CLM - P257

United Kingdom Atomic Energy Authority
RESEARCH GROUP

Preprint

SYMBOLIC PROGRAMMING FOR PLASMA PHYSICISTS

K. V. ROBERTS
R. S. PECKOVER

Culham Laboratory
Abingdon Berkshire

1970

CLM - P 257

SYMBOLIC PROGRAMMING FOR PLASMA PHYSICISTS

by

K.V. ROBERTS
R.S. PECKOVER

(Paper presented at the Fourth Conference on Numerical Simulation of Plasmas, U.S. Naval Research Laboratory, Washington, 2-3 November 1970)

A B S T R A C T

The symbolic use of Algol has been developed in order to provide a clear notation in which to code physics problems for solution on a computer. A scheme has been devised which enables real physics programs to be built up quickly from a standard skeleton program DUMMYRUN by adding modules peculiar to the problem. Each module is provided with a 'testbed' which enables it to be checked in a methodical way. Techniques are described for making an Algol program 'portable', so that it can readily be transferred from one computer to another, and the paper discusses the CACTUS package which performs an automatic conversion between certain alternative Algol hardware representations. It is explained how the use of Symbolic Algol can make the initial and boundary conditions clear. By way of comparison, the way in which a Fortran simulation program can be built up from a Universal Control Package (UCP) is briefly mentioned.

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

November, 1970

C O N T E N T S

1. INTRODUCTION

The details of computing are one of the burdens that must be borne by the physicist who believes in a particular case that only by a complex calculation will he be able to substantiate his theory or establish the intricacies of a real experimental situation. Nevertheless, it seems to us that the simulation of plasma behaviour, to take an example, is at present made a more difficult task than it need be partly because of the limitations of existing programming languages, and partly because each worker in the field usually constructs his own input, output and control facilities, and often does not develop his program in a systematic way. However, a common structure can be adopted for a wide class of time-dependent fluid flow problems.

To help the computational plasma physicist to keep closer to his problem, the symbolic use of Algol has been developed[1,2,3]. Further, a system has been devised which enables computer programs to be built quickly out of a set of standard prefabricated modules, with the addition of a few further modules peculiar to the problem. The style enables all details of mathematics and logic to be hidden at a lower level. The discussion will be based on the use of Symbolic Algol, (see Section 2), but the remarks about the need for a good adaptable program structure hold true also in Fortran (see Appendix).

The prefabricated general purpose modules deal with output, vector algebra, vector analysis, program control etc., act as a library, and provide the physicist with a program which has a logical structure and with the mathematical tools with which he is familiar.

Many programs in plasma simulation share the same superficial structure - a set of differential equations are solved as a function

of time - and this superficial structure can be made the same in a suite of computer programs, so that only the sections describing the actual physics need be different from case to case. Symbolic Algol programs can moreover be written in a machine-independent way, so that they will run quickly on any computer system, and with careful design they can be made to execute with high efficiency.

To construct worthwhile programs rapidly, modules with well-defined interfaces and dependable characteristics must be available - thiscan be achieved using tiny testbed programs the results of which are available as guarantees of confidence. The "standard empty program" DUMMYRUN described in Section 3 was tested in just this way before being used as the skeleton on which to hang more substantial programs. Such programs as ROLLS (a 2D program for studying enclosed convection) and TRINITY (a 3D MHD program) have been tested this way and provide good illustrations of the way in which initial conditions and boundary conditions (see Section 6) can be set up easily in Symbolic Algol I. These were initially programs using leapfrog schemes but the modular structure is flexible enough for other schemes both explicit and implicit to be incorporated without major surgery. Such flexibility is illustrated in Section 7.

2. SYMBOLIC ALGOL I

Let us first recall how symbolic Algol techniques can be used to express programs that solve sets of partial differential equations. These techniques were briefly reported at Culham[1] and a more complete account can be seen in papers to be published soon[2,3].

For mathematical text books, a fairly standard notation has been adopted for such topics as vector algebra, and analysis. This notation is highly compressed and can be co-ordinate-free. For example,

the vector magnetic field is written tersely as $\underline{B}$ instead of in the expanded form $(B_x(x,y,z), B_y(x,y,z), B_z(x,y,z))$ and such expressions as (curl $\underline{B}$) $\wedge$ $\underline{B}$ are independent of both the particular co-ordinate system used and the effective number of dimensions.

Consider the equation of charge conservation

$$\frac{\partial q}{\partial t} + \text{div } \underline{j} = 0 \qquad (1)$$

Using an explicit difference scheme we may express this in Algol 60 as:-

```
AQ[O] := Q - DT * DIV(J);                    (2)
```

where:

AQ is an array holding the charge;

O is the local origin of the difference scheme;

Q is the value of the charge at the 'old' time;

DT is the effective time increment;

J is the current vector;

DIV is a finite different operator.

Here Q, J, and possibly DT are real parameterless procedures. i.e. they depend only on implicit variables which represent the chosen vector component and lattice point in terms of the geometry of the problem and do not depend on any explicitly exhibited variable. The differential operator "div" is defined in mathematical physics for a Cartesian co-ordinate system by

$$\text{div } \underline{F} = \sum_{i=1}^{3} \frac{\partial F_i}{\partial x_i} \qquad (3)$$

In Symbolic Algol I it is represented by the quite analogous real procedure DIV which has a declaration of the form

real procedure DIV (A); real A; DIV: = SIGMA(DEL(A)); (4)

The procedure DIV here has an explicit argument, as div does in vector analysis.

A second example of the compactness of Symbolic Algol notation is in the difference form for the Vlasov equation. For a continuous distribution function f we may write the Vlasov equation in ordinary mathematical notation as

$$\frac{\partial f}{\partial t} + (\underline{v} \cdot \underline{\nabla})f + (\underline{a} \cdot \frac{\partial}{\partial \underline{v}})f = 0 \qquad (5)$$

without needing to say explicitly that f is a function of (x,y,z,u,v,w,t) wherever it is mentioned.

In Algol 60 this can be written

AF[O] := F - DT * (DOT(V,DEL(F)) + DOT (A,DELV(F))); (6)

Because procedures can be parameterless in Algol (unlike Fortran) we can express the distribution function as F, its current value, through the definition:

real procedure F ; F:= AF(O); (7)

where O is a matrix subscript and represents the current position on the lattice.

