JOURNAL OF COMPUTATIONAL PHYSICS **8**, 83–105 (1971)

The Solution of Partial Differential Equations Using a Symbolic Style of Algol

K. V. ROBERTS

Culham Laboratory, Abingdon, Berkshire, England

AND

J. P. BORIS

U. S. Naval Research Laboratory, Washington D. C.

Received October 27, 1970

A large class of physical problems reduces to the solution of nonlinear sets of partial differential equations. The success of mathematical physics in such problems depends closely on the available mathematical notation, which is both elegant and precise. Even so, certain information can only be obtained from numerical computations using finite-difference methods on discrete spatial and temporal grids. Existing computers are already powerful enough to handle many interesting problems in two and three dimensions, and the main limiting factor is now the time taken to develop the necessary programs. In this paper a symbolic style of writing Algol is developed using three-dimensional magnetohydrodynamics as an example. Programs written in this way are clear and concise and can be written and modified quickly with little chance for error; they can also be converted easily for use with any computer system. Thus the development of a working computational physics program need not take man-years of high-level scientific effort. The symbolic style of programming developed here overcomes the usual objections to large Fortran programs by introducing an operator formalism which is very close to the formalism developed for mathematical physics. One defines a set of difference operators which "look" and "feel" just like their differential counterparts and thus the details of the finite-difference scheme can be largely hidden from view. Although the Symbolic Algol 'prototype' codes developed in this way execute too slowly for full three-dimensional production runs, the inner loops can readily be converted, either automatically or by hand, into highly optimized versions in any chosen language.

1. INTRODUCTION

This paper describes a symbolic programming technique, based on the Algol 60 language, which can be used for the rapid production of computational physics programs. So far it has mainly been applied to the solution of sets of coupled

nonlinear partial differential equations, using finite difference methods on a discrete space-time mesh, but the technique appears to have other applications which lie outside physics, since it could also be used for constructing many different kinds of computer software. The programs are constructed in modular form; they are portable (i.e., they can be quickly adapted to any computer system for which an Algol 60 compiler is available); they are readily intelligible to theoretical and experimental physicists and easily updated; and they can be made as efficient as required.

Mathematical Symbolism

The successes of mathematics can be linked quite closely to the development of a widely accepted notation which is elegant, compact, and precise. In mathematical physics, for example, a vector-valued function can be represented by a single symbol, independent of the coordinates, with both the indexes and the functional dependence suppressed. A theoretician does not write

$$\text{(BX(IX, IY, IZ), BY(IX, IY, IZ), BZ(IX, IY, IZ))} \tag{1}$$

as in Fortran but simply uses B instead. A symbolic vector-analytic expression such as

$$\mathbf{\nabla} \times \mathbf{V} \times \mathbf{B} \tag{2}$$

can be used for problems in one, two, or three dimensions and for any type of coordinate system, whether it is rectangular, cylindrical, spherical, generalized orthogonal curvilinear, or whatever.

Fortran Difference Notation

It would seem that comparable successes for computational physics must await the development of an equally lucid, equally powerful symbolic method for programming problems directly. The extent of the difficulty may be gauged from the fact that a simple expression such as (2) can expand by nearly a factor 100 when converted to a three-dimensional difference form which is compatible with standard implementations of Fortran. Although the translation of ordinary mathematical notation into executable program statements ought surely to be a purely mechanical process, this is not yet the case in practice; the development and testing of a valid and useful scientific program often occupies several man-years of high-level scientific and mathematical effort. Furthermore the resulting code is usually bulky and hard to understand, so that it is difficult to modify the program or even to be quite sure that it is correct.

Symbolic Algol I

The sought-after simplicity, clarity and preciseness of mathematical physics can be reproduced almost exactly by programming in a symbolic style of Algol which is equally intelligible to compilers, numerical analysts and theoretical physicists alike. For example, the vector magnetic field equation

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{V} \times \mathbf{B}) + \eta \nabla^2 \mathbf{B}, \tag{3}$$

where η is the scalar resistivity and **V** is the velocity of the fluid medium in which the vector magnetic field **B** is embedded, can be integrated in difference form in Algol as

$$\text{NEW B:} = \text{B} + \text{DT} \times (\text{CURL(CROSS(V, B))} + \text{ETA} \times \text{DELSQ(B)}); \tag{4}$$

Here the physical quantities V, B and the analytic or algebraic operators CURL, CROSS, DELSQ are real procedures whose structure will be explained in Sections 4 and 5, while NEW B, DT, and ETA are real variables. Like Eq. (3) the Algol statement (4) is independent of the coordinate system and of the number of dimensions, and the actual choice is made by changing the definition of lower-level procedures, or the values of certain 'hidden' global constants. Statement (4) is also independent of the particular difference schemes used.

Privileged Variables

This style of Algol programming,[1] which will be called Symbolic Algol I to distinguish it from a second style to be mentioned later, relies on the use of **privileged variables** such as V and B in statement (4). A privileged variable is in fact a parameterless typed procedure ('function' in Fortran parlance), whose value, when invoked, is determined by the current state of variables which are declared to be global in scope to the actual procedure declarations and are therefore hidden from view. Thus the magnetic field B is represented by

real procedure B; B: = AB[C1, Q]; (5)

where AB is an array which stores the actual values, C1 is an integer specifying the component value (1, 2 or 3), while Q is an integer which labels the position on the mesh. Procedures such as CROSS, CURL, DELSQ manipulate the values of the hidden variables C1 and Q, so extracting the array values needed in the difference scheme.[2]

[1] Some of the ideas can be traced to earlier work by Dr. N. K. Winsor at Princeton.

[2] In the more general case of a tensor, C1 will signify the first component. Q has been chosen as a relatively rarely used letter, not too dissimilar to O (local origin).

