

GENERALIZED SYMBOLIC PROGRAMME FOR THE
ANALYSIS OF INTERLINKED CHEMICAL
ENGINEERING PROCESSES

A thesis submitted for the degree of

Doctor of Philosophy

in the

Faculty of Engineering

of the

University of London

by

ARTHUR WILLIAM WESTERBERG, B.Sc., M.Sc.

Department of Chemical Engineering,
Imperial College of Science and Technology,
London, S. W. 7.

June, 1964.

C7 24 JUL 1964

ABSTRACT

A description is given of a general-purpose computer programme (SPEED-UP) for the analysis and design of complex chemical engineering processes. The programme, which adopts the simulation approach, consists of roughly three sections:

1. specification and analysis of the plant structure,
2. reading of data, and
3. computation.

The process description presented to SPEED-UP is analysed and the best order for computation, including all iteration loops, is derived automatically. The language for specifying all data is designed for the chemical engineer and allows for the unit routines provided to be built onto in a straight-forward manner. Finally, a dynamic reordering, which can be based on a learning procedure to account for past numerical experience, is provided during the computation section.

ACKNOWLEDGMENTS

The author is deeply indebted to Professor R. W. H. Sargent for exceptionally willing guidance, for his many ideas, proddings, disagreements, and in particular for his many encouragements throughout this work. A special thanks is extended to Professor H. G. Denbigh for his interest and for the opportunity he extended to hold an assistant lectureship at Imperial College during my stay here. Miss Lilly Li, who has been a very cheerful and much too hard-working partner with the programming, also deserves my sincerest thanks, and finally the thanks one gives to an extremely patient typist and morale builder must go to my wife Barbara. Without the help of the above people the following work could never have been completed.

A. W. W.

TABLE OF CONTENTS

Title page	i
Abstract	ii
Acknowledgments	iii
Table of Contents	iv
Introduction	1
Scope of SPEED-UP	2
Reading of the Plant Structure	4
Implied Feed-back Streams	5
Arrangement of Units in Order of Computation - An Analysis of Flow-diagrams	7
Algorithm I	9
Algorithm II	11
Stream Weights	18
Data Specification	19
Description of the Instructions	23
Stream Specification	31
Error Detection during Data Specification	33
Correction Facilities for Input Data	35
Computation	40
Optimization	47

Details of the Compiler	50
Symbols and Instructions	50
Data Storage for a Unit	53
Interpretation of a Sample Data Instruction	56
Previous Related Work	58
Future Work	62
References	66
Appendix I, Description of the Lists Occurring in SPEED-UP	69
Appendix II, Algorithms I to IV	77
Appendix III, Algorithms for the Complete Programme	90
Appendix IV, Unit Routine Instructions	105
Appendix V, Structure of Table 2	122
Appendix VI, Structure of Table 3	126
Appendix VII, User's Reference Manual with Glossary	128
Tables 1 and 2	154
Figures 1 to 10	155

INTRODUCTION

It has become commonplace for the chemical engineer to turn to the digital computer for the solution of routine problems in process design, and the literature abounds with computer programmes and techniques for such standard problems as the rating of heat exchangers and the determination of the number of plates in a distillation column. The capabilities of most modern computers are such that the time required for these routine design calculations is determined in large measure by the speed of input and output of data, and this time in turn is in general small compared with the time for preparation of the input data and for interpretation of the results. There thus arises a need for more compact means of presentation of the data and results, preferably in the chemical engineer's own language rather than in terms oriented towards the needs of the computer.

In the design of complete processes these routine calculations are only a small part of the total design effort. The individual unit calculations are interlinked via the process streams, and in the past a final self-consistent design for the whole process has generally been achieved by an iterative procedure in which the successive corrections were left largely to the judgment and intuition of the designer. The convergence

of such procedures is often slow, and much time and effort is required. If one is more ambitious and attempts on overall optimization of the plant performance, the work is much more arduous and this traditional method of attack is usually quite impracticable.

Nowadays optimization is usually treated as a separate problem, and is normally carried out using a greatly simplified model of the process. The parameters of this model must be fixed on the basis of the results of detailed design of the individual units and again an iterative procedure is necessary, which again involves an enormous task of data interpretation and transcription.

This thesis presents in detail the logic for a general-purpose programme called SPEED-UP (Simulation Programme for the Economic Evaluation and Design of Unsteady Processes) which will provide for an integrated approach to the optimum design of complex processes, and thus eliminate much of the need for intervention by the designer in the intermediate stages.

Scope of SPEED-UP

The programme is intended to provide the logical framework of the overall design procedure, and adopts the "simulation" approach. That is, it sets up a mathematical model of

the process from the specification, which can be used as many times as required to evaluate the performance of the plant, either for different input conditions, or for different values of design parameters within the process. It can therefore be used equally well for design or for investigating behaviour of an existing plant if the appropriate data are available.

The complete programme will consist of five main parts:

1. Reading of the plant structure.
2. Arrangement of units in order of computation.
3. Specification of data.
4. Computation of plant performance for fixed values of the inputs and plant parameters.
5. Readjustment of the input and parameter values, and possibly also the structure of the plant, to obtain desired or optimum conditions.

This thesis deals mainly with the first four stages of the programme, and will be generally directed to the case of dynamic simulation rather than steady state design.

READING OF THE PLANT STRUCTURE

The traditional plant flow-diagram is in essence a block diagram providing two kinds of information: the nature of the units comprising the plant, and the interconnections between them. This graphical representation of the plant structure cannot be accepted directly by the computer, and the first task is to find a suitable alternative representation. The type of unit is obviously indicated by its name, and it seems natural to describe each interconnection in terms of the units it links together. To make this latter description specific, it is necessary to give each unit a specific label, and also label all inputs and outputs; the procedure adopted was to number separately the inputs and outputs of a given unit, and to number units of the same kind. Thus a link can be specified by a statement of the type:

INPUT 2 OF HEAT EXCHANGER 14 IS OUTPUT 4 OF PLATE COLUMN 5

An interpretive routine has been written so that the computer will accept such statements directly as input data.

Since every unit has at least one input or output, the names of all the units in the plant appear in the set of link-statements which specify the interconnections. Thus such a set of link-statements provides a complete description of the plant, the order of the statements within the set being immaterial.

The next problem is to store the information contained in the link-statements in the computer in a compact form, suitable for further treatment. There is an extensive literature on the representation and properties of networks, or graphs, (see for example Berge⁽⁸⁾). Various incidence matrices^(8,9) or other matrices associated with the graph^(8,10) have been used to represent them, but for the present purpose these matrix representations are unsatisfactory for two reasons. Firstly, they specify only the number and direction of links between units, and do not contain details of the connections between the individual inputs and outputs. Secondly, they are inefficient in space utilisation. For example, the storage of the adjacency matrix of a plant containing n units requires n^2 storage-locations, while in general the number of links is very much less than this.

A more compact means of storage, which at the same time facilitates subsequent use of the information, is to compile lists of the input-connections to each unit, together with a list of the units. Again, the order in the lists is unimportant and can conveniently be the order of occurrences in the link statements as they are fed to the computer.

Implied Feedback Streams

In the simulation of dynamics some units transmit information

in both directions in the links connected to them; for example, consider a pipe in which the pressure drop determines the flow rate of material through it. The material flows down the pipe, while at the same time pressure information is transmitted back up through the pipe; that is the flow rate of the streams from the unit upstream to the pipe is affected by the pressure in the downstream unit. For this reason a special category of unit types, such as a pipe, can be included in SPEED-UP. Each link specified either to or from a unit included in this special category causes an additional special link in the reverse direction to be inserted to account for the implied feedback.

ARRANGEMENT OF UNITS IN ORDER OF COMPUTATION--AN ANALYSIS
OF FLOW-DIAGRAMS

The object of stage 2 of the programme is to arrange the units of the plant into a sequence of groups of units such that there is only a forward feed of information from group to group, with a minimum number of units within each group. It is easy to see that this provides the order in which the units should be computed so as to minimise the number of preliminary assumptions required.

If there are no recycle-streams and no feed-back of information within the plant, the groups of units referred to above will each consist of a single unit and the sequence provides a complete computation scheme. On the other hand any units contained in a closed loop of information must be grouped together in the sequence to retain the forward feed of information between groups. The computer may be large enough to treat such a group of units as a single entity and resolve all the describing equations of the different units within the group as a single simultaneous set of equations. In this case no iterative solution and no preliminary estimates will be required. However, in general it will be necessary to treat the units one at a time, and the need arises for a systematic method of "breaking" the recycle streams to provide a scheme for computing the units in sequence. Of course this means that the parameters of the inputs represented by the broken recycle streams must first be

estimated, and then an iterative procedure used to match the broken branches of the recycle stream. In general one would wish to order the units in the sequence in such a way as to minimise the number of parameters on which iteration is required.

These procedures are best illustrated by considering as an example the flow-diagram shown in Fig. 1. Here it is obvious that units G and H must be treated together as a group, and the same applies to units E, I, L, D, which form a closed loop. A possible computational sequence is then:

① AB ② F(GH)C ③ M ④ (DEIL)JK

If the computer can handle each of the groups in brackets en bloc, this analysis is sufficient, but otherwise a further ordering of the elements within the groups is required. It is immediately evident that computation of G before H required iteration on 3 parameters, whilst H before G requires iteration on only 2 parameters; HG is therefore the preferred order. Study of the group (DEIL) will reveal that the optimum order which minimises the number of parameters to be estimated is in fact LDEI. The final sequence is therefore:

① AB ② FHGC ③ MLD ④ EIJK

It should be noted that the inputs from the exterior are treated as units, and appear immediately before the unit which

requires them; outputs are irrelevant to the network analysis and do not have to appear in this part of the programme. It may further be noted at this point that the sequence:

① ② ③ ④ AFBHGCMLDEIJK

also satisfies the conditions laid down and therefore forms a possible computational sequence. However, unless the computer has a large single-level store, the first sequence will always be preferable. For example, on completion of the computation of unit A, all the input data for B is readily available in the working store of the computer and immediate computation of B, rather than F, will avoid unnecessary transfers of data between different levels of the computer store. For the same reason, there is no point in introducing information about external inputs until this is actually required. In short, it is desirable that the sequence follows as far as possible the natural tree structure of the plant.

The systematic analysis of flow-diagrams in this way is effected by two algorithms:

Algorithm I, for finding the groups and arranging them in sequence.

Algorithm II, for ordering the units within a group.

Algorithm I

The input stage of the programme has stored a list of the units in an arbitrary order (List 3), and separate lists of the inputs to

each unit (lists of type 4). The essence of Algorithm I is to take a unit from list 3 and form a sequence, or chain, of units by tracing a path backwards from unit to unit using the individual lists of input connections. Such a chain can terminate in one of three ways:

- a) It ends at an input from the exterior.
- b) It reaches a unit which is already a member of the same chain.
- c) It reaches a unit which is a member of a previously formed chain.

In case (a), the path is retraced back along the chain, listing the units in the order encountered to form the final computational sequence, until one reaches a unit with a second input, other than the one used in forming the chain; a new chain is then formed from this point.

In case (b), all the units in the chain following the first appearance of the repeated unit form a closed loop and must be grouped with the repeated unit. This group is henceforward treated as a single unit in the algorithm, and a new list of type 4 is formed for it to give all inputs to any of the units in the group. The chain-forming procedure is then continued from this group.

In case (c), the unit reached will already have been stored in the computational sequence, and the procedure is the same as if an input from the exterior had been encountered.

When all chains emanating from the original unit have been exhausted, the next unused unit on list 3 is taken and the procedure repeated. It is evident that this process will ultimately use all units in the plant, since list 3 is a complete list of the units; furthermore, all units contained in a closed loop of information will be grouped together, and no unit will be so grouped unless it is contained in a closed loop. The groups are therefore of minimum size, and the sequence of groups formed follows the tree-structure of the plant. The detailed steps of the programme which executes Algorithm I are given in Appendix II.

Algorithm II

Any proposed order of computation of the units forming a group implies the breaking of certain recycle streams, with the consequent need for estimating and iterating on the values of the parameters associated with these streams. The object of Algorithm II is to build up a sequence of units which minimises the number of such estimated parameters. It turns out that this can be done with the help of dynamic programming⁽¹¹⁾, for which the first step is to establish a "principle of optimality" associated with the building-up procedure. The required principle can be stated as follows:

"If a sequence of k units is in optimum order, then the subsequence of $(k - 1)$ units obtained by removing the first unit must also be

in optimum order."

Let us denote the i^{th} permutation of the k units in the group under consideration by $S_k(i)$, where i ranges from 1 to $k!$, and the corresponding number of parameters involved in the resulting feed-back streams by $N_k(i)$. If we label an optimum sequence S_k^* , with the number of feed-back parameters N_k^* , we have

$$N_k^* = \min_i [N_k(i)] \quad i = 1, 2, 3 \dots k! \quad (1)$$

To prove the principle of optimality, let us split this optimum sequence into its leading unit, A let us say, and the remaining subgroup of $(k - 1)$ units forming a sequence $S_{k-1}(j)$. Thus we have:

$$N_k^* = N' + N_{k-1}(j) \quad (2)$$

where N' is the number of parameters involved in streams flowing from the sub-group $S_{k-1}(j)$ to unit A . Now N' is obviously invariant to permutations within the sub-group S_{k-1} , so that it immediately follows from (1) and (2) that

$$N_{k-1}(j) = \min_m [N_{k-1}(m)] = N_{k-1}^* \quad (3)$$

for if this were not true, permutation of $S_{k-1}(j)$ to S_{k-1}^* would result in a reduction of N_k^* according to (2), which is contrary to hypothesis. This establishes the proof of the principle.

The principle provides a necessary condition, which must be satisfied by all subgroups of the optimum sequence, and it now remains to devise a procedure which ensures that all sequences satisfying this condition are considered, and the appropriate minimisation

carried at each stage.

Let us suppose that the $\binom{n}{k}$ subgroups of size k out of a total number of n units have been obtained, each arranged in optimum order. For each subgroup we now adjoin each of the remaining units in turn, so generating the set of all possible subgroups of size $(k + 1)$ which satisfy the principle of optimality. However, each possible selection of $(k + 1)$ units occurs $(k + 1)$ times in different permutations in this set, so that the next step is to evaluate the number of feedback parameters associated with each of the permutations; since all the N_k^* are known, this can be done by use of recurrence formulae similar to equation (2). The optimum permutation of each group of $(k + 1)$ units can then be selected, using equation (1), and these chosen permutations constitute the $\binom{n}{k+1}$ subgroups of size $(k + 1)$, each in optimum order.

The procedure may be started by taking the n units themselves as optimum subgroups since these obviously have no feedback parameters associated with them. The programme details for this building-up procedure are given as Algorithm II-C in Appendix II, but two further comments are relevant at this stage:

It may occur that two different permutations of a given subgroup result in the same minimum number of feed-back parameters, so that both are optimum sequences. However, as has already been noted, the number of external links to or from any subgroup is

invariant to permutations within the subgroup, so that both of these optimum sequences will generate an overall optimum sequence, with the same number of feed-back parameters in each case. Only one of the optimum subsequences need therefore be retained during the building-up procedure.

Secondly, for any given subgroup the total number of parameters associated with the links between the units is fixed, and only their distribution between feed-forward and feed-back paths is affected by permutation of the units. Thus minimisation of the number of feed-back parameters is equivalent to maximisation of the number of feed-forward parameters, and it is this latter procedure which is used in the programme because it allows the use of the same list-structure as that adopted for Algorithm I.

As with all applications of dynamic programming, the number of permutations which must be examined is dramatically reduced compared with a systematic examination of all possible permutations, and the relevant figures for various group sizes are given in Table 1. As usual, this advantage is achieved at the expense of a severe storage problem which quickly limits the size of group that can be handled; Table 1 also lists the number of cells required for the algorithm, using ALP list-processing language⁽¹²⁾ on a Ferranti Mercury computer.

The effort to find a more effective optimization technique revealed two very simple properties of networks which can be used to generate an optimum sequence of units for a large class of groups without the help of dynamic programming, and in most other cases greatly reduces the effective size of group to which this latter technique must be applied.

Consider units A, B, C arranged as shown in Fig. 2. The dotted links indicate that A and C form part of a group which links them with other units and perhaps directly or indirectly, with each other; unit B on the other hand is linked only to A and C, being fed from A by streams involving a total number of parameters w_1 , and feeding C by streams involving a total number of parameters w_2 .

The relative positions of A and C in the final optimum sequence will depend on the group as a whole, but the position of B relative to A and C depends only on the numbers w_1 and w_2 , since its position does not influence the other feed-back links in the sequence. There are six possible permutations of the three units which are listed in Table 2 together with the contribution to the total number of feed-back parameters due to the links with unit B.

It is obvious from this table that if A precedes C in the final sequence, the optimum position for B is between the two;

that is, following A and preceding C. It can also be seen that if C precedes A the position of B depends on the relative values of w_1 and w_2 ; if $w_1 > w_2$, then B should follow A, whereas if $w_1 < w_2$, B should precede C, and for $w_1 = w_2$ either position provides an optimum sequence. It follows that this latter rule for determining the position of B provides the correct answer regardless of the relative positions of A and C, and eliminates the need for further consideration of unit B in the group ordering process. Its removal and final correct placing is therefore effected by the following algorithm:

1. Remove B from the group.
- 2a) If $w_1 > w_2$, record that B must immediately follow A in the final sequence.
- b) If $w_1 < w_2$, record that B must immediately precede C in the final sequence.
3. Constitute a new link from A to C with a number of parameters equal to the lesser of the two number w_1 or w_2 .

The second simplification results from consideration of two-way links of the type shown in Fig. 3a. If we subtract a constant, c , from the number of parameters associated with each link as in Fig. 3b, the total number of feed-back parameters in any sequence containing A and B is reduced by the constant c , regardless

of the relative positions of A and B in the sequence. This follows from the obvious fact that only one of the links between A and B is a feed-back link in any given permutation, and either involves the correction due to c .

Since subtraction of the constant c affects the totals of feed-back parameters associated with all the different permutations by the same amount, it does not affect the choice of the optimum permutation which minimises this total. Now, if we choose c to be the lesser of the two numbers w_1 and w_2 , one of the links in Fig. 3b will have the effective number of parameters reduced to zero, and since such a link can never affect the number of feed-forward or feed-back parameters in a permutation it can be ignored in the subsequent analysis. We may therefore replace any two-way link by a unidirectional link with the appropriate nett number of parameters associated with it, for the purpose of finding the optimum computational sequence of a group.

Such replacements may create links of the type shown in Fig. 2, or even destroy the group character of the new effective network. Should this last event occur, a new application of Algorithm I will break the network down into a sequence of smaller effective subgroups. Hence repeated application of Algorithm I. followed by removal of units simply linked as in Fig. 2 by use of the algorithm

given above and replacement of any resulting two-way links by effective unidirectional links may go a long way towards the final ordering of any given group, leaving possibly a residue of small subgroups to which the dynamic programming algorithm must be applied.

An example illustrating the power of this method is given in Fig. 4, and the details of the two additional algorithms required are given as Algorithm 2A and 2B in Appendix II.

Stream Weights

In deriving Algorithm II it has been convenient to speak of minimising the number of parameters associated with each stream, but this is not necessarily what is really required. It may be that certain parameters give rise to long iterations which have very poor convergence properties, while the unit performance is very insensitive to the values of other parameters. Obviously one wishes to minimise the overall iteration time, and one way in which this might be achieved is to weight the different parameters in proportion to the difficulties that they introduce when they occur in an iteration loop. Algorithm II can then be used to minimise the weighted sum, i.e. stream weight, rather than the number of parameters itself.

DATA SPECIFICATION

There are six main types of data required for the complete process calculation:

1. The flow-diagram or process structure.
2. Data on the nature and properties of the materials being processed. (e.g. enthalpies, reaction rate data, particle size distributions).
3. Data giving characteristics of individual plant units (e.g. number of plates in a column, pressure-drop correlations, lagging thickness).
4. Data on feed-streams and other inputs to the plant.
5. a) The initial state of the plant, in the case of simulation of dynamic behaviour
or b) The performance specifications, in the case of design.
6. Information on the results required, their accuracy, and the method of setting them out.

For a general-purpose programme it is desirable to provide maximum flexibility in allowable forms of data, compact presentation, readily understandable instructions with a minimum of symbolism, automatic crosschecking and fault detection, and ease of correction of individual items. To some extent these are conflicting requirements, and SPEED-UP has alternative ways of dealing

with data according to circumstances.

The unit routines in a general-purpose programme will probably contain many refinements and alternatives, only a few of which will be used in any specific case. For example, a distillation column routine may allow for a feed or product-withdrawal on every plate. Thus many of the data will be zero, and to avoid tedious repetition the convention normally adopted in SPEED-UP is that any variables not explicitly specified are automatically set equal to zero. This convention means, however, that the programme cannot check for unintentional omissions in specifications. Partly for this reason therefore, it is possible for the writer of a unit routine to designate certain variables as "special variables," which must then always be specified explicitly, even if zero; the computer will detect omissions of such data and register a fault, indicating what data are missing. This facility is especially useful for data which would normally be specified in functional form, such as enthalpies and equilibrium-ratios whose evaluation is desired several times during the execution of a unit routine.

