Dr John von Neumann,
with the compliments of
the authors.
[illegible] . 16/1/48

CODING FOR A.R.C.

by

Andrew D. Booth

and

Kathleen H.V. Britten.

Institute for Advanced Study,
Princeton.

Sept. 1947.

ERRATA.

P 11. line 3. For "are" read "is".

P 11. para. 2.02, line 4, For "2P" read "2^{2P}".

P 14. Memory position 42 should read : $a_1b_2 + a_2b_1 + (a_1b_1)_2 = c_2$

P 17. para. 2.05, line 4. For "sin" read "sin(x)".

line 10. For "immediatly" read "immediately".

line 10. For "adopted" read "adapted".

P 19. Orders 28 and 33, read L(8) and R(1) respectively.

Orders 42 and 44, for "(n)" read "M(n)".

P20 .. 4 lines from bottom, for "sins" read "sines".

P 21. Orders 8, 13, and 19, for $S_L(6)$ read L(6).

Order 29, for 149 read +140.

P 27. For "S_L" read "L".

P 28. For "S_L" read "L".

PREFACE.

Although this report deals specifically with the coding of problems for A.R.C. (Automatic relay computer) it can be considered as the successor to our previous work on the design of electronic computers. This is because we have made the code of A.R.C. identical with that projected for the electronic machine.

The advantage of this arrangement is obvious, we shall be able to transfer any problem which has been coded for A.R.C. directly to the electronic machine when this becomes available. In addition, the experience which we gain in using the simpler machine may form a valuable guide to possibly advantageous modifications of the code for the faster computer.

We again wish to thank the Rockefeller Foundation and the British Rubber Producers' Research Association for valued encouragement and support.

A.D.B.
K.H.V.B.

Sept. 1st, 1947.

TABLE OF CONTENTS.

CODING FOR A.R.C.

1.0 General introduction.

The present report deals specifically with the coding of problems for A.R.C. (Automatic relay calculator), but, despite this, may be considered to form a continuation of our previous report on the design of electronic computers. The reason for this universality lies in the fact that, as will become evident from 1.1, the number of logically complete codes which do not use bizarre operations is limited. In addition, it is our aim to have a standard procedure so that, when our electronic machine becomes available, only limited replanning will be necessary.

1.1 The nature of a code.

It was explained, in the first report, that although logically simple arithmetic operations of the type +, −, shift, were sufficient to programme more complex operations such as multiplication and division, it was nevertheless desirable to include unit operations of this more complicated type in order to save time. We shall now take up the design of a code in somewhat greater detail.

In the first place, to be satisfactory, a code must be <u>arithmetically</u> complete. By this is meant that it must include sufficient basic operations to make possible the generation of any function defined by the ordinary rules of arithmetic. Let us examine this in more detail, and suppose that we take as our basic set only the operations +, −, L and R shift and control shift. The most obvious operation upon which to test the code is that of multiplication.

Consider two purely binal numbers a and b of n and m digits respectively. Then to form the product we must do one of two things :-

if $a_L \neq 0$
   1.1 Add b into partial product already formed in adding device A.
   1.2 Shift b one place to right.
or if $a_L = 0$
   2.1 Shift b one place to right.
a_L being the extreme left hand digit of a.

The question now arises :- can the decision as to which of the alternative procedures is appropriate be decided on the basis of the postulated order set? We shall show that the answer is in the affirmative. Assume first that the left shifting operation transfers the L.H. digit of any number in the accumulator to the extreme right hand position of the register. Let the first set of orders (1.1 above) be stored at memory location 2^{-P1} and the second set (2.1) at 0. Consider the following process :-

1.0 0 to ${}_cR$.
1.1 M(a) to ${}_cA$.
1.2 Left shift.
1.3 A to M(a).
1.4 R to ${}_cA$.