Privileged variables have other applications. In mathematical physics, for example, the same notation is normally used for the resistivity η whether it is a constant, or a function which depends on the coordinates x, y, z, or the physical variables B, ρ, T, (where ρ is the density and T the temperature). Correspondingly in Symbolic Algol we might write

real procedure ETA; ETA: = CONSTANT × TEM ↑ 1.5, (6)

where TEM is itself a privileged variable which describes the temperature.

Control Phrases

There is another use for parameterless typed procedures which is important in establishing an intelligible structure for the program. When using the leapfrog scheme, for example, in which alternate points of the mesh are calculated at each timestep, we can write

if THIS POINT IS EVEN **then** INVOKE DIFFERENCE EQUATIONS; (7)

Here THIS POINT IS EVEN is a Boolean procedure which is defined by **Boolean procedure** THIS POINT IS EVEN;

THIS POINT IS EVEN: = I + J + K + N = 2 × ((I + J + K + N) ÷ 2); (8)

where I, J, K, N are the integral coordinates of points on the space-time mesh. INVOKE DIFFERENCE EQUATIONS is a procedure which performs the required action (see Appendix). Other typical control phrases are:

TWO DIMENSIONS

PERIODIC X;

RIGID Y; (9)

LAX WENDROFF SCHEME;

All these procedures set hidden 'switches' which determine the type of problem to be solved, or the difference scheme to be used. Three features of Algol make this use of control phrases feasible; the ability to have functions with no parameters (which are not allowed in standard Fortran), the absence of any real restriction on the length of identifiers, and the omission of the unnecessary word 'CALL.'

In Fortran the set of phrases (9) would have to be replaced by something like

```
CALL TWODIM
CALL PRDICX
CALL RIGIDY
CALL LAXWEN
```
(10)

which is much less satisfactory.

Modular Structure

In mathematical physics, the same formalism can often be used to solve a wide range of problems; for example vector algebra and vector analysis, matrix algebra and Hamiltonian dynamics are all used in this general way. A similar generality applies to Symbolic Algol, because the procedures which implement DOT, CROSS, GRAD, CURL and other operators are largely problem-independent, so that they can be coded and tested once-for-all and then stored in a random-access file in a convenient modular form, for use by any programmer.

Once this principle has been recognized, it is natural to look for other sections of a typical physics program which can be prefabricated and "packaged" in this way. The development of a universal set of Symbolic Algol modules for solving general time-dependent problems involving partial differential equations has been initiated at the Culham Laboratory by R. S. Peckover and one of the authors (KVR) and this work will be reported elsewhere [1]. It has been found that virtually the whole program can be built up from prefabricated symbolic modules, including both the physics and the organizational sections.

Portability

Algol is a universal language except for two features; the hardware representation and the input–output procedures. The former need present no real problem because Peckover has shown that in most cases the conversion from one hardware representation to another can be carried out automatically using a context editor. In the scheme developed by Peckover and Roberts [1], all the output is channelled through one small module (OUTALGOL), containing less than 20 cards, which is simply replaced by an alternate version in switching to a different computer system.

The same Symbolic Algol modules have currently been used with 10 different Algol 60 or Algol W compilers, seven different types of computer system including the IBM 360, ICL 1900 and CDC 6600, and six different on-line systems of which three were commercial computer utilities. This suggests that it should be possible to set up 'universal' suites of modules for computational physics, comparable to the standard notations used in mathematical physics.

Testing and Optimization

Although Symbolic Algol I enables programs to be developed and tested very quickly, these programs execute quite slowly because of the large number of nested procedure calls involved; typically 30–100 times more slowly than optimized Fortran. Several methods have been used successfully to overcome this problem.

All the methods depend on the obvious fact that different sections of the program are executed with widely different frequencies. Consider for example a three-dimensional production run, using a $64 \times 64 \times 64$ mesh and computing for 256 timesteps, or 2^{26} space-time steps in all. While the main difference equations will be executed 2^{26} times, some of the more complex logical statements in the initialization procedures will be executed only once, while (hopefully) those in failure procedures will be executed less than once on average. In optimizing the final version for production runs it is only necessary to recode those sections which are executed more than (say) 2^{18} times, and these usually form only a small proportion of the total program.

We have found it practicable to test a program with a comparatively small number of time steps using a coarse mesh; say $8 \times 8 \times 8$ for eight timesteps, or 2^{12} in all. Then the slow speed of Symbolic Algol I is quite immaterial. In fact because all the space-time steps are essentially the same, many programming errors can be found by checking the results of a *single* space-time step, and this can readily be done on-line [1].

Once the program has been checked out in Symbolic Algol I, a detailed series of trial runs can be carried out, again using a relatively coarse mesh. The sections which have high execution frequency are then recoded in optimized Algol, Fortran or assembly language, and the trial runs repeated, when any coding errors should produce a discrepancy in the results. Once the optimized version has been verified, the number of mesh points can be increased and production runs made.

Symbolic Algol II

Although this optimization procedure is not difficult to carry out by hand, it is a purely mechanical process and therefore capable of being automated. A minor transformation of statement (4) (which can itself be carried out automatically), will convert it into the equivalent Algol statement

```
EQUATE(B, SUM(B, MULT(DT, SUM(CURL(CROSS(V, B)),
                       MULT(ETA, DELSQ(B))))));      (11)
```

where EQUATE is a procedure, and all the other identifiers represent typed procedures. These can be defined in such a way that they calculate and then print or punch the symbols needed to implement the optimized version in any desired language. This technique, to be referred to as Symbolic Algol II, was suggested

by the authors [2, 3] and has been implemented by Dr. M. Petravic and Dr. G. Kuo-Petravic [4, 5] who have generated optimized Fortran, Algol, ICL KDF9 Usercode and IBM 360 assembly language. The latter two versions seem to be about as fast as a good Fortran compiler and a good Fortran programmer could jointly produce. Symbolic Algol II will be briefly discussed in Section 6.

