The Autocode Programs developed for the Manchester University Computers

by R. A. Brooker

Summary: The article describes two programs written for the Manchester University Computers, which enable these machines to accept programs written in a simplified form.

INTRODUCTION

The preparation of programs for an automatic digital computer is a technique which usually takes several weeks to become familiar with, depending on the logical complexities of the machine in question. For the casual user, with a problem that does not require to make full use of the high speed of such a machine, it is possible to devise simplified coding procedures which are both quicker to learn, taking a day or two at most, and easier to use. These make use of a *conversion* program which enables the machine itself to accept "programs" written in an idealized language or "instruction code" and convert them to the precise instructions which the machine eventually obeys. The present article describes two such conversion programs (or Autocoding programs as they are sometimes known) written for the Manchester University Computers (Ferranti's, Mark I and Mercury). The first of these programs, the Mark I Autocode, is now obsolete, but is of interest because the difficulties in programming which it was designed to overcome are characteristic of many machines still in use, e.g. the Ferranti Mark I*. These programming difficulties are discussed and compared with those of Mercury, for which the second autocode programming system is designed. The second and main part of this article is a users' description of this latter program.

THE AUTOCODE PROGRAM FOR THE MARK I COMPUTER

A users' description of this program has been given elsewhere (Brooker, 1956). The following "program" for evaluating the limit of the arithmetic-geometric sequence

$$a_{n+1} = \tfrac{1}{2}(a_n + b_n),\ b_{n+1} = \sqrt{a_n b_n},\ (a_0 = 1,\ b_0 = 0\cdot5)$$

will give some idea of the "instruction" code, and hence of the degree of simplicity introduced.

$2\ r_1 = 1$	sets a_0 (2 is an entry point)
$r_2 = 0\cdot5$	sets b_0
$1\ r_3 = r_1 \times r_2$	forms $a_n b_n$
$r_4 = r_1 + r_2$	forms $a_n + b_n$
$r_1 = r_4/2$	forms a_{n+1}
$r_2 = F_1(r_3)$	forms b_{n+1} where F_1 is $\sqrt{}$
$r_4 = r_1 - r_2$	$a_{n+1} - b_{n+1}$ (>0)
$\rightarrow 1,\ r_4 > 0\cdot000001$	tests for convergence
$(\rightarrow 2)$	starts the program.

The main difficulties encountered in conventional programming for the Mark I Computer are (1) the scale factor difficulty, (2) the use of the two-level storage system, and (3) the language difficulty.

The scale factor difficulty arises because the Mark I is a fixed binary-point machine. It is therefore necessary to arrange the calculation so that all the quantities remain within the capacity of the storage registers (40 binary digits) and yet can still be represented to the required accuracy. This difficulty is normally *overcome* by studying the magnitude of the intermediate quantities arising in the calculation and assigning *fixed* scale factors to individual numbers or groups of numbers so that their maximum values are just within the permitted range, usually ± 1. In an autocode program the difficulty can be *removed* in one of two ways. The first is to work throughout with double length arithmetic keeping, say, 40 binary digits for the fractional part of a number, and 40 digits for the integral part. Alternatively, each number occurring in the calculation may be associated with its own *adjustable* scale factor: an interpretive scheme for doing this has been described by Brooker and Wheeler (1953). This so-called *floating-point* technique requires the use of routines for carrying out operations of the form $a.2^p + b.2^q$, etc., and for this reason the time of execution of a floating-point instruction is very long (addition 40 msec, multiplication 24 msec, division 90 msec), compared to that of the corresponding fixed-point machine instructions. However, this is partly offset by the fact that fewer instructions of the floating-point variety are needed. A floating-point system is, in fact, used in the Mark I Autocode, although it is the author's opinion that the former alternative would have been more generally satisfactory for the Mark I machine.