The real procedure DOT is the finite difference analogue of the inner vector product i.e.

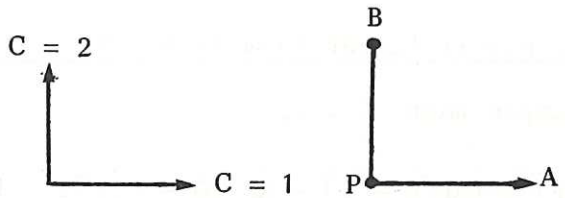
$$\underline{a} \cdot \underline{b} \rightarrow \text{DOT(A,B)} \qquad (8)$$

To show the way in which Symbolic Algol I is able to build up new procedures from older ones in a hierarchic fashion, the nested structure of DEL can be exhibited. The abbreviation RP will be used for real procedure and IP for integer procedure.

```
RP DEL(F); real F;  DEL = (EP(F)-EM(F))/(2*DS);
RP EM(F); real F; begin O := O - DO; EM := F;O := O + DO; end
RP EP(F); real F; begin O := O + DO; EP := F;O ;= O - DO; end
IP DO;
DO:= if C eq 1 then 1 else if C eq 2 then PI else PI*PJ;
```

(9)

O is the current lattice point. C is the index specifying the direction.



If C = 1, DO corresponds to the length PA, and the statement O:= O + DO moves the current origin from P to A. If C = 2 the origin is moved to B, and if C = 3 it is moved in the z-direction.

This may be compared with the mathematical hierachy

$$\nabla_h \psi \equiv \frac{1}{2h}(E^+ - E^-)\psi \tag{10}$$

where

$$\begin{aligned} E^+\psi(\underline{r}) &= \psi(\underline{r} + \underline{dr}) \\ E^-\psi(\underline{r}) &= \psi(\underline{r} - \underline{dr}) \end{aligned} \tag{11}$$

and

$$\left.\begin{aligned} \underline{dr} &= dx \text{ (x component)} \\ &\text{or } dy \text{ (y component)} \\ &\text{or } dz \text{ (z component)} \end{aligned}\right\} \tag{12}$$

With so many nested procedure calls Symbolic Algol I usually executes slowly, but it is very clear what the program is doing. Such an approach enables clear programming in the initial stages. Subsequently DEL could be made faster by introducing a vector DR with 3 elements containing 1, PI, and PI * PJ respectively. The procedure

definition for DEL could then be shortened into

```
RP DEL (F); real F; begin real F1;
integer DO;
DO: = DR(C);  O := O + DO; F1: = F;
              O := O - 2 * DO; DEL: = (F1 - F)/(2 * DS);
              O := O + DO; end;
```

(13)

A deeper level of optimisation is to replace the right hand side of equation (6) by an explicit linear form in AF $[O + \alpha]$ for a sum of terms in α. For example when $C = 1$,

$$DEL(F) := (AF[O + 1] - AF\ [O - 1])/(2 * DS); \qquad (14)$$

Optimisations of this kind can be carried out automatically and are discussed in detail elsewhere[2].

Converter programs exist[14] which enable equations to be converted automatically into an optimized code for any desired combination of output language, co-ordinate system, and different scheme. Languages implemented so far have been Algol, Fortran, KDF9 Usercode and IBM 360 assembly language. The optimized module is then used, in conjunction with the remainder of the original Symbolic Algol 1 program, to carry out the production runs.

3. STANDARD MODULAR STRUCTURE

Consider two programs that solve two sets of different time-dependent fluid equations. Evidently these programs could be designed to have much in common; almost everything, in fact, except the physics. We have developed a series of Algol modules which deal with standard tasks such as output, vector algebra and analysis, program control and so on, and which fit together to enable a wide range of physics programs to be constructed quickly. Some of these pre-fabricated modules have been put together to form a skeletal program

DUMMYRUN which has the general structure of a program that simulates time-dependent particle and fluid flows, although it actually does no physics. It is a "standard empty program" which with little effort can be converted into a running program for a real physical problem.

DUMMYRUN consists of a set of modules which are available as on-line files and has the following structure:

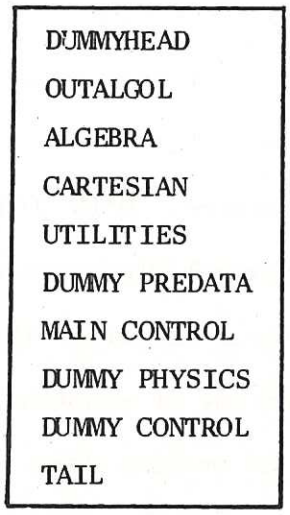


Fig.1

Let us build the program up piece by piece:

(a) An Algol program needs job control cards, and a begin for the outer block. } HEAD

(b) It needs an end for both the outer block and the inner problem-oriented block. } TAIL

(c) In order to output anything, an output channel must be specified and output procedures provided. } OUTALGOL

(d) The physics problems in which we are interested use vector algebra, difference schemes in Cartesian geometry, with various sorts of array output. } ALGEBRA, CARTESIAN, UTILITIES

(e) Clearly other standard library modules could be inserted, to generate on-line graphical display or to use cylindrical co-ordinates, for example.

(f) Prior to entering the inner (physics) block, dynamic array bounds must be set and the various modules activated. } PREDATA

(g) Most fluid simulation programs seem to require must the same elements, as we find by perusing programs written in the past both by ourselves and by others. These have been formalized into a standard control structure. MAIN CONTROL consists of a series of parameterless procedures whose names describe their functions (Fig.2). This provides a compact way of starting clearly what each section of the program does. Each of the procedures called by MAIN CONTROL must be defined and a series of modules are created to do this. Initially we need only two: CONTROL and PHYSICS. } MAIN CONTROL

```
'COMMENT' ----------------------------------------------------
          S-FILE*MAIN CONTROL.CPIP* (STANDARD MAIN PROGRAM);

'PROCEDURE'MAIN CONTROL;'BEGIN'
     LABEL THE RUN;                              REPORT(1,1);
     CLEAR VARIABLES AND ARRAYS;                 REPORT(1,2);
     SET DEFAULT VALUES;                         REPORT(1,3);
     DEFINE DATA SPECIFIC TO RUN;                REPORT(1,4);
     SET AUXILIARY VALUES;                       REPORT(1,5);
     DEFINE INITIAL CONDITIONS;                  REPORT(1,6);
     INITIAL OUTPUT;                             REPORT(1,7);
'FOR' N=NSTART'STEP'1'UNTIL'NSTOP'DO'
     'BEGIN'MAIN COMPUTATION CYCLE'.' T=T+DELTAT;
     ADVANCE ONE TIMESTEP; OUTPUT IF REQUIRED;
     'END' OF MAIN CYCLE;
                                                 REPORT(1,8);
                                                 REPORT(1,9);
     N=NSTOP; FINAL OUTPUT;TERMINATE THE RUN;
'END';
```

Fig.2

(h) CONTROL provides the procedures which control the progress of the calculation, i.e. which label the run, clear the core store, initialize the run, output when required, and tie up the loose ends after the run is complete.

