rustin(ed), Prentice Hall, 1971.
- [8] J. Gary, "PDELAN: A Mesh Operator Variant of FORTRAN", Department of Computer Science, University of Colorado, Boulder, Colorado, 80302, 1974.
- [9] P. Brown, "The ML/I Macro Processor", *Comm. ACM*, Vol. 10, pp. 618-623, 1967.
- [10] W. Waite, "A Language-independent Macro Processor", *Comm. ACM*, Vol. 10, pp. 443-440, 1967.
- [11] P. Brown, "A Survey of Macro Preprocessors", *Annual Review in Automatic Programming*, pp. 37-88, 1969.
- [12] S. Pollack and T. Sterling, "A Guide to PL/I", Holt, New York, 1969.
- [13] D. Knuth, "Structured programming with GOTO statement" *Computing Surveys*, 6, pp. 261-301 (1974)
- [14] Workshop on FORTRAN preprocessors for numerical software, Jet Propulsion Laboratory, Pasadena, Calif., Nov. 1974.
- [15] J. Cohen, "Experience with a conversational Parser Generating System", *Software-Practice and Experience*, Vol. 5, pp. 169-180 (1975).

A VECTOR LANGUAGE FOR THE SOLUTION
OF PDE PROBLEMS

by

John Gary
Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-068-75

April 1975

This research was supported by an ARPA grant AFOS-74-2732.

1. Introduction. We are mainly concerned with the inclusion of a vector capability within a higher level language used to construct codes for the solution of partial differential equations. The vector constructs are modeled after those in APL, except that we impose restrictions in order to produce efficient programs on a parallel computer such as the Texas Instruments ASC, the Cray computer, or a new version of the Illiac IV. The language should be designed so that it is easy for the user to differentiate between those constructs which can be compiled efficiently and those which can not. We should also impose restrictions so that implementation of an efficient compiler is not too difficult. The language should be designed so that it can be implemented on a serial computer by a preprocessor which generates an object FORTRAN program.

2. Vector definition. The usual scalar variables and expressions are included. The scalar types are logical, real, complex, double, integer. Arrays can be declared with fixed dimensions, for example

```
REAL ARRAY A[10,20], B[-10..10,30]
```

Note the use of the brackets. A[10] is equivalent to A[1..10]. The language could permit dynamic storage allocation, and perhaps recursive procedure calls. If so, then an efficient static storage allocation should be included for scalar operations involving arrays.

Vectors can be defined explicitly as follows //3., X+2.,Y,0.//. Unlike APL the // is used to explicitly bracket vector definitions.

An increment vector can be defined as 2..13..2 which has the vector value //2,4,6,8,10,12//. This definition was suggested by Gerry Fisher of the Burroughs Corporation. It is similar to a construct used in TRANQUIL. This vector is intended for use as an array subscript. It would probably have a different internal representation than the other vectors. It permits efficient access to the components of a vector. The increment vectors are declared as follows.

```
INC_VECTOR IV,JV
```

and can be set as indicated below. Here N,M, and I can be integer valued expressions.

```
IV=N..M..I
```

The only arithmetic operation which can be performed with these vectors is addition or subtraction with a scalar integer expression, for example,

```
IV=(2..37) + 5
```

Then IV has the value 7..42, or 7..42..1. A vector may be created from an array by use of a vector subscript. For example,

```
REAL ARRAY A[-10..10] , B[30]
INC_VECTOR IV
IV=7..21
A[IV-11] = B[IV]
```

Here we deviate from APL, since the origin of the subscripts need not be zero or one. Also we allow array declarations in order to obtain efficient computation with scalar array elements.

The rank of a vector is defined as in APL to be the number of (vector) subscripts. The rank could be obtained from a RANK function within a program. For example,

```

REAL ARRAY A[20,20,10]
INC_VECTOR IV,JV
JV = 1..5
IV = 1..10
N = RANK(A[IV,1,JV])

```

For a declared array, whose rank is fixed, we might allow $RANK(A)$, which would yield $RANK(A) = 3$. In this case N would have the value 2, since there are two vector subscripts. The "dimension" of a vector V is another vector with rank one whose components are integers equal to the number of components in each vector subscript of V . In the above example, the dimension of the vector $A[IV,1,JV]$ is $//10,5//$. The dimension could be obtained by a function call $DIMEN(A[IV,1,JV])$. This is essentially the same as in APL. Perhaps the dimension should be $//10,1,5//$ in this case, but we prefer to ignore scalar subscripts.

Storage can be allocated to dynamic vectors, that is vectors whose dimensions, and hence the storage space required, vary during the computation. This may not be necessary for most PDE codes. These vectors are declared by means of a block declaration. Perhaps this block name could then be put in COMMON. For example,

```

VECTOR_BLOCK VBUF[2000]
  REAL VECTOR VA,VB,VC
  INTEGER VECTOR VN,VM
END_VECTOR_BLOCK

```

This construct could be inefficient, particularly for short vectors and certainly if these vectors are used in scalar expressions, that is with a subscript containing a "FOR" index. Perhaps arrays with dynamic storage allocation for subroutines or procedures is better, particularly if recursive calls are not allowed, and the dynamic allocation can be overridden to create a static allocation. The

dimension of a dynamic vector is set when the vector appears on the left of a replacement statement (which could occur within a subroutine call). For example,

```
VA = //1.,2.,3.//
```

A vector of type LOGICAL can be used as a subscript, in which case the vector is regarded as a bit vector in the sense used on the Illiac IV. For example,

```
LOGICAL ARRAY IV[5]
REAL ARRAY A[5] , B[5]
IV = //1,0,1,1,1//
A[1..5] = //1.,2.,3.,4.,5.//
```

In the above 1 is assumed equivalent to true and 0 to false. Then the vector A[IV] is equivalent to //1,3,4,5// , that is the subscript values corresponding to zero in IV are dropped. This gives a vector similar to the "monotone" vector in TRANQUIL. If a vector is defined by means of a general vector subscript, there may be no way to efficiently access the components on a vector computer. The components may all be in the same memory bank, or the location of the components may be too random for the vector access mechanism on the parallel computer. The use of a logical vector as a subscript should permit efficient computation on a vector machine, provided the percentage of zeros in the logical vector is not too high. This depends on the flexibility of the vector access on the computer.

