United Kingdom Atomic Energy Authority
RESEARCH GROUP

Preprint

AUTOMATIC OPTIMIZATION OF SYMBOLIC ALGOL PROGRAMS

I. GENERAL PRINCIPLES

M. PETRAVIC
G. KUO-PETRAVIC
K. V. ROBERTS

Culham Laboratory
Abingdon Berkshire

1971

AUTOMATIC OPTIMIZATION OF SYMBOLIC ALGOL PROGRAMS

I. GENERAL PRINCIPLES

M Petravic*, G Kuo-Petravic*
and
K V Roberts

ABSTRACT

The symbolic style of programming referred to as Symbolic Algol I [1] appears to have a number of advantages when applied to the solution of sets of nonlinear partial differential equations. Programs written in that style are clear, elegant and concise and their modular structure enables large parts of the programs to be used over and over again for many different problems. Such programs, however, tend to be slow because they involve a large number of nested procedure calls at execution time.

Finite difference methods in several dimensions require in general that a relatively small number of equations be solved a large number of times and much is gained if these nested procedure calls are executed only once. This is achieved by a generator or translator program, written in Algol, which processes input written in a related style named Symbolic Algol II. Usually only finite difference equations in very compact symbolic form are input, while output is completely explicit and can be in a number of computer languages. Of greatest interest are User or Assembler codes automatically produced in this way. They are competitive in speed with fully hand-optimized Fortran versions and are produced effortlessly and error-free. Their speed, on the other hand, enables a full set of magnetohydrodynamic equations in three space dimensions to be solved in a reasonable time on a 60x60x60 or equivalent mesh, using an IBM 360/91 computer.

* Department of Engineering Science, Oxford University

U.K.A.E.A. Research Group
Culham Laboratory
Abingdon
Berks.

June, 1971

1. INTRODUCTION

This paper describes how Algol 60 can be used as a powerful macro-processor which enables the symbolic expressions of classical vector analysis to generate efficient target code automatically by the use of controlled side effects. The target languages produced so far have been IBM 360 Assembler code, Fortran, Algol and ICL KDF9 Usercode, but it appears that any language might be generated in a similar way. The target code can be optimized by physical symmetry declarations; for example if $V_\theta = 0$ (no rotation) and $\partial f/\partial z = 0$ for all functions f (no z-dependence), then appropriate declarations can be used to suppress terms in which such quantities occur as products. The method can be used for generalised orthogonal curvilinear coordinates and an example will be given. Finally it would seem that the method might be extended without difficulty to other kinds of symbolic formalism.

A previous paper [1] showed how Algol could be used for the symbolic solution of problems in Computational Physics, especially those in which sets of partial differential equations are solved by finite difference methods using a discrete mesh. A vector equation such as

$$\frac{\partial \underline{B}}{\partial t} = \mathrm{Curl}\ (\underline{V} \times \underline{B}) + \eta \nabla^2 \underline{B} \qquad (1)$$

can be programmed symbolically in the closely similar form

```
AB[C1,Q]:=CURL(CROSS(V,B))+ETA x DELSQ(B);          (2)
```

a style of programming which has been termed [1] Symbolic Algol I. Here the left side of (2) is an array element representing the magnetic field B in the C1-direction, (C1 = 1,2 or 3) at the mesh point Q , while most of the identifiers on the right side are symbolic operators or functions which are closely analogous to their counterparts in Eq. (1) and are represented by real procedures. Details of the choice of coordinate system, the number of dimensions, the boundary conditions and the difference scheme are excluded from Eq. (2) and are dealt with at a lower level just as in the familiar symbolic notations of mathematical physics.

Symbolic Algol I (SA/I) enables complex problems to be coded in a concise form which is virtually system-independent and should be readily intelligible to physicists because it is close to the mathematical language which they normally use. By way of example, Table I shows the partial differential

equations which are used in the 3D magnetohydrodynamic TRINITY code [1], while Table II shows the same equations programmed in SA/I. The differences are fairly minor and are partly due to the restricted class of symbols currently available on computer input devices such as the teletype or card punch. One might perhaps compare the relation between vector analysis and SA/I to that between classical and quantum mechanics. In the notation introduced by Heisenberg and Dirac the same symbols (e.g. q, p, H) are used as in the classical Hamiltonian theory but they now receive a different interpretation as operators or q-numbers; nevertheless many of the formulae look just the same.

The advantages of symbolic notation are clear enough. The Algol procedure-operators are neat and concise and have the same formal properties as their mathematical counterparts, so that the manipulation of statements and the construction of new expressions are quick, intelligible and easy to check for errors. A typical example is the operator CURL, represented in a Cartesian coordinate system by the short procedure:

```
real procedure CURL(A); real A; CURL:=RP(DEL(RP(A)))-RM(DEL(RM(A)));    (3)
```

Here RP and RM are rotation operators which rotate the 1,2,3 - components of vectors or tensors in either the positive (RP) or negative (RM) directions rather like spin operators in quantum mechanics (Fig. 1), while DEL is a finite difference operator. These rotation operators are reciprocal to one another so that

$$RP(RM(A)) = RM(RP(A)) = A, \tag{4}$$

while the property

$$RP(RP(A)) = RM(A) \tag{5}$$

is also often used in 3 dimensions. The use of vector and tensor operators ensures that statements are independent of the coordinate system (covariant), the components being hidden and appearing only at execution time.

All these properties combined with modularity and portability of programs [2] make SA/I a powerful tool for the quick and error-free development of large and complex physics or engineering programs. However, SA/I executes quite

slowly because of the great number of nested procedure calls and is therefore not too useful for 2D and 3D production runs although this depends on how well the Algol 60 compiler has been written. Its main application to date lies in the testing of prototype programs on a coarse mesh over a few timesteps. In this way standard test results are obtained for comparison with future better-optimized and faster versions of the same program, written for example in ordinary Algol, Fortran or Assembler code [1].

The aim of the present paper is to carry the theory of Symbolic Algol one stage further. We shall show that by a further slight transformation of a vector expression such as Eq. (2) it can be made to generate the optimized program automatically. In this new style of programming, which is termed Symbolic Algol II (SA/II), Eq. (2) in fact becomes

```
EQUATE(B,SUM(B,MULT(DT,SUM(CURL(CROSS(V,B)),MULT(ETA,DELSQ(B))))));      (6)
```

and the statements of Table II are replaced by those of Table III. The reason for this transformation is to replace the arithmetic operators +, -, x, /, = which occupy a privileged position in high-level languages by their generalized counterparts SUM, DIFF, MULT, QUOT which are real procedures, and EQUATE which is a procedure. Once this has been done these procedures can, of course, be given any interpretation that we choose, and they can in particular be made to generate optimized code in any desired programming language by means of side effects as explained in §2.

The languages generated so far have been IBM 360 Assembler code, optimized Algol and Fortran, and ICL KDF9 Usercode which is similar to Reverse Polish and therefore has a theoretical as well as a practical interest. The transformations from Table I and Table II to Table III obey prescribed rules and we have in fact carried them out automatically [3] using the STAGE 2 macro processor [4], although they are not difficult to perform by hand.

An SA/II generator program looks very like the corresponding SA/I calculational program except that some of the auxiliary statements must also be changed from form (2) to form (6), so that for example CURL becomes

```
real procedure CURL(X); real (X); CURL:=DIFF(RP(DEL(RP(X))),RM(DEL(RM(X))));
                                                                        (7)
```

The purpose of (7) is however rather different from that of (3) because instead of actually calculating numerical values directly, the program now works out which programming instructions are needed to calculate these values and then generates the instructions, either printing them or punching them out on cards or placing them in a file for subsequent execution. Clearly there is a close correspondence between the two processes of calculation and code generation - for example in both SA/I and SA/II the storage locations are evaluated by manipulating symbolic operators - and so CURL, DEL and other operators appear virtually unaltered. To pursue the quantum-mechanical analogy further, the conversion from SA/I to SA/II might be compared to second quantization in that the c-number wave-functions become q-number field operators but the equations stay the same.

An SA/II generator program is not a compiler, and it is shorter, more flexible and easier to write than most compilers. We shall endeavour to explain the relation in §6. Since it is coded in Algol and has at least as many nested procedure calls as SA/I it may also not be as fast as some compilers. However, the point is that each statement in Table III is executed only once , instead of many millions of times as in a normal run. Code generation from Table III actually occupies about 10 seconds on the IBM 360/91, which is comparable with ordinary job overheads and much less than the duration of a typical production run which may last for several minutes or even hours. The only real requirement is that the generated Assembler code should be efficient, and it turns out in practice to be slightly more efficient than the corresponding Fortran written by a good programmer and compiled with the IBM Fortran H Option 2 Compiler.