The second difficulty relates to the physical nature of the storage facilities. The programmer's task would be made easier if, instead of two levels of storage, that is a small "fast" working store and a large "slow" auxiliary store, there was just one single storage medium of indefinitely large capacity and rapid accessibility. However, since the adoption of floating-point arithmetic rules out the need for rapid accessibility, it is possible to regard the slow store (magnetic drum) as the working store, and use the fast store to record the interpretive routine for floating-point arithmetic, together with a trackful each of numbers and instructions. The content of a *track* of the drum is the minimum amount of information that can be transferred between the two levels of storage in one operation. Each track of the drum holds 32 numbers, so that the locations on 128 tracks can be labelled $r_0, r_1, r_2, \ldots r_{4095}$, the locations r_0–r_{31} being on track 1,

r_{32}–r_{63} being on track 2, and so on. To gain access to any one of them, the interpretive routine first determines the track on which it lies, then transfers the contents of the track in question to the fast store, and finally selects the particular location required. The access time for any operand is thus at least 40 msec (the time for a reading transfer), and for the corresponding recording operation at least 90 msec (the time for a writing transfer). These figures about match the times of floating-point operations, although the ratio of access time to computing time can be reduced still further by eliminating unnecessary magnetic transfers when the "magnetic" address of any operand involves the same track as the operand last selected (which may occur in the same instruction or in the previous instruction). Since operands will be selected from the "neighbourhood" of previous operands, this proves a useful time-saving device.

As regards the instructions, one trackful of these is normally kept in the fast store, and when these have been used up arrangements are made to replace it by the contents of the next track, or, in the case of a "jump" instruction, by that of whichever new track is involved. This means of access to instructions is efficient because the time taken to transfer each trackful is negligible compared with the time taken to execute the floating-point operations which they initiate: the ratio is still quite reasonable even where ordinary (machine) instructions are concerned, and indeed this track-changing business is a characteristic feature of the conventional programming technique used with the Manchester machine.

The third difficulty, the use of a strange language, also arises from the need to partition the fast store (like the drum) into blocks of 32 numbers for the purpose of magnetic transfers. As a result one is virtually compelled to use a scale-of-32 numbering system for all items of information. For this purpose the 32 symbols

/ E @ A : S I U ½ D R J N F C K T Z L W H Y P Q
O B G " M X V £

are employed.† The reasons for this are explained in ref. (1), and their validity is proved by the fact that although many alternative schemes have been suggested, none has survived in practice. Nevertheless, attempts to replace it are natural enough: it does not encourage beginners when they learn that the instructions

O / T /
@ E T C
R E T A

mean "add the contents of locations 24 and 34 and place the result in 44." For this reason it was decided to make a complete break with the conventional programming style and adopt a language as simple and as close as possible to elementary arithmetical formulae. This would be necessary in order to attract the occasional user, since the price paid, in terms of speed, for overcoming the first two difficulties, means that its use is more or less confined to *ad hoc* problems, and problems which might otherwise have been considered too small for a large-scale computer. As an extreme example of what it has been used for, it may be interesting to record what must surely be one of the smallest genuine calculations ever done on such a computer—the author was asked for assistance in calculating the cube root of 0·62315670985. The following program was therefore prepared and run on the machine, which was fortunately available at the time.

$$\begin{array}{llllll}
1 & r_1 = F_4(0\cdot 62315670985) & & & & \\
 & r_2 = r_1/3 & & (F_3 & \text{exp} & \\
 & r_3 = F_3(r_1)* & & F_4 & \text{log} & \\
 & \quad H & & * & \text{print}) & \\
 & (\;\cdot 1) & & & &
\end{array}$$

It was also considered important (again for psychological reasons) to make the description as short as possible. What the author aimed at was two sides of a sheet of foolscap with possibly a third side to describe an example. This was achieved [see ref. (1)] although, as time went on, various special operation codes were added, e.g. for the solution of differential equations, and arithmetic with complex numbers, which, together with hints on coding, brought the total to about nine sides of foolscap. Nevertheless, these still made a considerably smaller volume than the conventional programming manual which proves so indigestible to many would-be machine users.

Since its completion in 1955 the Mark I Autocode has been used extensively for about 12 hours a week as the basis of a computing service for which customers write their own programs and post them to us. These are then checked and prepared for the machine, and the results returned to the customer. The total delay is normally less than a week, while for members of the University who are on the spot it is, of course, much less. One customer's experience of this Autocode service has been described by Lunt (1957).