3. Basic vector operations. The usual arithmetic operations are performed component-wise on vectors. This requires the vectors to have the same dimension, that is they must be conformable in the sense of APL. Comparisons may be made between arithmetic vectors to form logical vectors of the same dimension. For example,

```

REAL ARRAY A[100] , B[100], C[100,20]
INC VECTOR IV
LOGICAL ARRAY LV 100
IV = 1..100
LV = A[IV] < 0.
B[LV] = 1.
B[¬LV] = 0.

```

Here \neg represents the logical operator "not". For simplicity in the lexical scanner, and for ease of typing, blanks should probably be delimiters and reserved words should be used. In this case the operators \neg and $<$ might be represented by NOT and LT. Note that the vectors on the left and right side of an assignment statement must have the same dimension. Again, there can be difficulty with the use of general vectors as subscripts. For example the following should be prohibited.

$$A[//1,1//] = //1.,2.//$$

since the storage assignment is not well defined. Perhaps subscript vectors should be restricted to increment vectors and logical vectors.

A vector can be created from an array by using "*" as a subscript. In the above code segment $C[*,1]$ would be a vector of dimension 100.

4. Indexed vector expressions. This is a construct which is not included in APL. We include it because it seems particularly well suited to mesh calculations for the solution of PDE problems. The following are some vector calculations which seem to us to be awkward using the APL notation for vectors.

$$\begin{aligned}
 a_{ij} &= b_{ij} * C_j & \text{for } 1 \leq i, j \leq n \\
 a_i &= b_{ii} * C_i & \text{for } 1 \leq i \leq n
 \end{aligned}$$

In APL the following operations

```
I = ?N
J = ?N
A[I;J] ← B[I;J] - B[J;I]
```

yield

```
A[I;J] ≡ 0
```

which is not what the notation suggests.

We allow arrays and vectors to use subscripts which have been flagged as "index" vectors. Operations involving such indexed vectors are performed componentwise by matching the indices; that is, the operations are performed pairwise on the components with the same index subscript. Only logical or increment vectors may be declared index vectors. Arrays or vectors with such vectors used as subscripts are treated differently from the usual vectors. For example,

```
INC VECTOR IV,JV
SET INDEX IV,JV
REAL ARRAY A[100], B[100,100]
IV = 1..100
JV = 1..100
B[IV,JV] = B[IV,JV]* A[IV]
```

Note that if IV and JV are not indices, but only vectors, the operation

```
B[IV,JV]* A[IV]
```

is not allowed since the vector operands do not have the same dimension.

When index vectors are used the expression

```
.5*( B[IV,JV] - B[IV,JV] )
```

yields the skew symmetric part of B.

An indexed expression has an associated index list. The index list associated with an array consists of all the index vectors which appear as subscripts of the array. The index list associated with a simple binary arithmetic operation consists of the union of

the index lists of the two operands. For example, the index list of the expressions

$$A[IV], \quad B[IV,JV]*A[IV], \quad B[IV,1]*A[JV]$$

are the following

$$IV, \quad (IV,JV), \quad (IV,JV).$$

Note that the last operation is really a direct product. If one subscript of an array or vector is an index, then we require that all vector subscripts be declared index vectors. If one operand of an expression is an indexed expression and the other is a vector in which no indices are used, then the operation could be carried out by regarding the first operand as a vector. Then the dimensions of the two vector operands must be the same. This provides a sort of mixed mode which permits indexed vectors to be converted into vectors.

In a replacement statement each index of the expression on the right must appear on the left, however, those on the left need not appear on the right. For example, an indexed array may be equated to a scalar, however a scalar variable can not be equated to an indexed vector. A notation is available which allows the entire range of an array subscript to be identified as an index. Note that $B[* , 1]$ is a vector, whereas $B[* 1, 1]$ is an indexed vector since the vector subscript is identified by a number and thus indexed. For example,

$$B[* 1, * 2] = B[* 2, * 1] * A[* 2]$$

Vectors can be removed from the set of index vectors by the command

`DROP_INDEX IV,JV`

As an example of the use of index vectors in a finite difference type of computation, consider the following


```

INC_VECTOR I,J
SET_INDEX I,J
REAL ARRAY U,V,W[64,-16..16,2], C[-16..16]
I = 2..63
J = -15..15
N1 = 1
N2 = 2
U[I,J,N2] = U[I,J,N1] + DT*C[J] * (U[I+1,J,N1] - U[I-1,J,N1])/DX

```

It will be easy to forget to include vectors in a SET_INDEX statement. The intended indexed operations using these vectors might be syntactically correct as vector operators. It might be desirable to build some redundancy into the language at this point by using special symbols for the indexed arithmetic operations. For example, +!,-!,*!,/!,**!. However, the use of special characters is troublesome because character sets differ widely from one system to the next, and they have an awkward appearance.

5. The INDEX_IF conditional statement. This statement allows a computation to be restricted to that portion of a mesh for which a Boolean expression is true. The mesh points are identified by index vectors. For example,

```

INC_VECTOR I,J
SET_INDEX I,J
REAL VECTOR U,V,W[100,100],X,Y[100]
INDEX_IF X[I]*Y[J] ≠ 0. THEN
    W[I,J] = 1./X[I] + 1./Y[J]
ELSE
    W[I,J] = EXP( V[I,J])
END_INDEX_IF

```

In this case the first replacement statement is executed only for those values of I and J for which $X[I]*Y[J] \neq 0$. The second replacement statement is executed for the complementary set of values of I and J. The index list of each expression within the scope of an INDEX_IF is extended to include the index list of the Boolean expression of the INDEX_IF. Thus the expression $1./X[I]$ in this example is

evaluated over a different set of values of I depending on the value of J. If an index from the INDEX_IF Boolean appears on the left side of a replacement statement, then so must all indices which appear in the Boolean. We say that the members of index list associated with the Boolean expression of an INDEX_IF are "linked". In the above example I and J are linked within the scope of the INDEX_IF.

Expressions within an INDEX_IF must be extended to contain all the linked indices. This conditional statement should compile reasonably efficiently on a well designed parallel computer.

6. Additional vector operators. APL includes many operators which can be applied to vectors. Perhaps a mesh language does not need quite so many. Also it may be better to include these operators in the form of internal functions, since the names may be easier to read and recall, particularly with a restricted character set. Therefore we do not use special symbols as does APL.