The basic idea of SA/II is quite straightforward and will be explained in §2, namely that side effects of typed procedures can be used to generate code. Some complications arise in a practical program because it is desirable to make the generated code as efficient as possible. As we have already mentioned, it should be possible in a problem in which $V_\theta = 0$ or $\partial f/\partial z = 0$ to declare these symmetry conditions to the generator program and to have it automatically eliminate all terms which are then known to be identically zero. This can be achieved by making use of the fact that the real procedures in a statement such as (6) pass numerical values to one another. The simplest version of SA/II outlined in §2 treats these values as dummies, but they can be employed for a variety of purposes including optimization. A zero value is assigned to basic terms or products which are known to be identically zero in the physical problem, and this value is used to control the way in which the code is generated. Unnecessary brackets and signs are eliminated

in a similar fashion.

Code generation is carried out by a real procedure TRIPOP (triple operator), and operators such as SUM, DIFF, MULT, QUOT simply call TRIPOP to generate the appropriate output. Although TRIPOP is quite short its working is fairly complex, and to avoid burdening the present discussion it will be described in detail elsewhere [5]. A brief account is however given in §3.

Another complication arises from the finiteness of real machines, for example the nesting store or arithmetic stack of the KDF9 has about 13 levels available to the user, the IBM 360 has 4 floating-point registers, and so on. We have found it efficient to generate the code for an infinite computer and then to map it on to the real machine, and how this is done for the IBM 360 will also be described in ref. 5.

Since Algol is likely to be useful for other types of symbolic manipulation it may be appropriate, after some examples have been given, to list those features of the language that we have found to be important, and this is done in §8. It will however become evident almost from the beginning that recursion and call-by-name play a significant part, together with a number of other facilities which are not available in a language such as Fortran. A more practical point is that although Algol allows identifiers to have arbitrary length, some compilers (including that of the IBM 360) distinguish only the first 6 characters. To make the discussion of this paper clearer we have in some cases used long English-language identifiers, but in the published version of the program itself [6] the identifiers are restricted to 6 characters or less to avoid possible clashes.

2. CODE GENERATION

The generation of code by an expression such as

```
SUM(X,MULT(Y,Z))                                    (8)
```

is simple to explain. We first need a procedure PRINT(<string>) which will output an arbitrary string of characters taking into account any special requirements to leave gaps at the beginning and end of the line. In terms of this we define the algebraic symbols:

```
procedure PLUS; PRINT('+');
procedure STAR; PRINT('*');                         (9)
```

and the symbols to be used as identifiers in the generated code:

```
real procedure X; begin PRINT('X'); X:= 1; end;
real procedure Y; begin PRINT('Y'); Y:= 1; end;     (10)
real procedure Z; begin PRINT('Z'); Z:= 1; end;
```

Finally there are two arithmetic operators:

```
real procedure MULT(A,B); real(A,B);
    begin real CALL; CALL:=A; STAR; CALL:=B; SUM:=1; end;    (11)
```

and

```
real procedure SUM(A,B); real(A,B);
    begin real CALL; CALL:=A; PLUS; CALL:=B; SUM:=1; end;    (12)
```

Explanation

Exactly what happens is illustrated by the tree shown in fig. 2. When the statement 'CALL:=A' is encountered, the real procedure PRINT is transmitted by name via X into SUM, causing a symbol 'X' to be generated. The local variable CALL in SUM is set equal to 1 but this value is disregarded. A '+' symbol is next produced. Finally, MULT is entered by 'CALL:=B' which generates the symbols 'Y', '*', 'Z' in the same way so that we get

```
X + Y * Z                                           (13)
```

as required.

Several points may be noticed:

(a) Typed procedures are used to generate the code because Algol does not allow ordinary procedures to be transmitted via nested argument lists in this way.

(b) The actual generation takes place by means of side effects; the statement

CALL:= A;

causes the string of symbols associated with the formal parameter A to be constructed, however complex an expression A may represent.

(c) A precise order is forced on these side effects, independently of the order in which the Algol compiler writer may choose to evaluate the terms in an arithmetic expression such as P+Q or PxQ.

(d) It is immaterial whether real or integer procedures are used, but we have chosen real procedures for compatibility with SA/I.

(e) At this stage the values associated with the real procedures have no significance; only the side effects are important. In §3 we shall however indicate how a suitably-chosen assignment of values can be used to remove expressions which are known to be identically zero.

Generation of Reverse Polish

The reader might be excused for wondering what exactly has been achieved so far; starting from the algebraic expression (13) we have converted it automatically or by hand into the SA/II form (8), and then proved that this is capable of reproducing the original expression. The real advantages are, first, that the generated code can represent a considerable expansion of the original which may be in symbolic vector or tensor form, and second, that by making small changes in the basic procedures such as PLUS, STAR, X, Y, Z, MULT, SUM we can generate the code in a lower-level and therefore more efficient language.

To illustrate this second point we show how to convert (8) into the Reverse Polish form

X, Y, Z, *, +, (14)

which is in 1-1 correspondence with the Assembler Language (or 'Usercode') of the ICL KDF9 on which Symbolic Algol was first developed. To do this we simply modify the five procedures (9) and (10) in order to add an extra comma, e.g.

```
procedure PLUS; PRINT('+,');                                    (15)
```

and then reverse the order of the second and third statements in the arithmetic operators (11) and (12), e.g. :

```
CALL:= A; CALL:= B; STAR; MULT:= 1;                             (16)
```

which then enables (8) to generate (14).

Brackets

The question of brackets has not so far been taken into account in generating algebraic expressions such as (13). Thus at present

```
X x (Y+Z)                                                       (17)
```

would be converted by

```
MULT(X,SUM(Y,Z))                                                (18)
```

into

```
X * Y+Z                                                         (19)
```

which is wrong, although the corresponding Reverse Polish string is still correct:

```
X,Y,Z,+,*,                                                      (20)
```

The simplest way of correcting this is to enclose the output from each arithmetic operator in a pair of brackets by means of two procedures OPENB, CLOSEB, so that for example (12) becomes

```
real procedure SUM(A,B); real(A,B);
begin real CALL; OPENB; CALL:= A; PLUS; CALL:= B; SUM:= 1; CLOSEB;end;   (21)
```

Then expressions (8) and (18) generate the correct output

```
(X + (Y*Z))                                                     (22)
```

and

```
(X * (Y+Z))                                                     (23)
```

respectively.

A complicated expression will now produce a large number of unnecessary pairs of brackets but although these make the expression difficult for a

human to understand they should not trouble a Fortran or Algol compiler; in fact they might well shorten the compilation time since questions of operator precedence need no longer be resolved. However in order to make the generated code more elegant and intelligible we have implemented a method for removing the unnecessary bracket pairs and this will be described in ref. 5.

Another difficulty which occurs with the KDF9 is that the nesting store or arithmetic stack has finite depth (in practice about 13), so that a sufficiently long string of identifiers unrelieved by operators could cause it to overflow. To monitor this situation (which has not yet occurred in the problems that we have treated) one can introduce a level counter and print out a warning to the user when overflow is detected. In most cases he can then permute the order in the SA/II expression to produce a better pattern. For example

$$\text{SUM(MULT(Y,Z),X)} \tag{24}$$

instead of (8) leads to

$$\text{Y,Z,*,X,+,} \tag{25}$$

which requires a smaller stack. Overflow is more likely to occur with the IBM 360 which has only 4 floating-point registers, and here we perform automatic dumping and restoring of registers when necessary as explained in ref. 5.

3. PROCESSING THE EQUATIONS

The central procedure of the SA/II generator program is EQUATE, which for Algol output is

```
procedure EQUATE(X,Y); real X,Y;
   begin real CALL;
        NEXTLINE; PRINT:= false;
        if X = 0 then go to EXIT;
        PRINT:= true; SIGN:= 0;
        CALL:= X; ASSIGN;
        if Y = 0 then TEXT(1,'0');
        SEMICOLON;
      EXIT:
   end;
```

(26)

A similar version is used for any other target code but the explanation is simpler for this particular example and may give some idea of the power of the symbolic method. ASSIGN generates the assignment symbol,'=.' for the IBM 360.

Suppose that (26) is called by the statement

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

(27)

as in Table III but without the resistive diffusion term. Because this is a vector equation it will be called 3 times with C1 = 1,2,3 and we shall suppose that B_z has been declared identically zero. The tree of real procedure calls is indicated in Fig. 3.

The auxiliary call to NEXTLINE does any editing that is required, e.g. ruling a line across the output page, and printing is then switched off. The left side X of the expression, in this case B_x, B_y or B_z, is next tested to see if it is identically zero, which as explained in §1 will be represented by a zero value. If so, there is no point in generating anything and the right side is skipped. (C1 = 3). Note that since X is a real procedure the statement if X = 0 then ..... involves further operations at a lower level*.

---

* Since X is real it is perhaps not good practice to test whether or not it it is exactly zero but we have found no difficulty on the ICL KDF9 or IBM 360. The procedure values are always set to integers or sums or differences of small integers and it is hard to see why trouble should arise with any computer. If it does, one can simply test for $X < \epsilon$ where $0 < \epsilon \ll 1$.

Assuming that the component is not identically zero we switch printing on again, make sure that no + sign will be printed, and reference X once more, subsequently calling the procedure ASSIGN. This series of actions will generate some expanded form such as

$$B1(/Q/).= \tag{28}$$

depending on the hardware representation and on the way in which array components are being referenced. Note that the numerical value in (28) is that of the formal parameter V.