In SPEED-UP any item of data can be specified either as a number, or as a function defined by an equation, a table, or a subroutine. The functional statements are executed by the computer in the order in which they are entered and therefore a sequence of

functional instructions in which a dependent variable becomes an independent variable in a later instruction is possible. Provided "old" numerical values exist for a variable before the sequence is executed the order may be reversed with the consequence that an "old" value of a variable will be used in a functional statement for another variable before it is set to its new value later in the sequence.

The special variables differ from the ordinary variables other than in the detection of omissions, but this difference is more easily understood when stage 4, the execution algorithm, has been discussed. Briefly, the ordinary variables are "preset" and then presented to the routine as a set of number; in contrast the special variables are not "preset" but are only available to the routine when requested. For this reason a set of functional instructions in which the ordinary variables are defined recursively is allowed, but a set in which the special variables are defined recursively is not allowed. Consider for example the case where a parameter H is defined as a function of X and then X is defined as a function of H . If both X and H are ordinary variables, "old" values are available for X and H before these instructions are encountered, and H is set using the old value of X , and X is then reset using this new value of H ; the recursive definition is in fact ignored. Similarly, it is easy to show that the

computational scheme will always terminate if either X or H is an ordinary variable. If however X and H are special variables, no "old" values exist and an infinite computational sequence, which is obviously impossible to evaluate, would be set up; that is, evaluate H as a function of X , but first evaluate X as a function of H , etc.

The detection of recursive definitions involving special variables is the same network problem as the detection of groups of units in the computation scheme, and thus such an error can be detected. However, the extra programme required would take too much time and space in the computer to justify its existence, and therefore it has been left to the user to make certain no such recursions exist.

Variables may be referred to by name or by several different symbols, and any one of them can be used for the same variable in different contexts if desired. For example, "LIQUID FLOW RATE," "LFR," and "L" can all be legal names for the same data item in a single unit routine; this set of legal names is fixed only by that routine. The format for a general symbol in SPEED-UP is

letters - number - full stop - number ;

the letters alone or the letters followed by a single number may also be used. The first number in the general symbol is used as a

component index and the second as a stage index; for example

X1.3

represents the variable X for the first component in the third stage. Only the second index has been allowed to range in SPEED-UP because of the limitations of the base language, Mercury Auto-code⁽¹³⁾, which has been used, but when SPEED-UP is written in ALGOL⁽¹⁶⁾ or FORTRAN⁽¹⁵⁾ both can easily be allowed to vary.

A range on the second index for the above symbol would be represented by

X1.3/15

and would mean the variables X1.3 to X1.15 inclusively.

Description of the Instructions

Following the link statements, the first data to be specified are the list of all components found in the plant, and a typical list appears in Fig. 5. This list has a twofold purpose; firstly, it provides the order and the number of component flow rates which will be required for each of the links connecting the units, and secondly, it is useful for error detection as will be discussed presently. Each link represents in general the flow of information from one unit to another within the plant, and for a general programme the information is most conveniently stored in a standardised order; therefore for each link, space is allotted for

temperature, pressure, and a flow rate for each possible component in the plant. Some streams may not of course contain all components and these particular flows will be set to zero.

For each unit, table, equation, or routine, a specification sheet indicating all parameters for which values must be given will be included in the programmer's manual for SPEED-UP, and an example of such a sheet is found in the appendix, page 140.

The actual form of allowable instruction for a unit is perhaps best illustrated by considering the sample specifications given in Fig. 6 for a distillation column. The instruction categories for normal data specification have for convenience in this discussion been indicated by a double spacing between them. The first category of instructions which is in fact just the first instruction, simply names the unit, input, table, equation, or routine to which the data following belongs.

The second category of instructions allows input and output numbers to be redesignated, as in the first instance input 5 is changed to input 7 for the remainder of the programme. The exact location of input 7 is set by the unit routine but could have the interpretation that the input is to be on plate 7. This facility allows the programmer to alter the feed plate in this instance without rewriting the set of link statements and reconstituting the computational sequence for the plant, as such an alteration does not affect the network structure of the plant.

The statement

OUTPUT 2 TO OUTPUT 3

has a similar interpretation, and for each change one such statement occurs at this point in the specification. The statement

INPUT 5 TO INPUT 7

followed by

INPUT 7 TO INPUT 5

interchanges the labelling of the two inputs and does not simply change input 5 to input 7 and back again to input 5.

The next category of instructions is for the specification of components, and these instructions are included to allow for the combination of components where convenient so as to reduce the dimensionality of the system or to allow an order of components other than that specified by the list of all components for the complete process. All the data for the unit will be interpreted in the order the components are listed here; for instance X1 will be a parameter associated with water and X2 with methanol.

The instruction

METHANOL (METHANOL + ETHANOL)

is interpreted: Combine in each input stream to the unit the flow rates of methanol and ethanol and treat the result as the input of methanol. The system has thus been reduced by one component, the same device one uses when doing computation by hand for

trace components.

Certain specified components may be split again in each exit stream, proportional to their sum in all input streams directly connected to the exit stream, if the request "split on exit" is added. The full instruction would then read:

METHANOL (METHANOL + ETHANOL) SPLIT ON EXIT .

There are two separate cases to be considered here; the first where several inputs are connected to several outputs as for example in a plate column, and the special case where certain streams enter and leave without mixing with any other streams, as for example the two streams in a conventional heat exchanger. A convention has been adopted for the input and output numbers of units in order that the computer can distinguish between the two cases; all inputs and outputs labelled by a number less than 100 are considered mixed, whereas any unmixed streams have matching input and output numbers greater than 100. As an example, input 101 only joins output 101 in a unit, but inputs 3, 5, and 7 mix and are all connected to outputs 3, 4, and 8.

If for a particular unit all the components listed on the "list of all components" are present and in the same order as on that list the instruction

ALL COMPONENTS

is provided. When used no further component instructions may be entered.

The third category of instructions is the set of stage specification instructions, and these instructions set the number of stages or sections or plates, etc. for the unit under consideration. The stage instructions, together with the component instructions, set the dimensionality of the unit, i.e. the unit will have "n" components and "m" stages. That these dimensions need to be specified before the actual data is entered is reasonable for otherwise no advantage can be made of reducing one dimension to allow an increase in the other in order not to exceed a maximum allowed storage.

The stage statements in Fig. 6 are interpreted to mean exactly what they say; the column has seventeen plates plus a total condenser and a partial reboiler. For steady-state calculations, the reboiler will behave as an additional stage whereas the condenser will not and therefore the column would be treated as an eighteen-stage system.

The next category of instructions is the parameter specification section. This set of instructions may be entered in any order, and they specify all initial conditions such as flow rates, etc. and all unit parameters such as diameters, lengths, etc., according to the type of unit. The allowable forms of instructions fall into

four types: The single parameter, the list, the constant, and the functional type. It is understood that a set of self consistent units, such as the MKS or the FPS system, is used throughout.

Single parameter: An example of this type of instruction is the first in Fig. 6:

$$P = 1 \quad .$$

The variable, in this case pressure, specified by the symbol "P" is to be set to the value one.

List: An example of this type is:

$$X_{1.0/17} = 1.3 \quad 0.95 \quad 0.90 \quad . \quad . \quad . \quad 0 \quad .$$

This instruction is interpreted as a sequence of single parameter instructions starting with $X_{1.0} = 1.0$ to the final $X_{1.17} = 0$, where X_1 in this case would be the liquid mole fraction of the first component, water.

Constant: An example of this type is given by:

$$\text{VAPOUR HOLDUP } 0/17 \text{ SET} = 0.01 \quad .$$

The interpretation is to set the parameters from VAPOUR HOLDUP 1 to VAPOUR HOLDUP 17 to the constant value of 0.01.

Functional: Two typical examples of this type in Fig. 6 are:

$$\text{REFLUX RATIO (T15) GIVEN BY EQUATION 6}$$

$$K_{2.0/17} (P, T_{0/17}) \text{ GIVEN BY ROUTINE 11} \quad .$$

The first specifies a single parameter, REFLUX RATIO, as a

function of the parameter T_{15} , and this functionality is given by Equation 6. The second, however, specifies a complete range of the variables K_2 from $K_{2.0}$ to $K_{2.17}$ as a function of "P" and of T_0 to T_{17} severally. As an example, the exact interpretation for the particular variable $K_{2.3}$ would be the same as if the instruction

$K_{2.3}(P, T_3)$ GIVEN BY ROUTINE 11

were used. The present value of the time as well as up to sixteen constants may be used as the independent variables in functional instructions. Constants are permitted in SPEED-UP for the case where some of the independent variables are to be held at set values; they are of use when a routine, equation, or table, for which more independent variables than are required exists, but the routine etc. will give the desired functionality, if some of the variables are held constant. This may occur, for instance, if a dew point routine, which is a function of compositions, exists for several units; in a particular unit however, one of the components may be missing and its composition may conveniently be replaced by a constant set equal to zero.

The last category of instructions comprises the single element, the asterisk, and has the function of indicating that the data specification for the unit is completed.

Up to this point the data specification has been in terms of a particular unit, a distillation column, to which the form is

obviously not inappropriate. It might therefore help to clarify now the underlying structure of this form of data specification to see how it can be extended to cover more general cases.

The facility of allowing input and output numbers to be changed has obvious implications when one considers the use of SPEED-UP in a more general form which allows for the computer to make these adjustments automatically.

The two dimensions of the system given by the number of components and number of stages can be extended for systems of higher dimensions by allowing several sets of instructions similar to the single set called stage statements. In fact SPEED-UP as it is now permits a second set of instructions similar to the stage statements for setting a second dimension to replace the number of components when that number has no meaning, as for a table, an equation, or a routine.

The instructions in the section for entering parameters can of course be extended to allow for further types, but the important feature here is the ease with which one can add, to the already existing unit routines, special routines to set holdups, heat capacities, levels, etc. To illustrate, in Fig. 6 the instruction

REFLUX RATIO (T15) GIVEN BY EQUATION 6

is in essence specifying temperature control on the reflux ratio.

By such a device controllers operating on a single unit can easily be incorporated into the actual unit routine even though they are not explicitly in the routine itself. However, one must not be misled into thinking that with the functional expression any sort of control can be added to any routine, as this is obviously not the case. The controlled, or dependent, variable must be an actual input parameter for the routine or the routine will simply ignore it.

Finally, it should be noted that functional statements allow for the free interchange of subroutines among the various unit routines so that several units can use the same correlations for equilibrium constants, or plate efficiencies, etc.

Stream Specification

A facility has been provided for the estimation of a stream's initial temperature, pressure, and component flow rates where such an estimation is required or desirable. In the case of dynamic simulation, the particular units and the computational sequence determine which streams should be estimated; the remainder of the streams can be derived and should be to avoid unnecessary inconsistencies.

Only a feedback stream may cause difficulty and require an estimate, and the feedback streams are determined directly by the computational sequence. In addition, the instantaneous outputs from any unit which has capacity are fixed by the state of the unit

and are independent of the instantaneous inputs; however, the outputs from any unit which has zero capacity are, in fact, equal to the instantaneous inputs. For dynamic simulation no unit has zero capacity, but some units, such as short tubes or junctions, have negligible capacity for a reasonable time scale suitable to the other units in the plant, and it is in the case that an instantaneous input to such a unit is an unknown feedback stream that the possibility of making initial estimates for that stream becomes desirable.

When establishing initial conditions the actual network structure of a group reduces if units of high capacity are present; each such unit becomes the head of a chain of units because all of its instantaneous outputs can be determined directly. Only a substructure of "zero" capacity units will remain in any group, and only if this substructure contains a loop will any difficult feedbacks occur for which a first estimate could prove helpful. Therefore, a direct application of Algorithm I to each group after all input links to units of high capacity have been removed followed by the reordering of any remaining subgroups by Algorithm II will yield these residual feedback streams.

For the dual role of simulation for steady-state design, the facility of estimating all feedback streams in each group is

desirable, and in fact the convergence of the iteration loop around a group will very likely depend significantly on these estimates.

The data for the streams to be estimated are entered just after the data for all the units, input streams, tables, equations, and routines. The format of the stream specification is very similar to that of a unit, but no instructions to set dimensions, such as number of components or stages, and no functional instructions are allowed. The reason for the latter restriction is obvious as the streams are direct outputs from the units and cannot therefore be tampered with arbitrarily, but the reason for the first restriction is that the format of data storage for a stream is standardized and is determined completely before this stage of data entry. Each stream has a storage space for temperature, pressure, and for the flow rates of each of the components listed on the list of all components for the plant. Any flow rates not specified are set to zero.

Error Detection During Data Specification

SPEED-UP is at present capable of detecting many of the errors which can occur during data input, and a list of the detected errors can be found on page 129. The types of errors which can occur will now be briefly discussed.

The most obvious mistake is spelling and this error in theory, can always be detected. SPEED-UP, as it now exists, will find almost all spelling mistakes, but because the method of translating a symbol is at present based on the alpha-numeric equivalent sum of the letters in the symbol, any spelling error accidentally yielding an acceptable sum will pass undetected (for further explanation see the section "Symbols and Instructions").

The next class of errors is instruction format errors, and all can be detected in the present SPEED-UP language.

Omissions constitute the next class to be discussed, and these really do pose a serious problem if they must always be detected. If the complete data for a unit is omitted this is easily found by simply tagging each unit which is on the list of all units within the plant as its data is entered; any not entered will be obvious at the end of the complete data input. All tables, equations, and routines at any time during the input in functional statements can simply be listed and again each of them on the list ticked as the data is entered.

Omissions of only part of a unit's data are not so straightforward unless a list of data to be entered is kept for each unit type, and a similar accounting done for each item. However, as mentioned earlier, many data will be zero and it is convenient to

allow intentional omissions to imply that the data are actually zero. For this reason the data have been split into two classes mentioned previously, the ordinary variables and the special variables. All special variables must be entered and a count of the number entered is kept; this number is compared with the total number which should have been entered and if the two do not agree the specific omission will be indicated. On the other hand, deliberate omission of an ordinary variable implies the value of the item is zero.

Finally, if any of the component statements are missing in the data specification for a unit, an error will be indicated. No check is made on the omission of any of the stage statements, however, as deliberate omission is taken to mean that the number of stages is either zero or, as is the case for the instruction

TOTAL CONDENSER

in Fig. 6 , the information may not alter the actual number of stages.

Correction Facilities for Input Data

As stated previously facilities by which errors may easily be corrected are very desirable for a general-purpose programme such as SPEED-UP. The present correction facilities for data input to units, input streams, tables, and equations are of two types; either single items of the data may be corrected or the complete data may be re-entered. Either form of correction consists of an additional entry

headed by the name of the unit (input stream etc.) whose data is to be corrected and terminated by an asterisk as for the original entry. The correction can be inserted into the complete data tape at any point following the original entry but must be added before the point at which stream data are specified.

If any dimension of the unit's (input stream, etc.) data, i.e. either the number of components or the number of stages, is to be corrected, the entire data for the unit must be re-entered, and for this case a complete re-entry identical in form to the original is allowed; the "old" data including any changes in input and output numbers is completely ignored from then on. The "old" data cannot conveniently be corrected because the whole of the storage has to be altered; for instance for an increase in the number of components extra space is required for the compositions, etc. associated with these extra components.

If however only input and output numbers or parameter values or specifications are to be altered, a special facility is available. The data to be corrected is entered in the same manner as for the first entry; but the extra instruction

CORRECTION

is inserted immediately after the unit's (etc.) name, and this instruction causes the computer simply to correct those parts of

the "old" data affected. An error will be registered if either a component or stage statement is entered, but aside from this the instructions follow the same pattern as previously.

The CHANGE statements take on a slightly modified meaning in this case because all corrections will be made with the assumption that the previous statements have been executed. For example, if the instruction

OUTPUT 3 TO OUTPUT 7

was in the original data entry, the statement

OUTPUT 7 TO OUTPUT 5

will result in the original OUTPUT 3 being changed to OUTPUT 5.

Additional CHANGE statements other than those in the original set are also allowed.

The parameter statements may also be corrected but not in an arbitrary fashion, again because of storage allocation problems. Any corrected special variable must be respecified over the same range and by the same instruction type; for example, if originally the special variables K1 were specified by

K1.1/17 SET = 20

a correction statement must also be of the constant type and over the same range. A legal correction would be

K1.1/17 SET = 10 .

This restriction on special variables eliminates the possibility

of correcting any omissions in them by inserting a correction entry, and a complete re-entry of the entire data will therefore be required.

Any variable specified by a functional statement can only be corrected by a functional statement over the same range and with the same ^{or fewer} number of independent variables. Other than these two restrictions any correction in functional statements is allowed; for instance

T1/17(P,Q0/16) GIVEN BY EQUATION 15

may be corrected by the instruction

T1/17(Z0/16, X) GIVEN BY TABLE 3 .

The order of the functional statements is altered by a correction entry as each new statement cancels the original but puts the new entry at the end of the order for execution. Thus a correction entry may be used to alter the order of a chain of execution of functional statements if desired.

No ordinary variable not originally specified by a functional statement may be corrected by one, but it may otherwise be corrected arbitrarily, including the correction of any omissions. For instance, the ordinary variable

TCB 1/17 SET = 100

can be corrected by the three statements

TCB 1/5 SET = 100.8

TCB 6/9 = 95 90 85 70

TCB 10/17 SET = 60 .

Finally, stream data may also be corrected but only by a complete re-entry as the data specification for a stream is so short.

COMPUTATION

Algorithm 10 in the appendix is an explicit statement of the execution algorithm, which is stage 4 of the programme. Stage 4 consists of the successive calculations of the different units of the plant, in the order obtained in stage 2, using the data specified in stage 3. The unit is a basic element in SPEED-UP, and it is assumed that a routine is provided for each unit type which can compute outputs for given inputs and the present state and design parameters for the unit.

Before each unit routine is executed the entering streams are converted from the standard form (explained on page 31) to the order and with the combined component flows specified by the component instructions in the data for the unit. A list is then compiled for the unit routine indicating where all the preset input streams are located and where all output streams should be stored. Next all functional statements specifying ordinary variables are executed thus presetting them; in each case where the value of a special variable is required in a statement it is first evaluated and this value used to set the ordinary variable. The data are now completely set for the unit and the standard routine for the unit type is entered.

During the execution of the routine values of the special variables can be requested as many times as required and these will be stored at any location specified by the unit routine. Iterations to set bubble point temperatures, for example, can obtain new estimates of equilibrium constants as many times as required for convergence.

The final portion of the unit routine produces and stores the output streams in the specified locations; control then returns to the main programme, and all output streams are reconverted to the standard format, components being split where specified, and then stored.

Iteration cycles are automatically set up around each group in the plant, and each of these terminates when the values for the severed streams agree to within a specified maximum error. Each group is computed once through completely for each iteration cycle, and the feedback streams are checked for convergence as they are encountered. If a convergence check fails, no more data other than the outputs for any of the remaining units is stored as only new estimates of the subsequent feedback streams are required for the next iteration.

The problem of obtaining new estimates for the severed streams is one in numerical analysis, and several techniques have been

devised and tested in the literature. The crudest method is successive substitution, i.e.

$$\underline{x}_n = \underline{f}(\underline{x}_{n-1})$$

where the new estimate is simply the computed values from the last iteration step. This method will probably work for dynamic simulation where the time steps are very small; however, one would not wish to use a method which is so crude as to cause a time step smaller than that required for stability by any of the unit routines if possible. A modified direct substitution which weights the last guess with the computed values is also possible, i.e.

$$\underline{x}_n = \underline{\alpha} \underline{f}(\underline{x}_{n-1}) + (\underline{1} - \underline{\alpha}) \underline{x}_{n-1}$$

where $\underline{\alpha}$ is the weighting matrix. Second order convergence is possible for the proper choice of $\underline{\alpha}$, but this choice involves partial derivatives and is therefore extremely difficult to find. Further methods, such as Newton's Method ⁽³⁴⁾ with approximated partial derivatives, the hyperbolic method as mentioned by Heaslet and Lomax, ⁽³²⁾ and the modified Aitken's Δ^2 method ⁽³³⁾ as suggested by Wegstein, ⁽³¹⁾ all of which were tested by Cavett, ⁽³⁴⁾ are also available.

No numerical work has been done with SPEED-UP as no unit routines exist at present; this portion of work constitutes the next step to be taken, and until then a choice as to which of the

various numerical procedures to use remains undecided. The choice will of course depend on the use of SPEED-UP; if it is being used for steady-state calculations a very powerful method which will converge in almost all cases will be required, but for dynamics the requirements may possibly be relaxed.