2. A Model Problem

To make the discussion more concrete, Symbolic Algol I will be described in the context of a model problem, TRINITY, which solves a simple set of three-dimensional magnetohydrodynamic (3DMHD) equations on a rectangular Eulerian mesh, using the leapfrog scheme for the dynamical terms and the Dufort–Frankel scheme for the diffusion terms [6, 9]. The basic equations are

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \rho \mathbf{V}, \tag{12}$$

$$\frac{\partial(\rho V_i)}{\partial t} = -\frac{\partial}{\partial x_j}(p_{ij}) + \nu \nabla^2 \rho V_i\,, \tag{13}$$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{V} \times \mathbf{B}) + \eta \nabla^2 \mathbf{B}, \tag{14}$$

$$\frac{\partial T}{\partial t} = -\nabla \cdot T\mathbf{V} + (2-\gamma)\, T\nabla \cdot \mathbf{V} + \kappa \nabla^2 T$$

$$+ (\gamma - 1)\,\eta \mathbf{J}^2/\rho + (\gamma - 1)\,\nu[(\nabla \times \mathbf{V})^2 + (\nabla \cdot \mathbf{V})^2]. \tag{15}$$

Here $p_{ij} \equiv (\rho T\delta_{ij} + V_i V_j + (B^2/2)\,\delta_{ij} - B_i B_j)$ and $\mathbf{J} \equiv \nabla \times \mathbf{B}$. The fluid field variables are the mass density ρ, the three-component fluid velocity $\mathbf{V}$, the three-component magnetic field $\mathbf{B}$, and the temperature T, (kT in actuality). γ is the ratio of specific heats for the fluid, typically 5/3. The three transport coefficients η, the electrical resistivity, ν, the viscosity, and κ, the thermal conductivity, will be assumed to be constant over the entire system and independent of time. These restrictions are not necessary for the symbolic approach. On the contrary, the symbolic approach is shown in its best light for extremely complex equations, difference schemes, geometries, etc., because the complexities can be treated once and then hidden from sight. Here, however, we minimize these complexities for two reasons:

(a) For the purposes of this paper the symbolic methods are most transparent in a simple problem. A good example should not require a great deal of specialized physical understanding since the method is intended to be quite general.

(b) The TRINITY program was originally designed, and has in fact been used, for full three-dimensional MHD production runs. To keep the execution time within reasonable bounds for a 3D run, even in a fully optimized code on the IBM 360/91, great simplicity is required.

Difference Scheme

A Cartesian coordinate system is used with a grid spacing of δs (constant) in each of the three coordinate directions, with each of the eight field quantities specified at each grid point. A fully time- and space-centered leapfrog difference scheme is used to ensure second-order accuracy for the dynamic terms. This explicit difference formulation, in which the 3D mesh is fully staggered in time, is particularly well suited toward writing high-speed, optimized codes and conserves global ρ, global $\rho\mathbf{V}$, and local $\nabla \cdot \mathbf{B} = 0$ identically.

Figure 1 shows one plane of the staggered mesh. The field quantities (ρ, $\mathbf{V}$, $\mathbf{B}$, T) on the grid are thought of as specified at two different times, separated by δt, depending on whether the sum of $i + j + k$ is "even" or "odd." Thus the O-points all have discretized physical variable values specified at time t, say, and the X-points have their mesh values specified at time $t + \delta t$. When τ, the timestep number,

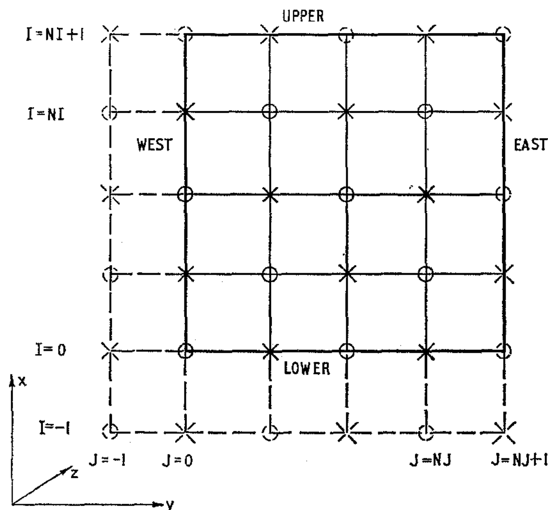

FIG. 1. One plane of the staggered mesh. The physical region is a cube, containing $NI \times NJ \times NK$ cells. Because of the periodic symmetry, the three faces which may be denoted by East, Upper and North do not have to be calculated and act as guard planes. An extra set of guard planes is provided outside the West, Lower and South faces. The figure shows one plane of the mesh for an even value of K, and NI = NJ = 4. Points O are recalculated at even steps, and points X are recalculated at odd steps. Points ○ and × are guard points, set by symmetry.

is even, only the even mesh variables O are advanced in time by $2\delta t$. Thus, after this timestep in which all gradients and fluxes are computed using the time- and space-centered X-point variables, the O-points now contain variables at $t + 2\delta t$. These new values are then suitable to advance the odd points in time in a fully time- and space-centered manner.

Since the O-points and X-points interact primarily through gradients and derivatives of each other, it is natural for the meshes to decouple. In fact as explained in Ref. 6 (Section IIG), there are eight uncoupled meshes in three dimensions. Except for the density, these are linked together by the Dufort–Frankel scheme which is used for the diffusion terms.

Periodic geometry is normally used in TRINITY, although other boundary conditions have been implemented for specific physical problems [9].

The actual Symbolic Algol I program equivalent to Eqs. (12–15) is given in the Appendix.