The whole of the testing and generation of the right side is now contained in the deceptively simple statement

$$\underline{\text{if}}\ Y = 0\ \underline{\text{then}}\ \text{TEXT}(1,'0'); \tag{29}$$

Some possible cases for the right side Y are:

a. Constant. Since printing is now switched on the numerical representation of the constant is generated.

b. Non-subscripted variable. The character string corresponding to this variable is generated.

c. Array variable. If this variable has been declared to be identically zero it will not be printed and the value 0 will be returned, causing the second part of the if statement to print '0'. If it is not identically zero it will print its own character representation, including any necessary vector or tensor components.

d. Arithmetic expression. The outermost arithmetic operator calls TRIPOP which switches off the printing and tests the whole of the expression, TRIPOP being called again recursively by each of the internal arithmetic operators, including those in CURL and CROSS. If it is finally determined to be identically zero then '0' is printed. Otherwise this outermost TRIPOP initiates a second scan which prints all those sub-expressions which were found to be not identically zero the first time they were tested.

As an example, Table IV shows Algol 60 target code generated for the IBM 360 for Maxwell's two equations

$$\frac{\partial \underline{E}}{\partial t} = \text{Curl}\ \underline{H}, \quad \frac{\partial \underline{H}}{\partial t} = -\ \text{Curl}\ \underline{E}$$

in orthogonal curvilinear coordinates using a mesh of size (14x10x42). The source statements were:

```
for C1 = 1,2,3 do
   EQUATE(EFIELD,SUM(EFIELD,MULT(DCT(8),CURL(HFIELD))));
for C1 = 1,2,3 do
   EQUATE(HFIELD,DIFF(HFIELD,MULT(DCT(7),CURL(EFIELD))));          (30)
```

In this case no symmetry conditions were imposed. DCT7 and DCT8 are timestep factors which will usually be the same.

4. PHYSICAL CONSTANTS AND VARIABLES

Provision is made for handling non-subscripted variables which may be constants or functions only of the time, and scalar, vector and tensor functions. A typical 'declaration' specified by the user is for example

```
real procedure B; B:= VECTOR(5,1,'B',1,1,0);                    (31)
```

This calls the real procedure VECTOR which in the Algol 60 output version reads

```
real procedure VECTOR(ORDINAL NUMBER,LENGTH,S,V1,V2,V3);
    integer ORDINAL NUMBER,LENGTH,V1,V2,V3; string S;
    begin
        VECTOR:= if C1 = 1 then V1 else if C1 = 2 then V2 else V3;
        if not PRINT then go to EXIT;
        SIGN IT; TEXT(LENGTH,S);
        COMPONENT; SHIFT;
     EXIT:
   end;                                                         (32)
```

The parameter ORDINAL NUMBER is not being used here; it controls the actual storage region used by B which is important in some Assembler Language versions and is retained for consistency. LENGTH gives the length of the identifier string 'B' which will be used in target statements, instructions and comments, in this case 1. The last 3 parameters V1, V2, V3 define whether or not the x, y, z components are identically zero. In this case the user has specified V3 = 0 so that B_z is taken to be identically zero. Because VECTOR returns the value 0 whenever C1 = 3 the z-component of the magnetic equation will be suppressed altogether (§8), and all terms in which B_z occurs as a product will be suppressed on the right side of any other equation. Otherwise when PRINT = true the procedure VECTOR will proceed as follows:

SIGN IT	Examine the value of a global variable SIGN and output either a preliminary '+', '-', or no sign.
TEXT(LENGTH,S)	Output the string S of length LENGTH, in our example 'B'.
COMPONENT	Output the current value of the component C1, e.g. '2'.
SHIFT	Output '[' or '(/' followed by the signed numerical value of the current displacement from the local mesh origin, and finally a closing bracket, ']' or '/)'.

Typical output for Algol on the IBM 360 is

$$+B2(/Q+73/) \tag{33}$$

and similarly for Fortran except for the absence of the slashes. Several other versions have been developed; for example the displacements can be handled symbolically so that the code does not have to be regenerated when the mesh size is changed or one might generate 'BY' instead of 'B2' for clarity.

The operators RP, RM act on the global variable C1 as determined by algebraic and analytic operators such as DOT, CROSS, DIV, CURL and therefore enable the correct component label to be calculated. Similarly, vector translation operators EP, EM which are called whenever a space derivative occurs are used to calculate the position on the lattice relative to the central point Q of the local mesh 'molecule'. In eq. (33) this appears as a numerical displacement of 73 words within the region of core store occupied by the array B2.

A scalar is handled in a precisely similar way by the real procedure SCALAR which requires only a single value 0 or 1 and does not need to call COMPONENT, and correspondingly for constants on the one hand, and tensors on the other.

Core Storage Layout

We have already mentioned that the generated code can be in any chosen computer language. Whatever the language, however, we obviously have to be able to print some form of address at which the value of a particular variable defined on a specific mesh point is stored. The addressing can be completely symbolic, e.g. in Algol via an array

$$B[C1,Q + DX - DY], \tag{34}$$

it can be coded numerically as in KDF9 Usercode

$$V34P6M15 \tag{35}$$

(location 34 of variable block 6 modified by register 15), or it can be partly numerical and partly symbolic as in IBM 360 assembly code.

$$\left.\begin{array}{ll} 0052(,DISPQ) & BY(I,J-1,K) \\ 0312(PLUSDZ,DISPQ) & BZ(I,J,K+1) \end{array}\right\} \tag{36}$$

Here the displacement in bytes defines both the variable and the position in the xy-plane relative to the centre of the molecule, DISPQ is a general

register which contains the current center, while PLUSDZ is a register which shifts one mesh unit in the z-direction. The comments on the right side give the Fortran notation. Several other choices are possible. In any case the address will depend on the variable in question, on the coordinate direction on to which a vector variable is being projected, and on the mesh point at which the expression should be evaluated. These 3 quantities are defined in all the Symbolic Algol programs [1] by the variable name and by the global integer variables C1 and Q.

Because the generator program does not evaluate the physical expressions itself it always has to leave the central point of the difference scheme undetermined and refer to it as Q, or else to point to the register in which the current value of Q will be kept during the subsequent execution run. These registers can be referred to directly (M15) or symbolically (DISPQ). The variable name on the other hand is known at the time of the generation run and so the numerical value of C1 (or, if the variable name is coded numerically, some arithmetic combination of that code number and the value of C1) can be output. Table V shows an example of IBM 360 assembly code produced in this way.

Although in Algol or Fortran we could represent a scalar variable by an array with as many subscripts as there are dimensions n, and a vector variable by an (n+1)-dimensional array, this is often not desirable. The use of multi-dimensional arrays requires more registers and address calculations and slows down execution. 1-dimensional arrays are therefore to be preferred for the representation of physical variables when the execution count is high.

Decisions must thus be made about whether to have separate arrays for different vector components of the same variable or, if not, in which order they are to be stored in a single array. Even for scalar variables it has to be decided in which way a 3-dimensional set of values is going to be mapped into a 1-dimensional array. These strategic decisions cannot all be made automatically and the best solution will depend on particular circumstances like the type of the mesh, the symmetries of the problem, the order in which the mesh is scanned, the total number of mesh points, the size and type of core and backing storage available and so on. In practical terms this means that even when the output language has been selected, small changes to the generator program will be necessary from problem to problem. In most cases, however, these changes will be confined to one procedure (SHIFT), and will

consist in replacing one simple function of the component C1, the lattice displacement vector (K1, K2, K3), and the core storage 'distances' (DELTA1, DELTA2, DELTA3) between elements one lattice spacing apart by another simple function of the same integer variables. Since any SA/II program will in this way be mesh dependent, we give a brief description of the mesh that the TRINITY program assumes [1].

Finite Difference Mesh used for TRINITY

The mesh is assumed to be Cartesian, of up to 3 dimensions, and equi-spaced. The numbers of mesh points in the x, y and z directions are denoted by PI, PJ and PK respectively, so that the total number of mesh points is SIZE = PI x PJ x PK. This includes six guard planes introduced to enable the same difference expressions to be used on the physical boundaries of the volume as inside it. The mesh points are counted in the x-direction, starting along the intersection of the first y- and the first z-plane numbered by integers I, J and K. The numerical relation is

$$Q = 1 + I+1 +(J+1)xPI + (K+1)xPIxPJ \qquad (37)$$

For output in Algol or Fortran we choose separate arrays for each vector component in which the values are packed with increasing number Q. All the arrays are then of the same size equal to PI x PJ x PK and are functions only of one index Q.

This way of mapping the 3-dimensional mesh on to 1-dimensional arrays implies that two mesh points which are adjacent in the y-direction correspond to array elements PI storage locations apart, and that if the points are adjacent in the z-direction the corresponding elements will be PI x PJ locations apart. The single mesh index changes by DELTA3=PI x PJ if the shift is in the z-direction. It is the vector displacement operators EP and EM that cause, on a single application, a shift by one mesh point in the direction determined by the current value of C1. The number of shifts in the x, y and z directions is denoted in the program by the global integers K1, K2 and K3 from which, knowing PI and PJ, the corresponding change in Q can be calculated.