THE MERCURY AUTOCODE

Turning now to the Ferranti Mercury, it is appropriate to start by considering how the difficulties outlined above are modified by the design of the machine. Firstly, the scale factor difficulty has been virtually removed by the inclusion of built-in floating-point arithmetic. Indeed, the machine has been criticized in this respect as not providing fixed-point arithmetic facilities. However, there would be little point in providing fixed-point facilities with the 40-digit accumulator because, owing to the design of the machine, these would be little faster than the corresponding floating-point operations. The main advantage of them would be that they could be made to operate on numbers of 40 digits rather than 30 digits, which is the precision of a floating-point number. Fixed point arithmetical facilities and logical

† Any set of 32 symbols would suffice but these were chosen because they occur on the upper case of a standard typewriter.

operations are provided for 10-digit words, and these prove quite adequate for the "red tape" of organization.

As regards the question of two-level storage, one can only say that this has been alleviated, but not entirely removed. The fast store of Mercury amounts to 1024 40-digit registers (but only half this is available for instructions) which is four times the size of that on the Mark I computer. The auxiliary magnetic drum store is still, therefore, essential—and could only be dispensed with if the fast store were extended to at least 8000 numbers. A feature of Mercury which compares unfavourably with the Mark I is the *ratio* of access times to the two forms of storage. Thus for the Mark I the time to transfer a trackful (128) of instructions from the drum to the fast store is 40 msec, and the access time to each of these instructions within the fast store is approximately 1 msec. For Mercury, however, the corresponding times are 24 msec, and 0·1 msec, a ratio which is six times that of the Mark I. Fortunately, however, the larger size of the fast store makes references to the magnetic drum store considerably less frequent.

Finally, we have to compare the language difficulty of the two machines. Because of the comparatively large fast store, the representation of instructions is not influenced to anything like the same extent by the physical nature of the auxiliary drum store, and a more or less conventional decimal code is used both for the address part, and the function and "B" digits of an instruction. Thus in place of

O / T /	we now have	400	24
@ E T C		420	34
R E T A		410	44

Nevertheless Mercury has its own special difficulties which arise from using the 10 binary digits of the address part of an instruction to refer to information in 2048 registers

$$0, 1, 2, \ldots, 2047.$$

To overcome this, the address part of an instruction may refer either to the 1024 double registers (40 binary digits)

$$0, 2, 4, \ldots, 2046$$

or the 1024 single registers (20 binary digits)

$$0, 1, 2, \ldots, 1023$$

or the 1024 half registers (10 binary digits)

$$0, 0+, 1, 1+, \ldots, 511, 511+$$

according to the nature of the instruction in question. In the case of half registers a duplicate set of instructions is provided which enables the instructions to reach the second quarter of the store, namely the half registers

$$512, 512+, \ldots, 1023, 1023+.$$

To some extent this feature of the machine can be concealed by writing all addresses in the "medium" number system as above, and arrange for the necessary doubling and halving of addresses to be done by the input routine. Nevertheless it has to be revealed at some stage in a programming course and can usually be relied on to mystify pupils for a day or two!

It follows from the preceding discussion that a large-scale autocode program is very appropriate for use with Mercury. The built-in floating-point arithmetical facilities mean that the resulting program will not be of an interpretive nature, but will consist of ordinary machine instructions and hence will be almost as fast as a conventional program. The size and accessibility features of the two levels of storage mean that it is worth while to accept these and not to disguise them. Indeed, many problems will be so small that there will be no need to consider auxiliary storage at all, and therefore this need only be described for the benefit of "advanced" programmers. With these difficulties removed it is possible to concentrate on the language question and the form of presentation of the calculation. Thus instead of the rather elementary instructions of the Mark I Autocode, amounting to what is nothing more than a two- or three-address instruction code, one can instead arrange to interpret more complicated algebraic formulae. There is a practical limit as to how far one can go in this direction because of the restrictions on the number and style of the symbols available on commercial teleprinter equipment. What is really needed is a "two-dimensional" teleprinter and corresponding keyboard which will cope with both affices and suffixes. In the absence of such an instrument, it is necessary to arrange for a mathematical expression to be treated as a one-dimensional sequence of symbols. This does not restrict the programmer from using affices and suffixes in his *manuscript* form of the program, but it does mean that the format of the expression must be capable of being interpreted unambiguously when arranged in one-dimensional form. Thus, for example, the manuscript form