3. Storage Allocation for The Variables

Field Variables

The eight dependent variables are represented on a three-dimensional, rectangular, uniformly spaced finite difference mesh, with NI, NJ, NK intervals in the three directions, respectively. In periodic geometry we write

$$\mathrm{PI} = \mathrm{NI} + 2, \qquad \mathrm{PJ} = \mathrm{NJ} + 2, \qquad \mathrm{PK} = \mathrm{NK} + 2; \tag{16}$$

then the total number of storage locations required for each variable is

$$\mathrm{SIZE} = \mathrm{PI} \times \mathrm{PJ} \times \mathrm{PK}. \tag{17}$$

The grid spacing is DS in each direction, and the three independent coordinates are

$$\begin{aligned} x &= \mathrm{I} \times \mathrm{DS}, \\ y &= \mathrm{J} \times \mathrm{DS}, \\ z &= \mathrm{K} \times \mathrm{DS}, \end{aligned} \tag{18}$$

where

$$\begin{aligned} -1 &\leqslant \mathrm{I} \leqslant \mathrm{NI}, \\ -1 &\leqslant \mathrm{J} \leqslant \mathrm{NJ}, \\ -1 &\leqslant \mathrm{K} \leqslant \mathrm{NK}. \end{aligned} \tag{19}$$

The active region of calculation is

$$\begin{aligned} 0 &\leqslant I \leqslant NI - 1, \\ 0 &\leqslant J \leqslant NJ - 1, \\ 0 &\leqslant K \leqslant NK - 1, \end{aligned} \tag{20}$$

and the six extra planes $I = -1$, NJ, (all J and K) etc. are considered to be guard planes. These are provided so that the difference formulation can be extended unmodified right up to the boundaries of the physical system. In the leapfrog algorithm adopted, the entire problem has been cast as a three-level, nearest-neighbour difference solution and thus only one level of guard planes need be supplied.

For actual core references to the dependent field variables, a single-subscript notation is chosen. The three-dimensional arrays are represented by a single subscript

$$Q = 1 + (I + 1) + (J + 1) \times PI + (K + 1) \times PI \times PJ, \tag{21}$$

where Q is declared a global integer, representing the current center of all difference-equation formulations. It is clear that Q has the permissible range of $1 \leqslant 0 \leqslant SIZE$, which is chosen for compatibility with Fortran.

For the vector quantities **V** and **B**, a second subscript $C1 = 1, 2, 3$ is attached to array storage declarations to distinguish the three vector components. Thus, for TRINITY, the appropriate array declarations are

$$\textbf{real array}\ A\,RHO,\ A\,TEM[1 : SIZE],\ AV,\ AB[1 : 3, 1 : SIZE]; \tag{22}$$

The letter A preceding the variable names distinguishes between actual and privileged variables and has the mnemonic of "array". Thus the continuity equation can be written and executed as

$$A\,RHO[Q]: = RHO - DT \times DIV(RHO \times V); \tag{23}$$

and reads as:

"The new ρ at the current origin is equated to the old ρ minus δt times the divergence of ρ times **V**."

Although Q and C1 are really array subscripts, it is more correct to view them as pointers in the discussion of Symbolic Algol that follows. They point both to a core location and to a physical location in the rectangular problem space. As global pointers they get modified by the various operators and then, in turn, all quantities which depend on these pointers have modified values when referenced.

4. Privileged Variables

In Eq. (23) of the preceding section the continuity equation was written in a symbolic notation in which the references to ρ (RHO) in the right- and left-hand sides took different forms. When a new value is being inserted into A RHO[Q], direct reference to the actual core location is necessary. When numerical values are merely being picked up as members of an intermediate calculation, on the other hand, the Algol language permits a very much compressed notation through the use of **privileged variables**. As explained briefly in Section 1, a privileged variable is a parameterless typed procedure whose value, when invoked, can be controlled by the current state of any of the variables which are declared to be global in scope to the actual procedure definition. In the case of the TRINITY program the main global variables concerned are the pointers Q and C1 of the previous section.

In the example of Eq. (23), RHO, V, and possibly DT are privileged variables; no subscript or component references appear or are necessary. The utility of privileged variables over their closest Fortran counterparts is enormous because of the structure of the Algol language and the restrictions of Fortran. Computational compactness such as shown by Eq. (23) cannot be rivalled by a corresponding Fortran program because (a) there must be at least one argument, possibly a dummy, for every Fortran function, and because (b) Fortran has no real analog to the Algol call-by-name facility.

The first restriction reduces the possibility of program brevity and hinders notational simplicity. As remarked earlier, the ability to achieve brevity in scientific notation has aided the rapid development of both physics and mathematics and thus compactness should be one of the prime goals of the symbolic style of programming. The second restriction of Fortran follows from a subtle but extremely important point of programming philosophy and will be treated in the next section.

The simple privileged variables declared in TRINITY are listed below

real procedure RHO; RHO: = A RHO[Q];
real procedure TEM; TEM: = A TEM[Q];
real procedure V; V: = A V[C1, Q]; (24)
real procedure V2; V2: = A V[C2, Q];
real procedure B; B: = A B[C1, Q];
real procedure B2; B2: = A B[C2, Q];

where '2' refers to the second component j in tensor expressions such as $V_i V_j$, p_{ij}. Note that both V and V2 point to the same array, and similarly for B, B2.

All these definitions rely on the global variables A RHO, A TEM, A V, A B

Q, C1, etc., which must therefore be preset to realistic values before any of the above privileged variables are invoked. These variables are used in the following way: suppose C1 = 1, and Q is defined by Eq. (21), where I, J, and K also have preset values. Then

$$\text{RHO} \times \text{V}$$

when referenced, has the value

$$\text{A RHO[Q]} \times \text{A V[1, Q]},$$

or in a more explicitly scientific notation

$$\rho(i, j, k)\ V_x(i, j, k).$$

We are making a distinction here between privileged variables and the more general class of parameterless procedures. Privileged variables are typed real or integer and generally very short and their definitions can often be written in a single line. By contrast, other uses for parameterless procedures take explicit advantage of the long-identifier facility of Algol, a prohibition in most Fortran implementations. In fact, parameterless procedures can be used in a number of ways to make Algol more concise and readable by following through one of the basic tenets of the Algol language, that the logic should read like English and the computation like mathematics. Thus in TRINITY, for example, parameterless Boolean procedures are defined to carry out various checks on the current problem state implicitly and return the value, **true** or **false**, depending on what they find out. One of the more important examples is the Algol statement (7)

if THIS POINT IS EVEN **then** INVOKE DIFFERENCE EQUATIONS;

THIS POINT IS EVEN is a Boolean-type procedure, and INVOKE DIFFERENCE EQUATIONS is an Algol procedure equivalent to a Fortran subroutine. The definition of the Boolean procedure is given by Eq. (8). In passing we note that the corresponding Fortran to statement (7) might be

IF((I + J + K + N).EQ. 2* (I + J + K + N)/2)) CALL INDIFQ. (25)

This is superficially shorter than (7) but requires comments for documentation while (7) does not. Further the Algol logical check may appear in many places. Thus, in statement (7) the body of THIS POINT IS EVEN need only be changed once in one place to modify the entire program logic.