The compress and expand operators of APL are included as vector valued functions with two arguments. The first argument of compress¹ is a logical vector and the second is an arithmetic vector. These vectors must be of rank one. The value of the compress function is another vector of the same arithmetic type as the second argument whose dimension equals the number of ones in the logical vector. For example,

```

LOGICAL ARRAY L[100]
REAL ARRAY U[100], A[100,100]
VECTOR_BLOCK VBUF[1000]
    REAL VECTOR V,W
END_VECTOR_BLOCK
INC_VECTOR I,J
SET_INDEX I,J
I = 1..100
L[I] = U[I] < 0.
V = COMPRESS(L,U)

```

In this example the vector V consists of the negative components of U . It is awkward to have the result of the compression be an indexed vector since the index as well as the vector must be set. If we allow

$$V[J] = \text{COMPRESS}(L[I], U[I])$$

then the notation would not be consistent with that used previously since both V and J would be altered. The EXPAND function is defined in a similar manner. The result vector has the same dimension as the logical vector. For example, if the following statement is added to the above program segment

$$W = \text{EXPAND}(L[I], V)$$

then W is obtained from the vector $U[I]$ by zeroing out the positive components of U .

A reduction or inner product operator is included. In this case we do use a special symbol, namely $*\downarrow$. If both operands are vectors, then the last vector subscript of the first operator is reduced by taking the inner product with the first vector subscript of the second operand. This corresponds to matrix multiplication, and is the convention used in APL. If both operands are indexed expressions, then the reduction is carried out over all repeated indices. If the reduction is to be restricted to specified indices, then these indices are specified within brackets. For example, consider

$$\begin{aligned} &A[I, J] * \downarrow U[J] \\ &A[J, I] * \downarrow U[J] \\ &A[J, I] * \downarrow [J] U[J] \\ &A[*1, *2] * \downarrow U[*1] \\ &A[I, J] * \downarrow [J] A[I, J] \\ &V * \downarrow V \end{aligned}$$

If we assume that I , J , and V are vectors, then the first expression multiplies the matrix A with the vector U , the second multiplies the

matrix A with the vector U, the second multiplies the transpose of the matrix A by the vector U, the third and fourth are the same as the second, the fifth computes the inner product of the rows of A with themselves, and the last computes the inner product of the vector V with itself. If the reduction is to be carried out along specific subscripts of vector operands (not indexed vectors), the subscripts can be indicated by their sequence number in the subscript list. For example,

$$A[*,*] * \downarrow [2:2] A[*,*]$$

This yields the same computation as the fifth example given in the group above.

Other internal functions or operators might be desirable. The RANK and DIMEN functions have already been discussed. A function to find the maximum and another to find the minimum of the components of a vector should be included (ceiling and floor in APL). These might be VMAX and VMIN. The usual FORTRAN functions such as SIN, EXP, etc. should be extended to vectors without change of name.

7. Vector valued functions. The user needs the ability to define functions whose value is a vector which can be used in vector expressions. Subroutines may also communicate arguments which are vectors. A FORTRAN-like syntax could be used for this. A vector block could be placed in COMMON to provide another manner to communicate vectors between separate subprograms. In addition to separately compiled subroutines procedures in the style of ALGOL could be included in the language. This would provide global variables. If a subprogram argument is an indexed vector expression, then it must be converted to

a vector for transmission to the subprogram. The following is an example of a function

```

REAL VECTOR FUNCTION FN(X)
REAL VECTOR X
COMMON VBUF,NMAX,EPS
VECTOR BLOCK VBUF[1000]
      REAL VECTOR TA,TB
END VECTOR_BLOCK
REAL EPS
INTEGER NMAX,N
FN=1.+X
TA=X
N=1
WHILE (VMAX(ABS(TA)) GT EPS) AND (N LT NMAX) DO
      N=N+1
      TA=TA*X/N
      FN=FN+TA
END WHILE
END FUNCTION

```

In this example the storage for the vector result FN would have to be declared in the calling program. It might be necessary to require that FN appear in a VECTOR_BLOCK declaration in the calling program.

8. Some comments on the design of a vector language. Perhaps most users would be adequately served by vectors formed from arrays using increment vectors. Thus the vector declaration could be eliminated which would certainly simplify the compiler. However, we do not see a reasonable way to transmit vector arguments to subprograms, return a vector valued function, or to define compress and expand operations if this restriction is made. Therefore we feel it is desirable to include variables of vector type.

The structures and types allowed in the language should be considered more carefully. Should a more general type of structure be allowed as in ALGOL 68, or a record structure as in PASCAL? Should we allow the value of a function to be any structure permitted in the

language? What seems natural to the average scientific programmer should be considered here as should the cost of implementation and the effect on run-time efficiency.

The control structures used in the language should be re-designed taking into account the ideas of Knuth, Wirth and others. This is a very active area in Computer Science at the moment; perhaps the design of a good set of control commands will be easier in the future.

We have not included any I/O features which is a bad omission. We especially need to include format-free I/O, good graphics commands, the ability to handle a memory hierarchy, etc.

The design of a vector language raises optimization questions which may be different from those which arise with a scalar language. The parse tree now contains vector operands and this should allow better optimization.

REFERENCES

- [1] N. Wirth (1972), "The Programming Language PASCAL", Eidgenössische Technische Hochschule Zurich.
- [2] IVTRAN (1973), "The IVTRAN Manual", Massachusetts Computer Associates, Wakefield, Massachusetts, 01880.
- [3] D. Lawrie (1973), "Memory-Processor Connection Networks", PH.D. thesis, University of Illinois, Comp. Sci. Rep. 557.
- [4] P. Budnik and D. Kuck (1969), "A TRANQUIL Programming Primer", Department of Computer Science, Rep. 816, University of Illinois.
- [5] H. Katzan (1970), "APL Programming and Computer Techniques", Van Nostrand, New York.
- [6] Löcs and Gary (1974), "A FORTRAN Extension for Data Display", IEEE Trans. on Comp., 1257-1263.
- [7] M. Wilson, "Flexible Subarray Facilities for Classical Programming Languages", IBM Houston Scientific Center Technical Report No. 320-2426, IBM Corp., 6900 Fannin Street, Houston, Texas, 77025.
- [8] A. Haberman (1973), "Critical Comments on the Programming Language PASCAL", Acta Informatica, 3, 47-57.