It is sometimes desirable to interleave the variable arrays so that the variable values at one mesh point Q are stored sequentially in the memory, e.g. in the order

$$RHO,V1,V2,V3,B1,B2,B3, TEM \qquad (38)$$

followed by the same sequence for the next mesh point (Q+1) and so on. Adjacent values of the same variable are then(say) 32 bytes apart. This enables physical data planes to be transferred readily to and from the backing store as a single block, and can be done either in Assembler code or, in Fortran, by the use of EQUIVALENCE statements. In all cases the SA/II program can readily be adapted to calculate the correct word or byte position in the store.

A similar situation exists when the core store is too small to hold the complete set of physical data. The largest size of mesh on which TRINITY has been run has 60x80x48 points and requires two IBM 2301 drums to accommodate the 7 Mbytes of data. All the calculation then takes place within 3 sectors of a rotating quadruple buffer, the fourth sector being used to transfer data to and from the core store in parallel with the calculation. The SHIFT procedure has been adapted so that it always refers to the correct areas of buffer storage.

5. ORTHOGONAL CURVILINEAR COORDINATES

Symbolic vector algebra and analysis on a Cartesian lattice in SA/I have been described elsewhere [1] and few changes are required for SA/II. It may however be of interest to demonstrate how the method has been extended to generalized orthogonal curvilinear coordinates, with spherical polar coordinates as a special case.

The generalized definitions for divergence and curl can conveniently be written as

$$\text{div } \underline{A} = \frac{1}{h_1 h_2 h_3} \sum_i \frac{\partial}{\partial q_i} \left(h_i^+ h_i^- A_i \right) \tag{39}$$

$$\text{Curl } \underline{A} = \sum_i \frac{\underline{a}_i}{h_i^+ h_i^-} \left(\frac{\partial (h_i^- A_i^-)}{\partial q_i^+} - \frac{\partial (h_i^+ A_i^+)}{\partial q_i^-} \right) \tag{40}$$

where $(\underline{a}_1, \underline{a}_2, \underline{a}_3)$ are unit vectors, $\underline{h} = (h_1, h_2, h_3)$ are scale factors, and +, - denote positive and negative cyclic rotation respectively.

These are translated into SA/II as

```
real procedure DIV(A); real A; DIV:= MULT(DOT(DEL(MULT(A,HPHM)),R2DQ),RH1H2H3);
```
(41)

```
real procedure CURL(A); real A;
 CURL:= MULT(DIFF(RP(MULT(DEL(RP(MULT(A,H,))),R2DQ)),
                                RM(MULT(DEL(RM(MULT(A,H))),R2DQ))),RHPHM);
```
(42)

where

```
real procedure DEL(X); real X; DEL:= DIFF(EP(X),EP(X));
```
(43)

This leaves to be defined the real procedures

H	$\rightarrow (h_1, h_2, h_3)$	(vector function)
RH1H2H3	$\rightarrow 1/(h_1, h_2, h_3)$	(scalar function)
RHPHM	$\rightarrow 1/(h^+ h^-)$	(vector function)
R2DQ	$\rightarrow 1/(2.DQ)$	(vector)

(44)

The mnemonic 'R' means reciprocal and signifies that division is avoided in the interests of efficiency, and for similar reasons $h_1 h_2 h_3$ and $h^+ h^-$ are defined separately instead of being constructed from $\underline{h}$.

When using the leapfrog scheme [1] we find it useful to store the scale factors and quantities that are constructed from them on a subsidiary mesh centered on the point Q, which may either contain 7 points if only one displacement $\pm \Delta q_i$ occurs at a time (Fig. 4) or 27 if they occur together. Then for example RHPHM becomes a real procedure which generates the code

$$\text{RHPHM1[I], RHPHM2[I], RHPHM3[I]} \tag{45}$$

with I in the range (-3,3) as in Table IV.

The generated target module is therefore still independent of the coordinate system although it does depend on the symmetry. To run the target program we must reload the variables on the subsidiary mesh whenever they alter. In spherical polars the scale factors are

$$h_r = 1,\ h_\theta = r,\ h_\varphi = r\text{Sin}\theta \tag{46}$$

so that we can minimize the amount of recalculation by assigning the variables in order of $q_1 = \varphi$, $q_2 = \theta$, $q_3 = r$ with the innermost scan over q_1, although this is not necessarily the best choice on other grounds.

Some further improvements can be made if the coordinate system is specified at generation time; for example h_r might be automatically suppressed since it is known to be unity.

6. THE PROBLEM PROGRAM AND THE GENERATOR PROGRAM

By no means all of the problem program need be constructed automatically. For a typical computational physics program involving the solution of sets of coupled partial differential equations it is only necessary that those sections which have a high execution count should be coded in the most efficient form. Execution counts vary widely, e.g. with a 60x60x60 leapfrog mesh run for 500 timesteps the coding for an internal point of the mesh will be executed 5×10^7 times while some of the initialization statements will be executed only once.

One therefore starts by developing a strategy for the problem which takes into account its modular structure, the layout of core and backing storage, the communication between program modules, the choice of language and programming style for each module and so on. Those parts of the program whose execution count is low should be optimized according to considerations other than those of CPU efficiency; for example they should be made intelligible or easy to write and to debug.

In the final production program some of the modules may be in SA/I, some in conventional Algol 60, some in Assembler code which has been automatically generated from SA/II, and others in hand-coded Assembler code, Fortran or any other appropriate choice.

So far as possible the production code is to be developed from a more symbolic prototype code by the simple process of successively 'unplugging' some of the modules and replacing them by faster but equivalent versions, and this must be taken into account in designing the structure and the linkages. Portability should also be planned right from the start, so that low-level modules designed for one computer system can be rapidly replaced by those written for another.

The Generator Program

An Assembler code replacement for an SA/I module of the prototype program is constructed by incorporating the corresponding SA/II module into the generator program which is then run (Fig. 5). We visualize a series of related physical problems, run either on one computer at a single laboratory or on a range of different computers at several laboratories. If the generator program has been constructed properly only a small part of the work has to be repeated for each new situation. This saves time and makes it easier for one person to understand a range of programs, since they share a family resemblance like members

of related biological species.

The generator program ought to be highly modular since there are a number of requirements which may well need to be changed independently. Examples are:

a. Computer System on which the code is generated.
   i. Algol character representation
   ii. System output procedures
b. Computer System for which the code is generated.
   i. Character representation.
   ii. Language.
   iii. System facilities.
c. Number of dimensions and coordinate system.
d. Mesh and storage layout.
e. Symbols used for variables and constants.
f. Numerical methods and difference schemes.
g. Physical equations.
h. Boundary conditions.
i. Physical coefficients (e.g. thermal conductivity).

Fig. 6 shows the relation between the modules which have been developed so far while Table V provides a list. In §7 we shall go through this list briefly, indicating what the various modules do. A detailed description together with a program listing and test runs will be published elsewhere [5].

Relation to Compilers and Macro-Generators

In mathematics or theoretical physics it is always possible for an author to devise a new notation or extension to the accepted 'language' and having defined it for the reader, to use it throughout a paper or a course of research. This flexibility has been largely unavailable in computing science, where it has been customary to use rather standardized and limited languages such as Fortran, Algol or PL/I which are constructed either by manufacturers or by international committees, and translated by compilers which are expensive and time-consuming to write and to maintain. The only degree of freedom left to the individual has been the ability to define sets of library subroutines or procedures which in effect become additions to the standard notation. There is no limit on the extensions which can be achieved in this way but programs tend to run slowly if they make considerable use of procedure calls, as in SA/I.

Macro-processors such as STAGE2 [4] enable any string of symbols to be given a meaning. The user is free to define sets of macros which convert any string into any other string and eventually, into the code of some high or low-level language whose efficiency depends solely on his own ingenuity. Thus the full flexibility of mathematics is achieved provided that the character set is wide enough.

SA/II appears to lie somewhere in between. The formal structure of the input string is constrained by the syntax of Algol so that there is usually an excessive number of brackets as in Table III, but the manipulations that can be carried out are quite general and it is remarkably easy for the individual user to make alterations or additions to the 'language' by changing the basic procedures of the generator program. An SA/II generator program is also quite short; usually only a few hundred Algol statements. Thus we have effectively at our disposal an ultra high-level 'language' compiler which is difference scheme and problem dependent, but is also easily changed by any user.

7. STRUCTURE OF THE GENERATOR PROGRAM

The current version divides logically into 6 main modules and 15 submodules as shown in Table VI.

I. BASIC OUTPUT. In transferring the generator program to a new computer system the first task is to rewrite this module, which forms a link to the standard Algol output procedures of the computer on which the generator is being run. (This need not of course be the same as the computer for which the optimized code is being produced). The ICL KDF9 version occupies about 25 cards and it can usually be rewritten for another system in a few hours. The module defines the output channel and sets up standard formats, and contains procedures which enable the output to be manipulated in a straight-forward system-independent way; e.g.