We close this section by defining one more privileged variable. Because of its

complexity it demonstrates more forcefully the power of Symbolic Algol:

real procedure P;

P: = **if** C1 = C2 **then** (RHO*(TEM + V × V) + 0.5 × DOT(B, B) − B × B)

else (RHO × V × V2 − B × B2); (26)

This privileged variable is the entire tensor pressure used by TRINITY, defined in a couple of lines for all locations and vector components.

In the simplest and least efficient 3D Fortran notation, using triple subscripts, the P_{xx} component of (26) would read

PXX = RHO(I, J, K)*(TEM(I, J, K) + VX(I, J, K)*VX(I, J, K))

+ 0.5*(BY(I, J, K)*BY(I, J, K) + BZ(I, J, K)*BZ(I, J, K)

− BX(I, J, K)*BX(I, J, K)). (27)

There are five more distinct components to P_{xx} and a statement like (27) would have to be written for each of these with great chance for error.

5. Operators in Symbolic Algol Programming

Having defined the basic physical variables, we now introduce a set of operators which act on these, analogous in some ways to the operators used in quantum mechanics.

Rotation Operators

Two small integer arrays are declared globally to help with the symbolic treatment of the component index C1:

integer array CP, CM[1 : 3];

C1 is restricted to the three values, 1, 2, and 3 signifying the x, y and z components, respectively. In many vector and differential operations such as sums, cross products, and curls, it is advantageous to be able to rotate the component indices in a cyclic manner. Thus we define

CM[1]: = 3; CM[2]: = 1; CM[3]: = 2; CP[1]: = 2; CP[2]: = 3; CP[3]: = 1; (28)

The mnemonics for P and M are plus and minus and indicate cyclic rotation in the plus and minus directions, respectively. The rotation is actually carried out

by two very short real procedures which act as clockwise and anticlockwise rotation operators:

```
real procedure RP(A); real A; begin C1: = CP[C1];
                              RP: = A; C1: = CM[C1]; end;
real procedure RM(A); real A; begin C1: = CM[C1];
                              RM: = A; C1: = CP[C1]; end;     (29)
```

The vector product can then be defined as

```
real procedure CROSS(A, B); real A,B;
      CROSS: = RP(A × RP(B)) − RM(A × RM(B));     (30)
```

This use of short nested procedures is typical of Symbolic Algol. Note that although side effects are used to alter the value of C1, the original value is restored before the procedure eventually relinquishes control; the side effect is therefore 'reversible.' To see how (30) works, suppose that we wish to calculate the y component

$$(A \times B)_2 = A_3B_1 - A_1B_3 .$$

When CROSS is entered, C1 has the value 2. The first occurrence of RP causes a clockwise rotation and generates the value C1 = 3, so extracting A_3. Then a further rotation extracts B_1. Similarly, two successive anticlockwise rotations produce the second term in (31). Just as in ordinary vector algebra, both A and B may themselves be complex vector expressions involving algebraic or analytic manipulations.

Translation Operators

Mesh translations are carried out by means of a basic operator

```
integer procedure DQ; DQ: = if C1 = 1 then 1 else if C1 = 2 then PI else PI × PJ;
                                                                        (32)
```

There are two vector translation operators, one being the inverse of the other;

```
real procedure EP(F); real F; begin Q: = Q + DQ; EP: = F; Q: = Q − DQ; end;
                                                                        (33)
real procedure EM(F); real F; begin Q: = Q − DQ; EM: = F; Q: = Q + DQ; end;
```

These allow a vector difference operator DEL to be defined;

```
real procedure DEL(F); real F; DEL: = (EP(F) − EM(F))/DS2;     (34)
```

where DS2 = 2 × DS, and then CURL can be defined in terms of DEL;

real procedure CURL(A); **real** A;

CURL: = RP(DEL(RP(A))) − RM(DEL(RM(A))); (35)

Broadly speaking, there is a one-to-one correspondence between the operator formalism available in analysis, numerical analysis and Symbolic Algol. For example once we have defined a vector lattice translation operator in numerical analysis:

$$E_x f(x, y, z) \equiv f(x + \delta s, y, z), \tag{36}$$

etc.
then a centered difference operator $\tilde{\nabla}$ can be defined by

$$\tilde{\nabla} \equiv (E - E^{-1})/(2 \cdot \delta s) \tag{37}$$

and then

$$\widetilde{\text{Curl}}\, \mathbf{A} \equiv \tilde{\nabla} \times \mathbf{A} \tag{38}$$

Two slight differences may be mentioned; firstly we cannot take inverses for granted in Symbolic Algol but have to define them separately, and secondly we cannot use the vector product operator in (35) in the same way as in (38) because the latter application is rather unusual; one of the operands of × acts on the other.

Call by Name

This use of operators may seem unfamiliar to those versed in Fortran, since it relies on the Algol 'call-by-name' facility. An elementary discussion of the facility will therefore be given here.