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

$$U_2 = U_1 + DLT * DXX(U_1)$$

where DXX represents the operator

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

and $DLT = \Delta t$. 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 compatible 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 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 readability 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 scalar 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 scalar 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 object 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

$$\frac{\delta u}{\delta t} = \frac{\delta^2 u}{\delta x^2} \quad u = u(x, t) \quad \begin{array}{l} 0 \leq x \leq 1 \\ 0 \leq t \end{array}$$

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

The problem is made discrete by use of a mesh in x and t , $x_j = j\Delta x$, $0 \leq j \leq J$, $\Delta x = 1/J$. 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

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

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

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 $U_1 + U_2$, then

$$DX(U_1 + U_2)(I) = ((U_1(I+1) + U_2(I+1)) - (U_1(I) + U_2(I)))/DLX$$

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

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

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

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, $2 \leq I < 127$. 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(SC*D'(W))
```

```
ENDSETJACOB
```

The unknown vector is $\{W_i\}$ with components in the range $2 \leq i \leq 63$.

The expression defines a linear system of equations for W_i of the following form

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

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

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

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 δW

$$A\delta W = -F(U2)$$

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 δW of the Jacobian system,

$$U2 = U2 + \delta W$$

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 too 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

x	x	x	o		
x	x	x	o		
o	x	x	x		0
	.				
	.				
	.				
	0			x	x
				x	x
				o	x
				x	x

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))*0.5
    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(*)
END/ARRAYBLOCK
```

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, Englewood 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)

Boundary Conditions for the Method of Lines
Applied to Hyperbolic Systems*

by

John Gary
Department of Computer Science
University of Colorado
Boulder, Colorado 80302

TR #CU-CS-073-75

July 1975

* This research was supported by an ARPA grant AFOS-74-2732.

1. Introduction. A fundamental problem in the numerical solution of hyperbolic equations is the proper approximation of the boundary conditions. For example, the leapfrog scheme applied to the following hyperbolic problem is unstable.

$$\begin{aligned} u_t + u_x &= 0 & 0 \leq x \leq \pi \\ u(0, t) &= \sin(-t) & 0 \leq t \\ u(x, 0) &= \sin(x) \end{aligned} \quad (1)$$

The mesh for this scheme is $x_j = j\Delta x = j\pi/J$ for $0 \leq j \leq J$. The exact solution is $\sin(x-t)$. If a second order difference approximation for the spatial derivative is combined with a leapfrog scheme for time, then the following scheme is obtained

$$\begin{aligned} u_0^n &= \sin(-t_n) \\ u_j^{n+1} &= u_j^{n-1} - 2\Delta t(u_j^n - u_{j-1}^n)/\Delta x \\ u_j^{n+1} &= u_j^{n-1} - \Delta t(u_{j+1}^n - u_{j-1}^n)/\Delta x \quad 1 \leq j \leq J \end{aligned}$$

This scheme is unstable. If the outflow boundary is modified as indicated below, then the scheme is stable and has second order accuracy.

$$u_j^{n+1} = u_j^n - \Delta t(u_j^n - u_{j-1}^n)/\Delta x$$

When the method of lines is used for the simple linear hyperbolic equation (1) with periodic boundary equations, then the resulting difference scheme is stable, provided an ODE solver with automatic step-size adjustment is used to solve the system of ordinary differential equations. Even if the ODE solver uses an Euler forward time-step scheme, the integration will converge as the mesh spacing is taken to zero, since the ODE solver will take the step time-step small enough as a function of the mesh size to guarantee convergence. It will not be the case that

$\Delta t = 0(\Delta x)$. Note that the semi-discrete approximation produced by the method of lines with periodic boundary conditions can be written in the form

$$\underline{u}' = A \underline{u} \quad \underline{u} = (u_{-J}, \dots, u_{J-1}) \quad x_j = j\pi/J$$

where the matrix A is

$$A = -\frac{1}{2\Delta x} \begin{vmatrix} 0 & 1 & 0 & \dots & -1 \\ -1 & 0 & 1 & 0 & \dots \\ 0 & -1 & 0 & 1 & \dots \\ \vdots & & & & \\ 1 & & & 0 & -1 & 0 \end{vmatrix}$$

The solution of this differential equation is given in terms of an exponential matrix as

$$\underline{u}(t) = \underline{u}(0)e^{At}$$

Since the matrix is skew symmetric and cyclic its eigenvalues are pure imaginary and its eigenvectors are orthogonal. Therefore, the solution is bounded independently of the number of mesh points. This implies that the solution of this semi-discrete approximation will converge to the solution of the original equation (1). Therefore any spatial discretization which yields a skew symmetric, cyclic matrix will define a convergent method of lines approximation. Stability in a finite difference scheme for hyperbolic problems is in a sense associated with the temporal discretization.

Unfortunately, the method of lines does not necessarily produce a stable scheme when the boundary conditions are not periodic. However, the method of lines does seem to be more likely to yield a stable scheme

than a leapfrog time discretization.

Our purpose is an experimental study of some boundary difference approximations for use on hyperbolic systems where the method of lines is used for the temporal discretization. Our results will refer mainly to the Runge-Kutta-Fehlberg ODE solver, although we intend to experiment with the Adams method of Shampine [9] in the future. We have found it important to include test cases for hyperbolic systems (more than one independent variable) for which the characteristics lie on both sides of the boundary. This is in agreement with comments by Chu [3] and Sundstrom [10]. We are interested in boundary approximations which can be incorporated into a general PDE solver to treat hyperbolic systems in two dimensions. Such solvers for parabolic equations in one dimension are described by Sincovec and Madsen [11], Carver [2], Loeb [6], Bowen [1], Hastings [12] and others. Because of our interest in general hyperbolic systems, we cannot consider boundary approximations stated in terms of specific variables for specific equations. We can only consider algorithms which can be presented in a general framework. We will test two such algorithms.

Of course, such a general algorithm requires the user to apply it in such a way as to produce a properly posed hyperbolic problem. We must allow the user the flexibility to set the boundary conditions. Eventually, we might be able to supply an optional check to see that the boundary conditions are consistent with the hyperbolic system.