BLANKS(N)	Output N blank spaces
LINE	Start a new line
OUTNUM(L,F,X)	Output the value of an arithmetic expression X in format F, length L
TEXT(L,S)	Output a string S of length L*.

The complete set is enough to generate code and comments in any language.

II. CHARACTERS. This also contains no submodules. It is made up of a dozen or so simple statement procedures which enable one to refer to symbols like (, + ; = etc. by symbolic names : OPENB, COMMA, PLUS, SEMICOLON, EQUALS. Although this somewhat slows down code generation it makes the program much easier to read and quicker to write, particularly in view of the awkward way in which string quotes are often represented in Algol. Examples are:

```
procedure EQUALS; TEXT(1,'=');
procedure CLOSEB; TEXT(1,')');
procedure OPENSB; TEXT(1,'[')
procedure MINUS; TEXT(1,'-');                    (47)
```

The first argument gives the length of the string, in this case 1.

---

* The simpler procedure PRINT discussed in §2 did not contain the parameter L, which helps the output procedures to organize the layout of the line.

III. MATHEMATICS A. The first mathematics module comprises the submodules

ARITHMETIC

ALGEBRA

The former of these contains a set of procedures

SUM(X,Y)	MULT21(X,Y)
DIFF(X,Y)	QUOT(X,Y)
MULT(X,Y)	SUM3(X,Y,Z)

which are all dealt with by a single fairly complex procedure TRIPOP (triple operator) to be explained elsewhere [5], e.g.

```
real procedure SUM3(X,Y,Z); real X,Y,Z; SUM3:= TRIPOP(5,X,Y,Z,1,1);        (48)
```

The first argument of TRIPOP is an operation code, the second to fourth are the operands, and the last two specify the first or second indexes (in the case of a second-rank tensor). Procedure SUM3 is convenient when handling 3D scalar products and divergences while MULT21 is used for handling tensor contractions. The others deal with the arithmetic operations +, -, x, /.

The ALGEBRA submodule contains the rotation operators RP and RM and the vector-algebraic operators DOT and CROSS.

IV. MATHEMATICS B. In the simplest version the second mathematics module contains the two submodules

1. CARTESIAN ANALYSIS LEAPFROG
2. SPACE AND TIME SCALES

As its name implies, the first of these submodules depends on the mesh geometry and on the difference schemes used (although not on the physical problem or on the computer system). The procedures EP, EM, DEL, GRAD, DIV, CURL, SAV, DELSQ are fairly direct translations of the SA/I versions already published [1]. The other submodule deals with the constants Δt, $2\Delta S$ and $(\Delta S)^2$ and can be also readily extended.

V. OUTPUT ORGANIZATION. This module depends on the output language. It may contain up to 6 submodules of which number 5 is omitted in the Algol target version:

(V.1) TRANSLATION LOGIC. Contains most of the logic needed to generate the output code and to eliminate expressions which are identically zero, as well as unnecessary brackets. It therefore deserves a more detailed description which is given in ref. 5.

7. STRUCTURE OF THE GENERATOR PROGRAM

The current version divides logically into 6 main modules and 15 submodules as shown in Table VI.

I. BASIC OUTPUT. In transferring the generator program to a new computer system the first task is to rewrite this module, which forms a link to the standard Algol output procedures of the computer on which the generator is being run. (This need not of course be the same as the computer for which the optimized code is being produced). The ICL KDF9 version occupies about 25 cards and it can usually be rewritten for another system in a few hours. The module defines the output channel and sets up standard formats, and contains procedures which enable the output to be manipulated in a straight-forward system-independent way; e.g.

BLANKS(N)	Output N blank spaces
LINE	Start a new line
OUTNUM(L,F,X)	Output the value of an arithmetic expression X in format F, length L
TEXT(L,S)	Output a string S of length L*.

The complete set is enough to generate code and comments in any language.

II. CHARACTERS. This also contains no submodules. It is made up of a dozen or so simple statement procedures which enable one to refer to symbols like (, + ; = etc. by symbolic names : OPENB, COMMA, PLUS, SEMICOLON, EQUALS. Although this somewhat slows down code generation it makes the program much easier to read and quicker to write, particularly in view of the awkward way in which string quotes are often represented in Algol. Examples are:

```
procedure EQUALS; TEXT(1,'=');
procedure CLOSEB; TEXT(1,')');
procedure OPENSB; TEXT(1,'[')
procedure MINUS; TEXT(1,'-');                    (47)
```

The first argument gives the length of the string, in this case 1.

---

* The simpler procedure PRINT discussed in §2 did not contain the parameter L, which helps the output procedures to organize the layout of the line.

III. MATHEMATICS A. The first mathematics module comprises the submodules

ARITHMETIC

ALGEBRA

The former of these contains a set of procedures

SUM(X,Y)	MULT21(X,Y)
DIFF(X,Y)	QUOT(X,Y)
MULT(X,Y)	SUM3(X,Y,Z)

which are all dealt with by a single fairly complex procedure TRIPOP (triple operator) to be explained elsewhere [5], e.g.

```
real procedure SUM3(X,Y,Z); real X,Y,Z; SUM3:= TRIPOP(5,X,Y,Z,1,1);        (48)
```

The first argument of TRIPOP is an operation code, the second to fourth are the operands, and the last two specify the first or second indexes (in the case of a second-rank tensor). Procedure SUM3 is convenient when handling 3D scalar products and divergences while MULT21 is used for handling tensor contractions. The others deal with the arithmetic operations +, -, x, /.

The ALGEBRA submodule contains the rotation operators RP and RM and the vector-algebraic operators DOT and CROSS.

IV. MATHEMATICS B. In the simplest version the second mathematics module contains the two submodules

1. CARTESIAN ANALYSIS LEAPFROG
2. SPACE AND TIME SCALES

As its name implies, the first of these submodules depends on the mesh geometry and on the difference schemes used (although not on the physical problem or on the computer system). The procedures EP, EM, DEL, GRAD, DIV, CURL, SAV, DELSQ are fairly direct translations of the SA/I versions already published [1]. The other submodule deals with the constants Δt, $2\Delta S$ and $(\Delta S)^2$ and can be also readily extended.

V. OUTPUT ORGANIZATION. This module depends on the output language. It may contain up to 6 submodules of which number 5 is omitted in the Algol target version:

(V.1) TRANSLATION LOGIC. Contains most of the logic needed to generate the output code and to eliminate expressions which are identically zero, as well as unnecessary brackets. It therefore deserves a more detailed description which is given in ref. 5.

(V.2). SIGNS, SPACES, NUMBERS AND FUNCTIONS. A number of standard utilities are provided here, some of which depend on the output language or format. For example a line overflow in Fortran requires that a continuation symbol should be punched in column 6; overflow in assembly language is handled in a different way while at the end of an Algol statement a semi-colon is required. Signs, integer and real numbers and elementary mathematical functions are also provided for. The most important procedure is EQUATE which has been discussed in §3.

(V.3). VARIABLE CLASSES. Contains procedures which deal with constants, scalars, vectors and tensors of which (32) is an example.

(V.4). COMPONENTS AND DERIVATIVES. Contains the procedures COMPONENT and SHIFT which generate the code for referencing storage locations, including any subsidiary calculations that are needed.

(V.5). COMMANDS AND REGISTER CHECKS. (IBM 360 Assembler code only). Contains procedures REGISTER REGISTER, REGISTER STORAGE which issue IBM 360 instruction mnemonics, and STORE IF OVERFLOW which determines whether or not the required register is already in use, if so copying it into a reserved location in core store.

(V.6). INITIALIZATION. Contains a procedure START which initializes the variable of the generator program.

VI. PROBLEM DEFINITION. This module is provided by the user and consists of three submodules which have been kept as simple and as close to the physics as possible:

1. PHYSICAL CONSTANTS AND VARIABLES
2. CONTROL STATEMENTS
3. SOURCE STATEMENTS

Table III gives the source statements for TRINITY while (31) is an example of a variable declaration. Typical initialization statements are

$$\text{NDIM:= 3; PI:= PJ:= PK:= 8;} \tag{49}$$

(use an 8 x 8 x 8 mesh in 3 dimensions).

8. CONCLUDING REMARKS

A satisfactory solution to the slowness of programs written in Symbolic Algol I has been found. Using a style known as Symbolic Algol II it has been possible to translate finite difference equations automatically into fully explicit codes in a number of target languages. The best of these codes are fully competitive in speed with hand-optimized Fortran and are fast enough to make the solution of time-dependent magnetohydrodynamic equations in three space dimensions a feasible proposition. In a recent exercise, a 3D plasma code in rotating spherical coordinates was designed and written in about four days using the SA/II method. The same translator program can be used for other systems of fluid equations and also for problems in two dimensions. Although the advantages of the method are smaller when applied to simpler problems, repetition of effort can still be avoided. More importantly, the use of well-tested procedures reduces programming errors, in particular those of a numerical nature which are often impossible to detect except experimentally through a comparison with another calculation. Though at first sight trivial this may prove to be one of the important attractions of the method.