Call-by-name is closely linked to the concept of a generalized formal parameter in Algol. In general, when a parameter (or 'argument') is passed between modules, what the usual compiler does is to put the address of the core location of the parameter in a place where the called module will know to look. The called module "knows" where to find the address of the first parameter, the second, and so on. When the value of the parameter is needed, the contents of the appropriate core location are picked up. Where Algol and Fortran usually differ, in effect, is in what the contents of these pre-specified core locations mean.

In Algol the parameter is generally thought of as a reference to a function which returns the actual value of the parameter every time the formal parameter name is referred to by the called subroutine. The external variable, function, expression, etc. which corresponds in the calling sequence to the formal parameter being referenced, is reevaluated at very such reference. If the procedure (routine) being called changes something on which the actual parameter value depends, then the value of the parameter changes during the execution of the called procedure.

The great flexibility and power of this generalization can be seen by reference to the rotation operator RP(A) defined by (29), which clearly corresponds to an algebraic expression of the form $A^* = RAR^{-1}$. This very simple procedure only works because of call-by-name. In (29), A is a formal real (by name) parameter which is assumed to be a vector, depending on the globally declared component index C1. Thus RP evaluates A for a cyclically shifted component index, and then restores that index to the value it had on entry.

Suppose R is involved elsewhere in the program in the following way:

REAL VARIABLE: = RP(V);

where V is the privileged velocity variable. The Algol compiler constructs code whose essential actions are as follows:

(a) A function reference is constructed which returns the value V: = AV[C1, Q] whenever the argument function is referenced.

(b) This function reference is put in a place where RP "knows" to "look" for it.

(c) Control is passed to RP which immediately "looks" at the prespecified function reference and equates this reference to its internal dummy function A.

(d) The current value of C1 is rotated to CP[C1] = C1 + 1 (modulo 3). This has the effect of moving the component pointer C1 cyclically from x to y, y to z, or from z to x.

(e) The return value of RP is set equal to A which is evaluated through the external referencing to V = AV[C1, Q] where C1 is now the *new* value of the component pointer.

(f) The value of C1 is restored to its original value, C1: = CM[C1]; before exiting from the procedure.

(g) The program control is returned to the calling program which assigns the return value of RP to the real variable (or to any other expression in which RP is used).

On the surface this sequence of calculations and actions seems very simple and natural but the distinction from Fortran must be pointed out again. The important points of difference occur in steps (c) and (e). In step (e), the parameter reference is not resolved during construction of the calling sequence. The external variable, function, expression, conditional expression, or whatever, is not evaluated by Algol until precisely that time when the formal parameter is actually referenced within the called procedure. If A in the above example were referenced several times with C1 and Q being changed between references, each usage would bring a different

An important feature of call-by-name as used in TRINITY is the "reversible side effect." In the example given above, C1 was rotated cyclically, a side effect, and then was reset to the value it had on entry after the required value of the formal argument A had been evaluated. These reversible side effects allow all of the vector algebra and difference formulae to be evaluated behind the scenes. Other controlled side effects of typical procedures are used in Symbolic Algol II for generating code.

Scalar product

Another of the TRINITY operators will now be given to illustrate the method further. In its most symbolic form, the scalar product can be implemented as:

```
real procedure DOT (A, B); real A, B; DOT: = SIGMA (A × B);    (39)
```

where in three dimensions:

```
real procedure SIGMA (F); SIGMA: = F + RP(F) + RM(F);    (40)
```

These formulae correspond to the algebraic expression

$$a \cdot b \equiv \sum_i a_i b_i \qquad (41)$$

The dot product can however be written in another form which is less symbolic but which will execute much faster than the form (39). This other form (42) is computationally equivalent to (39) but removes several of the expensive calling sequences:

```
real procedure DOT (A, B);  real A, B;
begin                       real SUM;
C1: = CP[C1];               SUM: = A × B;               (42)
C1: = CP[C1];               SUM: = SUM + A × B;
C1: = CP[C1];               DOT: = SUM + A × B;
end;
```

Whatever the initial value of C1, three rotations span the entire range and bring it back to its initial value. Many of the operators can be speeded up in this way without changing the physical statements at the highest level.

6. Optimization of Symbolic Programs

Direct calculations using the fully symbolic versions of TRINITY are not very fast; the computer CPU is not used efficiently, in general, when multiple nesting of calling sequences is necessary, and a great fraction of the actual calculation is redundant. The only solution to this problem, for large 3D calculations, seems to be optimization of the inner loops of the symbolic code, by one method or another.

Various levels of hand coding are possible and most of these have actually been implemented, giving identical results on several computers:

(a) *Fully Symbolic Form* (*Algol*). All operators and variables are treated symbolically. Each operator, in general, consists of a single line definition. Thus, for example, DOT in (39) is defined in terms of the summation operator SIGMA. This is the slowest method of all, and can be extended almost arbitrarily far. Initial and boundary conditions can be treated symbolically, for instance.

(b) *Optimized Symbolic Form* (*Algol*). This approach is considerably faster but otherwise very little different from (a): the two methods differ only in the operator definitions. Rotation and displacement operators are not used but otherwise form (b) is obtained from form (a) by substituting the DOT operator (42) for (39), say.

(c) *Optimized Algol Form.* Here the basic outlines of the symbolic program are retained; the data structure, the initial conditions, the boundary conditions, etc. Direct substitutions for all of the operators in the difference equations greatly increase program efficiency but increase the time needed to check out the program. A great saving of execution time is achieved by using the optimized code for production.

(d) *Standard Fortran Form.* Here the program is basically as in (c) above but the Fortran compiler is used. On the IBM 360/91 the H-level (OPT = 2) compiler generates exceedingly tight code, as compared to the IBM Algol compiler, and either Fortran or assembly language should eventually be used for the inner loops where a large number of long runs are being contemplated.

(e) *Optimized Fortran Form.* When all possible tricks are employed, such as storing momentum and magnetic flux quantities to save recalculation, and using a singly subscripted notation to optimize on register usuage, a factor of two or three more can be obtained. On the IBM 360/91 a $40 \times 40 \times 20$ grid can then be executed on a production basis at about 2.5 secs/step or about 1400 steps/hr. Almost all production work so far done with TRINITY has been with codes of this type.