2. Computational results indicating stability and accuracy of the method of lines. In this section we consider difference approximations for the system (1). These are semi-discrete approximations of the form

$$\underline{u}' = A\underline{u} + \underline{f} \quad (2)$$

where $\underline{u} = \underline{u}(t) = (\dots u_j(t) \dots)^T$ is a vector of mesh point values. In this section we will look at the eigenvalues of A and the norm of the exponential matrix

$$\|e^A\| \quad (3)$$

for four finite difference approximations. If this norm is bounded independent of the spatial mesh, then the semi-discrete approximation is stable. This follows from the integral form of the solution of (2)

$$u(t) = u(0)e^{At} + \int_0^t f(\tau)e^{A(t-\tau)} d\tau \quad (4)$$

If the eigenvectors of A are orthogonal, then a bound for the norm of the exponential matrix can be obtained from the eigenvalues of A . Therefore, we compute these eigenvalues and also the norm (3) in order to gain insight into the stability of the following four schemes.

A. An inconsistent scheme. Here a one-sided difference is used at both boundaries in spite of the fact that the solution should be specified at the inflow or left boundary. This must yield an unstable approximation. The approximation is consistent, and if it were also stable, then it would be convergent. That is, if the norm of the exponential matrix

$$e^{At}$$

were bounded independently of the mesh spacing the approximation would be convergent, which is impossible since no boundary condition has been specified on the inflow boundary. The scheme is

$$\begin{aligned} u_0'(t) &= -\frac{1}{\Delta x}(u_1(t) - u_0(t)) \\ u_j'(t) &= -\frac{1}{2\Delta x}(u_{j+1}(t) - u_{j-1}(t)) \quad 1 \leq j < J \\ u_J'(t) &= -\frac{1}{\Delta x}(u_J(t) - u_{J-1}(t)) \end{aligned} \quad (5)$$

B. A second order scheme. This scheme is the same as the previous one except

$$u_0(t) = \sin(-t)$$

It is only first order at the boundary, but the overall accuracy should be second order.

C. Fourth order with a third order boundary. This scheme is given below. Oliger [7] has shown that subtle changes are required in this spatial approximation when it is used with a leapfrog time discretization, in order that the resultant difference scheme be stable. However, it seems to be stable without modification when it is used with a variable step ODE solver.

$$\begin{aligned} u_0(t) &= \sin(-t) \\ u_1'(t) &= -(2u_0 - 3u_1 + 6u_3 - u_4)/6\Delta x \\ u_j'(t) &= -(2u_{j-2} - 16u_{j-1} + 16u_{j+1} - 2u_{j+2})/(24\Delta x) \\ u_{J-1}'(t) &= -(u_{J-3} - 6u_{J-2} + 3u_{J-1} + 2u_J)/(6\Delta x) \\ u_J'(t) &= -(-2u_{J-3} + 9u_{J-2} - 18u_{J-1} + 11u_J)/(6\Delta x) \end{aligned} \quad (6)$$

D. A fourth order scheme with a fourth order boundary approximation.

This is the same scheme as the one above except that one-sided fourth order differences are used at the boundary.

$$u_1'(t) = -(-6u_0 - 20u_1 + 36u_2 - 12u_3 + 2u_4)/(24\Delta x)$$

$$u_{J-1}'(t) = -(-2u_{J-4} + 12u_{J-3} - 36u_{J-2} + 20u_{J-1} + 6u_J)/(24\Delta x) \quad (7)$$

$$u_J'(t) = -(6u_{J-4} - 32u_{J-3} + 72u_{J-2} - 96u_{J-1} + 50u_J)/(24\Delta x)$$

The above four schemes can all be written in the matrix form of equation (2). The maximum of the real parts of the eigenvalues of the matrix A for these four schemes are given in Table I. The eigenvalues of A were determined by using the IMSL QR routine on the CDC 6400 at the University of Colorado. The exponential matrix was determined by summing its series expansion. The norm is that induced by the vector maximum norm. For the inconsistent scheme (A) the eigenvalues are all pure imaginary with a double or triple root at zero depending on whether J is even or odd. The instability of this scheme is evident from the norm of the exponential matrix but not from the eigenvalues.

Schemes (B) and (C) would appear to be stable from this analysis. However, we might expect the solution of scheme (D) to show exponential growth in time since its matrix has an eigenvalue with positive real part.

In order to provide a more complete test of these four schemes we wrote a code for these schemes applied to equation (1). This provides a direct test of the stability and accuracy of these schemes.

Table II shows the error obtained with the various schemes after integration to the indicated value of $t=T$ using the mesh resolution

determined by J . Note that the number of intervals per wave is $2(J-1)$ since the mesh runs from $x=0$ to $x=\pi$ and $J+1$ is the number of mesh points. Scheme (A) is clearly unstable. Schemes (B) and (C) seem to be stable which is consistent with the results in Table I giving the characteristics of the matrices corresponding to these schemes. Scheme (D) seems to be weakly unstable when the system is solved with the RKF ODE solver. However this scheme seems to be stable when the Runge-Kutta scheme with a fixed ratio $\Delta t/\Delta x$ is used.

3. A variable coefficient problem. A hyperbolic problem which is more typical of many applications than equation (1) is the following defined on the interval $0 \leq x \leq \pi$.

$$u_t + \cos(t)u_x = \cos(x-t)(\cos(t)-1) = r(x,t)$$

$$\text{If } \cos(t) \geq 0 \text{ then } u(0,t) = \sin(-t) \quad (8)$$

$$\text{If } \cos(t) \leq 0 \text{ then } u(\pi,t) = \sin(\pi-t)$$

$$u(x,0) = \sin x$$

The solution of this problem is $u(x,t) = \sin(x-t)$. The mesh is $x_j = j\pi/J$, for $0 \leq j \leq J$. In this problem the inflow and outflow boundary alternate between the two endpoints of the interval. When $\cos(t) \geq 0$ the left boundary is the inflow point. This makes the use of an ODE solver awkward if the method of equation (6) is used to define the system of differential equations. When $\cos(t) \geq 0$ the unknowns are $(u_1(t), \dots, u_J(t))$ and when $\cos(t) \leq 0$ the unknown vector has shifted to $(u_0(t), \dots, u_{J-1}(t))$.

Therefore we differentiate the boundary condition so that the system of differential equations always contains the same unknowns.