The underlying principle is that instead of writing a problem program by hand, one constructs a generator problem which writes it automatically. Because this generator program is built up from prefabricated modules and is also highly symbolic it can be developed and altered very quickly.

In essence we are using Algol as a powerful macro-generator which is capable of substituting one expression into another as well as performing many subsidiary calculations. Because a value is associated with each substitution, extra information can be carried along which allows some optimization to be done. A further extension might be to relate this value to a generalized variable type (e.g. logical, integer, real, complex, quaternion, matrix or whatever), so that any necessary conversions can be carried out and the basic operators SUM, DIFF, MULT etc. can be interpreted in the appropriate way in each case. This is close to the procedure which is followed in mathematics which allows operators such as +, -, x to be freely generalized to new classes of object. Other interesting possibilities are to apply the SA/II technique to other kinds of program such as operating systems and compilers, and to other types of computer such as the CDC STAR.

The features of Algol 60 that appear to be necessary or useful for this kind of work are:

a. Call by name. Needed for the symbolic substitution of one expression into another.

b. Parameterless typed procedures. Just as in mathematics, a function need have no explicitly-indicated arguments. (Fortran does not allow this).

c. English-language identifiers of any length, with blanks ignored. Can be used to make programs more intelligible and to avoid bulky comments.

d. Elimination of the unnecessary word 'CALL'.

e. Ability to have several statements on one line. Both d. and e. make programs more concise and attractive.

f. No overhead on procedure declarations. Often these declarations are only one card long, and one line in the compiler listing, instead of several pages as in Fortran.

g. Block structure for variable scopes. Global variables can be passed into a procedure implicitly without the need for a bulky COMMON deck or argument lists.

h. Recursion. Typed procedures can be substituted into one another without restriction as required by the mathematical physics.

i. Side effects. Available also in other languages, but mentioned here as being crucial to the whole method.

9. ACKNOWLEGEMENTS

The early planning of SA/II was carried out in collaboration with Dr J P Boris. We should like to thank Dr F Hertweck and his colleagues at the Institut für Plasmaphysik, Garching, Federal Republic of Germany, for making available to us the excellent facilities of the IBM 360/91 Computing Centre at the Institute. We should also like to thank Mr R S Peckover for many discussions on Symbolic Algol, and Dr N K Winsor for providing an improved compiler.

REFERENCES

1. K.V. ROBERTS AND J.P. BORIS, 'The Solution of Partial Differential Equations using a Symbolic Style of Algol', Journ. Comp. Phys., in the press.

2. K.V. ROBERTS AND R.S. PECKOVER, Symbolic Programming for Plasma Physicists, paper presented at the 4th Conference on Numerical Simulation of Plasmas, U.S. Naval Research Laboratory, Washington D.C., November 1970, to be published. Culham preprint CLM-P 257.

3. 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', to be published.

4. W.M. WAITE, 'The Mobile Programming System STAGE 2', Comm. ACM 13 415 (1970).

5. M. PETRAVIC, G. KUO-PETRAVIC AND K.V. ROBERTS, 'Automatic Optimization of Symbolic Algol Programs, II Code Generation', to be published.

6. M. PETRAVIC, G. KUO-PETRAVIC AND K.V. ROBERTS, 'The Symbolic Algol II Generator Program', to be submitted for publication in Computer Physics Communications.

Table I

3D MHD Equations used in the TRINITY Code

Continuity equation

$$\frac{\partial \rho}{\partial t} = - \nabla . \rho \underline{v}$$

Momentum equation

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

Magnetic equation

$$\frac{\partial \underline{B}}{\partial t} = \nabla \times (\underline{v} \times \underline{B}) + \eta \nabla^2 \underline{B}$$

Temperature equation

$$\frac{\partial T}{\partial t} = - \underline{\nabla} . (T \underline{v}) + (2 - \gamma) \, T \, \underline{\underline{\nabla}} . \underline{v} + \kappa \nabla^2 T + (\gamma - 1) \, \eta j^2 / \rho + (\gamma - 1) \, \nu [(\nabla \times \underline{v})^2 + (\underline{\nabla} . \underline{v})^2]$$

Pressure

$$p_{ij} = \rho T \delta_{ij} + \rho v_i v_j + \frac{B^2}{2} \delta_{ij} - B_i B_j$$

Current

$$\underline{j} = \underline{\nabla} \times \underline{B}$$

Table II

3D MHD Equations Programmed in Symbolic Algol I

```
procedure INVOKE DIFFERENCE EQUATIONS;
begin
     CONTINUITY EQUATION: DT: = 2 x DELTA T; C1:= C2:  =  1;
     Q: = 1 + I+1 + (J+1) x PI + (K+1) x PI x PJ;
     NEW RHO: = RHO - DT x DIV(RHO x V);

     MOMENTUM EQUATION:   DT: = 2 x DELTA T/(1 + NU/EPS);
          for C1: = 1,2,3 do
          AV[C1,Q]: = (RHO x V + DT x (-DIV2(P) + NU x DELSQ(RHOx V)))/NEW RHO;
          ARHO[Q]: = NEW RHO;
     MAGNETIC EQUATION:          DT: = 2 x DELTA T/(1 + ETA/EPS);
          for C1: = 1,2,3  do
          AB[C1,Q]: = B + DT x (CURL(CROSS(V,B)) + ETA x DELSQ(B));
     TEMPERATURE EQUATION:       DT: = 2 x DELTA T/(1 + KAPPA/EPS); C1: = 1;
          ATEM[Q]: = TEM + DT x (-DIV(TEM x V) + KAPPA x DELSQ(TEM)
                   + (2 - GAMMA) x SAV(TEM) x DIV(V) + (GAMMA - 1) x (ETA x
                   SQM(CURL(B))/SAV(RHO) + NU x (SQM(CURL(V)) + DIV(V) ↑ 2 )));
end;

real procedure P;
P: = if C1 = C2 then (RHOx(TEM+V x V) + 0.5xDOT(B,B) - BxB) else (RHOxVxV2-BxB2);
```

Note The differences in the treatment of p and j between Tables I, II and III are not essential.

Table III

3D MHD Equations Programmed in Symbolic Algol II

```
CONTINUITY EQUATION:
EQUATE(NEWRHO,DIFF(RHO,MULT(DT(1),DIV(MULT(RHO,V)))));
MOMENTUM EQUATION:
   for C1:= 1,2,3 do
EQUATE(V,QUOT(SUM(MULT(RHO,V),MULT(DT(2),DIFF(SUM(DIV
(DIFF(TEN(B,B),MULT(RHO(TEN(V,V)))),MULT(NU,DELSQ(MULT(RHO,V)))),
GRAD(SUM(MULT(RHO,TEM),MULT(RNUM(0.5),DOT(B,B)))))), NEWRHO));
MAGNETIC EQUATION:
   for C1:= 1,2,3 do
EQUATE(B,SUM(B,MULT(DT(3),SUM(CURL(CROSS(V,B)),MULT(ETA,DELSQ(B))))));
TEMPERATURE EQUATION:
EQUATE(TEM,SUM(TEM,MULT(DT(4),SUM(DIFF(SUM(MULT(MULT(DIFF
(RNUM(2.0),GAMMA),SAV(TEM)),DIV(V)),MULT(KAPPA,DELSQ(TEM))),DIV(MULT
(TEM,V))),SUM(QUOT(MULT(DIFF(GAMMA,RNUM(1.0)),MULT(ETA,SQUARE
(CURL(B)))),SAV(RHO)),MULT(DIFF(GAMMA,RNUM(1.0)),MULT(NU,SUM
(SQUARE(CURL(V)),EXP(DIV(V),INUM(2)))))))));
```

Note Equations (1.5) and (1.6) have been incorporated into the main equations although they can be defined separately. The 4 calls of the procedure DT take into account the Dufort-Frankel factors [1].