(f) *Optimized Assembly Language.* This is the fastest version, running with

100 % efficiency, and can be obtained by hand coding version (e). Not more than a factor 2 improvement on (e) can however be expected, at least on the IBM 360/91, because the H-level compiler for Fortran is so good, but since the conversion is comparatively straightforward it would probably be worthwhile.

Automatic Optimization

Clearly an automated optimization facility is to be preferred over direct hand optimization of the symbolic program. An approach suggested by the authors [2, 3] results from the observation that the symbolic version (a) already contains enough information to generate an optimized version of itself. The privileged variables RHO, V, B, T can be modified in such a way that they print or punch their own names as side effects, while the operators such as CURL, CROSS manipulate the values of C1 and Q which are needed in subscript expressions or index register settings. Clearly the arithmetic operators $+$, $-$, $\times$, $/$ and the assignment symbol $:=$ cannot perform this type of action, but they can be converted either by hand or automatically to the Symbolic Algol II form, in which expressions such as

$$\mathrm{A} + \mathrm{B}, \qquad \mathrm{A} := \mathrm{B} \tag{43}$$

become

$$\mathrm{SUM(A, B)}, \qquad \mathrm{EQUATE(A, B)}, \tag{44}$$

respectively. Then these procedures can be made to perform any desired action.

Dr. M. Petravic and Dr. G. Kuo–Petravic have succeeded in generating optimized versions of TRINITY in Algol, Fortran, ICL KDF9 Usercode and IBM 360 assembly language from the same Symbolic Algol II difference scheme, by suitable choice of the lower-level procedures [4]. They have also taken the obvious next step by writing a macro-preprocessor to allow the notation of the symbolic program to be even closer to mathematics [5]. One serious deficiency of Algol, for instance, is the lack of a facility permitting user definition of infix operators. This preprocessor therefore converts the statement

$$\mathrm{DB/DT} = \mathrm{CURL(V} < \mathrm{X} > \mathrm{B)} + \mathrm{ETA} \times \mathrm{DELSQ(B)} \tag{45}$$

into the Symbolic Algol II statement (11) or alternatively, it will convert (4) into (11).

The Symbolic Algol II optimization program is itself highly modular, and it can be used for a wide range of problems. Besides generating any necessary linkages and control statements, it can also carry out a certain amount of physical optimization; for example if informed that $\partial/\partial y$ and $\partial/\partial z$ are zero, it will avoid generating

any expressions in which these occur as products, together with any redundant operators, so that

$$a + g(\partial f/\partial y) + c \tag{46}$$

is replaced simply by

$$a + c. \tag{47}$$

This enables three-dimensional vector expressions such as Eq. (43) to be used for one- or two-dimensional problems, as is normal in mathematical physics.

Broadly speaking, the quality of the code which can be generated in this way is about the same as that of the Fortran hand version (d). It would be difficult to take account of the physical symmetry of the problem automatically, by storing fluxes for subsequent re-use, but this slight loss of efficiency is balanced by the improved efficiency of assembly language as compared to Fortran. A further optimization which could be carried out in a straightforward way would be the elimination of common sub-expressions; for example the combination of constants $e^2/\hbar c$ might be replaced by a single constant α, while terms such as $a - a$ could simply be removed.

7. Concluding Remarks

A symbolic style of Algol programming has been developed for application to [illegible]

dimensional problem as an example. This program, called TRINITY, uses an explicit leapfrog difference scheme, centered both in space and time, to advance the variables **V**, **B**, ρ, and $T \equiv P/\rho$. A rectangular Cartesian mesh is used. The method however seems capable of generalization to any type of difference scheme, explicit or implicit, Lagrangian or Eulerian, rectangular or curvilinear.

In the symbolic method of using Algol, vector-valued functions can be represented by single coordinate-free symbols, with positional and vector indices suppressed. Thus, using essentially classical vector-, tensor-, and differential-calculus notations, numerical algorithms for the solution of partial differential equations can be prescribed with a fraction of the effort needed for writing a computer program in a more standard programming style. As an added benefit, of course, the chance for programming errors to occur is very much reduced. This symbolic style of Algol has now been used on several computer systems in the USA, UK, and Germany.

Symbolic Algol permits the details of the vector and numerical analysis to be

built into a hierarchy of compact operators which can be defined in an extremely general manner. The clarity, flexibility, and generality which one obtains by this method have one compensating drawback; the symbolic codes run very much more slowly than their optimized counterparts. Optimization of one form or another, as discussed in Section 6, will almost certainly be required for 3D runs of any realistic and interesting scope.

A typical 3D programming job therefore proceeds as follows:

(a) A 'prototype' code, containing all of the necessary physics and numerical analysis but running very slowly, is written in Symbolic Algol I. Programming errors are not easy to make and those that do occur can usually be found very quickly. Much of the work can be carried out on-line, taking advantage of a rapid turnround for short jobs.

(b) Because the running time for a 3D explicit code goes roughly as the inverse fourth power of the mesh spacing, a series of test problems of increasing size can be run, with the final space mesh only about two or three times coarser than that needed for the full production runs. These test runs serve as reference cases for optimized versions of the codes.

(c) The prototype code is reduced to a computationally equivalent but highly optimized form to permit efficient production runs, either by hand or automatically. Only the inner loops need be processed in this way.

(d) When completed, the fast production version is put through the same series of test runs as for the prototype, and comparison with these reference tests is made. This gives a positive check on the accuracy of the final code and enables any errors to be tracked down and eliminated quickly.