CONTROL

(i) PHYSICS provides a slot where the real physics is to be inserted. Initially it contains only a dummy procedure: (Fig.3)

```
FILE*DUMMY PHYSICS*

'PROCEDURE'ADVANCE ON TIMESTEP;'BEGIN'LINE
   TEXT(' '**WE**HAVE**ADVANCED**ONE**TIMESTEP** ' ')
  LINE;'END';
```

PHYSICS

Fig.3

PHYSICS can be augmented, if required, by procedures which solve Poisson's equation or which deal with standard boundary conditions, for example.

(j) Through the module OUTALGOL, mentioned earlier in DUMMYRUN, is funnelled all input and output. This makes the program portable, since the changes required to run on a different machine are all concentrated into one place.

DUMMYRUN

4. ACCEPTANCE TESTS

Each module is part of an assembly, and as with any engineering component that is to be used without constant attention, it is sensible to put the module through a proper set of acceptance tests to guarantee that each element of the module performs properly. Every module must be completely dependable. Further, by making the tests generally available, the authors of the program raise confidence

levels, and allow the user to see for himself any restrictions which may apply.

Let us introduce the concept of a "testbed". This a special small program written for the new module which uses and tests out all its features in as thorough but economical a way as possible. A testbed program consists of:

(a) The pre-tested modules required in order to use the new module, or to perform the tests.

(b) The new module itself.

(c) A specially written TRIAL module, which is the part of the program that runs through all the procedures in the module which is being tested.

For example, the TESTBED of the module *ALGEBRA* which is used in DUMMYRUN consists of 4 modules:- HEAD (which contains the job control cards); OUTALGOL (through which is funnelled all output); ALGEBRA itself, and its associated TRIAL MODULE (see Fig.4).

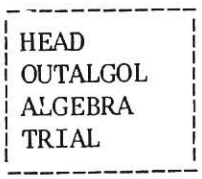


Fig.4

HEAD and OUTALGOL are modules which have been put through just such test previously. The results of this test are made available as part of the documentation for the module, and this facilitates conversion to other computer systems.

For further illustration let us consider the module CARTESIAN. It consists of procedures which are the finite difference analogues of DIV, GRAD, CURL. Of course these have different forms depending on the difference scheme and geometry; the forms in CARTESIAN are appropriate

for 3-D Cartesian geometry in which the operators are defined in terms of central difference formulae.

Analytically

$$\{\mathrm{curl}(\underline{u})\}_i \equiv \varepsilon_{ijk} \frac{\partial u_j}{\partial x_k} \tag{15}$$

In 3 dimensions, it is convenient to think of the dimensions (x,y,z) as (x_1,x_2,x_3) which can be written as $\{(x_{i-1},x_i,x_{i+1})/\text{modulo } 3\}$. In terms of these we may write

$$\{\mathrm{curl}(\underline{u})\}_i = \frac{\partial u_{i-1}}{\partial x_{i+1}} - \frac{\partial u_{i+1}}{\partial x_{i-1}} \tag{16}$$

i.e.

$$\{\mathrm{curl}(\underline{u})\}_i = \frac{\partial u_{i+2}}{\partial x_{i+1}} - \frac{\partial u_{i-2}}{\partial x_{i-1}} \text{ (modulo 3 again)}$$

The Algol procedure CURL is defined by

```
CURL(U) := RP(DEL(RP(U))) - RM(DEL(RM(U)));          (17)
```

in Symbolic Algol I. The procedure RP effectively increases the index by unity, and RM decreases it by unity. DEL is the analog of $\underline{\nabla}$ (see equation 9 above).

A test for CURL could be as follows:- Let f_1,f_2,f_3, g be any 4 scalar functions of x,y,z. Construct a general vector $\underline{A}$ s.t.

$$\underline{A} = \mathrm{curl}\ (f_1,\hat{x} + f_2,\hat{y} + f_3,\hat{z}) + \underline{\nabla}g \tag{18}$$

Then for the differential operators, the following 2 identities hold:-

$$\mathrm{div\ curl}\ \underline{A} \equiv 0 \tag{19}$$

and

$$\mathrm{curl\ grad}\ f_1 \equiv 0 \tag{20}$$

If the finite difference operators DIV, CURL, GRAD are defined in terms of central differences on a Cartesian mesh, these identities still hold. In a TESTBED for these, a TRIAL module would then contain:

RP F1; F1:= (user's choice: e.g. $x^4 + y^4 + z^4$);

RP F2; F2:= (user's choce: e.g. 3xyz);

RP F3; F3:= (user's choice: e.g. $x^3 + y^3 + z^3$);

RP G; G:= (user's choice: e.g. e^{x+y+z});

RP A; A:=

CURL(F1*E1 + F2*E2 + F3*E3) + GRAD(G);

and the printing procedures

PRINT VECTOR (CURL(GRAD(F1));

PRINT SCALAR(DIV(CURL(A));

The output is of course

0.0 0.0 0.0

and

0.0

The procedures E1, E2, E3 come from the module *ALGEBRA*, and are defined thus:-

RP E1; E1:= if C eq 1 then 1 else 0;

RP E2; E2:= if C eq 2 then 1 else 0;

RP E3; E3:= if C eq 3 then 1 else 0;

where C is an index indicating which component - x, y or z - is under consideration. E1, E2, E3 are in fact the unit vectors in the x, y, and z directions respectively.

The modules of which the skeleton program DUMMYRUN is composed have each been tested using the TESTBED approach. The progressive way in which such tests can be carried out is illustrated in Fig.5.