1.4	R to $_c$A.
1.5	$n - p_1$ left shifts.
1.5	+ M(1.8) to A.
1.6	A to M(1.9).
1.8	C to M(0 + contents of A after 1.4)

<u>Contents of A zero.</u>

2.1	M(b) to $_c$A.
2.2	Right shift.
2.3	A to M(b).
2.4	C to M(1.0).

<u>Contents of A 2^{-p_1}.</u>

1.9	M(ab) to $_c$A.
1.10	+ M(b) to A.
1.11	A to M(ab).
1.12	M(b) to $_c$A.
1.13	Right shift.
1.14	A to M(b).
1.15	M(1.8) to $_c$A.
1.16	-2^{-p_1} to A.
1.17	A to M(1.8).
1.18	C to M(1.0).

The notation used is that of our first report, but one or two points require explanation. Order 1.0 is needed since no facilities exist for clearing the register R. M(x) represents the memory location of the quantity (x). The essence of the process is to separate out the digit of a which is to multiply b and then to decide whether it is zero or unity by substituting it in a control shifting order.

Of course, it is not suggested that the above procedure forms a practical means of performing a multiplication; it is included mearly to show that the code is complete as far as arithmetic operations are concerned. Division can be treated in a similar fashion and likewise the extraction of square roots, or alternatively these operations can be programmed as iterations.

A precisely similar method can be used to establish the zero (or otherwise) character of a number of more than one digit, and then, if the operation |M| to $_c$A is available, sign can be tested by finding whether :-

$$|M| + M > 0.$$

This argument establishes the sufficiency of a code containing only the orders :

$$+ \,,\, - \,,\, L \,,\, R \,,\, |\;| \,,\, C \text{ to } M(x)$$

together with a sufficient number of purely transfer orders to make each arithmetic unit accessible to numbers from the memory.

A very simple code of this type would be justified only in a machine of enormously high speed in which the technical difficulties of increasing the number of orders might be prohibitive. This speed range is not, however, reached in any machine at present under discussion.

So long as a code is complete and includes the above basic operations it matters little what extra operations are inserted, and the nature of these will be governed by the saving of time which they effect and from their simplicity from the standpoint of engineering.

Evidently, from the point of view of speed, a multiplication order will be advisable, and , if feasible, one for division also: if the memory capacity of the machine is limited, the inclusion of basic operations of this nature is even more justified, since the large number of simpler orders needed to replace them would fill a significant proportion of the available memory.

The inclusion of a conditional transfer order is also well justified from the above points of view, the exact specification of its nature is, however, more speculative. Certainly an order which selected only one digit for discrimination would be of limited use in a machine with automatic multiplying and dividing facilities. Possible alternatives are :-

$$\left.\begin{array}{l} X \geqslant 0 \\ X < 0 \end{array}\right\} \quad (1)$$

$$\left.\begin{array}{l} |X| > 0 \\ |X| = 0 \end{array}\right\} \quad (2)$$

and a choice between them, on purely logical grounds, is more difficult since either can be derived from the other. Thus, given (1) nullity can be detected by examining |X| and [illegible], a process which requires 3 orders; whilst from (2), (1) can be derived by considering |X| + X in a process requiring 3 orders. From the engineering point of view, (1) is the simplest to apply in a machine which represents negative numbers by complements, whereas (2) is the easier in a machine involving direct subtraction. The decision thus lies in a study of the relative frequency with which it is desired to sense sign rather than nullity, and since it appears, from the examples already coded, that the former operation is more usual, we propose for the present to have (10) in our order set. In the relay machine which represents negative numbers by their complements this decision is also in accord with the desire for engineering simplicity.

1.2 The code of A.R.C.

We have seen, in 1.1, the type of order set needed to secure sufficiency, and it remains to discuss those further orders which will be available to A.R.C..