A proper application of this technique results in a well-structured, modular code in which the statements are either expressed in clear English-language control statements, or by mathematical expressions such as Eq. (43). A family of such codes will be presented elsewhere [1]. TRINITY is currently being adapted to run on a $60 \times 60 \times 60$ mesh on the IBM 360/91, using two IBM 2301 drums on separate channels to store the 2 million physical variables [7]. As foreshadowed in Refs. 8 and 9 it seems that such large 3D calculations are now quite practicable, and Symbolic Algol appears to be a suitable language in which to formulate them.

APPENDIX. The Trinity Difference Equations

The Symbolic Algol I difference equations are in virtual one-to-one correspondence with the partial differential equations of Section 2:

```
procedure INVOKE DIFFERENCE EQUATIONS;
begin
    CONTINUITY EQUATION: DT: = 2 × DELTA T; C1: = C2: = 1;
    Q: = 1 + I + 1 + (J + 1) × PI + (K + 1) × PI × PJ;
    NEW RHO: = RHO − DT × DIV(RHO × V);
    MOMENTUM EQUATION: DT: = 2 × DELTA T/(1 + NU/EPS);
      for C1: = 1, 2, 3 do
      AV[C1, Q]: = (RHO × V + DT × (−DIV2(P)
                   + NU × DELSQ(RHO × V)))/NEW RHO;
      A RHO[Q]: = NEW RHO;
    MAGNETIC EQUATION: DT: = 2 × DELTA T/(1 + ETA/EPS);
      for C1: = 1, 2, 3 do
      AB[C1, Q]: = B + DT × (CURL(CROSS(V, B))
                   + ETA × DELSQ(B));
    TEMPERATURE EQUATION:
      DT: = 2 × DELTA T/(1 + KAPPA/EPS); C1: = 1;
      ATEM[Q]: = TEM + DT × (−DIV(TEM × V) + KAPPA
                 × DELSQ(TEM) + (2 − GAMMA) × SAV(TEM)
                 × DIV(V) + (GAMMA − 1) × (ETA
                 × SQM(CURL(B))/SAV(RHO) + NU
                 × (SQM)(CURL(V)) + DIV(V) ↑ 2)));
end;                                                     (A1)
```

The adjustment to DT is a device which enables the Dufort–Frankel scheme to be used, without altering the form of the differential equation, and EPS $= \epsilon = \delta s^2/6\delta t$. Those operators which are not yet defined are fairly obvious (note that P is given by Eq. (26)):

```
real procedure DQ2;
    DQ2: = if C2 = 1 then 1 else if C2 = 2 then PI else PI × PJ;
real procedure DEL(F); real F; DEL: = (EP(F) − EM(F))/DS2;
real procedure DEL2(F); real F; DEL2: = (EP2(F) − EM2(F))/DS2;
real proredure DELSQ(F); real F; DELSQ: = (SAV(F) − F) × 6/DSSQ;
real procedure DIV(A); real A; DIV: = SIGMA(DEL(A));
real procedure DIV2(T); real T; DIV2: = SIGMA2(DEL2(T));
real procedure EM2(F); real F; begin Q: = Q − DQ2; EM2: = F;
                   Q: = Q + DQ2; end;
real procedure EP2(F); real F; begin Q: = Q + DQ2; EP2: = F;
                   Q: = Q − DQ2; end;
real procedure SAV(F); real F; SAV: = SIGMA2(SUM2(F))/6;
real procedure SQM(A); real A; SQM: = SIGMA(A × A);
real procedure SUM2(F); real F; SUM2: = EPS(F) + EM2(F);     (A2)
```

where

$$\text{DS2:} = 2 \times \text{DS}; \qquad \text{DSSQ:} = \text{DS} \times \text{DS}; \tag{A3}$$

The label '2' refers to the second index j, so that the correspondence is

$$\left.\begin{array}{ll} \text{DEL(F)} & \dfrac{\partial F}{\partial x_i} \\ \text{DEL2(F)} & \dfrac{\partial F}{\partial x_j} \\ \text{DIV2(T)} & \sum_j \dfrac{\partial T_{ij}}{\partial x_j} \\ \text{SAV(F)} & \sum_j (E_j + E_j^{-1}) F/6 \end{array}\right\} \tag{A4}$$

SAV is used to produce a space average over six surrounding mesh points, without disturbing the index i.

Finally, we note that C1 and C2 should be set to some legal value in the range 1–3 before evaluating the continuity and temperature equations, because they are left undefined by Algol when used in **for** loops. Conventionally the value 1 is chosen.

References

1. R. S. Peckover and K. V. Roberts, Symbolic Algol modules for the explicit solution of coupled partial differential equations, in preparation.
2. K. V. Roberts, Methods of computational physics, *in* "Proceedings of Culham Conference on Computational Physics," July 1969, Report CLM-CP (1969), paper A, Her Majesty's Stationary Office, London.
3. K. V. Roberts and J. P. Boris, Trinity programs for 3D magnetohydrodynamics, *in* "Proceedings of Culham Conference on Computational Physics," July 1969, Report CLM-CP (1969), paper 44, H.M.S.O., London.
4. M. Petravic, G. Kuo-Petravic, and K. V. Roberts, The automatic optimization of Symbolic Algol programs. I. General Principles, to be published.
5. G. Kuo-Petravic, M. Petravic, and K. V. Roberts, The Translation of Symbolic Algol I to Symbolic Algol II by the STAGE 2 macro-processor, Culham Laboratory Preprint CLM-P270, in preparation.
6. K. V. Roberts and D. E. Potter, Magnetohydrodynamic calculations, *Methods Comput. Phys.* **9** (1970), 339.
7. G. Kuo-Petravic, M. Petravic, and K. V. Roberts, The optimization of magnetohydrodynamic calculations in 3 dimensions, to be published.
8. J. P. Boris and K. V. Roberts, The optimization of particle calculations in 2 and 3 dimensions, *J. Comput. Phys.* **4** (1969), 552.
9. J. P. Boris, High-β stability in a three-dimensional diffuse pinch, *in* "1970 Sherwood Theoretical Conference," April 1970, Princeton University, Princeton, New Jersey, in preparation.