5. PORTABILITY

Collaboration between the staff of different laboratories can often make desirable the running of the same computer program on different machines. The transfer from one machine to another can be

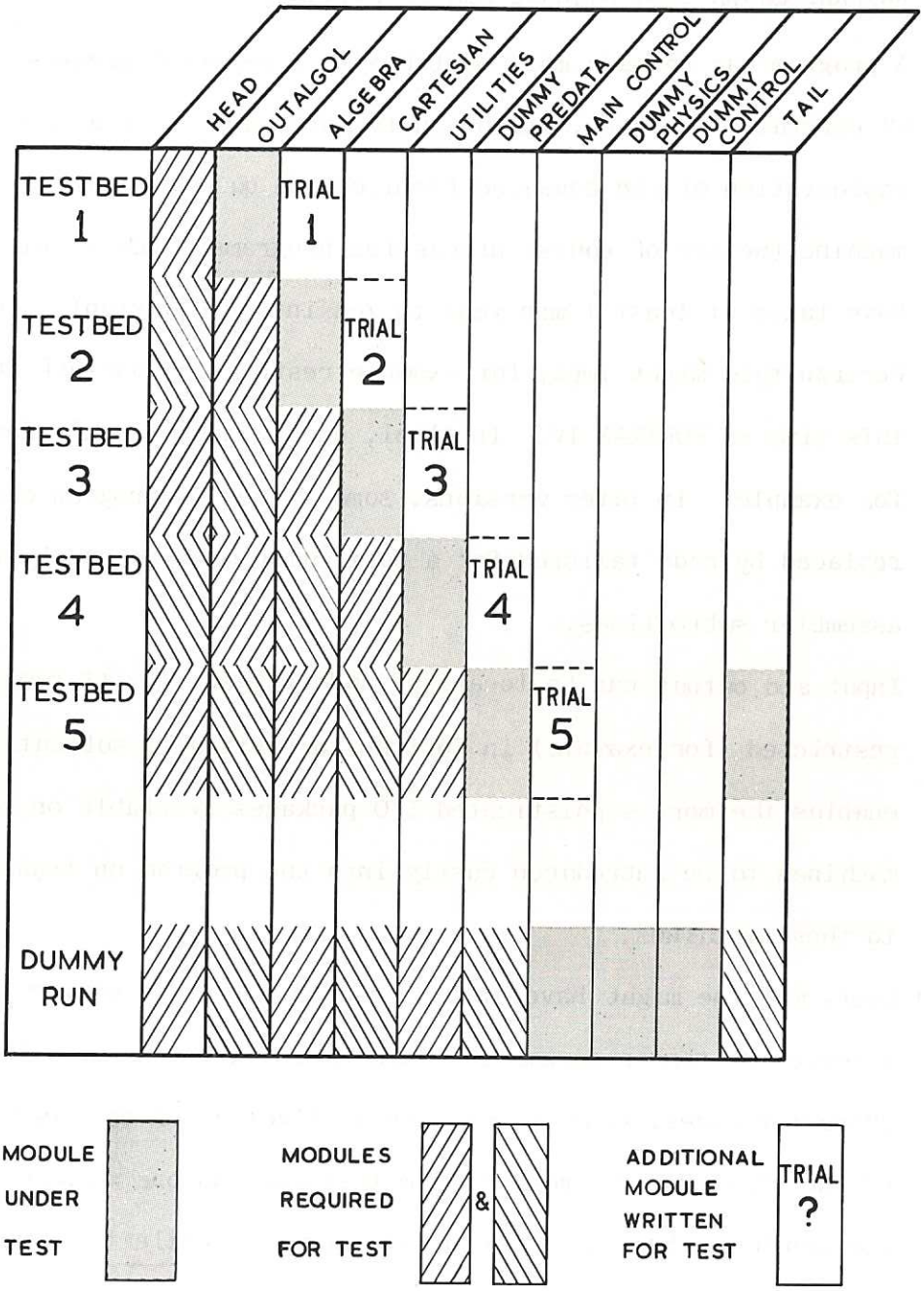


Fig.5. Listed along the top are the modules of which DUMMYRUN is constructed. The staircase shows the systematic way in which modules must be tested. For example TESTBED4 requires 4 pre-tested modules (DUMMYHEAD, OUTALGOL, ALGEBRA, CARTESIAN) in order to test UTILITIES. TRIAL4 is the specially written module containing the test material.

P 257

made more quickly, if the changes which must be made are localised into regions where such changes can be flagged.

A program can be made more portable in 3 separate areas:-

i) By carefully avoiding, in the early mark numbers of a program, exploitation of the advanced features and quirks peculiar to a machine (we are of course discussing programs which in the past have taken at least 1 man year to get into production). In Fortran this might imply for example restricting oneself at this time to FORTRAN IV. In Algol, one avoids Jensen's device for example. In later versions, some pieces of program can be replaced by code tailored for a particular machine and by fast assembler subroutines.

ii) Input and output can be localised and flagged. If I/O is restricted (for example) in Fortran to particular subroutines it enables the more sophisticated I/O packages available on some machines to be introduced easily into the program on transfer to those machines.

iii) Every machine might have available as a matter of course a series of tiny programs or macros which will change the character codes, (and in the case of Algol alter the representation of the Algol Basic Symbols) from that used on one machine to that for another. For the job control languages, while the variety of different machines continues, there seems no better way of grafting a program onto an unfamiliar machine than having the aid of someone with local knowledge.

Thus, at the Culham Laboratory we are deliberately writing our Fortran programs in ASA Fortran IV, which enables a program to be portable, and incidentally to be publishable and open to the criticism

of others. For example, since the name-list facility is not implemented on all big machines, it is not used, even though it is very convenient. In Algol, calls by value are avoided since some compilers do not have this feature. Again, since only the first six letters of an identifier are significant on an IBM machine, the long identifiers are chosen with care to ensure the distinct identity of each to the compiler.

For input and output, and the program transfer macros, we restrict our attention at this point to Algol; further comments on Fortran are found in the Appendix.

The Algol module OUTALGOL (see also Section 3 above) has been written to contain the high level procedure calls required to make simple-minded input and output requests; some of these are shown in Fig.6 in the form implemented on the KDF9 at Culham. The intention is to hide detail, irrelevant in more physical contexts. The advocacy of the use of such procedures, independent of implementation, is not new, see for example Michie, et al[5]. We report that their systematic use is worth the additional care in design initially. For the implementation on an IBM 360 see Figure 7. Similar OUTALGOL modules have been written for CDC 6600, the ICL 1900 series and GE 235.