E. A second order scheme for equation (8).

If $\cos(t) \geq 0$ then

$$u'_0(t) = \frac{d}{dt}(\sin(-t)) = -\cos(t)$$

otherwise

$$u'_0(t) = -(u_1 - u_0)/\Delta x + r(0, t) \quad (9)$$

If $\cos(t) \leq 0$ then

$$u'_j(t) = \frac{d}{dt}(\sin(\pi - t)) = \cos(t)$$

otherwise

$$u'_j(t) = -(u_j - u_{j-1})/\Delta x + r(\pi, t)$$

This scheme uses a differentiated form of the boundary condition at an inflow boundary and a one-sided first order difference approximation to the differential equation at an outflow boundary. The definition of the differential equation used to define the solution along the boundary line varies depending on the inflow-outflow nature of the boundary. However, the solution along these boundary lines is always determined by a differential equation.

F. A fourth order scheme for equation (8).

If $\cos(t) \geq 0$ then

$$u'_0(t) = -\cos(t)$$

otherwise

$$u'_0(t) = -\delta'_3(\underline{u})_0 + r(0, t)$$

Here δ'_3 is the third order difference approximation of $u_x(0)$ using (x_0, x_1, x_2, x_3) . The equation for $u'_j(t)$ is similar. The remainder of the system is identical with that of equation (6).

The results of using these schemes to approximate the solution of equation (8) is given in Table III. These results indicate that these schemes are stable. The norm of the second order approximation shows a slow linear growth with time. The error shows the expected asymptotic behavior with J (approximately). The behavior of this method on a more complex multidimensional problem awaits testing which we hope to carry out in the near future.

4. General boundary approximation algorithms. In this section we consider a program for the following, more general class of nonlinear hyperbolic equations.

$$\frac{\partial \underline{u}}{\partial t} = \frac{\partial}{\partial x}(\underline{g}(\underline{u}, x, t)) + \underline{h}(\underline{u}, x, t) \quad (11)$$

or the nonconservation form

$$\frac{\partial \underline{u}}{\partial t} = f\left(\frac{\partial \underline{u}}{\partial x}, \underline{u}, x, t\right) + \underline{h}(\underline{u}, x, t) \quad (11)$$

Here \underline{f} , \underline{g} , and \underline{h} are general vector valued functions and $\underline{u}(x, t)$ is the vector solution. We assume that boundary conditions are given at two end points $x=a$ and $x=b$. We consider two methods to specify these boundary conditions.

The first method requires the specification of a subset of the unknowns at each boundary point. Consider the left boundary $x=a$. The unknowns are $(u_1(x, t), \dots, u_M(x, t))$. The p unknowns $(u_{m_1}, \dots, u_{m_p})$ from the set $I = \{p_1, \dots, p_m\}$ are specified as follows:

$$\begin{aligned} u_{m_1}(a,t) &= S_1(u_{II}(a,t),t) \\ &\vdots \\ u_{m_p}(a,t) &= S_p(u_{II}(a,t),t) \end{aligned} \quad (13)$$

Here $u_{II} = (u_{m_1}, \dots, u_{m_{M-p}})$ is the complement of $u_I = (u_{m_1}, \dots, u_{m_p})$. The problem specification must include the integer p and the functions S_1, \dots, S_p at both boundary points. Note that p may depend on the time t . The functions S_i are used to set the values of u_I at the boundary. The values of u_{II} are computed from the hyperbolic equation, using one sided approximations for spatial derivatives.

For example, consider the variable coefficient problem given by equation (8). At the left boundary ($x=0$), if $\cos(t) \geq 0$, then for the number of boundary constraints we have $p=1$. The function $S_1(u_{II}, t) = S_1(t) = -\sin(t)$. Note that u_{II} is empty in this case. If $\cos(t) < 0$, then $p=0$ at the left boundary and u_I is empty. In this case the value of $U_0(t)$ (here $U_j(t)$ denotes the approximation to $u(x_j, t)$ on the "time line") is obtained from the differential equation

$$\frac{dU_0}{dt} = -\cos(t)\delta_3(U)_0 + r(x_0, t) \quad (14)$$

where δ_3 represents the onesided difference approximation.

When the ODE solver, such as the Runge-Kutta-Fehlberg is used, there is a slight problem in implementing this algorithm. When the characteristic slope $\cos(t)$ changes sign, the nature of the system of ordinary differential equations changes. When $\cos(t) > 0$, the unknowns in the system (8) are (U_1, \dots, U_j) , but when $\cos(t) < 0$ the unknowns are (U_0, \dots, U_{j-1}) .

The ODE solver always works with the full set of unknowns including the boundary values, that is (U_0, \dots, U_J) . However, in computing the "right side" functions in the Runge-Kutta steps the boundary constraints are applied to set the boundary values for variables in the U_I sets. If the system of ODE's is written

$$\frac{dU_j}{dt} = F_j(U_0, \dots, U_J, t) \quad (15)$$

and U_0 is in the constrained set U_I for $t = t_n + 1/2\Delta t$, then the function $F_j(U_0, \dots, U_J, t_n + 1/2\Delta t)$ used in the Runge-Kutta step is replaced by $F_j(S(U_{II}, t_n + 1/2\Delta t), U_I, \dots, U_J, t_n + 1/2\Delta t)$. Also, at the end of the step the value of U_0 computed by the ODE solver is replaced by $S(U_0, t)$ provided U_0 is still in the constrained set U_I . Obviously this requires modification of the ODE solver. There is no guarantee that this method will converge. In fact, as we will see shortly, it does not always converge. The algorithm can be implemented as part of a PDE package once the user has supplied the subroutines to evaluate p , the sets U_I , and the functions $S_i(U_{II}, t)$.

The second method is a generalization of the differentiated boundary conditions described in section 3 in equations (9) and (10). In this case the user is allowed to reset the time derivatives used by the ODE solver to compute the boundary values, that is

$$\frac{dU_{m,0}}{dt} = \hat{F}_{m,0}(U_0, F_0, t) \quad \text{and} \quad \frac{dU_{m,J}}{dt} = F_{m,J} = \hat{F}_{m,J}(U_J, F_J, t) \quad (16)$$

Here we assume a system of equations for the unknowns $u_{m,j}$ where there are M unknowns ($1 \leq m \leq M$) and the mesh points are $X_j(a=x_0 < x_1 < \dots < x_J=b)$.