Table IV

Maxwell's Equations in 3D Orthogonal Curvilinear Coordinates generated in IBM360 Algol

```
'COMMENT'----------------------------------------;
 HFILD1(/Q/).=HFILD1(/Q/)-DCT7*((EFILD3(/Q+  14/)*H3(/+2/)-EFILD3(/Q-  1
       4/)*H3(/-2/))*R2DQ2-(EFILD2(/Q+ 140/)*H2(/+3/)-EFILD2(/Q- 140/)*
       H2(/-3/))*R2DQ3)*RHPHM1;
'COMMENT'----------------------------------------;
 HFILD2(/Q/).=HFILD2(/Q/)-DCT7*((EFILD1(/Q+ 140/)*H1(/+3/)-EFILD1(/Q- 14
       Q/)*H1(/-3/))*R2DQ3-(EFILD3(/Q+   1/)*H3(/+1/)-EFILD3(/Q-   1/)*
       H2(/-1/))*R2DQ1)*RHPHM2;
'COMMENT'----------------------------------------;
 HFILD3(/Q/).=HFILD3(/Q/)-DCT7*((EFILD2(/Q+   1/)*H2(/+1/)-EFILD2(/Q-
       1/)*H2(/-1/))*R2DQ1-(EFILD1(/Q+  14/)*H1(/+2/)-EFILD1(/Q-  14/)*
       H1(/-2/))*R2DQ2)*RHPHM3;
'COMMENT'----------------------------------------;
 EFILD1(/Q/).=EFILD1(/Q/)+DCT8*((HFILD3(/Q+  14/)*H3(/+2/)-HFILD3(/Q-  1
       4/)*H3(/-2/))*R2DQ2-(HFILD2(/Q+ 140/)*H2(/+3/)-HFILD2(/Q- 140/)*
       H2(/-3/))*R2DQ3)*RHPHM1:
'COMMENT'----------------------------------------;
 EFILD2(/Q/).=EFILD2(/Q/)+DCT8*((HFILD1(/Q+ 140/)*H1(/+3/)-HFILD1(/Q- 14
       Q/)*H1(/-3/))*R2DQ3-(HFILD3(/Q+   1/)*H3(/+1/)-HFILD3(/Q-   1/)*
       H3(/-1/))*R2DQ1)*RHPHM2;
'COMMENT'----------------------------------------;
 EFILD3(/Q/).=EFILD3(/Q/)+DCT8*((HFILD2(/Q+   1/)*H2(/+1/)-HFILD2(/Q-
       1/)*H2(/-1/))*R2DQ1-(HFILD1(/Q+  14/)*H1(/+2/)-HFILD1(/Q-  14/)*
       H1(/-2/))*R2DQ2)*RHPHM3;
```

Table V

IBM 360 Assembly Code generated for the second component of the magnetic equation

The code is efficient, with no extra address-manipulation instructions. Out of the 48 instructions, 4 are concerned with stack overflow. Note that comments are also generated automatically.

	Operation code	Location	Variable	Level	
*				00	001
	LE	0,0308(,DISPQ)	B2+0+0	01	002
	LE	2,CONSTANT+008	DT	02	003
	LE	4,0296(PLUSDZ,DISPQ)	V2+0+0	03	004
	ME	4,0312(PLUSDZ,DISPQ)	B3+0+0	03	005
	LE	6,0300(PLUSDZ,DISPQ)	V3+0+0	04	006
	ME	6,0308(PLUSDZ,DISPQ)	B2+0+0	04	007
	SER	4,6		03	008
	LE	6,0296(MINSDZ,DISPQ)	V2+0+0	04	009
	ME	6,0312(MINSDZ,DISPQ)	B3+0+0	04	010
	SER	4,6		03	011
	LE	6,0300(MINSDZ,DISPQ)	V3+0+0	04	012
	ME	6,0308(MINSDZ,DISPQ)	B2+0+0	04	013
	AER	4,6		03	014
	ME	4,CONSTANT+016	RECDS2	03	015
	LE	6,0324(,DISPQ)	V1+1+0	04	016
	ME	6,0340(,DISPQ)	B2+1+0	04	017
	STE	0,STORAGE+000		05	018
	LE	0,0328	V2+1+0	05	019
	ME	0,0336(,DISPQ)	B1+1+0	05	020
	SER	6,0		04	021
	LE	0,0260(,DISPQ)	V1-1+0	05	022
	ME	0,0276(,DISPQ)	B2-1+0	05	023
	SER	6,0		04	024
	LE	0,0264(,DISPQ)	V2-1+0	05	025
	ME	0,0272(,DISPQ)	B1-1+0	05	026
	AER	6,0		04	027
	ME	6,CONSTANT+016	RECDS2	04	028
	SER	4,6		03	029
	LE	6,CONSTANT+004	ETA	04	030
	LE	0,0564(,DISPQ)	B2+0+1	05	031
	AE	0,0052(,DISPQ)	B2+0-1	05	032
	AE	0,0308(PLUSDZ,DISPQ)	B2+0+0	05	033
	AE	0,0308(MINSDZ,DISPQ)	B2+0+0	05	034
	AE	0,0340(,DISPQ)	B2+1+0	05	035
	AE	0,0276(,DISPQ)	B2-1+0	05	036
	STE	2,STORAGE+004		06	037
	LE	2,=E'+6.0000'+00'		06	038
	ME	2,0308(,DISPQ)	B2+0+0	06	039
	SER	0,2		05	040
	MER	6,0		04	041
	ME	6,CONSTANT+020	REDSSQ	04	042
	AER	4,6		03	043
	LE	2,STORAGE+004		02	044
	MER	2,4		02	045
	LE	0,STORAGE+000		01	046
	AER	0,2		01	047
	STE	0,0308(,DISPQ)	B2+0+0	01	048

Table VI

List of Generator Program Modules and Submodules

I. BASIC OUTPUT

II. CHARACTERS

III. MATHEMATICS A

- III.1 ARITHMETIC
- III.2 ALGEBRA

IV. MATHEMATICS B

- IV.1 VECTOR ANALYSIS
- IV.2 SPACE AND TIME SCALES

V. OUTPUT ORGANIZATION

- V.1 TRANSLATION LOGIC
- V.2 SIGNS, SPACES, NUMBERS AND FUNCTIONS
- V.3 VARIABLE CLASSES
- V.4 COMPONENTS AND DERIVATIVES
- V.5 COMMANDS AND REGISTER CHECKS
- V.6 INITIALIZATION

VI. PROBLEM DEFINITION

- VI.1 PHYSICAL CONSTANTS AND VARIABLES
- VI.2 CONTROL STATEMENTS
- VI.3 SOURCE STATEMENTS

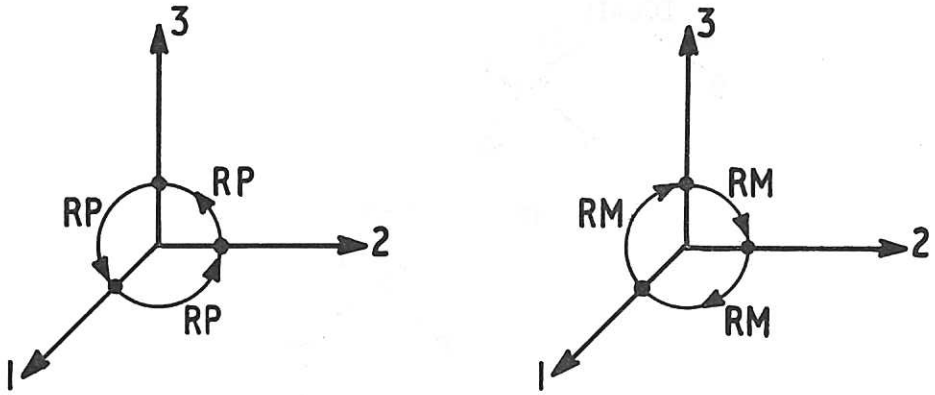

Figure 1
Positive and Negative Rotation Operators

The operators RP, RM rotate vector components cyclically in the positive and negative directions respectively, and satisfy the symbolic relations $R_+^3 = R_-^3 = R_+R_- = R_-R_+ = 1$, from which $R_+^2 = R_-$ etc. (Here $R_+ \equiv$ RP, $R_- \equiv$ RM)

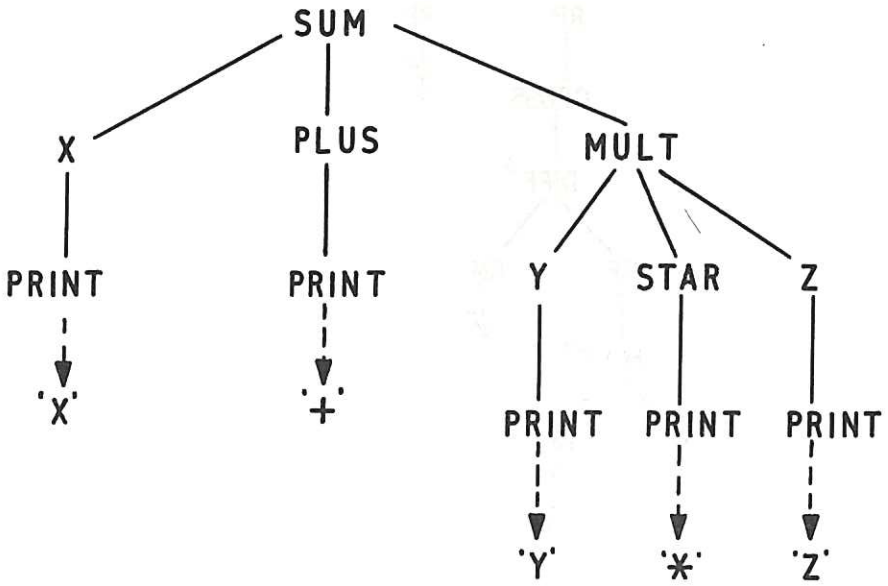

Figure 2
Generation of Code by the Expression SUM(X,MULT(Y,Z))