$$c_i = a_{(i-1)}a_{(i+1)} - b_{(i-1)}b_{(i+1)}$$

becomes in one-dimensional form

$$ci = a(i-1)a(i+1) - b(i-1)b(i+1).$$

In employing the only two bracket symbols on the Mercury teleprinter code to designate compound suffixes in this manner, the possibility of using them in the conventional way is sacrificed, and as a result the most complex algebraic expressions which can be treated by Mercury Autocode must be essentially parenthesis free. Thus, for example,

$$b_i = a_{(j+1)}(a_{(j-1)} - 1)$$

has to be written as

$$bi = a(j+1)a(j-1) - a(j+1).$$

The introduction of a second style of bracket symbols $\{\ \}$ would remove this difficulty, thus

$$bi = a(j+1)\{a(j-1) - 1\}.$$

THE AUTOCODE SYSTEM FOR THE MERCURY COMPUTER

The following is an account of a proposed Autocode system for the Manchester Mercury Computer. In this scheme the *program* consists of an ordered sequence of *instructions* which employ the following symbols:—

$$a\ b\ c\ d\ e\ f\ g\ h\ u\ v\ w\ x\ y\ z\ \pi$$
$$i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t$$
$$\cdot\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$
$$+\ -\ \times\ =\ >\ \geq\ *\ (\ ,\ )\ \simeq\ \rightarrow\ /\ \phi\ '$$

The program is ultimately presented to the machine in the form of a length of perforated paper tape which is scanned by a photo-electric tape reader, the input unit of the machine. The *program tape* is prepared by means of a manual keyboard perforator on which are engraved the standard symbols. The material is *punched* in the conventional fashion, namely from left to right and down the column. Each instruction is followed by two special symbols *CR* (carriage return) and *LF* (line feed) which are provided for this purpose. There is also an *erase symbol* ⋠ which is used for overpunching mistakes.

The instructions read from the tape are placed in the *instruction store* of the machine. The numerical quantities to which they refer are kept in the *working store* of the machine. The program will include instructions to set the initial values of such quantities, either directly or by reading them from a further *data tape* by a process similar to that by which the instructions themselves were read into the machine.

The instructions fall into two classes, the *arithmetical* instructions which perform the calculation proper, and the *control* instructions which "organize" the calculation (e.g. arrange to repeat cycles of arithmetic, select alternative courses of action, or, as already mentioned, read further numerical data into the machine). Instructions of this latter class are the characteristic features of *automatic* calculating machines: they distinguish them from desk machines which are "controlled" by the operator himself. Both kinds of instruction need to refer to the working store, so that it is appropriate to start by describing the notation used to refer to the numbers stored therein.

THE WORKING QUANTITIES

The numbers recorded in the working store are also of two kinds, *general variables* and *indices*, which, like the two kinds of instructions, relate mainly to the calculation proper and its organization. Thus the variables have numerical values in the range $10^{-70} < |x| < 10^{70}$ and are recorded to a precision of eight/nine decimals, while the indices are restricted to integral values in the range $-512 \leq i \leq 511$.

THE VARIABLES

These are divided into three sets as follows.

The *main set* consists of 480 variables which can be divided into a maximum of 15 groups associated with the *variable letters*,

$$a\ b\ c\ d\ e\ f\ g\ h\ u\ v\ w\ x\ y\ z\ \pi.$$

For example, they can be arranged as a single group of 480 variables

$$v_0\ v_1\ v_2\ .\ .\ .\ v_{479}$$

in which case the *directive*

$$v \rightarrow 479$$

is written at the head of the program. Alternatively they could be arranged in three equal groups, thus

$$a_0\ a_1\ a_2\ .\ .\ .\ a_{159}$$
$$b_0\ b_1\ b_2\ .\ .\ .\ b_{159}$$
$$c_0\ c_1\ c_2\ .\ .\ .\ c_{159}$$

the necessary directives being

$$a \rightarrow 159$$
$$b \rightarrow 159$$
$$c \rightarrow 159$$

It is intended that these groups shall reflect any natural grouping of the quantities occurring in the problem, and provided that the total number of variables does not exceed 480 the number and size of the groups is at the disposal of the programmer.

In addition to the main variables, there are fifteen *special variables* represented by the letters

$$a\ b\ c\ d\ e\ f\ g\ h\ u\ v\ w\ x\ y\ z\ \pi$$

employed without a suffix. These will be a common feature to every program which cares to use them. The special variable π may be assumed to have the value $3{\cdot}14159$ . . . until otherwise altered.