To effect transfer from one machine to another the character code must be altered, and the Algol Basic Symbols correctly represented. For this purpose the CACTUS package has been developed at Culham. On the Culham KDF9, the COTAN on-line system contains among its commands the facility for generating a "macro" command - i.e. a command which blocks together a series of commands in a file and activates them with a single command[15]. Macros have been written which change the representation of Algol Basic Symbols on the disc in such a way that

```
'PROCEDURE' OUTALGOL;
    'BEGIN' DECIMAL=LAYOUT('['S-NDDD.DD']');
     NTEGER=LAYOUT('['S-NDDD']');OUT=10 'END';
'INTEGER'  DECIMAL,NTEGER,OUT;

'PROCEDURE' BLANK;SPACE(OUT,1);
'PROCEDURE' INTVAR(N);'INTEGER'N;WRITE(OUT,NTEGER,N);
'PROCEDURE' IVAR(NAME,N);'STRING'NAME;'INTEGER'N;
    'BEGIN' TEXT(NAME);BLANK;TEXT('['='];INTVAR(N);'END';
'PROCEDURE' LINE;NEWLIN(OUT,1);
'PROCEDURE' PAGE;GAP(OUT,1);
'PROCEDURE' TEXT(T);'STRING'T;WRITET(OUT,T);
```

Fig.6. The module *OUTALGOL* includes such precedures as the above. REALVAR and RVAR are similar to INTVAR and IVAR.

```
'PROCEDURE' OUTALGOL.,
    'BEGIN' OUT .=1., 'END'.,
'INTEGER' OUT.,
'PROCEDURE' BLANK., OUTSTRING(OUT,'(' ')').,
'PROCEDURE' INTVAR(N).,'INTEGER'N., OUTINTEGER(OUT,N).,
'PROCEDURE' IVAR(NAME,N).,'STRING'NAME.,'INTEGER'N.,
    'BEGIN'TEXT(NAME).,BLANK.,TEXT('('=')')., INTVAR(N).,'END'.,
'PROCEDURE' LINE., SYSACT(OUT,14,1).,
'PROCEDURE' PAGE., SYSACT(OUT,15,1).,
'PROCEDURE' TEXT(T).. 'STRING' T., OUTSTRING(OUT,T).,
```

Fig.7. Some OUTALGOL precedures used on an IBM 360.

it comes to a Data Dynamics teletype in the form suitable for whichever machine is to be recipient. It is stored on the disc as "Wheteg Algol" which is a subset of the representation acceptable to the Whetstone compiler for instant execution and which is also acceptable to our Egdon compiler for batch processing. This flexibility enables the KDF9 to be used reasonably efficiently (viewed as a man+machine entity).

Given an Algol program in "Wheteg", it can be automatically translated into the forms required for use on a CDC 6600, IBM 360, ICL 1900 or GE 235, though still stored in an on-line file. The actual transfer can occur in a very simple fashion: the ISO paper tape code used by teletypes interfaces with all computers which connect with teletypes. Thus a paper tape can be produced for an Algol program which is then read back to another teletype (or the same one) connected up to a different machine. In this way the existence of different card codes on the various machines can be circumvented. The development of a program on a machine with a fast compiler and good debugging facilities, and the subsequent transfer to a machine with fast running times and a big core, seems an attractive method of computing effectively.

6. INITIAL CONDITIONS AND BOUNDARY CONDITIONS

The setting of initial conditions and boundary conditions is, in a substantial computer program, a nuisance. It takes up, in coding terms, far more statements and involves more intricate devices than the body of the calculation in which most of the time of the computation is spent.

This need not be so, and we describe in this section some of the tools with which we have provided ourselves. Some of these could be produced in Fortran, but there is no doubt that the availability of

parameterless procedures in Algol is a boon.

For example it is convenient to be able to set the values of an array in a simple fashion. Consider the triplet of Algol statements:-

```
TWO D;
FULL REGION;
SET SCALAR(PHI,SIN(2*PIE*X)*SIN(2*PIE*Y));
```

This sets the potential φ over the whole region of interest in a two-dimensional calculation, to be $\sin(2\pi x).\sin(2\pi y)$. Of course the setting of parameters and general spadwork needs to be done somewhere, but not at this point where we are only concerned with the mathematical formulation of the physics of the problem. Further assignment of values is not a clumsy matter but can be done briefly. For example the contents of the above procedures are:

```
procedure TWO D; begin NDIM:= 2; I:= J:= 0; K:= -1, NK:= 0; end;
```

(the problem is declared to be 2D, the values of I and J are cleared, and the k-direction is cut out).

```
procedure FULL REGION; begin IP1:= JP1:= KP1:= 0; IP2:= JP2:= NJ;
    KP2:= NK; end;
```

(the upper and lower bounds for the array indexes are set in terms of the array sizes).

```
procedure SET SCALAR (A,F); array A; real F;
    begin procedure SETS2(K); integer K;
        for J:= JP1 step 1 until JP2 do
        for I:= IP1 step 1 until IP2 do
        begin O:= 1 + (I + 1) + PI*(J + 1) + PI*PJ*(K + 1);
            A[O]:= F; end ;
    if NDIM eq 2 then SET S2(-1);
    else for K:= KP1 step 1 until KP2 do SET S2 (K);
```

(the array A is set to have the function value F evaluated at the relevant point on the lattice).

To set the value in 3 dimensions of the magnetic field $\underline{B}$ for each of its components B_X, B_y, B_Z in the interior of the region of interest, we may write

```
THREED;
INTERIOR REGION;
SET VECTOR (AB, Z*ZE1 + 0.5*X*Z*E2 + X*E3);
```

This sets the magnetic field to be

$$\underline{B} \equiv (Z^2, \tfrac{1}{2}ZX, X)$$

E1, E2, E3 are the unit vectors defined earlier (in Section 4); AB(C,O) is an array with 2 arguments: C (which determines the component x,y,z) and O (which determines the current lattice point).

The declaration

```
real procedure B; B:= AB(C,O);
```

enables B to be used in equations for the magnetic field as vector $\underline{B}$ would be.