CLM-P274

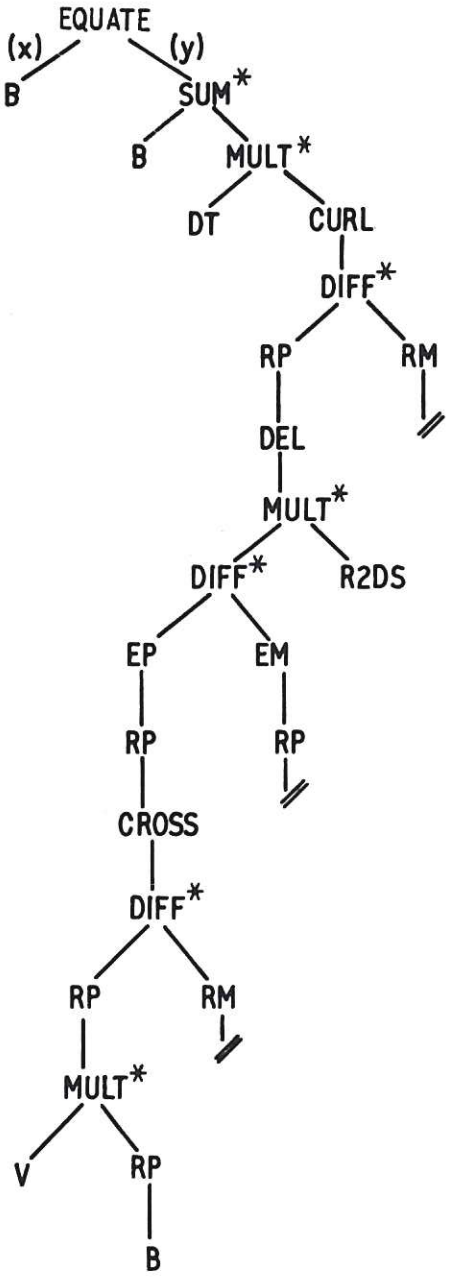

Figure 3

Procedure Tree for the Magnetic Equation (27)

Each arithmetic operator indicated by * calls the TRIPOP operator recursively to control the optimization process. Branches broken off with // are similar to the full branch shown on the diagram. Terminal symbols V, B, DT, R2DS generate the code, while RP, RM manipulate the global component variable C1, and EP, EM manipulate the mesh location.

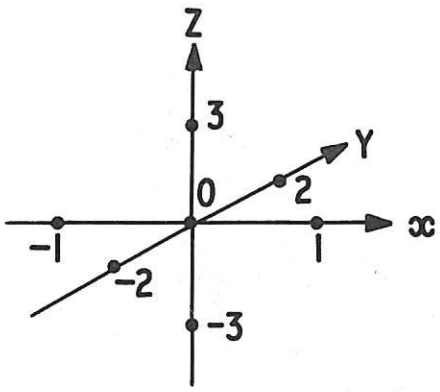

Figure 4

Subsidiary Mesh used to store the Scale Factors

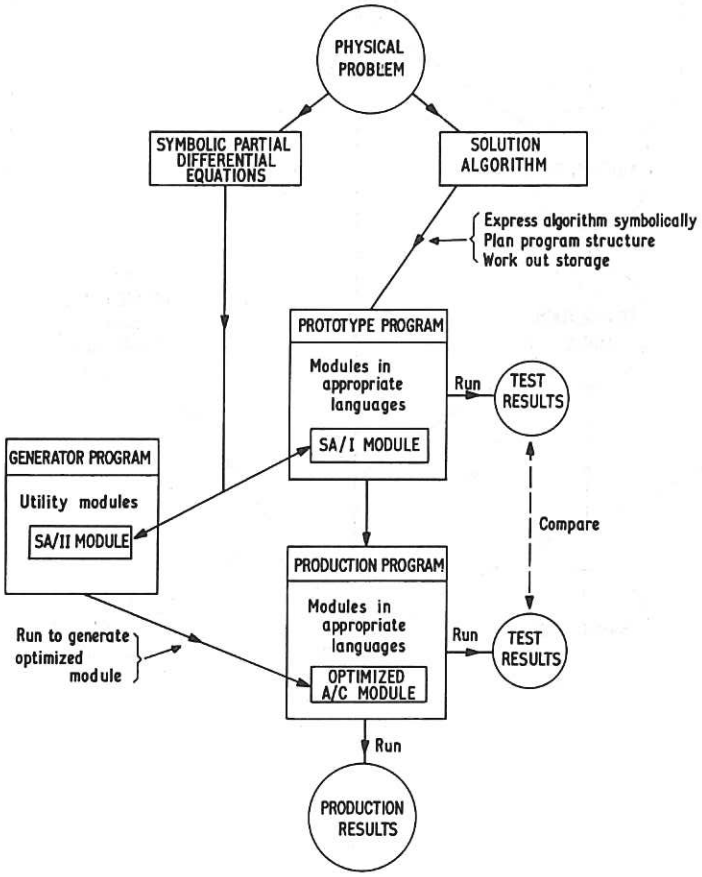

Figure 5

Programming Strategy

The method of solution is first tried out by writing a prototype program which is used to produce sets of test results. Modules of the prototype are next reconstructed either by hand or with the SA/II generator program to make them more efficient. Test results from the production program are then compared with those from the original tests to make sure that the updating has been carried out correctly.

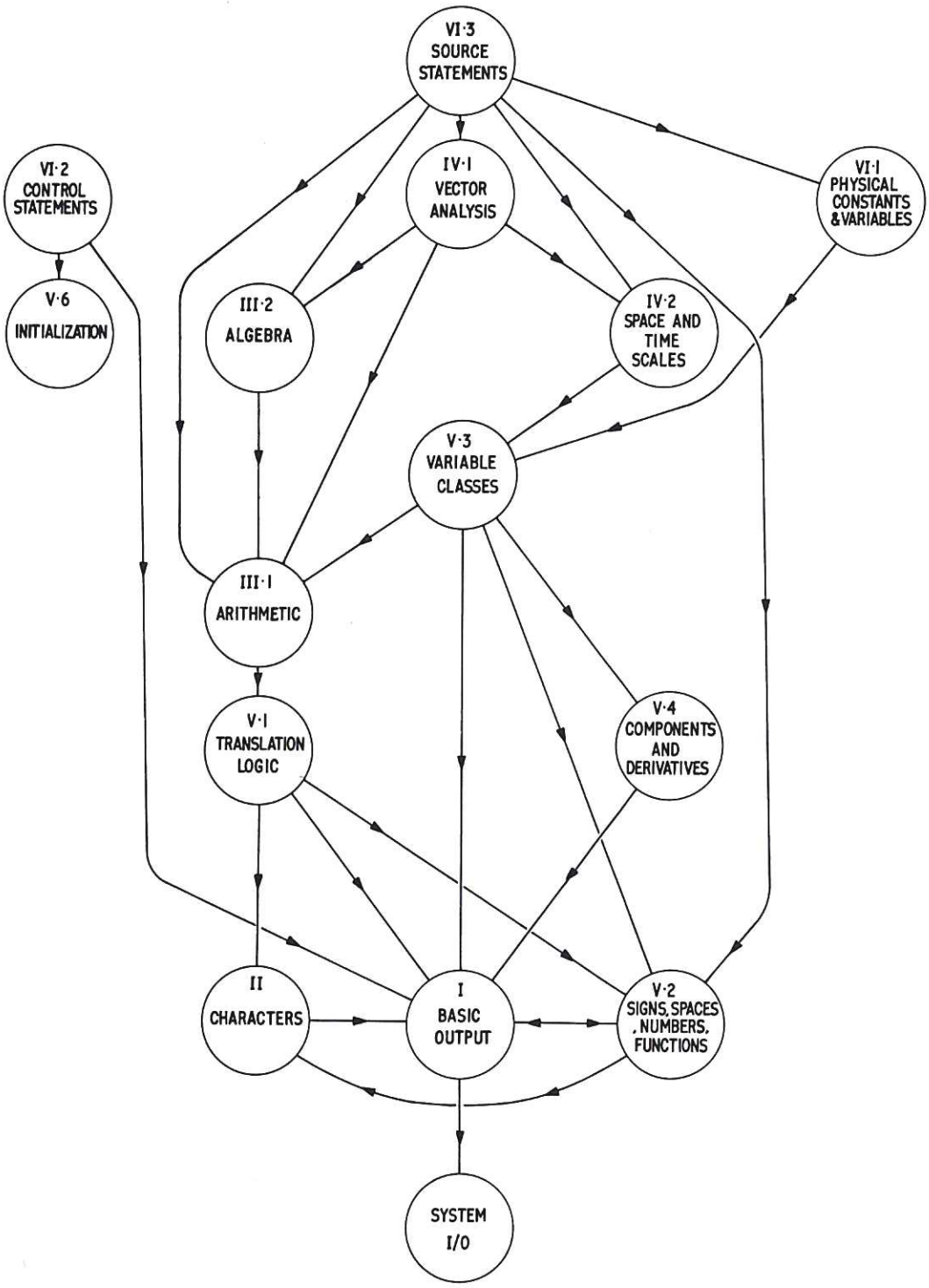

Figure 6

Relation between the modules of the generator program

An arrow indicates that one module makes use of procedures belonging to the other. Note that all output is channelled through a single short module BASIC OUTPUT so that only this module has to be changed to run the program on another computer.