The final portion of stage 4 is the implementation of group reordering during execution, and one of the studies to follow is the assignment of stream weights. At present there appear to be two possible effects to consider for assigning weights, both of which will undoubtedly need to be taken into account in the final version. The first is the size of the stream in terms of quantity of flow. Obviously if a stream has zero flow it can no longer influence the receiving unit but as flow increases the effect will be more and more pronounced, and to this extent, therefore, the weights should probably be some increasing function of heat and mass flow rates for the streams.

The second effect is the nature of the units into which and from which the streams flow. If a unit has an extremely high capacity, the inputs will have very much less immediate influence on the outputs than for a unit of negligible capacity. In dynamics then input streams to such units should tend to have lower weights than the inputs to the low capacity units.

In addition a learning procedure which modifies the weights of the streams in the light of previous experience with the group might also be devised to help find the best order for computation; this procedure could possibly involve a form of hill-climbing on the weights to minimise the time for computation.

Since the capacities of units as well as the flow rates of the streams vary with time, the best order for computation will also vary, and to take full advantage of this fact a dynamic reordering must be possible. This dynamic reordering, based on a learning procedure in which Algorithm II plays an integral part, will be provided by the final version of SPEED-UP.

A relatively simple learning procedure which still must be tested has in fact been programmed, and it consists of increasing the weights of the cut streams after every n iterations. Each weight is multiplied by a factor which depends on the relative change in the values for the stream between the final two iterations.

The factor is

$$a_j = 1 + b \frac{\sum_i (x_{ij,n} - x_{ij,n-1})^2}{\sum_i (x_{ij,n})^2}$$

where the $x_{ij,n}$ are the parameters in the j^{th} cut stream after each n^{th} iteration, and where b is a constant scaling factor.

If after m steps, where m is a multiple of n , the

iteration has not converged the group is reordered with the help of Algorithm II, and if any of the cut streams in the original sequence has had poor convergence properties its weight will have increased sufficiently to cause it to be a feed-forward stream in the new sequence. The best values for m and n will have to be determined by numerical experimentation.

In general some of the units in a group which has just been reordered will not require computation for the next iteration as their inputs will not have changed from the previous one. Consider for instance a simple loop in which unit A feeds unit B, unit B feeds unit C, unit C feeds unit D, and unit D feeds unit A again. If the original order for computation was ABCD and the new order is CDAB, the values for the stream from B to C have not changed from the previous iteration, and the same is true of the stream from C to D. However, the link from D to A was a feed-back stream in the original group and depending on whether the computed values for it have been stored or not either unit D or unit A will be the first requiring computation in this iteration. If successive substitution is used as the numerical procedure to obtain the new estimates for the cut streams, the values stored will be the actual computed values from the previous step, but for all other methods they will not be. To overcome this problem,

i.e. when successive substitution is not used, the actual computed values for all cut streams in a group must be retained, in addition to the other information stored, for every m^{th} iteration step.

After a reordering then the actual computed values for all streams which were feed-back and are now feed-forward streams are substituted for the averaged or extrapolated values (unless successive substitution is used). Computation commences with the first unit for which one of these streams is an input and continues through the remainder of the sequence as usual. If no such stream exists, i.e. the group order was not altered, computation begins with the first unit in the group.

OPTIMIZATION

No work has yet been done on stage 5 of SPEED-UP, but it should prove helpful to give some consideration to this stage in the light of the method of dealing with the first four stages.

In effect, stages 1 to 4 yield a mathematical model which define the plant completely and provide a functional relationship between the inputs, outputs, and plant parameters. The inputs and outputs need not be actual plant streams. The outputs can be any computed characteristic of interest for a particular unit, including for example gradients or other parameters which may be required for the overall optimization procedure, while it is convenient to treat all imposed conditions on the plant as inputs, and all adjustable conditions as plant parameters.

When using the programme for dynamic simulation, the mathematical model and the specified inputs and initial conditions will provide a complete description of the process, with no free parameters and no overdetermination. On the other hand, when the programme is used for design, the performance specifications fix the values of some of the variables, and perhaps place constraints on the values of others, leaving a number of free parameters which must be determined to provide optimum performance within the limits of the imposed constraints. Performance is usually defined in terms

of the value of some "objective function," and optimization implies finding the maximum or minimum of this function. Even the attainment of specific conditions can be regarded as an optimization problem, where the objective is to minimise the difference between the design behaviour and the desired behaviour of the plant. In this form it may be noted that overspecification is not a problem, unless the specifications impose incompatibly constraints, since the final result merely represents the best compromise possible.

For optimization, a variety of techniques is available, depending on the particular characteristics of the problem. For steady-state design the problem of finding an appropriate set of parameters reduces to a "hill-climbing" problem, and can be dealt with by one of the methods described by Powell^(1,2), the choice depending on whether a reasonable estimate of the objective function gradient is available or not. For optimum start-up problems or optimum behaviour over a time-interval, one is faced with a continuous variational problem of one of the types described by Katz⁽³⁾, and this is probably best tackled numerically by the first method suggested by him and used earlier by Horn⁽⁴⁾ on similar problems. If there is an appreciable number of optimizing variables or functions, the method of decomposition described by Jackson⁽⁵⁾ may well prove useful, and this fits very naturally into

the programme-structure provided by SPEED-UP. A useful comparison of the two main approaches to variational problems is given by Westcott, Florentin and Pearson⁽⁶⁾ while an up-to-date and comprehensive review of both variational and hill-climbing techniques is contained in the book by Rosenbrook and Storey⁽⁷⁾.

There appears to be as yet no published work on a systematic means of changing plant structure in order to meet an optimum criterion, but it is obvious that to facilitate this a simple method of altering the process description is required, and to this end stage 1 of SPEED-UP does provide for the simple addition or removal of units, and for changes in their connections.

DETAILS OF THE COMPILER

Symbols and Instructions

In this section details of the actual compiler portion of SPEED-UP will be given, including in particular the nature of the instruction format and some examples of an instruction's actual interpretation by the programme. A hierarchy of three subroutines exist for reading an instruction, (Algorithm III in appendix II) and these are:

1. a subroutine to read single characters from paper tape input,
2. a subroutine to read a symbol, that is a special sequence of characters on a tape, and
3. a subroutine to read a sequence of symbols which together constitute an instruction.

As stated earlier a complete symbol consists of a series of letters followed by two numbers separated by a full stop, but to be more precise a symbol has the complete format:

[letters] [number .] [number] [terminator],

and any or all of the first three parts may be missing for it to remain a symbol. The terminators are such things as a comma, a bracket, an asterisk, a carriage return, a letter shifting, etc.

An instruction is defined as a series of one or more symbols which appear on one type-written line on a page; thus the final

symbol of an instruction must always be terminated by a carriage return. An example of two instructions split into their composite symbols is given in Fig. 7, and the complete instruction reading algorithm is given as Algorithm III in appendix II.

Each instruction, as it is read into the computer, is split into a set of symbols, and each symbol is represented in turn by a set of four numbers. The first number is the alpha-numeric equivalent sum of the letters defined by $A = 2, B = 3, \dots, Z = 27$, the second is the value of the number occurring before the full stop, the third is the final number, and the fourth is the "value" of the terminator as defined by the tape code for the Mercury computer given in Fig. 8. If any portion (other than the terminator) is missing the number representing it is set of zero; the last four columns of Fig. 7 give the numerical interpretations of each of the symbols in the two sample instructions. It will prove convenient here to refer to the four numbers into which a symbol is split as the four fields of the symbol, field one for the letters, etc. To read a symbol all four fields are set to zero, and the first character on the tape is read. If it is a letter, its alpha-numeric value together with those for all following letters are summed in field one for the symbol. When a character other than a letter is encountered it is tested and if a digit, it and the subsequent digits for the number following are translated and put

into field three. A full stop causes the accumulated number to be transferred to field two allowing for a second number to be read into field three, and finally a terminator causes the routine to end with the value of the terminator stored in field four.

A series of symbols is read until a carriage return terminator is encountered, and at that point a complete instruction has been read.

Each instruction is interpreted completely after it is read by the computer; to illustrate consider the link statement

INPUT 2 OF TANK 4 IS OUTPUT 7 OF TANK 14 .

The symbols are

INPUT 2, OF TANK 4, IS OUTPUT 7, OF TANK 14 .

The alpha-numeric equivalents of the words "OF," "IS," and "OF" are subtracted from the second, third, and fourth symbols respectively leaving:

INPUT 2, TANK 4, OUTPUT 7, TANK 14 .

The programme tests each symbol in turn and finds that tank 14 should be listed on the input list for tank 4; the input and output numbers are 2 and 7 respectively. If either tank 4 or tank 14 have not been mentioned in earlier link statements, it is put onto a list of all units in the plant and an input list started for it. The computer is then ready to read the next link instruction.

At every stage of reading instructions the computer tests for only the set of legal instructions which may be entered at that point; for instance, until the link statements are terminated no tests are made for any other type of instructions, and although what is entered may be legal at some other point in the programme, if it is not legal then, it will simply be faulted as an incorrect instruction.

To understand the interpretation of a data statement for a unit the actual storage of the data must be explained.

Data Storage for A unit

In the following description the details are given for storing a unit's data as it is read. Each of the ordinary variables has an assigned location in a data table, which will be referred to as Table 1, and in which the actual numerical value of the variable is stored either when entered or when computed as the dependent variable of a functional statement. A convention has been adopted for the address of each variable such that this address can be computed by a standardized four constant equation in the number of components and the number of stages, together with the two indices on the variable. The first index is for components and the second is nominally for stages; for example X2.3 is the ordinary variable X for component 2 in stage 3. The variables are stored consecutively

for one component over all stages and then stored in consecutive blocks for all components.

For each ordinary variable a base address in Table 1 can be computed from an equation of the type:

$$\text{Base Address} = \alpha_1 + \alpha_2 C + \alpha_3 N + \alpha_4 NC$$

where C = the number of components,

N = the number of stages, and

α_1 , α_2 , α_3 , and α_4 are constants for each variable type.

The complete address for a specific variable then becomes:

$$\begin{aligned} \text{Address} = & \text{Base Address} + N(\text{Component index}) + \\ & (\text{Stage index}) \end{aligned}$$

where the two indices are contained in the second and third fields for the symbol, e.g. in a system with seven components the variable X2.3 would have a Table 1 address computed as follows:

$$\begin{aligned} \text{X2.3 address} &= \text{Base Address of symbol "X"} + 2(7) + 3 \\ &= \text{Base Address} + 17. \end{aligned}$$

The total number of spaces required on Table 1 is also computed by an equation identical in form to that for calculating the base address of a variable, only in this case the four constants vary for each unit type.

All special variable instructions, component statements, and functional statements for setting ordinary variables are recorded

on a second table, Table 2, for each unit; and the complete structure of Table 2 for a unit is given in appendix V . This second table is constructed with the help of ALP⁽¹²⁾, as these instructions may be of any length and therefore require no fixed amount of storage space. A sufficient description of Table 2 is given in the appendix so no further discussion of it will be included here.

The total number of entries required for each special variable type can be calculated by an equation similar to the above equation for the base address of an ordinary variable, and when the data have been entered completely for a unit the number of entries for each special variable type is compared to the computed number which should have been entered; if the two do not agree an error pointing out the specific omission is indicated for the unit.

For each unit a set of alpha-numeric sums of the legal symbol names for each type of data is required along with the various sets of four constants for calculating addresses, etc. These data are supplied to SPEED-UP with the unit routines which require them, and in this way the programme is independent of the particular routines used. For example, the routine for calculating a heat exchanger may be changed simply by entering the new routine complete with the new symbol data.

Interpretation of a Sample Data Instruction

To illustrate the interpretation of a data instruction consider again the second instruction given in Fig. 7. If L and T are ordinary variables and P is a special variable the interpretation would proceed as follows.

First, the list of legal symbols for all variables is searched to determine the type of the variable L. L will be found as an ordinary variable so the address of L3.0 in Table 1 can be calculated, and the symbol can now be identified solely by this address. The second symbol along with the terminator of the first indicates that a range of variables referred to and by convention the addresses of these variables follow in order from that of L3.0; that is, if L3.0 has a storage address of 350 then L3.1 has the address of 351, etc. The range is noted. The left bracket indicates that the instruction is a functional instruction and what follow are the names of the independent variables. The alpha-numeric sum of letters for the third symbol is located on the list of symbols, and it is found to be a special variable; a note is made of the symbol and that is not ranged, as indicated by the comma terminator. The second independent variable, T1, is also located, and as it is an ordinary variable its Table 1 address is calculated. This variable has a range which is found to be consistent with that of L3, but only the fact that T ranges from T1 is noted. The right

bracket indicates that no more independent variables follow and the last symbol must give the functionality, i.e. a table and in particular Table 3. A final note is made of this functionality, and Table 3 is listed on a list of all tables which must be entered (if it has not already been previously listed). The instruction is translated.

PREVIOUS RELATED WORK

The idea of user-oriented computer languages is of course not new. First there came the mathematically oriented languages like Autocode^(13,14), Fortran⁽¹⁵⁾, and Algol⁽¹⁶⁾, and more recently other languages have appeared, oriented towards the fields of operational research and systems analysis. The "Control and Simulation Language" (CSL) described by Buxton and Laski⁽¹⁷⁾ is a typical example, and is designed to facilitate the programming of problems concerned with operations involving classes of entities and decisions based on multiple conditions. The "General Simulation Programme" (GSP) of Tocher and Owen⁽¹⁸⁾ and the "General Purpose Systems Simulator" (GPSS) of Goruon⁽¹⁹⁾ are both languages for dealing with queuing problems in traffic flow through a system. The main purpose of these languages is to facilitate the description of situations which arise in production planning, and they deal with systems in very generalised terms.

Examples of programmes which solve logical problems are those concerned with critical path planning of projects, such as "Programme Evaluation and Review Technique" (PERT)⁽²⁰⁾ and "Resource Allocation and Multi-Project Scheduling" (RAMPS)⁽²¹⁾. In the chemical engineering field there are already the Kellogg "Flexible Flowsheet" programme⁽²²⁾, the "Generalised Interrelated Flow Simulation" programme (GIFS)⁽²⁵⁾ of the IBM Company, and the

Shell Company's CHEOPS, about which unfortunately nothing yet seems to have been published.

The Kellogg programme deals with the iterative procedure for determining steady-state overall heat and material balances for a process, but starts at the point corresponding to stage 3 in SPEED-UP; the flow-diagram analysis covered by Algorithms I and II must be done by the designer, and the computer presented with a sequence of units. The programme is set up for petroleum processes, for which the necessary physical data is permanently incorporated; it uses a series of standardised data-sheets which permit specification in chemical engineering terms and facilitate the transcription of the data onto punched cards. The GIFS programme is very similar and also requires a defined computational sequence. The physical data form part of the normal input data, which is again arranged on special data sheets, but the unit routines provided are at present more elementary.

Rosen⁽²⁴⁾, following earlier work by Nagiev^(25,26), describes a general method of solving complex steady-state material balance problems which does not require a stipulated computational sequence. He splits the material balance relationships into a set determined by the interlinkages between units of the plant and the individual sets of relationships between inputs and outputs for each unit,

and proposes an iterative solution based on alternation between the two types of equations. However, the system is treated as if there were links between all possible pairs of units and no attempt is made to take advantage of the structure of the network.

The decomposition of complex network problems into a set of smaller problems by taking advantage of the structure was pioneered by Kron⁽²⁷⁾ with his "tearing method." The possibility of matrix representation of networks has already been mentioned, and Ponstein⁽⁹⁾ shows how matrix methods may be used to establish Kron's results. In essence, Ponstein gives an efficient method of inverting a large matrix by use of partitioning, but, as he remarks, the method can only be applied to matrices having an underlying network structure and relies on certain properties of this network. Harary⁽¹⁰⁾ deals with the same inversion problem by seeking a permutation matrix which reduces the given matrix to block triangular form, and he obtains an algorithm for this purpose by considering the associated network ("digraph"). Steward⁽²⁸⁾ has studied methods of solving large set of non-linear algebraic equations by using the underlying graph structure, and obtains the same rules for partitioning as Harary. These partitioning techniques accomplish the same purpose as Algorithm I, but, as pointed out earlier, there are a number of disadvantages associated with using matrix

representations of the plant flow-diagram. Algorithm I has a closer similarity to the algorithm cited by Trakhtenbrot⁽²⁹⁾ for solving the famous "Cretan labyrinth" problem.

There appears to have been only one other attempt in the literature, by Rubin⁽³⁰⁾, to solve the group ordering problem. Rubin provides an algorithm based on systematic interchanges from an initial arbitrary sequence of units. The choice of the units to be interchanged at any stage is governed by the properties of a special matrix, derived from the incidence matrix after each interchange. Unfortunately the method finds only local optima, as was recognized by Rubin himself, who gives a counterexample in the same paper.

FUTURE WORK

At the present time only stages 1, 2, and 3 have been completed; and no doubt completion of some unit routines for stage 4 will reveal many possible improvements in the method of data specification, the procedures for fault detections, and the checking for inconsistencies. These however, are merely refinements in the computer programme, and there are at least three directions in which one might seek more fundamental improvements.

It has already been mentioned that no published work seems to exist on modifying plant structure to achieve optimum performance. This is often done in design work, but again the designer relies on intuition and his general knowledge of process behaviour. It is easy to see how to formalise this in a few specific instances, for example in the use of exchangers to achieve heat economy, but a systematic means of deciding what types of unit might give improvement in any situation would be an enormous step forward.

At present SPEED-UP is based on the simulation technique, which implies that each unit routine makes a predetermined choice of what variables it treats as "inputs," or independent variables. Although there is a definite direction of flow of material through the unit, the flow of information depends on the application; in dynamic simulation the two flows are in the same direction, but in

the design problem specifications may occur on either inlet-streams or outlet-streams. If the unit routine is written for dynamic simulation, each outlet-stream specification creates a backward flow of information, and hence an extra iteration loop in the whole procedure. Of course, it is possible to write different unit routines for each different purpose, but the number of different possible combinations of design specifications for some units makes this a formidable task, not only in writing the routines themselves, but also in providing for automatic selection of the appropriate combination in any given case. It would seem more worthwhile to seek more general ways of writing unit routines, so that the appropriate selection of independent variables for each unit can be made as part of the network analysis.

Finally, much work remains to be done on the various numerical techniques required for the programme. Methods are available for solving sets of ordinary differential equations, such as occur in the dynamics of multi-stage systems, and also for specific types of partial differential equations, which arise in the dynamics of continuous systems, but little experience is available on mixed sets of ordinary and partial differential equations. Steady state design in general gives rise to sets of non-linear algebraic equations, which can be dealt with by any of the methods mentioned

previously, or possibly by hill-climbing techniques. Here experience is mainly empirical, and there is a lack of a proper theoretical basis by which to judge alternative methods. The opposite is true of variational problems, where there is a need for more effective numerical techniques, particularly where the partial derivatives required in the adjoint equations cannot be obtained analytically, and for variable end-point problems such as occur in the study of optimum start-up procedures.

These three directions of research, if successful, will carry SPEED-UP to a higher level of sophistication, but the one direction which should follow immediately is that which deals with assigning weights in the reordering of groups so as to improve the convergence properties of the iteration loops devised for groups. The weights can possibly be chosen by a set of rules which account for flow rates, etc. initially, then modified as computation proceeds by a learning procedure of a type similar to those employed in "hill-climbing" techniques^(1,2,7). Such procedures will add the extra possibility of accelerating convergence by changing the way in which the set of equations is solved in the light of the difficulties encountered, which is precisely what the human designer does when faced with this problem.

Indeed, this is the keynote of SPEED-UP, for it is an attempt to formalise those procedures which a skilled engineer uses to solve the process design problem.

REFERENCES

1. Fletcher, R., and Powell, M.J.D., A Rapidly Convergent Descent Method for Minimisation, The Computer Journal, 6 (2), 163, (1963).
2. Powell, M.J.D., A Rapidly Convergent Method for Finding the Minimum of a Function of Several Variables without Evaluating Derivatives. Read before Institution of Chemical Engineers, Loughborough, Feb., 1964.
3. Katz, S., Best Operating Points for Staged Systems, I & EC Fundamentals, 1, 226, (1962).
4. Horn, F., and Troltenier, U., "Über den Optimalen Temperaturverlauf im Reaktionsrohr, Chem. Ing. Techn., 32, 282, (1960).
5. Jackson, R., A Generalised Variational Treatment of Optimisation Problems in Complex Chemical Plant. Chem. Eng. Sci. (in press).
6. Westcott, J.H., Florentin, J.J., and Pearson, J.D., Approximation Methods in Optimal and Adaptive Control. IFAC Congress, Basle 1963.
7. Rosenbrock, H.H., and Storey, C., Computational Methods in Chemical Engineering, Pergamon Press (in press).
8. Berge, C., The Theory of Graphs and its Applications, Methuen and Wiley, London (1962).
9. Ponstein, J., Matrix Description of Networks, J. Soc. Indust. Appl. Math., 2 (2), 233, (1961).
10. Harary, F., A Graph Theoretic Approach to Matrix Inversion by Partitioning, Numerische Mathematik, 4, 128, (1962). (Actual proof in: J. Assoc. Computing Machinery, 7, 255, (1960).)
11. Bellman, R., Dynamic Programming, Princeton University Press, Princeton N.J., (1957).
12. Cooper, D.C., and Whitfield, H., ALP: An Autocode List-Processing Language, The Computer Journal, 5, (1), 28, (1962-3).
13. Brooker, R.A., The Autocode Programmes Developed for the Manchester University Computers, The Computer Journal, 1, pp 15-21 and 124-127 (1958).
14. Nixon, W.L.B., A CHLIF3 Glossary, University of London Computer Unit, (1963).