An alternative approach is to use 3 component arrays

BX[O], BY[O], BZ[O]

rather than the single array AB(C,O). Obviously the technique does not depend on the method of storage of the information about the magnetic field.

In terms of these 3 arrays, we could set up $\underline{B}$ with the procedure calls

```
THREE D;
INTERIOR REGION;
SET SCALAR (BX, Z*Z);
SET SCALAR (BY, 0,5*X*Z);
SET SCALAR (BZ, X);
```

Another set of useful procedures involves setting values on lines and surfaces inside the region of interest. These are all tiny procedures, where the purpose of defining them is to remove the mechanics of the computing from the focus of attention. For example

```
procedure SET XLINE (A, JJ, KK, F); array A; real F; integer JJ, KK;
    begin J:= JJ; K:= KK;
    for I:= IP1 step 1 until IP2 do
    begin DEFINE O; A[O]: = F; end;end;
```

This sets the elements of the array A, corresponding to points on the line $y = y_j$, $z = z_k$ (i.e. J = JJ; K = KK) to have the functional values F.

This ruled line provides a simple way of setting values on a plane surface parallel to an axis. Thus

```
procedure SET YZ SURFACE (A, II,F); array A; real F; integer II;
    for K:= KP1 step 1 until KP2 do
        SET YLINE (A,II,K,F);
```

sets A = F on the surface $x = x_i$, (i.e. I = II).

These hierarchic definitions enable one to program as clearly as one can write mathematics.

Similarly with boundary values, if the boundary conditions can be set easily and can be seen to have been set correctly, a program is simpler to handle. The setting of some of the possible boundary conditions at the wall for a plasma experiment are shown in Fig.8.

```
WALL;
   GUARDX(RHO,RIGID,ZERO          ZERO,SINGLE STEPS TO,NI);
   GUARDX(JZ,SYMMETRIC, ZERO, X-STAGGERED PT, DOUBLE SETPS TO, NI);
   GUARDX(TEMP,RIGID,TO*Z*(1-Z), ZERO,DOUBLE STEPS TO,NI);
```

Fig.8

The procedure WALL (besides acting as a paragraph heading) sets the current origin on the wall and arranges for it to move along in the x-direction setting suitable values. The procedure GUARDX has mnemonic arguments to make the boundary conditions clear. The use of mnemonics is valuable for although the analytic condition may be simple, the difference form is often messy. The first set of arguments defines the physics. For example, we have a rigid boundary on which the density is set to zero, and the temperature to $T_0Z(1-Z)$. The z component of the electric current j is set to zero, and so is its gradient. The second set of arguments pick out the points where values are to be set. For example, using a leapfrog scheme one may only need values at alternate points, the mesh in use shifting by one interval between alternating time steps. The starting point is shifted backwards and forwards by X-STAGGEREDPT, and values are set at alternate points up to point I = NI.

7. FLEXIBILITY

The flexibility of a modular structure, properly constructed, enables programs to be developed quickly. Each module, besides being fully tested, has a corresponding dummy module. This is composed of the same procedures as the full-bodied module, but each is dummy. Such dummy modules enable parts of the program to be thoroughly tested without wasting time in execution and compilation of other pieces of program known to be in working order.

If having developed a program we decide to change the method of solution of the differential equation, if the program is sufficiently modular this can be achieved by the simple substitution of one form for another. This can best be seen in a concrete example.

Thus, let us consider a one-dimensional plasma made up of

electron and ion (singly ionized) fluids whose distribution functions satisfy the linearized Vlasov equations appropriate for a collisionless plasma

$$\left.\begin{aligned} \frac{\partial f_e}{\partial t} + \frac{\partial}{\partial x}(v_e f_e) + \frac{\partial}{\partial v}(a_e f_e) &= 0 \\ \frac{\partial f_i}{\partial t} + \frac{\partial}{\partial x}(v_i f_i) + \frac{\partial}{\partial v}(a_i f_i) &= 0 \end{aligned}\right\} \qquad (21)$$

where f_e, and f_i are the electron and ion distribution functions,

v_e, and v_i are the velocities of the local elements of the fluids,

a_e, and a_i are the accelerations to which they are subjected.

The accelerations a_e, and a_i satisfy

$$m_e a_e = - m_i a_i = - eE \qquad (22)$$

where

$$\frac{\partial E}{\partial x} = 4\pi e \int (f_e - f_i) dV \qquad (23)$$

the velocities f_e and v_i satisfy

$$\left.\begin{aligned} \frac{\partial v_e}{\partial t} &= \frac{-eE}{m_e} \\ \text{and} \\ \frac{\partial v_i}{\partial t} &= \frac{eE}{m_i} \,. \end{aligned}\right\} \qquad (24)$$

The plasma is assumed collisionless, and the ions and electrons interact only in as much as each species contributes to the electric field which acts on both of them.

To treat these equations by a finite difference method was advocated by Kellogg in 1965[6]. Of course several other methods have been advocated and implemented for this, e.g. the Waterbag Model (for the history of this see Berk and Roberts 1967[7]), expansion in terms of Fourier components and orthogonal polynomials[8,9] and the

popular sheet/rod model developed by Buneman and Dawson[10,11].

This approach does have its drawbacks[12], but is used here to illustrate the strength of symbolic techniques. Such techniques could be used in the other cases also.

The computational steps are clearly

```
solve Poisson's equation;
ion fluid advected;
electron fluid advected;
```

For a leapfrog scheme, the advection of the equations can be gathered together into a single procedure (Fig.9).

```
'PROCEDURE'LEAPFROG;      'BEGIN'DT=0.5*DELTAT;
    'FOR'I'EQUAL'IP1'STEP'1'UNTIL'IP2'DO'
    'FOR'J'EQUAL'JP1'STEP'1'UNTIL'JP2'DO'
    'BEGIN'DEFINE O;
     NEW FE[O]=FE-DT*(DELX(VE*FE) + DELV(AE*FE));
     NEW FI[O]=FI-DT*(DELX(VI*FI) + DELV(AI*FI));
    'END'; 'END';
```

Fig.9

IP1, IP2, JP1, JP2 are the bounds for I and J. FE is a procedure which returns the value of the electron phase fluid density appropriate for the current lattice point. It should be added that these difference equations, although apparently a forward difference in time, are actually centred in time and space. The procedure DT is set to 0.5*DELTAT where DELTAT is the time step interval. As quantities are defined on a staggered mesh, they are available at the correct time level when required:

Thus in this case we may write simply

```
SOLVE POISSONS EQUATION;
LEAPFROG;
```

```
'PROCEDURE' AUXILIARY CALCULATION; 'BEGIN'DT = DELTAT;

'REAL''PROCEDURE' ELECTRON PHASE FLUX;
     ELECTRON PHASE FLUX = DELX(VE*FE) + DELV(AE*FE);

'REAL''PROCEDURE' ION PHASE FLUX;
     ION PHASE FLUX = DELX(VI*FI) + DELV(AI*FI);

'REAL''PROCEDURE' FE NEW;
     FE NEW = SAV(FE) - DT*ELECTRON PHASE FLUX;
'REAL''PROCEDURE' FI NEW;
     FI NEW = SAV(FI) - DT*ION PHASE FLUX;

FILL   THE AUXILIARY POINTS'.'
'FOR'  I 'EQUAL' IP1 'STEP' 1 'UNTIL' IP2 'DO'
'FOR'  J 'EQUAL' JP1 'STEP' 1 'UNTIL' JP2 'DO'
 'BEGIN' DEFINE O;

     C = 2; NORTH = O+DOY;
          NEW FE[NORTH] = EP(FE NEW);
          NEW FI[NORTH] = EP(FI NEW);
            SOUTH = O-DOY;
          NEW FE[SOUTH] = EM(FE NEW);
          NEW FI[SOUTH] = EM(FI NEW);
     CC = 1; EAST = O+DOX:
          NEW FE[EAST] = EP(FE NEW);
          NEW FI[EAST] = EP(FI NEW);
            WEST = O-DOX;
          NEW FE[WEST] = EM(FE NEW);
          NEW FI[WEST] = EM(FI NEW);
      'END';
```

Fig.10. A 'procedure' AUXILIARY CALCULATION, for use in a 2-step Lax-Wendroff scheme. Procedures FE NEW and FI NEW contain a clear statement of how the first step of the scheme works.

in the Symbolic Equations module.

For a two-step Lax-Wendroff scheme, provisional values at an intermediate timestep must be calculated. The module could take the form of Fig.10.

In this case the Symbolic Equations module contains

```
LAX WENDROFF TWO STEP'.'
     SOLVE POISSONS EQUATION;
     AUXILIARY CALCULATION;
     SOLVE POISSON FOR AUXILIARY VALUES;
     LEAPFROG;
```

Provided the modules have been properly constructed, the change from one numerical difference scheme to another simply requires the replacement of one set by another (e.g. the Lax-Wendroff module is replaced by the leapfrog one). This makes the comparison of the different methods over such matters as speed, gross accuracy, and in particular velocity dispersion, a fairly straightforward matter. With the development of methods such as Fromm's[13] hybrid Lax-Wendroff the close monitoring of methods becomes of greater interest.

CONCLUSIONS

Many programs could be written with less wear and tear on the physicist (and with shorter development times) by adopting methodical techniques of prefabrication such as those described here.

The use of symbolic methods provides a way of defining a physical problem clearly in computational terms. Algol and Algol-like languages are well suited to the symbolic approach especially for the parts of program dealing with the logic and the physical equations; in Fortran a control package for time-dependent problems, and the use of pre-fabrication with acceptance tests has been successfully introduced.

REFERENCES

1. Roberts K.V. and Boris J.P. "Trinity: Programs for 3D Magnetohydrodynamics', IPPS Computational Physics Conference, Culham (1969), paper 44. (Report CLM-CP (1969), H.M.S.O.).

2. Kuo-Petravic, G. Petravic, M. and Roberts K.V., 'The Translation of Symbolic Algol I into Symbolic Algol II by the Stage 2 Macro Processor'. IPPS Computational Physics Conference (1970).

3. Roberts K.V. and Boris J.P. The Solution of PDE Using a Symbolic Style of Algol (to be published).

4. Boris J.P. and Roberts K.V. 'Galaxy', IPPS Computational Physics Conference, Culham (1969), paper 4. (Report CLM-CP, (1969) H.M.S.O.).

5. Michie D., Ortony A., Burstall R.M. Computer Programming for Schools (1968).

6. Kellogg P.J. Phys. Fluids, 8, 102, (1965).

7. Berk H.L. and Roberts K.V. Phys. Fluids, 10, 1269, (1967).

8. Knorr, G. Z. Naturforsch., 18a, 1304, (1963).

9. Armstrong T.P. Phys. Fluids, 10, 1269, (1967).

10. Baneman O. Phys. Rev., 115, 503, (1959).

11. Dawson J.M. Phys. Fluids, 5, 445, (1962).

12. Roberts K.V. and Weiss N.O. Math. Comp., 20, 272, (1966).

13. Fromm J.E. Comput. Phys., 3, 176, (1968).

14. Petravic M., Kuo-Petravic G. and Roberts K.V. 'A Program for the Automatic Production of Computer Codes from Difference Equations'. IPPS Computational Physics Conference (1970).

15. A User's Guide to COTAN: Culham Laboratory KDF9 manual. Section 8. (1968).

Acknowledgements

We are grateful to the following scientists for help and discussion at Culham: J.P. Christiansen, J.E. Crow, M.H. Hughes, M. Nordstrom.

APPENDIX

A UNIVERSAL CONTROL PACKAGE FOR FORTRAN PROGRAMS

Many time-dependent simulation programs are currently being written, and most of these are still programmed in Fortran. Whatever the specified set of differential equations may be, these programs usually have to carry out the same control processes, and the same general steps in the calculation, e.g.

DEFINE INITIALS CONDITIONS
START THE RUN
INITIAL OUTPUT

and so on. Often this part takes longest to write, and is hardest for newcomers to understand.

A Universal Control Package (UCP) is therefore being written at Culham which will contain a main control subroutine MAIN, together with utility and diagnostic subroutines, and which will form the foundation upon which a variety of actual simulation programs can subsequently be built. The package is being written in ASA Fortran, so that it can be used on any computer system with only trivial modifications. Because of this standardization of the structure, it should be easier for collaborating groups to exchange programs.

So far as possible UCP shares a common structure with DUMMYRUN, e.g. the Algol Procedure calls of MAIN CONTROL appear as comments in the UCP routine MAIN. (Fig.11). UCP is however less general, because there are no analogues for the symbolic modules which deal with vector algebra and analysis.