The *auxiliary variables* will not be introduced here because for a large number of applications the main (and special) variables will be sufficient.

INDICES

These quantities are represented by the 12 letters

$$i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t.$$

Although permitted integral values in the range $-512 \leq i \leq 511$, emphasis is placed on positive values because they are primarily intended to be combined with variables in the form, for example,

$$x_i \text{ or } x_{(n-1)} \text{ or } x_{(s+50)}$$

to represent a free suffix: that is, these expressions may represent any one of the variables

$$x_0\ x_1\ x_2\ .\ .\ .$$

depending on the particular value of the index in question. Thus if $n = 4$, then $x_{(n-1)}$ will refer to x_3. The last two expressions illustrate the most general form which a suffix may take, namely (index $\pm$ integer). In calculations of a repetitive nature an index will assume a range of values, and to arrange this it is necessary to be able to compute with indices as separate items in much the same way as variables.

In preparing the input tape, all expressions are recorded in a *one-dimensional* form, thus x_3, x_1, and $x_{(s+50)}$ appear as $x3$, xi, and $x(s+50)$, respectively. Consequently it is not possible to distinguish, in a product, between, say, xi meaning x_i and xi meaning "x times i". In order to resolve this difficulty, a convention will be introduced later for ordering the factors in a product.

NUMERICAL VALUES

Explicit numerical values will have to be introduced into the program at some stage, so that it is necessary to explain how these are written. The standard form is

integral part   decimal point   fractional part

omitting what is unnecessary. Thus, as is the case in writing suffixes, the decimal point can be omitted in whole numbers. However, absolute standardization of form is not necessary and, for example, the number 15 may also be written as

15·   15·0   015·0

All these and similar variations will be accepted by the machine.

Similarly $\sqrt{2}$ to six significant figures may be written

1·41421   01·41421   001·4142100   etc.

In the case of the number 1·414213562372095, however, only the first ten significant figures will be treated, and the remaining figures will be ignored (although they will be read from the tape).

THE ARITHMETICAL INSTRUCTIONS

The basic form of the instructions for computing variables may be illustrated by the following example:—

$$y = 2mna_{(m+1)} + a_m n + ma_n + 0{\cdot}01m + 0{\cdot}01n$$

which gives the new value of the variable to be altered (in this case y) in terms of other quantities.

In general, the right-hand side may involve any number of products which may each have any number of factors either variables, indices, or constants. As already mentioned, it is necessary to distinguish in the "one-dimensional" form

$$y = 2mna(m+1) + amn + man + 0{\cdot}01m + 0{\cdot}01n$$

between am meaning a_m and $a \times m$. The convention adopted is that an index immediately following a variable letter is treated as a suffix so that the above expression is interpreted as

$$y = 2 \times m \times n \times a_{(m+1)} + (a_m \times n) + (m \times a_n) + (0{\cdot}01 \times m) + (0{\cdot}01 \times n).$$

Further examples of instructions in this general class are:

$$a = 0 \quad x_k = x_k + 1 \quad x_n = x_0 + nh$$

As a further refinement a quotient sign (solidus) may be inserted before the last factor in any product. Thus

$$u = x/a + y/b + z/c$$

and

$$v = 2\pi u/n$$

are possible instructions.

The basic form of the instructions for computing indices may be illustrated by the following example:

$$i = 2mn + m + n + 1$$

The only difference between this and the previous class of instruction is that the quantities on the right-hand side are restricted to indices or whole numbers, and the use of the solidus is not permitted. Further examples of instructions in this class are:

$$i = 0 \qquad n = n + 1 \qquad r = 10p + q$$

FUNCTIONS

The following equations are a means of introducing certain elementary functions into the program. Hence the L.H.S. y stands for any variable, and the argument x for any R.H.S. expression.

y = sq rt (x)	
y = sin (x)	
y = cos (x)	
y = tan (x)	
y = exp (x)	
y = log (x)	i.e. log to base e
y = mod (x)	i.e. modulus of
y = int pt (x)	i.e. integral part of
y = fr pt (x)	i.e. fractional part of.