15. IBM Reference Manual, 704 Fortran Programming System (1961).
16. Backus et al. Report on the Algorithmic Language ALGOL 60, Numerische Mathematik, 2, 106, (1960).
17. Buxton J.N., and Laski, J.G., Control and Simulation Language, The Computer Journal, 5 (3), 194, (1962-3).
18. Tocher, K.D., and Owen, D.G., The Automatic Programming of Simulations, Proc. International Federation of Operational Research Societies Conference, Aix-en-Provence, pp. 50-67, (1960).
19. Gordon, G., A General Purpose Systems Simulation Programme, Proceedings of the Eastern Joint Computer Conference, Washington D.C., December 1961. (The Macmillan Company).
20. PERT, Program Evaluation Research Task, Phase I Summary Report, Special Projects' Office, Bureau of Ordnance, Dept. of Navy, Washington, July 1958.
21. Resource Allocation and Multi-Project Scheduling. RAMPS Training Text. CEIR (UK).
22. Peiser, A.H., and Kessler, M.M., The Computer Approach to Optimizing Plant Design, Refining Engineer, 32, p. C7, (1960).
23. Dodrill, W.H., Using GIFS in the Analysis and Design of Process Systems, Proc. of 1962 Fall Joint Computer Conference, p. 275, (1962).
24. Rosen, E.M., A Machine Computation Method for Performing Material Balances, Chem. Eng. Progress, 58 (10), 69, (1962).
25. Nagiev, M.F., Material Balance in Complex and Multistage Recycle Chemical Processes, Chem. Eng. Progress, 53 (6), 297, (1957).
26. Nagiev, M.F., The Theory of Complex Recycling Processes of Petrochemical Synthesis, Fifth World Petroleum Congress (1959).
27. Kron, G., Diakoptics, Piecewise Solution of Large Scale Systems, General Electric Company Report Nos. 57GL330, 57GL330-1, 57GL330-12, 1957-8.
28. Steward, D.V., On an Approach to Techniques for the Analysis of the Structure of Large Systems of Equations, SIAM Review, 4(4), 321, (1962).
29. Trakhtenbrot, B.A., Algorithms and Automatic Computing Machines, D.C. Heath & Co., Boston (1963).
30. Rubin, D.I., Generalized Material Balances, Chem. Eng. Progress Symposium Series, 58 (37), 54, (1962).

31. Wegstein, J.H., Accelerating Convergence of Iterative Processes , Commun. Assoc. Computing Machinery, 1 (6), 9, (1958).
32. Heaslet, M.A., and Lomax, H., NASA TR R-116: Numerical Predictions of Radiative Interchange Between Conductive Fins with Mutual Irradiations , National Aeronautics and Space Admin., Washington, D.C. (1961).
33. Hildebrand, F.B., Introduction to Numerical Analysis, 443, 450 McGraw Hill Book Co., Inc., New York (1956).
34. Cavett, R.H., Application of Numerical Methods to the Convergence of Simulated Processes Involving Recycle Loops, Paper presented during the 28th mid-year meeting of the American Petroleum Institute's Division of Refining, Philadelphia, Pa. (1963).

APPENDIX I

DESCRIPTION OF THE LISTS OCCURRING IN SPEED-UP

The list processing language, ALP⁽¹²⁾, has been used extensively in SPEED-UP, particularly for the solution of the unit and group reordering problems. Several lists are required, and the following detailed description for each list is to be used as a reference throughout the remainder of the appendices. In ALP each cell on a list has four fields which will be referred to as the "link", I, J, and K in that order. The first field, the link, contains the name of the next cell on the list or the value zero if the cell is a terminal cell for a list. The remaining three cells may contain either integers or cell addresses (i.e. the names of other lists) as desired by the programmer; addresses may be distinguished from integers.

Most lists are headed by a single cell which has contents structurally different from the remaining cells on the list, and where this occurs a headcell has been defined. The contents of the three fields I, J, and K for the headcell, where it exists, and for any one of the remaining cells are given for each type of list in the following description. Many of the cells can have one of a selection of contents, the choice depending on the type of

item listed; for instance the cell might contain the unit number if the listing is for a unit, or a headcell to another list if for an input stream. Each choice has a separate comment which ends with a full stop and which indicates the contents for the type of item specified.

Lists of type 1

Five lists, all of which are used as a dictionary to recognize and classify units, input stream, output stream, table, and equation names during the input of instructions.

List 1 a: list of units with unidirectional links.

List 1 b: list of units with implied feed-back links, such as a pipe with flow dependent on both upstream and downstream pressures.

List 1 c: list of input streams.

List 1 d: list of output streams.

List 1 e: list of table and equation names.

Structure

Headcell: Blank.

Typical cell: I = Alpha-numeric sum for name.

J = Address of list 2.

K = Capacity code (zero indicates a unit type which has a high capacity).

Lists of type 2

One list for each type of item listed on lists 1. A separate list 2 exists for all heat exchangers, for all tanks, for all plate columns, etc.; each unit, input stream, etc. mentioned during a link statement is listed on its appropriate list 2. (As a result items on list 1e above only have a headcell.)

Structure

Headcell: I = Address of printing routine.
 J = Address of alpha-numeric table for parameter
 reading routine.
 K = Code number for computing routine.

Typical cells: The cells on this list are in pairs.

First cell: I = Unit number.
 Input stream number.
 J = Address of input list (List 4) for unit.
 Input stream number.
 K = Address of parameter table (Relative).

Second cell: I = Blank.
 J = Number of outputs from unit.
 K = Blank.

List of type 3

One list, an unordered list of all units in the plant excluding input streams; list 3 is constructed during the input of the link

statements and replaced by list 6 during the execution of Algorithm I.

Structure

Headcell: Blank.

Typical cell: I = Address of list 4 for unit.

Lists of type 4 (Input lists)

One list for each unit or group. Each list of type 4 contains all units and input streams which feed the unit or group associated with the list. These lists are constructed for all units as the link statements are read into the computer, and for all groups during the execution of Algorithm I.

Structure

Headcell: I = Address of headcell to list 2 for unit type.

 0 for a group.

 J = Unit number.

 Address of list 5 for group.

 K = Tag 0 Normal unit or group.

 1 Scanned during Algorithm 1.

 Address of group list 4 if part of a group.

Typical cell: I = Feed unit's list 4.

 Input stream number.

 J = Output number.

 Address of headcell to list 2 for input streams.

K = Input number for unit.

0 if a group.

Lists of type 5

One list for each group. List 5 is the list of units in the group and is constructed during the execution of Algorithm I.

Structure

Headcell: I = Address of list 4 for the group.

Typical cell: I = Address of list 4 for unit.

An expanded version of list 5 is used during the group reordering algorithm, Algorithm II; the structure here is similar to the structure of list 6 for the entire plant, but only units are on these lists.

structure

Headcell: I = List 5 cell one level down (actual listing of the group or subgroup).

J = Number of units in subgroup.

K = Indicator.

Typical cell: I = List 4 for unit or subgroup.

J = List 2 for unit.

0 initially for subgroup but list 5 headcell one level down during reordering.

0 for fullgroup at all times.

K = Unit number.

List 5 for group or subgroup.

List of type 6

One list. It is constructed during the execution of Algorithm I, and its contents become the "final ordered sequence" of units, input streams, and groups within the plant.

Structure

Headcell: Blank.

Typical cell: I = Address list 4, unit or group.

0 for input stream.

J = Address of list 2 for unit, input stream.

0 for group.

K = Unit or input stream number.

List 5 for group.

List of type 7

One list. It exists as a working list during the execution of Algorithm I only.

Structure

Headcell: Blank.

Typical cell: I = Address list 4 of unit, group.

Input stream number.

J = Address of cell in list 4 under examination.

K = 0 for unit, group.

Address of list 2 for input stream.

Lists of type 8

One list for each group. List 8 is a list of units with a
 input and a single
 single/output removed during the execution of Algorithm 12A.

Structure

Headcell: Blank.

Typical cell: I = List 4 address of unit removed.

J = 1 Unit in I follows unit in K.

2 Unit in I precedes unit in K.

K = List 4 address of unit.

Lists of type 9 and 10

These lists are created and destroyed during the execution of Algorithm IIC, the dynamic programming algorithm. List 9 contains the names of all subgroups of the size under consideration at any given time, while list 10, a similar list which is being constructed, contains subgroups of a size one unit larger.

Structure

Headcell: Blank.

Typical cell: I = Address of input list (list 11).

J = Sum of weights for feed-forward streams
 within the group.

K = Equivalence number of group.

Lists of type 11

One list for each unit and subgroup during the execution of Algorithm IIC. These lists are modified input lists.

Structure

Headcell: I = Address of permutation list (list 12)
 for subgroups.

 Address of list 4 for unit.

Typical cell: I = Address of list 11 for unit (i.e. subgroup
 of size 1).

 J = Weight of link.

 K = Equivalence number for input unit.

Lists of type 12

Permutation lists for the subgroups during the execution of Algorithm IIC.

Structure

Headcell: None.

Typical cell: Address of list 4 for unit.

APPENDIX II

ALGORITHMS I TO IV

Algorithm I

Algorithm I reorders the units (and input streams) of a plant into a sequence of units and groups of units having only a forward-feed of information. The list processing language, ALP⁽¹²⁾, was used to facilitate the construction of the algorithm, and lists 3 through 7 are required.

Explicit Statement of Algorithm I

- I START.
- II Remove top item list 3. If list 3 is empty EXIT.
- III Consider item:
 - A. Part of group, replace by group and repeat step III.
 - B. Item tagged, repeat from step II.
 - C. Item not tagged, put at top of list 7. Continue.
- IV Read (do not remove) top item of list 7; repeat from step II if empty.
- V Consider item.
 - A. Item is input stream, read next item on list 3 and repeat step V.
 - B. Item is a unit or group, continue.
- VI Read next input to item on its list 4. If no more inputs continue from step IX, otherwise continue step VII.

VII Consider input.

- A. Input stream. Put at top of list 7 and repeat from step VI.
- B. Unit or group, consider.
 - 1. Part of group; replace by group and repeat step VIIB.
 - 2. Tagged; repeat from step VI.
 - 3. Not tagged.
 - a) Not listed on list 7. Put at top of list 7 and repeat from step IV.
 - b) Listed on list 7. Continue.

- VIII A. Remove all items on list 7 down to and including input.
- B. Form a list 5 for group of these items excluding input streams.
- C. Form input list (Type 4) for group.
- D. Put group at top of list 7.
- E. Repeat from step IV.

- IX A. Tag item.
- B. Remove top item from list 7 and put at bottom of list 6.
- C. Repeat from step IXB if item is an input stream.
- D. Repeat from step IV.

Algorithm II

Algorithm II reorders all groups in the plant to minimise the sums of weights associated with feed-back streams. The main algorithm directs the handling of each group in turn by four subroutines whose

repeated application finally results in the group being reordered. The first subroutine removes all units from a group which have a single input and a single output, and the second reduces all two-way links to nett unidirectional links. The third algorithm is in fact Algorithm I, but this time it is applied only to a group list (rather than to a list of all units in the plant) after all possible two-way links and single input-output units have been removed; thus a sequence of units and subgroups of units from within the group will be formed. The three algorithms are applied repeatedly to each of the subgroups formed by the above procedure until residual subgroups remain which only the dynamic programming algorithm can reorder.

These residual subgroups are reordered and the entire ordered group reconstructed; the procedure is repeated for all groups in the plant.

The four subroutines are described by:

Algorithm IIA Remove single input-output units,

Algorithm IIB Reduce two-way links to nett links,

Algorithm I As before, and

Algorithm IIC Dynamic programming algorithm.

Explicit Statement of Algorithm II

I START.

II Consider list 6 for the entire plant.

- III Set "level count" to zero.
- IV Locate next subgroup on present list.
 - A. No more subgroups, continue from step XI.
 - B. Establish a modified set of input lists (type 4) for each unit in the subgroup; retain only inputs from other units within the subgroup.
 - C. Proceed to step V if present level is not zero.
 - D. Establish a list B for the group.
 - E. Remove all two-way links and condense all multiple one-way links in group.
- V Remove all single input-output units and resulting two-way links from subgroup.
 - A. No two-way links removed in either steps IV E or V, continue from step VIII.
 - B. Proceed to step IX if group is completely reordered.
 - C. Otherwise, continue.
- VI Apply Algorithm I to subgroup. If no reduction results continue from step VIII.
- VII Redefine as the list under consideration the present list 5 of the subgroup just reordered in step VI. Increase the level count by one and repeat from step IV.
- VIII Apply dynamic programming algorithm to subgroup.
- IX
 - A. Continue from step X if level count is zero.
 - B. Insert reordered subgroup into listing on list 5 of the subgroup "one level" down.
 - C. Decrease level count by one.
 - D. Repeat from step IV.

- X Remove each entry in turn from group's list 8 and insert unit into specified location on group's list 5. Repeat from step IV.
- XI If level count is not zero repeat from step IX; otherwise EXIT.

Algorithm IIA

In algorithm IIA all units with a single input and a single output are removed from a group.

Explicit Statement of Algorithm IIA

- I START
- II Search group for unit with only one input from and only one output to other units within the group. If no such unit exists, EXIT.
- III Remove the unit from the group list 5.
- IV Consider units A, B, and C. (Labelling refers to Fig. 2.)
 - A. Unit A and unit C are same unit.
 - 1. $w_1 \geq w_2$. Put unit B at head of list 8 and record that it must follow unit A. Repeat from step II.
 - 2. $w_2 > w_1$. Put unit B at head of list 8 and record that it must precede unit C. Repeat from step II.
 - B. Unit A is not unit C.
 - 1. Unit A does not directly feed unit C.
 - a) $w_1 \geq w_2$. Create a link from unit A to unit C with weight w_2 and repeat from step IVA1.
 - b) $w_2 > w_1$. Create a link from unit A to unit C with weight w_1 and repeat from step IVA2.

2. Unit A does directly feed unit C.

a) $w_1 \geq w_2$. Increase w_3 by w_2 and repeat from step IVA1.

b) $w_2 > w_1$. Increase w_3 by w_1 and repeat from step IVA2.

Algorithm IIB (See Fig. 3.)

In algorithm IIB two-way links are reduced to nett unidirectional links.

Explicit Statement of Algorithm IIB

- I START.
- II Consider next unit in group (call it unit A). If no more exist EXIT.
- III Consider next input unit to unit A (call it unit B). If no more inputs repeat from step II, otherwise continue.
- IV Is unit A listed on input list for unit B?
 - A. No. repeat from step III.
 - B. Yes, continue.
- V Compare w_1 and w_2 (See Fig. 3a).
 - A. $w_1 > w_2$.
 1. Decrease w_1 by w_2 .
 2. Remove link from unit B to unit A.
 3. Repeat from step III.
 - B. $w_2 > w_1$.
 1. Decrease w_2 by w_1 .

2. Remove link from unit A to unit B.
 3. Repeat from step III.
- C. $w_1 = w_2$.
1. Remove link from unit B to unit A.
 2. Repeat from step VB2.

Algorithm IIC

In algorithm IIC a group of units is reordered, with the help of dynamic programming, to maximise the total number of parameters associated with feed-forward links. Two definitions are required to understand the statement of the algorithm.

Weight: A weight is the (weighted) number of parameters associated with a link.

Equivalent subgroups: Two subgroups of units are equivalent if they contain the same units.

Explicit Statement of Algorithm IIC

- I START.
- II Create a fictitious unit which has as its input list the list of units within the group. Set the weights of these created links to zero, and create list 9 with this fictitious unit as its sole member.
- III Set the present subgroup size to zero.
- IV Create a new empty list 10.
- V Consider the present subgroup size.
 - A. Present size is equal to size of group. The top subgroup on list 9 is a best permutation for the group. EXIT.

- B. Present size is less than size of group. Continue.
- VI Consider the next subgroup (Call it subgroup X) on list 9. If list 9 is empty relabel list 10 as list 9, increase the present subgroup size by one, and repeat from step IV, otherwise continue.
- VII Consider the next input unit in list 11 of subgroup X (call it unit A). If subgroup has no more inputs repeat from step VI, otherwise continue.
- VIII Create a permutation list of type 12 for a new subgroup Y by putting unit A ahead of subgroup X.
- IX Determine the sum of the weights in feed-forward links in subgroup Y. (This sum is conveniently calculated from an equation similar to equation 3, page 12)
- X Compare subgroup Y to all other subgroups on list 10.
- A. Subgroup Y is equivalent to another subgroup. Compare the sum of weights for feed-forward links for the 2 subgroups.
1. The sum is greater for subgroup Y. Replace the other subgroup by subgroup Y on list 10. (Note that the input list to the two subgroups is the same so it need not be formed again.) Repeat from step VII.
 2. The sum is less or equal for subgroup Y. Repeat from step VII.
- B. Subgroup Y is not equivalent to another subgroup. Continue.
- XI Put subgroup Y on list 10.
- XII Form the input list of type 11 of units feeding subgroup Y. (This is conveniently done by combining the input lists for subgroup X and unit A. In as much as all units not in subgroup X are on its input list one need only duplicate this list except the entry for unit A. Each input unit to unit A is then checked against this duplicate list and if present the weights are summed to combine the links, otherwise the unit must be in subgroup Y and is ignored. The completed list is the input list for subgroup Y.) Repeat from step VII.

Algorithm III

Algorithm III reads one instruction on a data tape, and splits it into a sequence of symbols, the last of which is terminated by a carriage return or an equal sign. Two subroutines are used; the first reads the next relevant character on a tape, and the second reads one complete symbol translating it into the four fields described for a symbol on page 51. These two subroutines are described by Algorithms IIIA and IIIB respectively.

Explicit Statement of Algorithm III

- I START.
- II A. Set the tape read index, "N", to zero.
B. Set the symbol counter to zero.
- III Read the first symbol (Algorithm IIIB).
A. Symbol is a carriage return only, repeat III.
B. Otherwise, continue step V.
- IV Read the next symbol (Algorithm IIIB).
- V Store symbol.
- VI Increase symbol counter by one.
- VII Check symbol terminator.
A. Carriage return or equal sign, EXIT.
B. Otherwise, repeat from step IV.

Algorithm IIIA

This algorithm reads the next relevant character, ignoring erases, space characters, etc. on a tape.

Explicit Statement of Algorithm IIIA

- I START.
- II Read the next character on tape and modify by adding the contents of the tape read index, N.
 - A. Character is one to be ignored, repeat II.
 - B. Character is letter shift, set N to 32 and repeat II.
 - C. Character is figure shift, set N to zero and repeat II.
 - D. Otherwise, EXIT.

Algorithm IIIB

Algorithm IIIB reads one symbol from the tape accumulating the alpha-numeric sum of the letters in field 1 of the symbol, the two numbers in the symbol in fields 2 and 3 respectively, and the value of the terminator in field 4. Algorithm IIIA is used as a subroutine.

Explicit Statement of Algorithm IIIB

- I START, clear the four fields of the symbol.
- II Read next relevant character (Algorithm IIIA).
 - A. Letter, proceed step III.
 - B. Otherwise, proceed step IVA.
- III Add alpha-numeric equivalent of letter to the first field and repeat step II.

- IV Read next character (Algorithm IIIA).
 - A. Digit, proceed step V.
 - B. Full stop, proceed step VI.
 - C. Terminator, put value in field 4, EXIT.
- V Set field 3 = $10 \cdot (\text{Present contents}) + \text{digit}$, and repeat from step IV.
- VI Transfer contents of field 3 to field 2, clear field 3, and repeat from step IV.

Algorithm IV

During Algorithm IV the cells on Table 3, which are dependent on the computational sequence (see page 126), are completed for all units and input instreams. Table 3 includes the address of the data table for the next item in the sequence together with an indicator stating whether the item is in a group, and if it is also whether it is the first or last unit. For the last unit of a group the address of the data table for the first is included in case a further iteration is required, and all feed-back streams are tagged.

Explicit Statement of Algorithm IV

- I START, consider List 6 at beginning.
- II Locate and classify first item on List 6 (Algorithm IVA).
- III Store classification and address for storing item's data during the execution algorithm (cells 2 and 8).
- IV If item is not part of a group, continue from step VI.