The vectors $\underline{F}_0 = (F_{1,0}, \dots, F_{M,0})^T$ and \underline{F}_j are the time derivatives obtained using one sided difference approximations in the hyperbolic system at the boundary. The vectors $\underline{U}_0, \underline{F}_0$ and the time t are supplied to a user written subroutine which must then determine the set I and return values for $\frac{dU_{p,0}}{dt} = \hat{F}_{p,0}$, for $p \in I$. The remaining time derivatives for $p \notin I$ are the values $F_{p,0}$ obtained from onesided differences in the hyperbolic system. This method is going to be difficult to explain to a user. However, it is the only method that has, so far, worked reliably.

We will illustrate this method by the following example. This is a system with characteristics of different sign. Chu [3] and Sundstrom [10] have noted the difficulties of setting boundary conditions for such systems.

$$\begin{aligned} \frac{\partial u_1}{\partial t} &= \frac{3\partial u_1}{\partial x} - \frac{4\partial u_2}{\partial x} & 0 \leq t \\ \frac{\partial u_2}{\partial t} &= \frac{2\partial u_1}{\partial x} - \frac{3\partial u_2}{\partial x} & 0 \leq x \leq b \end{aligned} \quad (17)$$

This system is derived from

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{\partial u}{\partial x} & u &= u_1 - u_2 & u_1 &= 2u + v \\ \frac{\partial v}{\partial t} &= -\frac{\partial v}{\partial x} & v &= 2u_2 - u_1 & u_2 &= u + v \end{aligned}$$

Therefore the following boundary conditions are proper, since they amount to a specification of the characteristic variable on the inflow boundary.

$$\begin{aligned} \text{at } x=0 & \quad v = 2u_2(0,t) - u_1(0,t) = -\sin 2\pi t \\ \text{at } x=b & \quad u = u_1(b,t) - u_2(b,t) = \sin 2\pi(b+t) \end{aligned} \quad (18)$$

We have chosen the boundary conditions to correspond to the following solution

$$\begin{aligned} u_1(x,t) &= 2\sin 2\pi(x,t) + \sin 2\pi(x-t) \\ u_2(x,t) &= \sin 2\pi(x+t) + \sin 2\pi(x-t) \end{aligned} \quad (19)$$

To use the first method of setting the boundary conditions we must specify the set U_I at each boundary point. There is no unique choice here, since neither u_1 nor u_2 are characteristic variables. We will try to specify u_1 at each boundary from the given boundary conditions, namely

$$\begin{aligned} \text{at } x=0 \quad u_1(0,t) &= 2u_2(0,t) + \sin 2\pi t \\ \text{at } x=b \quad u_1(b,t) &= u_2(b,t) + \sin 2\pi(b+t) \end{aligned} \quad (20)$$

In this case $I = \{1\}$, $U_I = \{u_1\}$, $II = \{2\}$, $U_{II} = \{u_2\}$, and $p = 1$ at both boundary points.

A derivative rather than a constrained boundary condition can be obtained by differentiation of the above equation, namely

$$\begin{aligned} \frac{du_{1,0}}{dt} &= 2\frac{du_{2,0}}{dt} + 2\pi\cos 2\pi t \quad \text{at } x=0 \\ \frac{du_{1,0}}{dt} &= \frac{du_{2,0}}{dt} + 2\pi\cos 2\pi(b+t) \quad \text{at } x=b \end{aligned} \quad (21)$$

The derivative du_2/dt on the right can be computed from the hyperbolic system using one sided differences and then used in the user supplied routine to compute du_1/dt by equation (21) above.

As our results show neither of these methods given by equations (20) and (21) work satisfactorily. They both specify the inflow characteristic variable. The outflow characteristic should be computed using one sided differences. In equation (20) the inflow characteristic is specified by the boundary constraint. However, there is certainly error in the computed value of u_2 used on the right side of the boundary

constraint. This error can be transmitted to the other boundary and reflected back. The boundary condition probably should not allow much error in the incoming characteristic.

We tried a third type of boundary condition obtained by differentiating the boundary constraint and combining it with the equation for the outgoing characteristic variable obtained from the hyperbolic system.

That is, at $x = 0$

$$-\frac{du_{1,0}}{dt} + 2\frac{du_{2,0}}{dt} = -2\pi\cos 2\pi t$$

$$\frac{du_{1,0}}{dt} - \frac{du_{2,0}}{dt} = F_{1,0} - F_{2,0}$$

Here $F_{1,0}$ is an approximation to

$$\frac{3\partial u_1}{\partial x} - \frac{4\partial u_2}{\partial x}$$

and $F_{2,0}$

$$\frac{2\partial u_1}{\partial x} - \frac{3\partial u_2}{\partial x}$$

obtained using one sided differences. These equations yield

$$\frac{du_{1,0}}{dt} = 2F_{1,0} - 2F_{2,0} - 2\pi\cos 2\pi t$$

$$\frac{du_{2,0}}{dt} = F_{1,0} - F_{2,0} - 2\pi\cos 2\pi t$$
(22)

There are errors in computing $F_{1,0}$ and $F_{2,0}$, but these will cancel out in the computation of the time derivative of the inflow characteristic ($v=2u_2-u_1$) when this boundary condition is used. Perhaps this is the reason for the superior performance of condition (22) over (20) and (21). However, we do not have a solid theoretical understanding of these results.

5. Some computational results. These results all refer to the solution of equation (17) using a fourth order centered finite difference approximation in the interior and third order one sided differences near the boundary to approximate the spatial derivation $\partial/\partial x$. The Runge-Kutta-Fehlberg [5] method was modified to allow use of the "constrained" boundary condition (20). The "derivative-constrained" condition (21) and the "derivative-characteristic" condition (22) were also used. The parameter ϵ refers to the error tolerance used in the Runge-Kutta-Fehlberg. The variable J is the number of mesh points, and $x=b$ is the right boundary. The results depend on b, probably because of the way the error is reflected between the two boundaries. The error is the relative error in the computed solution at the indicated time $t=T$. The parameter N_E is the number of evaluations of the time derivative required in the integration. Each time step requires 6 evaluations (5 if it follows an unsuccessful step).

There seems to be little difference between the results for the constrained-boundary (20) and the derivative-constrained method (21), except for a slight difference in the number of functional evaluations. This difference can be largely eliminated by omitting the error estimate for the constrained boundary variables - at least this was our experience for the single equation (1). Only the characteristic derivative method (22) is free from the error growth which is probably due to multiple reflections from the boundaries. Note that the severity of the error growth depends on the length of the interval (the parameter b). The error reinforcement upon reflection is probably dependent on the phase angle which in turn depends on b. Of course, these results are based on a single, simple test case and may not apply to a given problem.