Examples of instructions in this class are

u = cos ($lx/a + my/b + nz/c$)

a = log ($xx + yy$)

w = sq rt (xxx)

x_n = cos (nd)

The following class of instructions involves functions of two variables, which may each be replaced by R.H.S. expressions.

z = divide (x, y)	i.e. x/y
z = arctan (x, y)	i.e. arctan (y/x)
z = radius (x, y)	i.e. $\sqrt{(x^2 + y^2)}$.

Examples of instructions in this class are:

z = divide ($x + y, x - y$)

u = arctan ($aa - bb, 2ab$)

a_i = radius (x_i, y_i)

Finally, there is the instruction i = int pt (x) for converting a variable (or any R.H.S. expression) to an index (the largest whole number less than x is taken).

THE CONTROL INSTRUCTIONS

These are the instructions which organize the calculation and which, for this reason, are sometimes known as the "red tape."

The most important instructions in this class are the *jump* instructions. Arithmetical instructions are normally obeyed in the order in which they are listed, but from time to time it is necessary to select alternative courses of action as in the following sequence of instructions for calculating:

$$f(x) = \begin{cases} e^x, & x < 0 \\ 1 + \sin x, & x \geq 0 \end{cases}$$

```
           ————————
           ————————
      jump 1, x ≥ 0 ——┐
      f = exp (x)     │
      jump 2       ———│——┐
   1) f = sin (x)  ←——┘  │
      f = 1 + f          │
   2) ————————     ←—————┘
      ————————
```

In this example the first instruction is a *conditional* jump, i.e. if the condition (in this case $x \geq 0$) is satisfied then control "jumps" to the instruction *labelled* 1) and then continues to obey instructions from there onwards; otherwise, if the condition is not satisfied, then the next instruction is obeyed in the usual way. Any instruction can be labelled in this way with an integer in the range 1–120 inclusive.

The second jump instruction in the above example is an *unconditional* jump and needs no further explanation. The general form of a conditional jump instruction is

$$\text{jump } n,\ \alpha > \beta \qquad (\text{or } \alpha \geq \beta)$$

where n is a specific label and α, β are the quantities being compared. These must be either both variables (including a numerical constant) or both indices (including a numerical integer). It is *not* possible to compare a variable directly with an index without first converting the index to variable form. Examples of conditional jump instructions are

jump 1, $x > y$	jump 8, $1{\cdot}41421 \geq a$
jump 50, $i \geq 2$	jump 97, $r > s$

Associated with the above are the instructions

$$\text{jump } (n) \qquad n) = 3 \qquad n) = m$$

where n, m denote any index letters.

The instruction $n) = 3$ makes a subsequent jump (n) equivalent to jump 3. One use for this will be to mark the point of return when calling in a subsequence, thus

```
main sequence        subsequence
————————      ┌→ 10) ————————
————————      │        ————————
————————      │        ————————
n) = 3        │        ————————
jump 10 ——————┘        ————————
3) ————— ←————┐        ————————
————————      │        ————————
————————      └——————— jump (n)
```

In the same way the instruction $n) = m$ makes a jump (n) instruction transfer control to one of several different points depending on the computed value of m. This device is known as a multi-way switch.

CYCLES OF OPERATIONS

Two special instructions are provided to simplify cycles of operations. These take the form

```
  i = p(q)r          or      i = p(−q)r
┌→ ————————               ┌→ ————————
│  ————————               │  ————————
│  ————————               │  ————————
│  ————————               │  ————————
└—— repeat                └—— repeat
```

and arrange to execute the intervening instructions for values of i running from p by increments of q (or $-q$) to r. Any index may be used in place of i and p, q, r, may be any integers or other indices, subject of course to the restriction that $r - p$ is a multiple of q, otherwise the cycle will continue indefinitely. Cycles within cycles are permitted up to a nest of 8 deep. The following sequence,

```
        x = 0                 "a_ij" = a_n(i−1)+(j−1)
        i = 2(1)n
  ┌———→ p = i − 1             a_11  a_12  .  .  .  a_1n
  │     j = 1(1)p             a_21  a_22  .  .  .  a_2n
  │  ┌→ q = np + j − 1
  │  │  x = x + a_q
  │  └——— repeat
  └—————— repeat              a_n1  a_n2  .  .  .  a_nn
```

for example, illustrates a cycle within a cycle (for calculating the sum of the off-diagonal elements of a symmetric matrix).