- A. If unit is last in a group, continue from step V.
 - B. Consider next input to unit.
 - 1. No more, continue from step V.
 - 2. Feed-back stream, tag.
 - 3. Not a feed-back, continue.
 - C. Repeat from step IVB.
- V
- A. Locate and classify next item in computational sequence (Algorithm IVA).
 - 1. No more, store address of first item in entire sequence, tag (cell 1).
 - 2. Store address of data table for next item (cell 1).
 - B. Store table 3 of present item.
 - C. Repeat from step III if not last item in sequence, otherwise EXIT.

Explicit Statement of Algorithm IVA

- I START.
- II Locate next item on list under consideration.
 - A. No more, continue from step V.
 - B. Unit or input stream, continue from step III.
 - C. Group, continue from step IV.
- III Consider item.
 - A. Part of a group.
 - 1. First unit in a group, set classification indicator to -3.

2. Last unit in a group, set classification indicator to address of data table for first item in present group.

3. Otherwise, set classification indicator to -2.

B. Not part of a group.

1. Unit, set classification indicator to zero.

2. Input stream, set classification indicator to -1.

C. EXIT.

IV Consider list 5 for the group, repeat from step II.

V Consider list.

A. List 6, set indicator to indicate previous item was the last item of entire computational sequence, EXIT.

B. List 5 for a group, consider again List 6, repeat from step II.

APPENDIX III

ALGORITHMS FOR THE COMPLETE PROGRAMME

Overall Algorithm for the Complete Programme

- 1 Read link statements.
- 2 Reorder plant into a computational sequence of units and groups of units.
- 3 Assign capacity weights to all streams and reorder groups to determine feed-back streams for which initial estimates should be entered; print request list for these streams.
- 4 Read data for all units and input streams.
- 5 Assign permanent storage addresses for all streams.
- 6 Read data for all estimated streams.
- 7 Set Table 3 for all units (etc.) for computational sequence established in Step 3 and compute all initial stream values.
- 8 Estimate initial stream weights.
- 9 Reorder all groups and set Table 3 of all units (etc.) accordingly.
- 10 Execute computation algorithm for normal time steps until simulation is completed.

Step I

During step 1 all link statements, which define the plant and its structure, are read by the computer. List 3, all lists of type 4, and all lists of type 2 are completed during this step.

Algorithm for Step 1

It will prove convenient here to define four cells, in which the information from each link statement is stored, as follows:

Cell 1	Contents:	name of receiving unit,
2		name of source unit,
3		input number,
4		output number.

Explicit Statement of Algorithm for Step 1

- I START.
- II A. Clear the four cells and set the unit type indicator to one.
B. Read the next instruction (Algorithm III), no more, EXIT.
- III Consider the next symbol.
 - A. No more, continue from step X.
 - B. Name of unit with unidirectional links, continue from step IV.
 - C. Name of unit with implied two-way links, set unit type indicator to two, and continue from step IV.
 - D. Input stream name, continue from step V.
 - E. "INPUT", continue from step VIII.
 - F. "OUTPUT", continue from step IX.
- IV A. Continue from step VI if unit has been mentioned in previous link statement.
B. Create an input list of type 4 for unit.
C. Put name of unit on List 7.

- V Put name on appropriate List 2.
- VI Subtract alpha-numeric equivalent of "IS" from the next symbol if it exists.
- VII A. Consider cell 2.
 - 1. Not clear, consider cell 1.
 - a) Clear, put name in cell 1.
 - b) Not clear, put name in cell 2.
 - 2. Clear, put name in cell 2.
- B. Repeat from step III.
- VIII A. Store number in cell 3.
- B. Consider cell 2.
 - 1. Clear, set equal to -1.
 - 2. Not clear, continue.
- C. Continue from step IX.
- IX A. Store number in cell 4.
- B. Consider cell 2.
 - 1. Value is not set to -1, copy contents of cell 2 into cell 1.
 - 2. Otherwise, continue.
- C. Subtract the alpha-numeric equivalent sum of "OF" from the next symbol.
- D. Repeat from step III.
- X Insert name of source unit (cell 2) on input list for receiving unit (cell 1). Record the proper input and output numbers.

XI A. Consider unit type indicator.

1. Present value is two, insert name of receiving unit on input list of source unit, code input and output numbers and store.
2. Present value is one, continue.

B. Repeat from step II.

Step 2

Step 2 reorders the units and input streams in the plant into a sequence of units and groups of units with only a forward feed of information; the algorithm for this step is Algorithm I.

Step 3

Step 3 reorders each of the groups in the plant using Algorithm II. All inputs to units with capacity are removed from the group structure as well as all implied links from units listed on List 1b. The remaining streams, i.e. ones to units with negligible capacity, are weighted according to a preassigned value specified by the unit routine. The setting of initial values to the streams will follow the sequence derived in this step so all feed-back streams remaining after the reordering will require an iteration loop, and a request list is issued to indicate these streams so a preliminary estimate may be entered for them in Step 6.

Explicit Statement of Algorithm for Step 3

- I START.
- II Consider next unit type listed on Lists 1.
 - A. No more, continue step IV.
 - B. Capacity code for type is zero, remove all inputs to all units of this type. (Zero code indicates a high capacity).
 - C. Capacity code is not equal to zero, weight all inputs to units of this type with the capacity code value except remove all implied feed-back streams from units listed on List 1b when encountered.
- III Repeat step II.
- IV Execute group reordering programme.
- V Print the names of all residual feed-back streams from reordered groups.
- VI EXIT.

Step 4

Step 4 reads and stores all the data which must be specified for all the units and input streams, including all tables and equations.

Explicit Statement of the Algorithm for Step 4, the Data Read Algorithm

- I START, set level indicator to 1.
- II Read next instruction (Algorithm III).
- III Consider terminator to instruction.
 - A. Equal sign, continue step IV.
 - B. Carriage return, enter at step corresponding to the present value of the level indicator.

IV Variable data statement.

A. Consider type of variable specified.

1. Special variable.

a) Consider present value of level indicator.

- i. '9', reserve storage space for data.
- ii. '12', set storage space available as that used for old data.

b) Read data and store.

2. Ordinary variable, read and store in Table 1.

B. Repeat from step II.

1) Instruction is "LIST OF ALL COMPONENTS". Set level indicator to 2 and repeat from step II.

2) A. Consider instruction.

1. Asterisk, set level indicator to 3.
2. Component name, list on list of all components.

B. Repeat from step II.

3) A. Consider instruction.

1. Asterisk, EXIT.
2. Name of unit, input stream, table, or equation.
 - a) Entered before, set level indicator to 4.
 - b) Not entered before, set level indicator to 3.

B. Repeat from step II.

- 4) Consider instruction.
 - A. "CORRECTION", re-enter old data and set level indicator to 10, repeat from step II.
 - B. Otherwise, indicate old data no longer required, set level indicator to 5, and continue at level 5.
- 5) If data is for a table or equation, set level indicator to 8 and continue at level 8, otherwise consider instruction.
 - A. "CHANGE", set level indicator to 6 and repeat from step II.
 - B. Otherwise, continue at level 6.
- 6) A. Consider first symbol.
 1. "OUTPUT" or "INPUT", list indicated number change.
 2. "ALL COMPONENTS", set level indicator to 8, and indicate that the list of components is the same as the list of all components for the plant.
 3. Instruction is "COMPONENTS", set level to 7.B. Repeat from step II.
- 7) Individual component name, list on component list for unit. If last component set level indicator to 8. Repeat from step II.
- 8) Consider instruction.
 - A. Stage instruction (always less than 3 symbol in length), alter the number of stages accordingly, and repeat from step II.
 - B. Otherwise, set level indicator to 9, and continue at level 9.
- 9) A. Consider instruction.
 1. Asterisk, end of data, store data, set level indicator to 3.
 2. Functional data statement, record on Table 2.B. Repeat from step II.

Correction Facilities:

- 10) If data is for a table or equation, continue at level 11, step B.
Consider first symbol.
 - A. "CHANGE", set level indicator to 11, repeat from step II.
 - B. Data statement, set level to 12, continue at level 12.
- 11) Consider instruction.
 - A. Two symbols in length, continue at level 6, step A1.
 - B. Otherwise, set level indicator to 12 and continue at level 12.
- 12) Remove old listing for instruction on Table 2, continue at level 9.

Step 5

A third table, Table 3, for each unit, input stream, etc. is constructed as indicated on page 126. The information which is tagged, such as the addresses for the inputs and outputs, is permanent throughout the entire programme, whereas the remaining information, such as the address of the data table for the next unit in the computational sequence, is temporary because it is dependent of the computational sequence indicated by List 6.

Step 5 fixes all the permanent addresses for all streams and determines the maximum reserve storage required for any of the groups, as a group's data is not stored permanently over that of the previous time step until the iteration on the feed-back streams converges. The remaining permanent data in Table 3 for all units (etc.) are also fixed at this time.

Explicit Statement of Algorithm for Step 5

- I START.
- II Set the "maximum group storage cell" to zero, consider List C at the beginning.
- III Next item on list.
 - A. No more, continue at step IX.
 - B. Unit, set counter on number of inputs to zero, continue from step IV.
 - C. Input stream, continue at step VI.
 - D. Group, continue at step VIII.
- IV Next input to unit.
 - A. No more, continue at step VII.
 - B. Input stream or unit, continue.
- V A. Assign next stream address.
 - B. Store address on output list for feed item.
 - C. Increase counter on number of inputs to unit.
 - D. Repeat from step IV.
- VI A. Set number of inputs to zero and number of outputs to one.
 - B. Set address cell for input streams to zero (cell 8).
- VII A. Store computing and printing codes (cells 5 and 6).
 - B. Store number of inputs and outputs (cells 3 and 4).
 - C. Store Table 3 for item.
 - D. Increase "group store cell" by storage requirements of present item.

- E. Repeat from step III.
- VIII A. Set contents of "group store cell" to zero.
- B. Substitute group's list 5 for list under consideration.
- C. Repeat from step III.
- IX Consider list.
- A. List 5 of a group.
 - 1. Set contents of the "maximum group storage cell" to $\text{Max}(\text{present contents, contents of group store})$.
 - 2. Re-enter List 6 as list under consideration.
 - 3. Repeat from step III.
- B. List 6, reserve storage for largest group (number of cells indicated by the maximum group storage cell), EXIT.

Step 6

The programme for reading the data of all estimated streams is similar to the programme for reading data for units; however, all data are treated as ordinary variables and none may be specified functionally.

Explicit Statement of Algorithm for Step 6

- I START, set level indicator to 1.
- II Read the next instruction (Algorithm III).
- III Enter at step corresponding to the present value of the level indicator.

- 1) Consider instruction.
 - A. Asterisk, EXIT.
 - B. Unit name, set level indicator to 2 and repeat from step II.
- 2) A. Consider instruction.
 1. Asterisk, set level indicator to 1.
 2. "INPUT", locate storage address for stream's data, clear storage space, and set level indicator to 3.
 - B. Repeat from step II.
- 3) A. Consider instruction.
 1. Asterisk, set level indicator to 2.
 2. Data instruction, read and store data specified by instruction.
 - B. Repeat from step II.

Step 7

Each unit routine has two entry points; the first is used to set the initial values for the outputs of the unit, and the second is used during the simulation of dynamics, i.e. the usual entry point. Step 7 uses the computational sequence established in step 3 to set Table 3 for each of the units and input streams in the plant (Algorithm IV). The execution algorithm for step 10 is then entered, but in this case the first entry point for each unit routine is used. Wherever unknown feed-back streams occur (for which initial estimates were requested in step 3) an iteration is performed until these converge on the values required for consistency in the initial state of the plant.

Step 8

Step 8 assigns initial weights to the streams in all groups, and then reorders them using Algorithm II. The algorithm to set the weight of a stream is one of the projects on which research must be carried out as mentioned in the section on future work. However, a programme which established a weight proportional to the total flow rate of the stream has been written, and the algorithm for it is presented below.

Explicit Statement of Algorithm for Step 8

- I START, consider list 6 at beginning.
- II Next group on list 6.
 - A. No more, continue from step X.
 - B. Set initial "total of flow rates" to zero for group, continue.
- III Next unit in group, no more continue from step Vi.
- IV Next input to unit.
 - A. No more, repeat from step III.
 - B. Input is from another unit in group, continue step V.
 - C. Otherwise, repeat step IV.
- V
 - A. Add component flow rates for input and store.
 - B. Add this total to the "total of flow rates" for group.
 - C. Repeat from step IV.

- VI Consider the group at the beginning again.
- VII Next unit in group, no more, repeat from step II.
- VIII Next input to unit.
 - A. No more, repeat from step VII.
 - B. Input from unit within group, continue from step IX.
 - C. Otherwise, repeat step VIII.
- IX A. Set stream weight to:

$$\text{Stream weight} = \text{stream flow} \times \left(\frac{\text{Maximum permissible sum of weights}}{\text{total of flow rates for group}} \right)$$
 - B. Repeat from step VIII.
- X EXIT.

Step 9

Step 9 reorders all the groups according to the weights established in step 8 (Algorithm II). Table 3 for each of the units and input streams is then set according to this new sequence (Algorithm IV).

Step 10

Step 10 is the execution algorithm for the programme. The unit routines are executed for the units according to the order specified by List 6, with special iteration loops established around each of

groups. (Step 10 is also used to set all initial conditions for the streams in the plant as part of step 7, but a special entry point is used for each of the unit routines in this case.)

Explicit Statement of Algorithm for Step 10

- I START, the unit or input stream under consideration is the first unit on List 6.
- II Enter Tables 1, 2, and 3 for item.
- III Preset all specified Table 1 data which is given functionally.
- IV A. Unit, preset all inputs, continue from step V.
B. Input stream, store stream, continue from step VIII A2.
- V Enter unit subroutine (set Table 2 on request).
- VI Store unit's Table 1 unless an iteration is indicated.
- VII Reset next output into standard format.
 - A. Feed-back stream.
 1. Compare with previous version.
 - a) Comparison is good, continue.
 - b) Comparison is not good, indicate another iteration for group is required, consider the number of iterations completed so far.
 - i. A multiple of n , adjust the stream weight according to the magnitude of the difference in the comparison.
 - ii. Otherwise, continue.
 2. Store (weighted) average of two versions.
 3. Repeat from step VII.

B. Not a feed-back stream, store stream, repeat from step VII.

C. No more, continue from step VIII.

VIII Next unit to be entered.

A. Iteration is indicated.

1. Last unit in group was just completed, consider the number of iterations.

a) A multiple of m iterations.

i. Reorder the group using the present weights.

ii. Set Table 3 for all units in group using the present computational sequence.

iii. Locate the first unit which has as an input a stream which was feed-back but is now feed-forward. If none exists consider the first unit in the new sequence. Repeat from step II.

b) Otherwise, reconsider first unit in group and repeat from step II.

2. Otherwise enter next unit and repeat from step II.

B. No iteration is indicated.

1. Last unit in group was just completed.

a) Store all doubly-stored Table 1's.

b) Enter next item, repeat from step II.

2. Otherwise, repeat from step VIII A2.

C. No more units for present time step, increment time.

1. No more time steps to be taken, EXIT.

2. Otherwise, repeat from step I.

APPENDIX IV

UNIT ROUTINE INSTRUCTIONS

The following section presents the requirements for a unit routine which is written for SPEED-UP. A knowledge of the Mercury computer and of Mercury Autocode^(13,14) is assumed, and the following topics are covered.

1. General requirements.
2. The "state" of the fast store of Mercury on entry.
3. The available work space.
4. The required state of the computer on exit.
5. Inputs and outputs.
6. Setting initial conditions for outputs.
7. Stage instructions.
8. Detailed structure for Table 1.
9. Details on detecting omissions of special variables.
10. Data for symbol translation.
11. Requesting special variables to be set.
12. Resetting the ordinary variables.
13. Errors.

General requirements

A unit routine must exist for each unit type in the plant being simulated by SPEED-UP, and each routine will be supplied with the

following data just before its execution.

1. the design parameters of the unit.
2. the state of the unit and its inputs at the beginning of a time step, and
3. the inputs at the end of the time step.

From these data the routine must be able to compute:

1. the outputs, and
2. the state of the unit,

both at the end of the time step.

The "State" of the Fast Store of Mercury on Entry

The main variables in the fast store of Mercury have been set by the directives⁽¹³⁾

```

A → 335
D → 15
B → 95
C → 31 .

```

These of course may be changed to suit the unit routine; however, the locations of the data will be described in terms of these directives.

The contents of the fast store on entry to the routine are:

MAIN VARIABLES	CONTENTS
A0	The present value of the time.
A1	The time increment.
A2 to A319	Table 1 for the present time.
A320 to A335	Junk.

D0	Drum address of first input (the remaining inputs follow in sequential blocks).
D1	The first output number.
D2	The drum address for storing the first output.
D3 to D14	Output number followed by drum address for each of the remaining outputs.
D15	Junk.
B0 to B95	Junk.
C0 to C35	Junk.

INDICES

S	The number of components.
T	The number of stages.
I'	The number of inputs.
J'	The number of outputs.
I to R	Junk.
K' to P'	Junk.
R', S'	Reserved.
T'	Junk.

AUTOCODE SPECIAL VARIABLES

A to π	Junk.
A' to Z'	Reserved.

The available Work Space

Fast store: The entire fast store except for the indices R' and S' and the variables A' to Z' (Autocode special variables) may be

used during the unit routine. As will be pointed out later, certain of the main variables should be left unused if possible to avoid unnecessary drum transfers when the SPEED-UP special variables are being set.

Drum: In addition to the fast store 2000 drum locations have been reserved for the unit routines to be used as desired.

The locations are from cells 2362 to 4361. They may not be assumed to be clear on entry.

The Required State of the Computer On Exit

Certain sections of the fast store must be set as follows on exit:

MAIN VARIABLES	CONTENTS
A2 to A319	Table 1 for the end of the time step.
D0	As on entry.
D1 to D14	As on entry.

The remaining cells in the fast store (other than those which are not available to the routine) and the 2000 cells on the drum may be left containing junk.

Inputs and Outputs

As stated above the address for the drum locations of the inputs and outputs of each unit are provided for the routine in the main variables D0 to D14. The actual structure of a stream data is:

CELL	CONTENTS
1	Stream weight.
2	Output number.
3	Input number.
4	Temperature.
5	Pressure.
6 to J1 (Max.)	Individual component flow rates in the order of the component instructions for the unit.

Each stream has five cells in addition to the component flow rates, and the number of flow rates is stored in the index S; thus the inputs are stored in blocks of "S + 5" cells with the first starting at the address D0. The inputs are in random order, but the input number is part of the data for them so as they are brought forward each input can be identified and stored appropriately by the routine.

In addition to the drum addresses of the outputs, however, the output numbers are provided in the main variables D1 to D14 as described. The reason this extra information is provided is that the outputs which are feed-back streams required double storage. Only the necessary space is provided, and the output numbers are not transferred to the separate double-store locations; thus they are not available to the unit routine in these cases. The drum address provided for each output is for the first cell described in the

structure above, but the first three cells should not be touched by the routine. The doubly-stored outputs actually overlap by these three cells so as not to waste the space.

Setting Initial Conditions for Outputs

Each unit routine must have an entry which is used when initial conditions for the outputs from the units are being set. Provided sufficient data is supplied by the data for the initial state of a unit, most unit routines will be able to extract the outputs without knowing the inputs to the unit. The remaining unit routines, i.e. those for units of low capacity, will require the inputs in order to extract the initial outputs and as discussed previously in the body of the thesis these inputs can cause difficulty in establishing initial conditions for streams if they are feed-back streams in the computational sequence. The final sequence for each group should, therefore, be such that the difficulty in setting initial conditions is minimised, and for this reason the writer of a unit routine is provided with a facility for weighting the inputs to any unit for which the routine will be used. With the help of Algorithm II the best sequence for determining initial conditions can be found. The actual entering of the weights is explained in the section "Data for symbol translation."

The weights are assigned as follows:

WEIGHT	TYPE OF UNIT ROUTINE
0	For all routines which can extract the data for the initial conditions of the outputs without requiring the initial inputs.
$n > 0$	For all routines which require the initial inputs to set the initial outputs. The higher the value of n the worse it would be to break streams feeding units of this type.

, 1

The next three sections lead up to the section "Data for symbol translation" and are presented to answer many of the questions which would otherwise arise if that section were presented alone. The first section describes the exact use of the stage statements in SPEED-UP and then points out the data which must be provided for each such statement. The second section, together with a previous discussion in the body of the thesis and Fig. 9, illustrates how the four constants are determined for calculating base addresses of the ordinary variables in the Table 1 for a unit (etc.), and finally the last of the three sections discusses and illustrates the method of setting the four constants used in detecting omissions of special variables. The section "Data for symbol translation" then states explicitly how these constants, etc. must be prepared for each unit routine so that SPEED-UP can use them when translating a user's programme.