Development and Diagnostics

It has been found useful to 'grow' an actual simulation program from UCP like a tree, checking it out at each stage by means of both standard and ad hoc diagnostic subroutines. Typical examples of

```
C
      SUBROUTINE MAIN
C
C U2   MAIN CONTROL
C
C-----------------------------------------------------------------
      COMMON/COMUCP/
     1      NONLIN,   NOUT,   NPRINT,   NREAD,
     2      NSTART,   NSTEP,  NSTOP,
     3      DELTAT,   T
C-----------------------------------------------------------------
CL              1        PROLOGUE
C
C  LABEL THE RUN
      CALL LABRUN
                                        CALL REPORT(1,1)
C  CLEAR VARIABLES AND ARRAYS
      CALL CLEAR
                                        CALL REPORT(1,2)
C  SET DEFAULT VALUES
     CALL PRESET
                                        CALL REPORT(1,3)
C  DEFINE DATA SPECIFIC TO RUN
     CALL DATA
                                        CALL REPORT(1,4)
C  SET AUXILIARY VALUES
     CALL AUXVAL
                                        CALL REPORT(1,5)
C  DEFINE INITIAL CONDITIONS
     CALL INCOND
                                        CALL REPORT(1,6)
C  START THE RUN
     CALL START
                                        CALL REPORT(1,7)
C  INITIAL OUTPUT
     CALL DSPLAY(1)
                                        CALL REPORT(1,8)
C
C-----------------------------------------------------------------
CL              2        MAIN CALCULATION LOOP
C
     DO 20 NSTEP=NSTART,NSTOP
C
     T=T+DELTAT
C    ADVANCE ONE TIMESTEP
     CALL STEPON
C
C    OUTPUT IF REQUIRED
       CALL DSPLAY(2)
C
  20   CONTINUE
                                        CALL REPORT(1,9)
C
C-----------------------------------------------------------------
CL              3        EPILOGUE
C
      NSTEP=NSTOP
C  FINAL OUTPUT
```

Fig.11 The UCP FORTRAN routine MAIN. The structure is similar to that for MAIN CONTROL in DUMMYRUN (using ALGOL)

standard subroutines are

MESAGE	Print a message of up to 48 characters
IVAR	Print 'NAME = ⟨integer value⟩'
RVAR	Print 'NAME = ⟨real value⟩'

while a useful ad hoc subroutine is

CLIST	List names and values of all common non-subscripted variables in alphanumeric order, using IVAR and RVAR.

These subroutines allow information to be extracted very easily at critical points of a test run by inserting single cards, without the need for format statements. Preferably, all the diagnostic tests are grouped together in a single ad hoc subroutine REPORT, which is called at suitable intervals by the main part of the program (Fig.11). In this way, the program itself remains undisturbed.

For illustration, we consider the development of the FORTRAN version of TRINITY[1]. This is now being generalized so that it can deal with a 60 × 60 × 60 mesh. The 8Mbytes of data will be stored on 2 IBM 2301 drums on an IBM 360/91 configuration, and transferred in and out of the core each timestep, using a rotating quadrupole buffer, a generalization of the triple buffer used in GALAXY[4]. Of these, three sections of the buffer deal with the central plane (O) which is being calculated, and those on either side (N and S) which are needed by the difference scheme. The fourth or 'move' section (M) handles the data transfer. During the first part of the calculation of each plane, data is transferred out from the far-south plane (FS) on to the drums on two separate channels. Halfway through, the direction of data transfer is switched to bring data in for the far-north plane (FN).

The logic of such a scheme is quite complex, since it involves

1. MESH BLOCK

```
DS      =   0.100E 00
I       =      0
J       =      0
K       =      0
L       =      0
N       =      0
NFIPT   =      2
NI      =      2
NIJ     =      0
NJ      =      4
NK      =      6
NW      =      0
PI      =      4
PIJ     =     24
PIMIN1  =      3
PJ      =      6
PJA     =      3
PJB     =      4
PJMIN1  =      5
PK      =      8
PKMIN1  =      7
S       =      0
SE      =      0
SIZE    =      0
U       =      0
W       =      0
```

2. CONTROL BLOCK

```
NONLIN  =      6
NPRINT  =      6
NREAD   =      5
NSTART  =      0
NSTEP   =      0
NSTOP   =      2
DELTAT  =   0.1000E 00
T       =   0.0
```

```
3.  ROUTINE  =  1,  POINT  =  7
    OUTPUT   =  1
    ROUTINE  =  1,  POINT  =  8
```

```
STEP      =     0
S-PLANE   =     2
O-PLANE   =     3
N-PLANE   =     0
M-PLANE   =     1
4.        STORE C-PLANE ON DRUM
C-PLANE   =     1
D-PLANE   =     5
          CALCULATE ROW
ROW       =     2
PLANE     =     2
I-FIRST   =     2
          CALCULATE ROW
ROW       =     2
PLANE     =     2
I-FIRST   =     3
          FETCH D-PLANE FROM DRUM
C-PLANE   =     1
D-PLANE   =     3
```

Fig.12 Output from test runs for TRINITY using a small number of mesh points for which the structure is checked out.

the alternation implied by the leapfrog different scheme, as well as periodicity, guard points in the borders, rotating buffers, swiches in the direction of data flow, and keeping count of the location of 360 separate tracks on the drums. The logic can however be checked out independently of the physics, and, of a large extent, without any actual transfer of data. To do this, we replace those subroutines which do the actual work by dummies, which simply print out messages saying what they are meant to do - a 'rehearsal' for the real calculation, as it were. A small mesh can also be used for the tests, so that not too much printout is generated. Fig.12 shows an example, for which the GO step occupied only 0·28 secs of IBM 360/91 CPU time. The first two sections print out names and values of the variables in the Common blocks COMESH, COMUCP by means of statements

```
CALL CLISTM
CALL CLISTU
```

while section 3 is generated by subroutines called by MAIN. Section 4 monitors the logic of the calculation, using a $4 \times 6 \times 8$ mesh. The output is generated by statements such as

```
CALL IVAR ('ROW     ',J)
CALL IVAR ('S-PLANE ',MCS)
```

Using this type of methodical approach, it is being found that programs can be checked out much more quickly and economically than by the usual methods.