INPUT FROM A DATA TAPE

Instructions are provided to read numerical information from the input tape into the computing store. These are

read (x)

read (i)

which mean "*read* the next number on the tape and set the specified variable (or index) to this value." As each number is read the tape is advanced to bring the next number to the reading station. Numbers must therefore be punched in the order in which they are required. Each number is written in the manner already described (preceded by minus sign if negative) and terminated by *CR LF* (or a double space *SP SP*).

OUTPUT

To print results the simplest procedure is to write a ? symbol after the arithmetical instruction giving the relevant value of the quantity in question, e.g.

$$x = zyy + 1? \qquad i = i + 1?$$

This will cause the new value of x (or i) to be printed immediately after computation. Each number is printed on a new line to 10 decimal places, so that results obtained in this fashion will be listed in a single column at the left-hand margin of the page.

In case the results are required in tabular form, the following instructions are provided:

print (x, n, m) print (i, n, m)
space
newline

The *print* instruction causes the value of x (or i) to be printed in fixed decimal point style with n and m positions allowed, respectively, for the integral and fractional parts (n, m   15). One figure is always printed before the decimal point, but other non-significant zeros are suppressed. If negative a minus sign appears before the first digit printed. Each number is automatically followed by two spaces, but extra spaces can be "programmed" by means of the *space* instruction. The instruction *newline* combines the operations of *line feed* and *carriage return*.

All numbers printed by the machine can subsequently be read in again by means of the *read* instructions, i.e. input and output are complementary.

CONCLUSION

To illustrate the foregoing description the program for a small computation job is given below, together with the necessary input directives which make up the program tape.

$$\text{Tabulate } y \quad \sin x \quad \frac{1}{3}\sin 3x \quad \frac{1}{5}\sin 5x \quad . \; . \; . \quad \frac{1}{(4n \quad 1)}\sin(4n \quad 1)x$$

for x 0, $\frac{\pi}{20}$, $\frac{\pi}{10}$, . . ., π, and for specific values of n.

```
          chapter 1
 11   1)  m     4n   1                19
  2       r     1(1)20                 1
 15       x     rπ/20                 (5·5 msec)
  3       newline                    (90 msec)
  7       print (r, 2, 0)           (150 msec)
  4       y     0                      4
  2       k     1(2)m                  1
 13       u     sin (kx)             (12 msec)
 14       y     u/k     y             (5·5 msec)
  4       repeat                       5
  2       space                      (60 msec)
  7       print (y, 1, 9)           (420 msec)
  4       repeat                       4
  1       stop                         1
          close
  2       n     15      | starting     2
  4       across 1/1    | sequence   (100 msec)
          close         | for case
                        | n    15
```

The program proper needs no explanation. The numbers on the left give the number of *registers* occupied by each instruction in the instruction store, while those on the right give the execution time in units of 60 μsec, unless otherwise stated. It is hoped that these will serve in place of detailed rules as a general guide to the *length* and *speed* of a program. The rest of the program tape, namely

```
chapter 1
   .
   .
   .
 close
 n   15
across 1/1
 close
```

is concerned with the input organization. This somewhat elaborate format is necessary to deal with larger programs which, for technical reasons, have to be partitioned into *chapters*, each chapter being restricted in length to 896 registers. Very many programs, however, will not extend beyond one chapter, and for these the above format can be used without further explanation. The starting sequence is really a chapter by itself which is entered automatically by the final *close* directive. In this case there are just the two instructions

```
 n   15
across 1/1
```

which first set n and then "jump to the instruction labelled 1 in chapter 1" and hence initiate the calculation proper.

It is hoped to give details of further facilities, including partitioning into chapters, the auxiliary variables, solution of differential equations, etc., in a further paper.

REFERENCES

1. BROOKER, R. A.
   "The Programming Strategy Used with the Manchester University Mark I Computer," *Proc. I.E.E.*, Vol. 103, Part B, Supplement No. 1, p. 151 (1956).
2. BROOKER, R. A., and WHEELER, D. J.
   "Floating Operations on the EDSAC," *Mathematical Tables and other Aids to Computation*, Vol. 7, p. 37 (1953).
3. LUNT, S.
   "Process Development and Plant Design," *Trans. Soc. Instrument Technology*, Vol. 9, p. 87 (1957).