Stage Instructions

A set of instructions known as stage instructions are included in SPEED-UP (see page 27), but these instructions are slightly more general than the name implies. Each instruction can be either a set of letters such as

PARTIAL CONDENSER

or a number followed by a set of letters such as

8 PLATES .

The symbol for which the alpha-numeric sum will be checked by the programme is in each case that corresponding to the words in the instruction, whether preceded by a number or not. Each instruction can cause either the index in which the number of components or the index in which the number of stages is stored to be altered; it is up to the person writing the unit routine to decide which. Only when the number of components has no meaning will these instructions be used to set the component index, but such cases do arise as for instance tables, equations, etc.

Each instruction can alter the specified index by the addition of a constant or by the addition of a number which will be given in the instruction or by the addition of both. For instance, the first instruction above could cause one to be added to the number of stages when it is entered. In a second case the instruction

10 STAGES

might simply cause 10 to be added to the stage index (which was initially zero), whereas the instruction

8 PLATES

would cause the number 8 plus a constant, 2, to be added to the stage index in the case where the number of plates is two less than the number of stages for a unit (as for instance with a distillation column during the simulation of dynamics).

For each stage instruction to be used for a unit routine, the programmer of the routine must specify

1. the alpha-numeric sum of the words in the instruction,
2. the index affected, and
3. a constant (even if zero) which will be added to the specified index whenever the instruction is entered.

Any number which occurs in a stage statement is added to the present value of the specified index along with the constant indicated by the third data specified above.

Detailed Structure for Table 1

A brief discussion has already been given of the structure of Table 1 on page 53 under the heading "Data Storage for a Unit". To illustrate that discussion consider Fig. 9a in which a sample Table 1 is presented. The four constants for computing the base address of each of the variables is given in Fig. 9b, together with a sample address computation; to illustrate how the four constants are

determined for each type of variable consider the following example.

Example: Compute the four constants α_1 to α_4 for the variable Y in Fig. 9a.

Final address of $Y_{1.1} = nm + 2$, from Fig. 9a.

$$= \alpha_1 + \alpha_2 n + \alpha_3 m + \alpha_4 nm + 1 \cdot m + 1, \\ \text{by definition.}$$

Rearranging: $\alpha_1 + \alpha_2 n + \alpha_3 m + \alpha_4 nm = 1 - m + nm$

Equating coefficients: $\alpha_1 = 1$, $\alpha_2 = 0$, $\alpha_3 = -1$, and $\alpha_4 = 1$.

Details on Detecting Omissions of Special Variables

Each type of special variable, i.e. K's, C's etc., will have a set number of entries for it for each unit, once the number of stages and number of components have been fixed. The exact number can be computed from an equation which has the same form as the equation for computing base addresses for ordinary variables; that is:

$$\text{Total number of entries} = \alpha_1 + \alpha_2 n + \alpha_3 m + \alpha_4 nm$$

for n components and m stages.

Fig. 10 illustrates the use of this equation for an example, and also presents the computation for a particular case.

Three instances have occurred during the entry of the data in which this same form of equation has been used, and these are

1. the calculation of the size of Table 1,
2. the calculation of the base address of an ordinary variable,

3. the calculation of the total number of entries for special variables.

With this background the next section, the form of the symbol translation data which must be entered with each unit routine, can now be presented.

Data for Symbol Translation

For each unit routine (or routine for tables, equations, input streams, etc.) a table of data must be constructed which names the unit type, specifies its class (see structure for List 1), gives the computing code to specify the unit routine etc., and finally specifies the data for symbol translation given during data input. The data must follow precisely the order now given.

1. An integer indicating the class of the unit:

- 0 units to be included on List 1a.
- 1 units to be included on List 1b.
- 2 input streams (List 1c).
- 3 output streams (List 1d).
- 4 TABLE, EQUATION, or INPUT, the latter is used during the interpretation of data for streams (List 1e).

2. The alpha-numeric sum of the name for the unit type.

3. The unit type's name included between two full stops, as for example:

.HEAT EXCHANGER.

4. The computing code.
5. An integer which will be used as the weight for all inputs to units of this type when determining the order of a group for setting initial values for streams.

The number will be:

- 0 for unit routines which can calculate initial outputs without knowing the inputs (high capacity units).
- n where n ranges normally between 1 and 10 for all remaining units. The higher the number the more sensitive the unit is to its inputs.

6. Either

- 0 for a normal entry, or

The alpha-numeric sum of the name of the unit which has the same data translation table (as would normally follow from this point). In this case one additional number, the class of this other unit, follows and no more data is entered for the unit type (including no asterisks).

The latter case above is used when all the symbols for all the instructions and when all storage addresses are identical for two units, as is the case when two names for the same unit type are used.

7 - 11.

The next five numbers are α_1 , α_2 , zero (a dummy number), α_3 , and α_4 ; the α 's are the four constants for computing the total storage required in Table 1 for the unit type.

12. An asterisk.

The remaining data are entered in three sections each followed by an asterisk. For each section with no entries, the asterisk must be included.

Stage Statements

For each stage statement three numbers must be entered whether zero or not (see page 112).

1. the alpha-numeric sum of letters in the instruction.
2. 19 if the instruction modifies the number of components.
20 if the instruction modifies the number of stages.
3. A constant which will be added to the specified index (see discussion on page 113).

Ordinary Variables

Each entry for ordinary variables requires six numbers.

1. The alpha-numeric sum of letters for the variable name.
2. α_1 (for computing the base address of the variable in Table 1).
3. α_2 .
4. Zero (dummy number).
5. α_3 .
6. α_4 .

Special Variables

Each entry for the special variables requires six numbers also.

1. The alpha-numeric sum of letters for the variable name.

2. α_1 (For computing the total number of entries which must be made for this variable).
3. α_2 .
4. Identification code for the variable; the only restrictions are that this number must be a negative integer and must be different from that assigned to any other special variable.
5. α_3 .
6. α_4 .

If more than one symbol is to be used for any of the variables above (including the stage statements) an additional entry may be made. This entry must be identical to the original, except for the alpha-numeric sum which must be that of the extra symbol.

Requesting Special Variables to be Set

During the execution of the unit routine occasions arise when special variables are needed. In order to have them set the routine must construct a request list as described below, and in addition the Mercury special variable H' must be set to a drum address where these special variables will be placed as they are set by the main programme. This drum address must be one of those available to the routine, i.e. between 2362 and 4361, and must be such that all the requested special variables can be stored sequentially from that

address without exceeding the last available address, 4361. The request list is constructed in cells C0 to C31 in the fast store; each request uses four cells, therefore up to eight requests may be made at one time. If fewer than the eight possible requests are made the cell immediately following the last request must be set to zero.

The special variables for each of the consecutive requests are stored sequentially on the drum.

Structure of the request list.

CELL	CONTENTS
C0	Identification symbol of the special variable (see the previous section about identification symbols for special variables).
C1	Component number of the special variable.
C2	Initial index of the desired range.
C3	Final index of the desired range.

The remaining requests follow in repeated blocks of four cells.

For example, if the special variable K2.1/35 are requested, the setting would be as follows:

First cell	Identification code for K
Second cell	2
Third cell	1
Fourth cell	35 .

There are three entries back into the main programme for setting the requested special variables; the preferred entry is the first as

it requires the fewest drum transfers, second best is Entry 2, but if neither of these is possible then Entry 3 must be used.

Entry 1

Entry is by the instruction

DOWN $3/4$ - 2

This entry is to be used if the contents of the unprimed indices, other than S and T, may be destroyed, and if the cells 80 to 895 have not been altered. This entry may also be used if the last request for special variables was not followed by a RESTORE instruction and cells 80 to 895 have not been altered since.

Entry 2

Entry is by the instruction

DOWN $2/4$ - 2

This entry is to be used if the unprimed indices (other than S and T) may be destroyed, and cells 80 to 895 have been used but the present contents may be destroyed.

Entry 3

Entry is by the instruction

DOWN $1/4$ - 2

This entry must be followed immediately by a RESTORE instruction. It is used if the conditions for the other two entries cannot be met and therefore cannot be used.

Resetting the Ordinary Variables

If a unit routine requires the resetting of the ordinary variables specified functionally at any point, as might be the case when a Runge Kutta type integration is being used, the entry

DOWN 32/5 - 2

may be used. All of the variables will be reset.

Errors

All errors of a numerical nature must be caught by the unit routine. An exit then by the instruction

ACROSS 101/6 - 2

will cause the name of the unit to be printed, and this name will be followed by the usual dump already provided by the Mercury computer. The programme will then terminate.

APPENDIX V

STRUCTURE OF TABLE 2

The structure of Table 2 for a unit or an input stream is presented in this section. Each table consists of three separate lists which are constructed with the help of ALP⁽¹²⁾; the first list is a list of the components, the second is the list of all ordinary variables specified functionally, and the third is the list of specifications for all the special variables. On each of the lists one entry corresponds to one instruction in the data specification.

List 2.1

The list of components (the order of the entries on the list corresponds to the reverse order of the components for the unit).

Structure

Headcell: I = Zero.

J = Total number of cells in table.

K = Blank.

Typical cell: I = Component number corresponding to position on list of all components.

J = Zero, if component is not a sum of components.

Sublist headcell, if component is a sum of components.

K = Zero, component is to be left combined in outputs.

Not zero, component is to be split in outputs.

Sublist structure for list 2.1.

Headcell: None.

Typical cell: I = Component number for a component to be included
 in the sum.

J = Blank.

K = Blank.

List 2.2.

The list of all ordinary variables specified functionally.

Structure

Headcell: I = Zero.

J = Number of components for system.

K = Number of stages.

Typical cell: I = Address in Table 1 of dependent variable.

J = Zero.

K = Sublist headcell.

Sublist structure for list 2.2.

Headcell: I = Table, equation, or routine number.

To distinguish which type the number refers to, all routine numbers have 127 added and all tables 255, thus:

1 to 127 corresponds to Equations,
128 to 255 corresponds to Routines, and
256 to 383 corresponds to Tables.

J = Initial "stage" index for variable's range.

K = Final "stage" index.

Typical cell: I = Address in table 1, if independent variable is an ordinary variable.

Identification code, if independent variable is a special variable.

J = -1 for ordinary variable or component number for special variable.

K = Zero, independent variable not ranged.

One, independent variable ranged.

List 2.3

The list of all special variables.

Structure

Headcell: I = Zero.

J = Number of special variables stored on outside tables.

K = Number of "constants" specified in data.

Typical cell: I = Symbol for dependent variable.

J = Component number.

K = Sublist headcell.

Sublists for list 2.3

The sublists have the same structure as the sublists for list 2.2 above when the special variables are specified functionally, but the cells have the following structure if the special variables are specified by a constant or by a list of numbers.

Structure

Headcell: I = -1, all variables are specified by a constant.
 -2, all variables are specified by an outside
 list.

J,K Specify range as above.

Typical cell: I = Blank.

J = Address in Table 2 of constant value for
 first case above.

 Address of outside list for second case above.

K = Blank.

APPENDIX VISTRUCTURE OF TABLE 3

The structure of Table 3 for a unit or an input stream is presented in this appendix. All cells which are tagged (only for this description) with the letter "p" have permanent contents, whereas all remaining cells have contents which depend on the computational sequence specified by List 6. Each output address listed is tagged with a (+) if the stream is feed-forward and with a (-) if the stream is feed-back; the address is permanent but the tag depends on the order for computation.

CELL	TAG	CONTENTS
1	$\pm$	(+) Address of data for next item on List 6. (-) Address of data for first item on List 6.
2	None	0, if unit which is not a part of a group. -1, if input stream. -2, if unit which is a part of a group but not the first or last unit in the group. -3, if first unit in a group. Address of data for first unit in group, if the unit is the last unit in group.
3p	None	Number of inputs.
4p	None	Number of outputs.

5p	None	Computing code.
6p	None	Printing code.
7	None	Storage address for data after computation (this will be a temporary address for any unit in a group).
8p	None	Address of first input (all remaining inputs follow in sequential blocks).
9p	$\pm$	(+) Address of first output if feed-forward stream. (-) Address of first output if feed-back stream.
10		and following cells, the addresses of the remaining outputs.

APPENDIX VII

USER'S REFERENCE MANUAL WITH GLOSSARY

The complete programme tape for SPEED-UP, which is to be prepared by the user, consists of the following sections.

1. The link statements.
2. The list of all components treated in the plant.
3. The data for all units and input streams, and for all tables, equations, and routines mentioned in these data.
4. Initial estimates for all streams which are requested just after the link statements are read.
5. The total time for which the simulation is desired.

The output from the computer is as follows,

Following the link statements.

1. The number of statements read.
2. A list of the units and groups in the plant in an order which has only a forward feed of information.
3. A list of the streams for which initial estimates should be entered in section 4 above.

Following the remainder of the programme.

1. The initial condition of the streams in the plant.
2. The state of each stream after each time step.

3. After each group reordering the new computational sequence for the plant.

The following errors are detected.

Link statements.

1. Spelling or spurious characters.
2. Input or output numbers used more than once for a particular unit.
3. Two outputs from an input stream.
4. Implied feed-back stream to an input stream.
5. Too many link statements (computer capacity exceeded).

List of all components.

1. List of all components missing.
2. More than 32 components on list.
3. Two components with identically translated symbols.

Data statements.

1. Spelling.
2. More than 222 tables, equations, and routines mentioned.
3. More than 511 special variables set by list type instructions.
4. Table 2 overflow (a total of too many component, functional, and special variable statements).
5. More than 15 constants.
6. Correction for data not previously entered.
7. Stage and component statements entered during a correction entry.

8. Non-existent output or input number changed.
9. More than 31 complete re-entries.
10. Drum storage exceeded.
11. Range on variable not identical with range in previous entry during a correction entry.
12. Range on numbers entered for a list instruction incompatible with number of data numbers entered.

Stream data.

1. Errors 1 and 12 above.
2. Stage statement or component statement entered.
3. Functional statement entered.

GLOSSARY

Asterisk

The terminator for all data entries in SPEED-UP. It is used to terminate

1. The link statements.
2. The list of all components.
3. The data for a single unit, input stream, table, equation, or routine.
4. The end of all data for units and input streams.
5. The end of a single stream specification.
6. The end of the stream specifications associated with a single unit.
7. The end of the section for stream specifications.

Chain of execution

The order in which all ordinary variables specified functionally in a unit's (input stream's) data will be computed. It corresponds to the order in which the statements occur in the data specification.

Change statements

The change statements in a unit's data are used to alter any incorrect input or output numbers given in the link statements. The

heading for the statements is the single instruction

CHANGE

and it is followed by a list in any order of the numbers which are to be changed. The format of the instruction to change an input number is

INPUT n_1 TO INPUT m_1

and for altering an output number

OUTPUT n_2 TO OUTPUT m_2

where n_1 and n_2 are the old numbers and m_1 and m_2 the new ones. Each instruction is interpreted independently of any of the other ones listed, for example the two instructions

INPUT 3 TO INPUT 5

INPUT 5 TO INPUT 7

would cause the "old" input numbers 3 and 5 to be changed to 5 and 7 respectively and would not cause input 3 to be changed to input 5 and then to input 7.

See Correction entry for the special interpretation of CHANGE statements during such an entry.

Complete re-entry

A method of correcting the data for a unit, input stream, table, equation, or routine. The data is re-entered in its complete form

as though the previous data had never been specified; all the old data will automatically be cancelled. No more than 31 complete re-entries may be made, and no more than is absolutely necessary should be made as the original entry wastes time and space in the computer.

Component instructions

The component instructions for a unit or input stream consist of either the single instruction

ALL COMPONENTS

if the components for the unit or input stream are the same as those listed on the list of all components for the entire plant, or the instruction

α COMPONENTS (or 1 COMPONENT)

where α is an integer followed by a list of the components by name, each on a separate line. No unit or input stream may have more than 28 components.

If some of the component flow rates are to be combined into the flow rate for one component in all the inputs to the unit, the format is

$$C_1 (C_1 + C_2 + \dots + C_n)$$

for the component name, where C_1 is the resulting component and $C_2, C_3, \dots, C_i$ are any components not listed elsewhere on the component instructions for the unit or input stream.

If in addition to combining several components into one the user

wishes to split these components again in all the outputs from the unit he may use an instruction format

$$C_1 (C_1 + C_2 + \dots + C_n) \text{ SPLIT ON EXIT} \cdot$$

An example of component instructions for a unit is given in Fig. 6.

Constants

Up to 15 constants may be used as independent variables in functional expressions. They must be specified in the unit's data by single parameter, list, or constant type instructions.

Constant type instruction

The format of a constant type instruction is

$$\text{Variable } \alpha/\beta \text{ SET} = d_1$$

where the variable may be any ordinary or special variable, α and β are two integers ($\beta > \alpha$) and d_1 is a data number.

Some examples are

$$P \ 1/21 \ \text{SET} = 3.14159, \ 3$$

$$FR5.9/17 \ \text{SET} = 200$$

Correction entry

Portions of a unit's (input stream's, etc.) data may be corrected by a correction entry. The entry may be made at any time

after the original entry but before the beginning of the stream specification section. The format is

Class 1 Name.

followed immediately by the instruction

CORRECTION

Class 2 Change statements.

Class 3 Component statements, none.

Class 4 Stage statements, none.

Class 5 Data.

Class 6 A single asterisk on a separate line.

The following restrictions apply to correction entries.

1. All change statements should be corrected under the assumption that all previous ones have been executed.
2. Change statements in addition to those in the original data may be specified.
3. No component or stage statements may be corrected.
4. Special variables must be respecified by the same type of instruction and over identical ranges when corrected.
5. All functional data must be corrected by functional statements and over identical ranges for the dependent variable. The number of independent variables may be reduced, but in no case can the number exceed that in the original statement. Each corrected

- functional statement for the ordinary variables is removed from its original position and put at the end of the chain of execution as it is entered.
- 6. All ordinary variables specified numerically may be corrected by any arbitrary set of new numerical specifications (i.e. by single entry, list, and constant type statements).

Data number

A data number is any number appearing on the r.h.s. of an equal sign during data specification. It may be either a fixed point or a floating point number and may be terminated by either a CR LF (see Fig. 8) or a double space SP SP.

The fixed point format is:

[sign] [digits] [. decimal digits] [terminator]

omitting any part which is unnecessary other than the terminator.

In all cases only the first nine or ten significant figures will be retained. Some examples are:

1 +1.7 15.0 015 +0015.0 -6.7 1000 -0.7 -.7

A floating point number is a combination of two fixed point numbers, the first of which is terminated by a comma. The second number is the power of 10 by which the first is multiplied to obtain the final number, and it must have an integral value. The final number must lie between 10^{-70} and 10^{70} , and some examples are:

1,-10	i.e.	10^{-10}
-2.5666,1	i.e.	-25.6661
1000,-3.0	i.e.	1
1.23456,-7	i.e.	0.0000001234567

Note: Data numbers for SPEED-UP have the identical format as data numbers for Mercury Autocode⁽¹³⁾.

Equation

For each equation mentioned in the data for any of the units or input streams an entry similar to that for a normal data specification for a unit must be made. It must conform with the specification sheet provided and may be made at any time during the input of data for the units and input streams; that is an equation's data entry may be made prior to any mention of it in a functional statement elsewhere.

Equation, specification sheet

Class 1	Name, for example EQUATION 15
Class 2	Change statements, none.
Class 3	Component statements, none.
Class 4	Stage statements. One stage statement with the following format must be entered:

n_1 CONSTANTS (or 1 CONSTANT)

where n_1 is an integer.

Class 5

1. Special variable, none.
2. Ordinary variables. The following ordinary variables must be entered (or their values will be set to zero) using any combination of list, constant, or single-entry type instructions.

Functional type instructions are not allowed.

A single entry specifying the equation type

TYPE = n_2

where n_2 is an integer.

For each constant over the range 1 to n

C

for example:

C1/3 = 1 2 3

Class 6

An asterisk, on a separate line.

Correction: entries and complete re-entries are allowed.

External input

An external input, to distinguish it from a unit or an external output, is a basic element in SPEED-UP which can only have one output and no inputs, i.e. it behaves as the source of a stream from outside the plant. It is generally referred to by the specific name "Input stream" which must not be confused with the name "Input" alone.

External output

An external output, to distinguish it from a unit or an external input, is a basic element in SPEED-UP which can only have one input and no outputs, i.e. it behaves as the sink for a stream leaving the plant. It is generally referred to by the specific name "Output stream" which must not be confused by the name "Output" alone.

An external output is a "Dummy" unit and should only appear in the link statements; no data should be entered for it.

Functional type instruction

The format of a functional type instruction is

$$D(I_1, I_2, I_3, \dots, I_n) \text{ GIVEN BY } F$$

where the dependent variable D is any variable or range of variables, and the independent variables $I_1, I_2, \dots, I_n$ (at least one must exist) are either single variables or variables with ranges consistent in number with D . F is any particular table, equation, or routine.