These computations were performed on the CDC 6400 at the
University of Colorado.

References

- [1] S. Bowen, "AMPLCT, A Numerical Integration Routine for Systems of Stiff Differential Equations", Department of Meteorology, University of Michigan, report #033390-2-T (1971).
- [2] M. Carver, "The FORSIM System for Automated Solution of Sets of Implicitly Coupled Partial Differential Equations", in Advances in Computer Methods for Partial Differential Equations, Vichnevetsky (ed.), AICA, Rutgers University (1975).
- [3] C. Chu and A. Sereny, "Boundary Conditions in Finite Difference Fluid Dynamic Codes", Jour. Comp. Phys., V. 15, pp 476-491 (1974).
- [4] T. Elvius and A. Sundstrom, "Computationally Efficient Schemes and Boundary Conditions for a Fine Mesh Barotropic Model Based on the Shallow Water Equations", Tellus, v. 25, pp 132-156 (1973).
- [5] T. Hull and W. Enright, "A Structure for Programs that Solve Ordinary Differential Equations", Technical Report #66, Department of Computer Science, University of Toronto (1974).
- [6] A. Loeb, "Users Guide to a New User-oriented Subroutine for the Automatic Solution of one dimensional Partial Differential Equations", in Advances in Computer Methods for Partial Differential Equations, Ivchnevetsky (ed.), AICA, Rutgers University (1975).
- [7] J. Olinger, "Fourth Order Difference Methods for the Initial Boundary-Value Problem for Hyperbolic Equations", Math. Comp., V. 28, pp 15-25 (1974).
- [8] A. Sunderstrom, "Efficient Numerical Methods for Solving Wave Propagation Equations for non-homogeneous Media", C4576-A2, Uppsala University (1974).
- [9] L. Shampine and M. Gordon, "Computer Solution of Ordinary Differential Equations", Freeman and Company, San Francisco (1975).
- [10] A. Sundstrom, "Note on the paper, 'Boundary Conditions in Finite Difference Fluid Dynamic Codes'", Jour. Comp. Phys., v. 17, pp 450-454 (1975).
- [11] Sincelar and N. Madsen, "Software for Nonlinear Partial Differential Equations", Math. Software II Conf., Purdue University (1974).
- [12] J. Hastings and R. Roble, "A Technique for Solving the Coupled Time-Dependent Electron and Ion Energy Equations in the Ionosphere", NCAR manuscript, Boulder, Colorado 80302 (1972).

	J	Max λ_r	$\ A\ _\infty$
Inconsistent 2nd order scheme (A).	6	0.0	50.
	11	0.0	199.
	21	0.0	798.
Consistent 2nd order scheme (B).	6	0.0	2.6
	11	0.0	3.3
	21	0.0	4.4
4th order with 3rd order boundary (C).	6	-0.04	2.5
	11	-0.03	3.6
	21	0.0001	4.4
4th order with 4th order boundary (D).	6	-0.54	5.8
	11	0.26	10.1
	21	0.26	15.9

Table I. Behavior of the matrix A
of the semidiscrete scheme $\underline{u}' = A\underline{u} + \underline{g}$.
Here λ_r denotes the real part of an
eigenvalue of A.

	J	T=6.28	T=62.8	T=1256.
(A) Inconsistent, RKF ODE solver.	11	9.44	665.	unstable
(B) Second order spatial, RKF ODE solver.	11	0.056	0.057	0.058
(C) Fourth order spatial. Fourth order Runge-Kutta with fixed $\lambda=\Delta t/\Delta x=1.8$. Third order at boundary.	6	0.061	0.069	0.070
	11	0.0059	0.0067	0.0068
	21	0.00039	0.00042	0.00042
(D) Fourth order spatial. Fourth order Runge-Kutta with fixed Δt . Fourth order at boundary.	6	0.052	0.066	0.069
	11	0.0031	0.0038	0.0039
(C) Fourth order spatial. RKF ODE solver. Third order at boundary.	6	0.012	0.016	0.015
	11	0.0043	0.0043	0.0043
	21	0.00024	0.00024	0.00025
(D) Fourth order spatial. RKF ODE solver. Fourth order at boundary.	6	0.033	0.025	unstable
	11	0.00069	0.0047	

Table II. Error for various schemes applied to equation (16)

	J	T=6.28	T=101	T=201	T=402	T=804
(E) Second order, solved by RKF	6	0.21	0.99	1.13	1.57	2.18
	11	0.065	0.33	0.56	0.88	1.39
	21	0.015	0.10	0.18	0.35	0.72
(F) Fourth order, solved by RKF, 3 rd order at body	6	0.043	0.20	0.23	0.26	0.30
	11	0.0022	0.012	0.020	0.034	0.052
	21	8.4E-5	6.1E-4	1.1E-3	2.2E-3	4.7E-3

Table III. Error for the solution of equation (21). Here T =time, and $\|u\|$ is the maximum norm of the solution.

	J	b	ϵ	T	N_E	Error
Constrained boundary (20)	11	1.0	0.01	1.0	348	0.052
	11	1.0	0.01	2.0	684	0.14
	11	1.0	0.01	4.0	1344	0.33
	11	1.0	0.01	10.0	3312	0.27
	11	1.0	0.01	20.0	6600	0.63
	6	0.5	0.01	1.0	354	0.011
	6	0.5	0.01	4.0	1350	0.031
	6	0.5	0.01	10.0	3312	1.16
	6	0.5	0.01	20.0	10122	213.00
Derivative-constrained (21)	11	1.0	0.01	1.0	318	0.052
	11	1.0	0.01	2.0	6.2	0.14
	6	0.5	0.01	1.0	318	0.011
	6	0.5	0.01	4.0	1194	0.031
Characteristic-derivative (22)	11	1.0	0.01	1.0	174	0.047
	11	1.0	0.01	2.0	342	0.061
	11	1.0	0.01	4.0	666	0.060
	11	1.0	0.01	10.0	1638	0.060
	11	1.0	0.01	50.0	8094	0.060
	11	0.5	0.001	1.0	606	3.1E-3
	11	0.5	0.001	20.0	11592	3.1E-3

Table IV. Error for solution of equation (17)
with various boundary conditions.