Some examples of functional statements are

$P(T)$ GIVEN BY TABLE 3

URF4.3/7 (CLK, TRV1/5, HJKL7/11) GIVEN BY EQUATION 2

$Z(X1, X2, X3, X4, X5, X6, X7, X8, X9, X10)$ GIVEN BY ROUTINE 81

Group

A collection of units. From any unit in a group a path may be traced through the outputs of the units in the group which will lead back to the starting unit, thus one or more of the inputs to each member of the group is dependent on one or more of the outputs of the same unit.

Input

1. The source unit or input stream in a link statement.
2. The actual stream which feeds the unit or output stream.

Input number

A number associated with an input to a unit. A convention on input numbers is established for each unit type and is given on its specification sheet. The numbers must lie in the range 1 to 125.

Input stream

See External input.

Input stream, specification sheet

Class 1	Name, for example: INPUT STREAM 15
Class 2	Change statements, none.
Class 3	Component statements, the component statements for an input stream follow the standard format (see <u>Component statements</u>).

Class 4 Stage statements, none.

Class 5 1. Special variables, none.

 2. Ordinary variables. The following ordinary variables must be entered (or their values will be set to zero) using any combination of list, constant, single-entry, or functional type instructions.

Single items

TEMPERATURE

PRESSURE

For each individual component flow rates (the ordering corresponds to the order of the component instructions for this input stream).

FLOW RATE

for example:

FLOW RATE 1/6 = 1 2 3 4 5 6

Class 6 An asterisk on a separate line.

Correction entries and complete re-entries are allowed.

Instruction

One line on a type-written page of a SPEED-UP programme (usually). The one normal exception is the list type instruction in which the data numbers following the equal sign may extend over several lines.

An arrow used as the last character of a line which does not contain an equal sign will cause the CR character following it to be ignored, and the next line will be treated as a continuation of the same instruction. For example:

```
8 PL→
AT→
ES
```

will be read as

```
8 PLATES
```

by SPEED-UP.

Link statements

The following sample instructions illustrate the acceptable formats for the link statements; any legal unit, external input, or external output name together with any number in the range 1 to 125 may of course be substituted for the sample units (etc.) mentioned. Any input number or output number in the range 1 to 125 may also be substituted for those given, but these numbers must of course correspond to the conventions required by each of the unit (etc.) routine specifications.

Between two units:

```
INPUT 4 OF TANK 8 IS OUTPUT 3 OF PLATE COLUMN 1
```

```
OUTPUT 3 OF PLATE COLUMN 1 IS INPUT 4 OF TANK 8
```

Between a unit and an external input:

INPUT 6 OF HEAT EXCHANGER 15 IS INPUT STREAM 71

INPUT STREAM 71 IS INPUT 6 OF HEAT EXCHANGER 15

Between a unit and an external output:

OUTPUT 101 OF PACKED COLUMN 5 IS OUTPUT STREAM 5

OUTPUT STREAM 5 IS OUTPUT 101 OF PACKED COLUMN 5

List of all components

The list of all components in the plant must follow directly after the link statements and before any of the data for the units. It must be headed by the instruction

LIST OF ALL COMPONENTS

and followed by the names of all components in the plant. The list must be terminated by an asterisk on a line by itself, and it must not contain more than 31 component names.

Each component name must consist of one or more letters which may or may not be followed by a number. If two component names result in the same numerical translation by the computer an error will be indicated specifying the two components by number.

An example of a list of all components is given in Fig. 5.

List type instruction

The format of a list type instruction is

Variable $\alpha/\beta = d_1 \ d_2 \ \dots \ d_{\beta-\alpha+1}$

The variable may be any ordinary or special variable, α and β must

be integers ($\beta > \alpha$), and d_1 through $d_{\beta-\alpha+1}$ are data numbers.

Some examples are

$K1.3/5 = 20 \quad 21 \quad 22.37$

ENTHALPY 4/5 = 0.11,-6 0.13,-6

The data numbers for a list type instruction may extend over several lines; this type of instruction is the only type which does not normally have to occupy a single line.

Ordinary variable

Certain variables will be specified as being ordinary variables in a unit's (input stream's, etc.) data, as opposed to being special variables. They have the following properties which in general distinguish them from the special variables.

1. Deliberate omissions will cause the values of the ordinary variables to be set to zero.
2. Any ordinary variable which appears as an independent variable in a functional statement must be given an initial numerical value or be given by an earlier functional statement; otherwise the initial value used will be zero.
3. Recursive definitions are allowed in functional statements but will be ignored during the setting of the ordinary variables.
4. Corrections (see Correction entry):
 - a) Any ordinary variable given functionally must be

corrected functionally and over the identical range.

- b) Any ordinary variable not given functionally originally may not be corrected functionally.
- c) All ordinary variables specified numerically may be corrected by any arbitrary set of new numerical specifications (ie. list, constant, and single entry type instructions).

Output

A stream for which the unit or input stream referred to is the source.

Output number

A number associated with an output from a unit. A convention on output numbers is established for each unit type and is given on its specification sheet. The numbers must lie in the range 1 to 125.

Output stream

See External output.

Range

The range of a variable is indicated by the two integers specified on each side of the stroke given with the variable's name; for example X1.3/15 means all variables ranging from X1.3 to X1.15, ie X1.3, X1.4, X1.5, ... X1.15.

Routine

Each routine in the data for any unit or input stream must exist as a library routine for SPEED-UP. If required a set of constants may be provided in a manner conforming with the routine specification sheet.

Routine, specification sheet

Class 1 Name, for example

ROUTINE 7

Class 2 Change statements, none.

Class 3 Component statements, none.

Class 4 Stage statements. One stage statement with the following format must be entered:

n_1 CONSTANTS (or 1 CONSTANT)

where n_1 is an integer.

Class 5 1. Special variables, none.

2. Ordinary variables. The following ordinary variables must be entered (or their values will be set to zero) using any combination of list, constant, or single-entry type instructions. Functional type instructions are not allowed.

For each constant over the range 1 to n_1

for example

$$C1/3 = 1 \quad 2 \quad 3$$

Class 6 An asterisk on a separate line.

Corrections and complete re-entries are allowed.

Single-entry type instruction

The format of a single-entry type instruction is

$$\text{Variable} = d_1$$

where the variable is any single variable (no range) and d_1 is a data number. Some examples are

$$P = 1$$

$$Z3.6 = -5.998, -7$$

$$\text{TEMPERATURE } 5 = 67.9$$

Special variables

Certain variables will be specified as being special variables in a unit's (input stream's, etc.) data, as opposed to being ordinary variables. They have the following restrictions which in general distinguish them from the ordinary variables.

1. All special variables must be specified even if zero as any omissions will be faulted.
2. No recursive definitions involving special variables is allowed; however, if one is entered it will not be detected and therefore

not be faulted during data input.

3. During a correction entry special variables must be respecified by the same type of instruction and over the same ranges (see Correction entry).

Specification sheet

Each unit type, input stream, table, equation, and routine has a distinct specification sheet in which a list of the data which must be entered and the precise nomenclature for entering it is given. The data are split into the following six classes on each sheet.

- | | |
|---------|--|
| Class 1 | Name. |
| Class 2 | Change statements (standard format). |
| Class 3 | Component statements (standard format). |
| Class 4 | Stage statements. |
| Class 5 | Data 1. Special variables.
2. Ordinary variables. |
| Class 6 | Terminator, an asterisk on a separate line in all cases. |

Stage statements

The fourth class of statements in the data for a unit or input stream. They may be entered in any order and must correspond to the stage statements required by the specification sheet for each unit routine, input stream, etc.

Stream

A stream is a link between any two units, between a unit and an input stream, or between a unit and an output stream in the plant. Each link statement specifies one stream.

Stream specification

A stream specification may be used to enter into the computer initial data for any stream in the plant; however, any data for streams which were not requested just after the link statements were analysed will be ignored. All requested streams not set in this section of data input will have all values initially set to zero.

The streams are identified by the unit they feed together with their final input number (after all CHANGE statements in the unit's data). The name of the unit appears on the first line followed by the instruction

INPUT α_1

where α_1 is the input number. The data follows. If any additional inputs follow immediately for the same unit the data is terminated by a single asterisk, and the next input will be assigned automatically to the same unit. In this case the instruction

INPUT α_n

alone identifies the particular stream. However, if the next stream is to be identified by a different unit a second asterisk is used, and

the specification repeats the above pattern.

When the stream specification data is completed a third asterisk after the final stream's data terminates this section of data read.

Schematically the structure is

```

Unit 1 name
INPUT  $\alpha_1$ 
(Data)
*
INPUT  $\alpha_2$ 
(Data)
* *
Unit 2 name
INPUT  $\beta_1$ 
(Data)
*
  :
  :
Unit  n  name
INPUT  $\pi_1$ 
(Data)
*
  :
  :
INPUT  $\pi_m$ 
(Data)
* * *
```

Table

For each table mentioned in the data for any of the units or input streams an entry similar to that for a normal data specification for a unit must be made. It must conform with the specification sheet provided and may be made at any time during the input of data for the units and input streams; that is, a table's data entry may be made prior to any mention of it in a functional statement elsewhere.

Table, specification sheet

Class 1	Name, for example TABLE 3
Class 2	Change statements, none.
Class 3	Component statements, none.
Class 4	Stage statements. Two stage statements with the following formats must be entered: $n_1 \text{ VARIABLES (or 1 VARIABLE)}$ $n_2 \text{ ENTRIES}$ where n_1 and n_2 are integers.
Class 5	1. Special variables, none. 2. Ordinary variables. The following ordinary variables must be entered (or their values will be set to zero) using any combination of list,

constant, or single-entry type instructions.

Functional type instructions are not allowed.

Over the range 1 to n_1

INCREMENT

for example: INCREMENT $1/3 = 2 \ 3 \ 5$

STARTING VALUE

DIMENSION

Over the range 1 to n_2

ENTRY

The entries are made as a series of single vectors in which the first variable ranges from its specified starting value in equal increments of the size specified to its final value in each vector. The second variable then ranges from its starting value over a matrix of vectors, etc.

The dimension of the first variable is the number of elements in each vector, of the second the number of vectors in each matrix, etc.

Class 5 An asterisk on a separate line.

Correction entries and complete re-entries are allowed.

Unit

A unit, to distinguish it from an external input or an external

output, is a basic element in SPEED-UP which can have both inputs and outputs.

Unit number

Each unit name consists of two parts, the type of the unit (such as HEAT EXCHANGER or TANK) and a number which must be distinct from the number assigned to all other units of the same type. The number may be any integer in the range 1 to 125.

Unit's data

See Specification sheet.

Unit type

A class of units for which one routine exists in SPEED-UP. For instance, plate column and heat exchanger are two types of units.

Table 1 - Analysis of Algorithm IIC

Group size N	Cells required by existing computer programme	Maximum number of subgroups examined	
		With D. P.	Without(= N!)
2	10	2	2
3	24	9	6
4	48	28	24
5	85	75	120
6	148	186	720
7	318	441	5040
8	626	1016	40,320

Table 2 - Contribution to feed-back parameters due to the links
connecting unit A to unit B and unit B to unit
C in Fig. 2

	<u>Permutation</u>	<u>Feed-back Parameters</u>
Unit A precedes unit C	ABC	0
	BAC	w_1
	ACB	w_2
Unit C precedes unit A	CBA	$w_1 + w_2$
	BCA	w_1
	CAE	w_2

Fig. 1. Typical Flow Diagram

(Numbers indicate the number of parameters associated with the link)

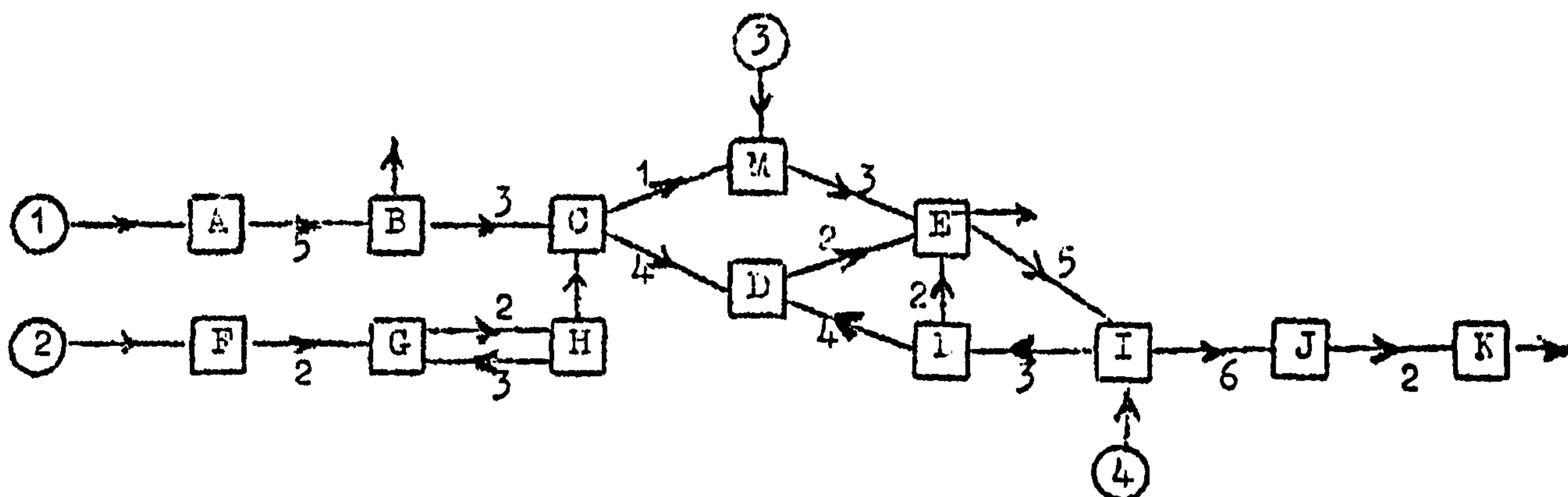
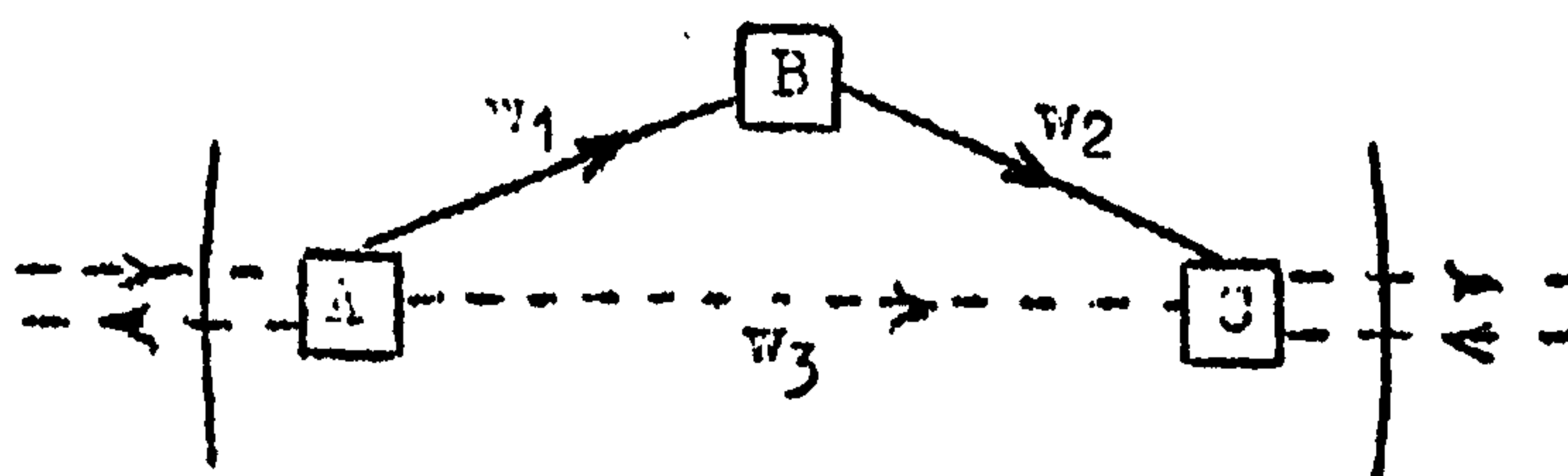
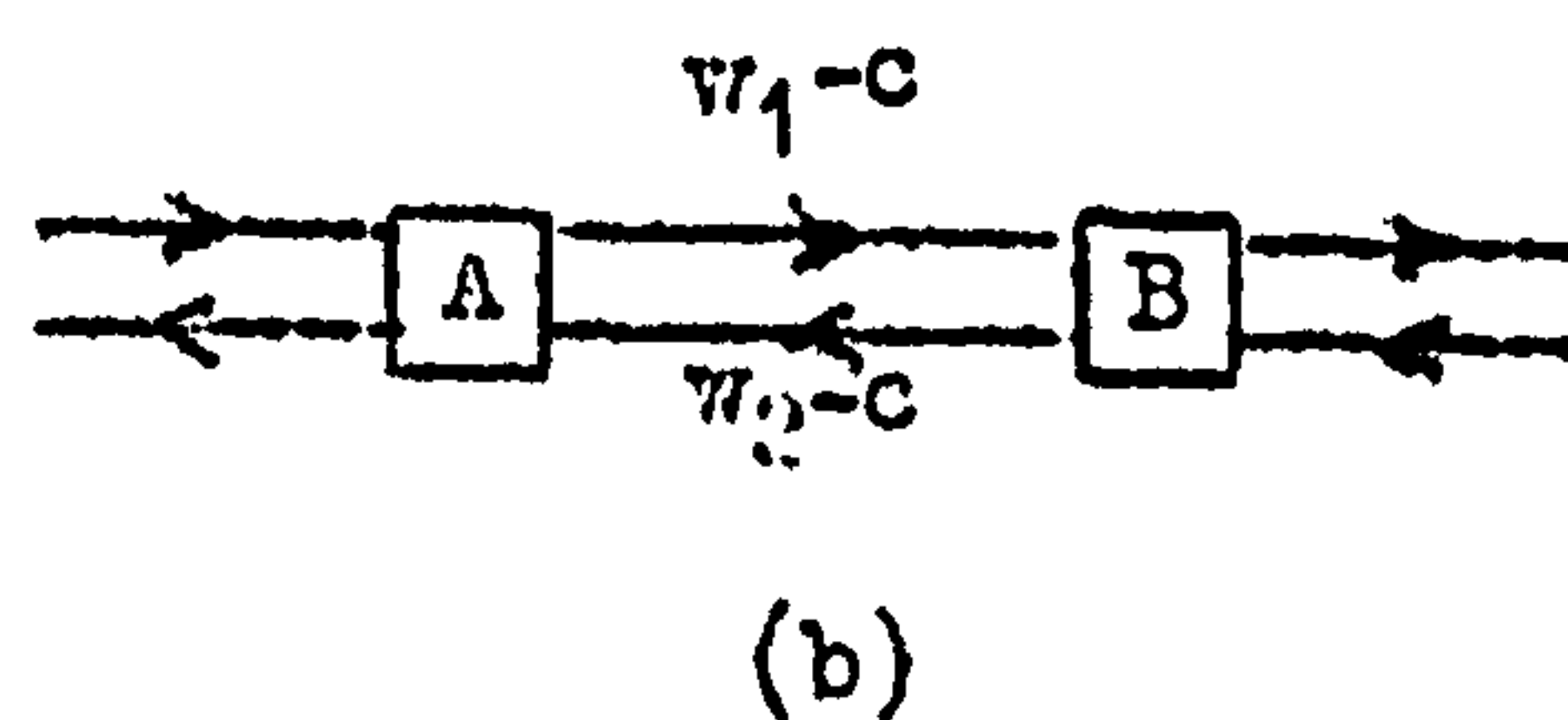
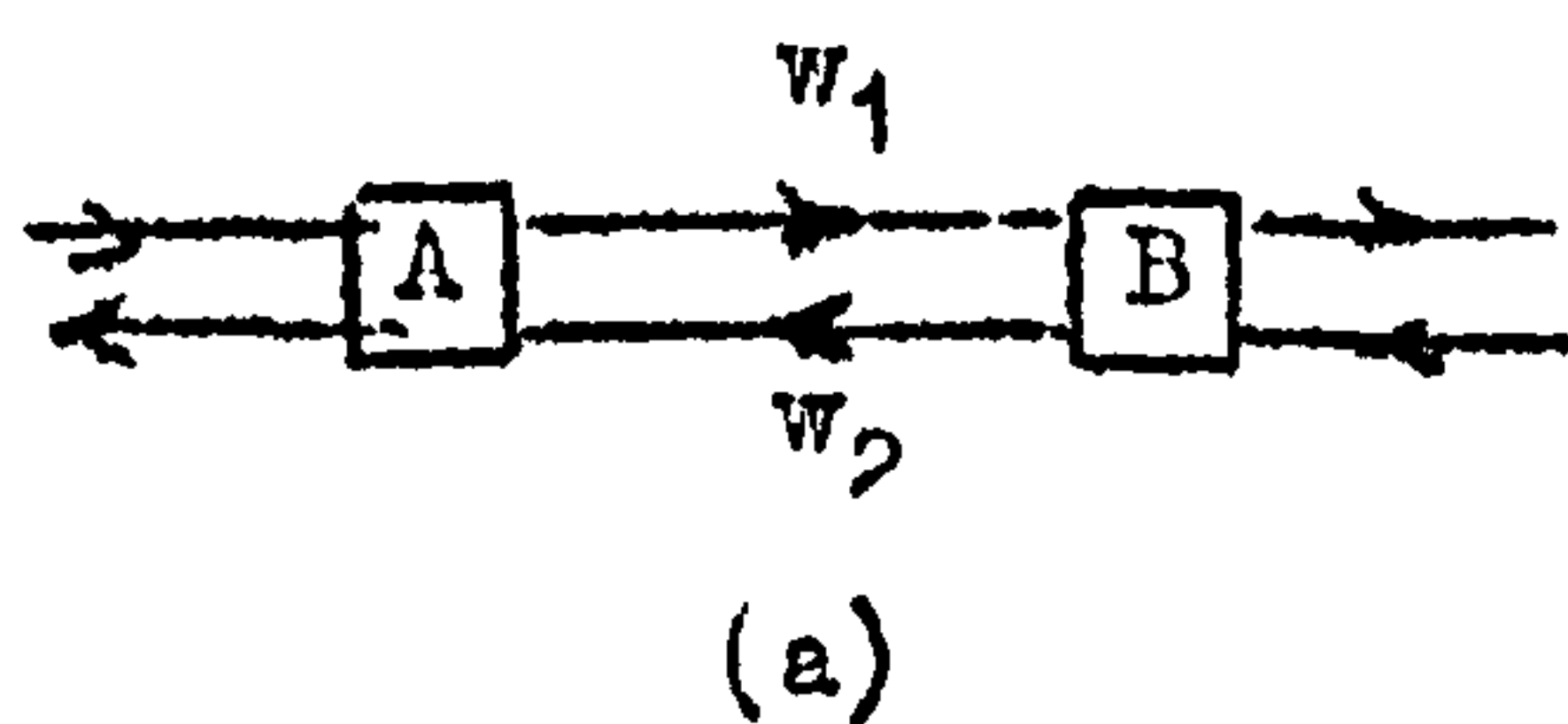
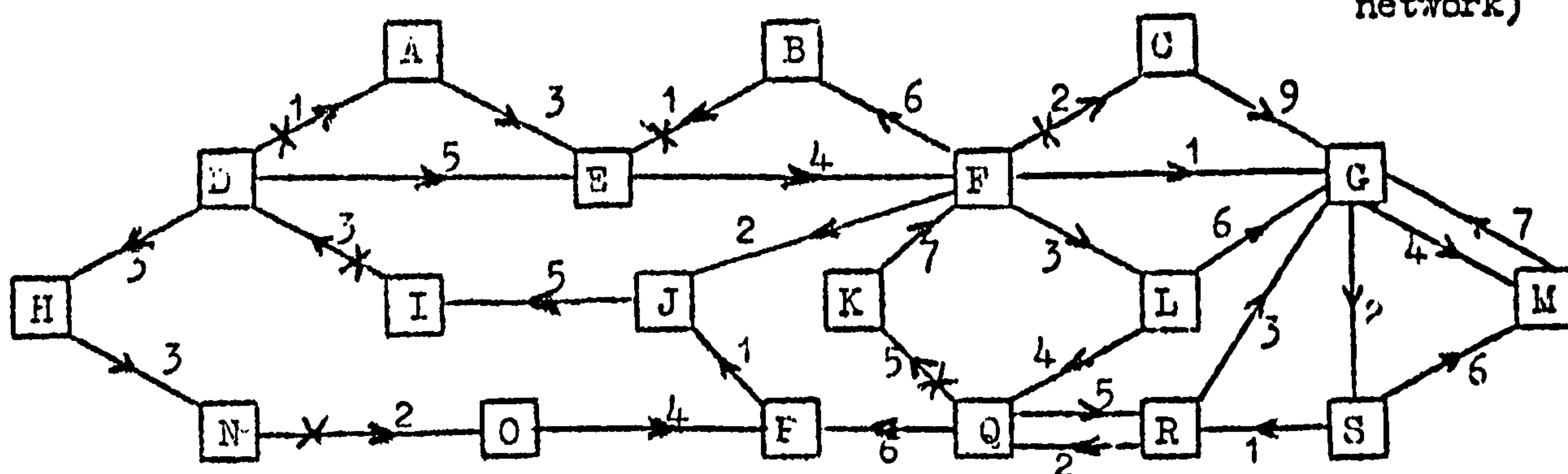
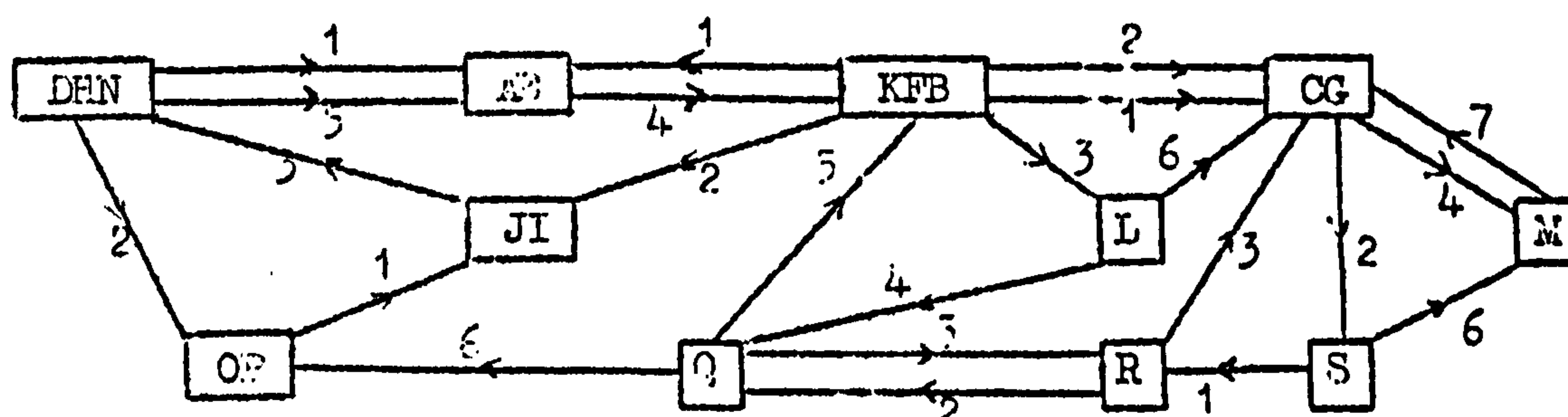
Fig. 2. The Simply Linked UnitFig. 3. Two-way Link

Fig. 4. Reduction and Ordering of a Group

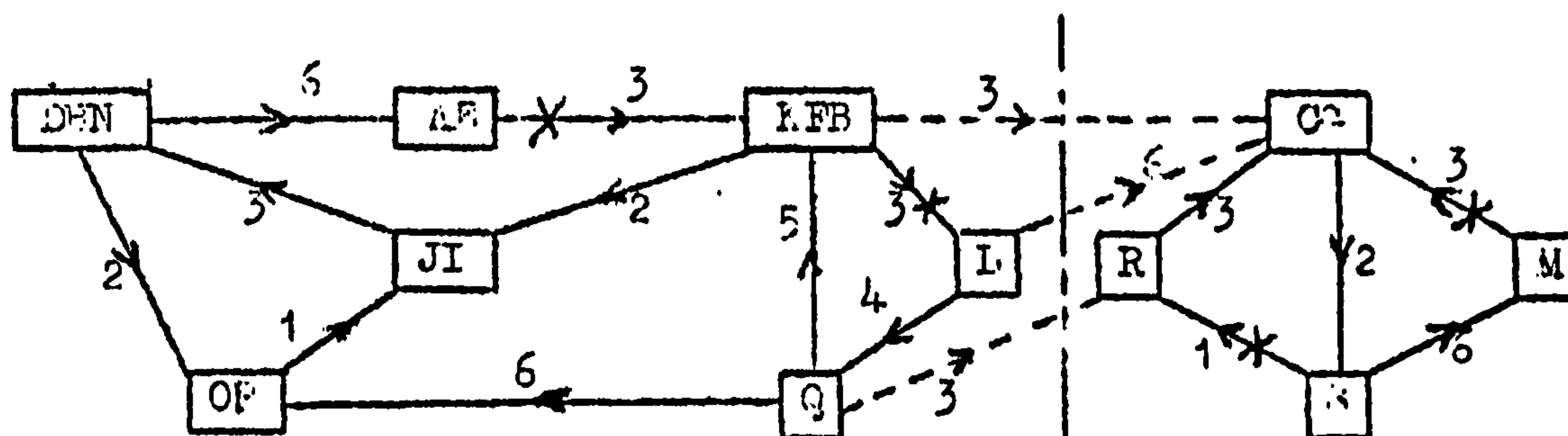
(Note Crosses indicate the link retained in the following reduction when simply linked units are removed from the network)



a) Original group



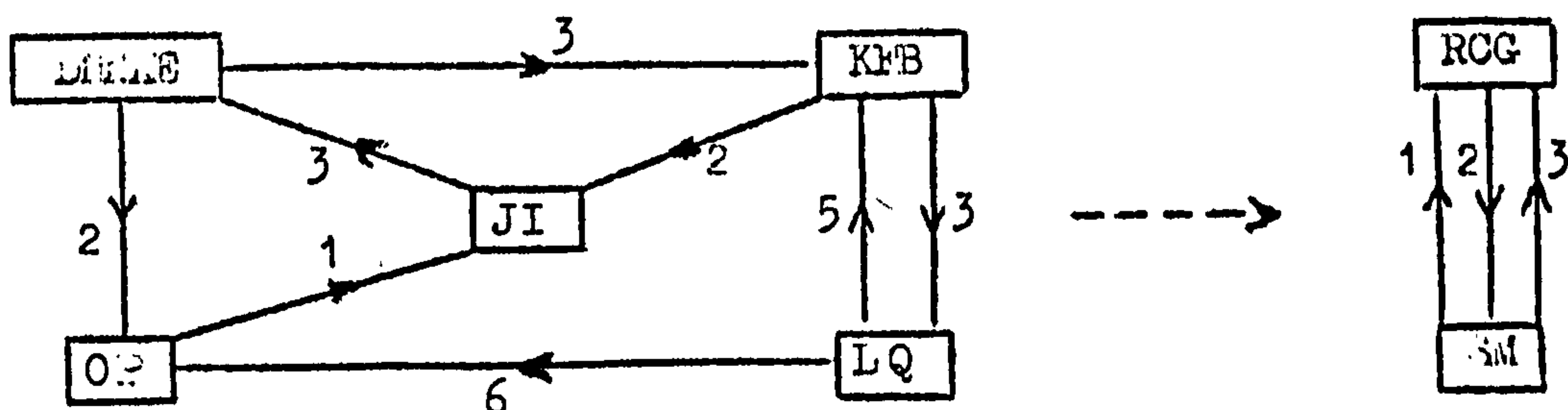
b) First reduction of simply linked units



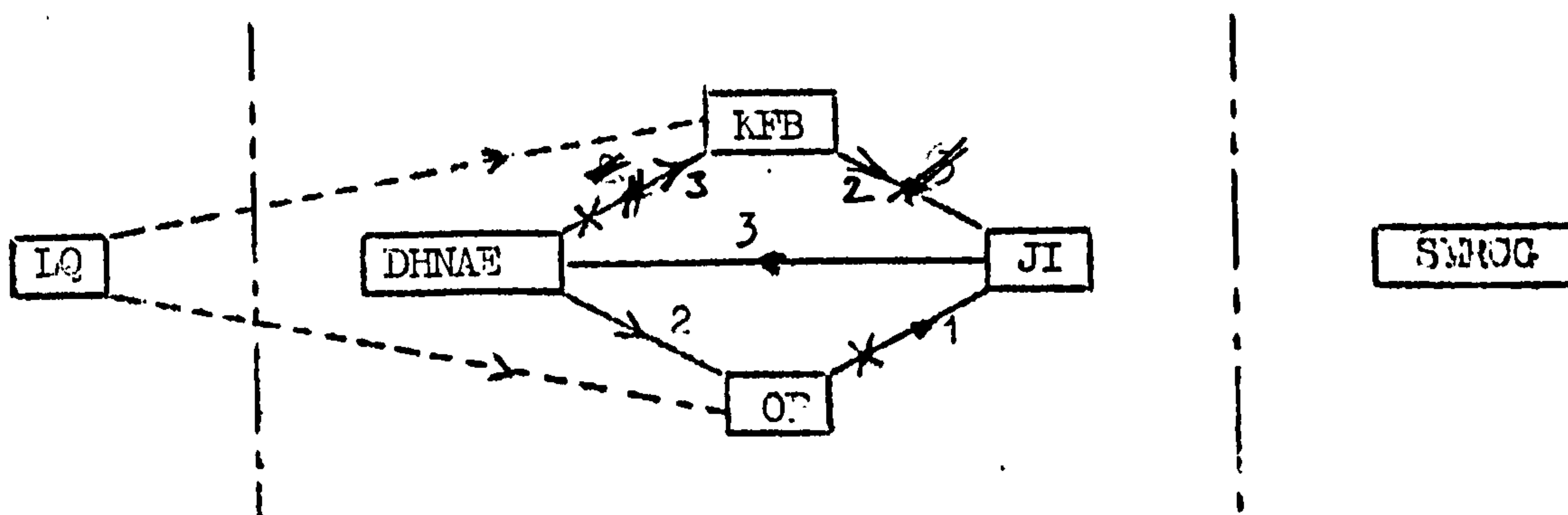
c) Decomposition by Algorithm I after replacement of 2-way links

(cont'd)

Fig. 4. (cont'd)



d) Second reduction of simply linked units



e) Decomposition by Algorithm I after replacement of 2-way links

$$\text{LQ, JI, } \overline{\text{KFB}}, \text{DHNAE, } \overline{\text{OP}}, \text{SMRCG}$$

f) Final sequence after third reduction of simply linked units

Fig. 5 Sample List of All Components

LIST OF ALL COMPONENTS

WATER
 METHANOL
 ETHANOL
 PROPANOL
 H2
 N2
 CH4
 O2
 NH3
 x

Fig. 6 Sample Data Input Instructions

PLATE COLUMN 12

CHANGE

INPUT 5 TO INPUT 7
 OUTPUT 2 TO OUTPUT 3
 INPUT 7 TO INPUT 5

2 COMPONENTS

WATER
 METHANOL (METHANOL + ETHANOL)

17 PLATES

TOTAL CONDENSER
 PARTIAL REBOILER

P = 1

X1.0/17 = 1.0 0.95 0.90 0.85 0.80 0.75 0.70 0.65
 0.60 0.52 0.47 0.41 0.34 0.27 0.21
 0.15 0.08 0.
 X2.0/17 = 0 0.05 0.10 0.15 0.20 0.25 0.30 0.35
 0.40 0.48 0.53 0.59 0.66 0.73 0.79
 0.85 0.92 1

VAPOUR HOLDUP 0/17 SET = 0.01

T 0/17 = 100 98 96 94 92 90 88 86 84 82 80
78 76 73 70 67 64 62.8
LIQUID HOLDUP1/17(L1/17,VO/16) GIVEN BY ROUTINE 10
REFLUX RATIO (T15) GIVEN BY EQUATION 6
K1.0/17 (P, TO/17, X1.0/17) GIVEN BY TABLE 6
K2.0/17 (P, TO/17) GIVEN BY ROUTINE 11
REBOILER HEAT(TIME, CONSTANT 1) GIVEN BY TABLE 5
CONSTANT 1 = 3.756
CONDENSER COOLING (T17, L17) GIVEN BY ROUTINE 43
L1/17 SET = 21.7
VO/ 7 SET = 12.5
V8/17 SET = 30
LIQUID HOLDUP 0 = 17.3
BOTTOM PRODUCT RATE (LIQUID HOLDUP 0) GIVEN BY EQUATION 2

*

Fig. 7 A Breakdown of two Sample Instructions into a Series of Symbols and their Numerical Equivalences

Instruction I) 8 PLATES

Instruction II) L3.0/4(P,T1/5) GIVEN BY TABLE 3

Inst.	Symbol	Letters	Number .	Number	Terminator	Numerical equivalents
I	1.			8	letter shift	0 0 8 27
	2	PLATES			carriage return	79 0 0 30
II	1	L	3.	0	/	13 3 0 23
	2			4	(	0 0 4 5
	3	P			,	17 0 0 15
	4	T		1	/	21 0 1 23
	5			5	)	0 0 5 6
	6	GIVEN BY TABLE		3	carriage return	136 0 3 30

Fig. 8 THE TAPE CODE FOR THE FERRANTI MERCURY COMPUTER

TAPE	VALUE	PRINTER	
		FIGS.	LETS.
•	0	FIG.	SHIFT.
• •	1	1	A
• •	2	2	B
• • •	3	•	C
• •	4	4	D
• • •	5	(	E
• • •	6	)	F
• • • •	7	7	G
• •	8	8	H
• • •	9	=	I
• • •	10	=	J
• • • •	11	-	K
• • •	12	$\approx (v)$	L
• • • •	13	L.F.	M
• • • •	14	SP	N
• • • • •	15	,	O
• •	16	o	P
• • •	17	>	Q
• • •	18	>	R
• • • •	19	3	S
• • •	20	→	T
• • • •	21	5	U
• • • •	22	6	V
• • • • •	23	/	W
• • •	24	$\phi (x)$	X
• • • •	25	9	Y
• • • •	26	+	Z
• • • • •	27	LET.	SHIFT.
• • • •	28	•	•
• • • • •	29	$\frac{1}{(n)}$	?
• • • • •	30	C.P.	$\pi (E)$
• • • • •	31	*	*

FERRANTI LTD.

COMPUTER DEPARTMENT

Office & Works,

WEST GORTON,

MANCHESTER, 12

London Computer Centre:

21, PORTLAND PLACE,

LONDON W.1

List CS 124

December 1956

Fig. 9a Sample Table 1 for unit

Classes of variables: X, Y, L, V, and P.

X and Y range for all n components and all m stages.

L and V range for all m stages only.

P is a single parameter.

MAIN VARIABLE(S)	CONTENTS
A_0	Time
A_1	Time increment
A_2 to A_{m+1}	$X_{1.1}$ to $X_{1.m}$
A_{m+2} to A_{2m+1}	$X_{2.1}$ to $X_{2.m}$
$\vdots$	
$A_{(n-1)m+2}$ to A_{nm+1}	$X_{n.1}$ to $X_{n.m}$
A_{nm+2} to $A_{(n+1)m+1}$	$Y_{1.1}$ to $Y_{1.m}$
$\vdots$	
$A_{(2n-1)m+2}$ to A_{2nm+1}	$Y_{n.1}$ to $Y_{n.m}$
A_{2nm+2} to $A_{(2n+1)m+1}$	L_1 to L_m
$A_{(2n+1)m+2}$ to $A_{(2n+2)m+1}$	V_1 to V_m
$A_{(2n+2)m+2}$	P

Total storage = $(2n + 2)m + 2$

Fig. 9b Constants for computing total storage and the base address of variables in Fig. 9a

General symbol is $Z \beta.\theta$, where β and θ are integers.

$$\text{Base address} = \alpha_1 + \alpha_2 n + \alpha_3 m + \alpha_4 nm$$

$$\text{Final address} = \text{Base address} + \beta m + \theta$$

<u>Variable</u>	α_1	α_2	α_3	α_4
X	1	0	-1	0
Y	1	0	-1	1
L	1	0	0	2
V	1	0	1	2
P	2	0	2	2
Total storage	2	0	2	2

Example: Compute the address of Y3.8 for $n = 6$ components and $m = 15$ stages.

$$\text{Base address} = 1 + (0)6 + (-1)15 + (1)(6)15 = 76$$

$$\text{Final address} = 76 + (3)15 + 8 = 129$$

The variable Y3.8 will therefore be stored in A129 in the unit's Table 1.

Fig. 10 Total number of entries for special variables

Classes of special variables: K, C, E, D, and U.

K and C range for all n components and m stages.

E ranges for all m stages.

D ranges for all n components.

U is a single parameter.

$$\text{Total number of entries} = \alpha_1 + \alpha_2 n + \alpha_3 m + \alpha_4 nm$$

<u>Variable</u>	α_1	α_2	α_3	α_4
K	0	0	0	1
C	0	0	0	1
E	0	0	1	0
D	0	1	0	0
U	1	0	0	0

Example: Compute the number of entries for K when n = 5

components and m = 16 stages.

$$\text{Total number of entries} = 0 + (0)5 + (0)16 + (1)(5)16 = 80$$