The fundamental difference between A.R.C., and the electronic computer, envisaged in our first report, is one of speed. In general, the extra operations which we have inserted are mearly devices to limit the number of transfers of data which have to be made during a calculation. For example: R to M,; this eliminates the necessity of the orders:

$$\begin{array}{l} R \text{ to } {}_cA \\ A \text{ to } M \end{array}$$

which would be needed under our previous code. Again: A to R is particularly useful in that it makes R available as temporary storage, for example when polynomials if the type:

$$a_n x^n + a_{n-1} x^{n-1} \text{ --- } + a_0$$

are to be calculated, the particularly elegant iteration:

$$\begin{array}{ll} a_n \text{ to } R & a_n \\ M, R \text{ to } {}_cA & a_n x \\ + a_{n-1} \text{ to } A & a_n x + a_{n-1} \end{array}$$

$$
\begin{array}{ll}
\text{A to R} & a_n x + a_{n-1} \\
\text{M} \times \text{R to cA} & a_n x^2 + a_{n-1}x \\
+ a_{n-2} \text{ to A} & a_n x^2 + a_{n-1}x + a_{n-2} \quad \text{etc.}
\end{array}
$$

is available, in which x stays in M, and no transient storage is required for the partial answers.

Another modification of the original code is the generalisation of the shifting orders L and R. These are modified to: L(n) and R(n) and read, shift contents of A (n) places to left or right respectively. In L(n) the left hand non-sign digits of A are shifted into the right hand end of the register R whose contents are likewise shifted to the left. In R, however, the right hand digits of A are lost. These extensions are particularly advantageous in conversions to and from the binary scale, and, as it is proposed that A.R.C. shall perform its own conversions, a considerable saving of time will result from this modification. An additional virtue of the new orders lies in the saving in time which they produce in interpolation processes.

The absolute control transfer order C to M(x) also comes in for modification. It has been found that a much more flexible form of this order is : C to M(x+ k) which reads :

"Advance control k orders."

The advantage of this version lies in the fact that k depends only upon the particular sequence in which the control finds itself, whereas in C to M(x) the position (x) depends not only on the basic sequence being executed but also on the position of this sequence in the whole scheme of computation. Thus a partial substitution will be required every time the sequence is used; the new variant eliminates this necessity.

Since A.R.C. has only 21 binal accuracy, orders cannot be stored two at a time in M; this means that the partial substitution order A_R to M is no longer needed, although A_L to M is, of course, retained.

In view of the limited high speed memory capacity of A.R.C. (256 numbers) a most important feature will be its ability to record and draw on information punched on tele-type tape; this function of the control we now proceed to discuss.

There are four possible situations in which the machine makes use of its low speed memory:

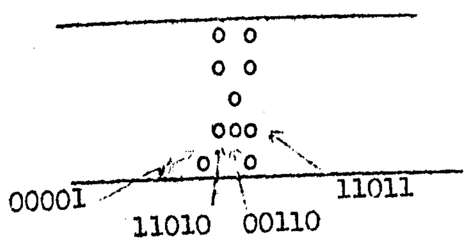
1) Decimal and order input to machine.
2) Input of binary data such as tables calculated by the machine and stored for future reference.
3) Output of the machine in binary form for its own subsequent use.
4) Decimal output of the machine required for communication with the human operator.

Since the basic construction of the digital (as distinct from abstract) code depends largely on:

a) Available tele-type equipment.
b) Coding adopted for decimal data.

we will now consider these two factors rather more fully.

Normal teletype equipment operates on a five hole code and a natural choice of punching would be to represent our 20 binal by four groups of five punchings so that, for example, 11011, 00110, 11010, 00001 would appear as :



Although this scheme makes the best possible use of the available tape space, it would be very wasteful when storing decimal numbers. Our procedure then would be the following: suppose the number 9864 is to be recorded, we write down the binary representation of each digit, viz:

9 = 1001
8 = 1000
6 = 0110
4 = 0100

and then represent the decimal number by its tetrad of binary digits :

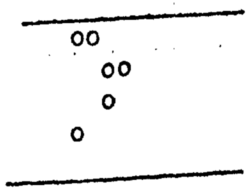
9864 = 1001,1000,0110,0100.

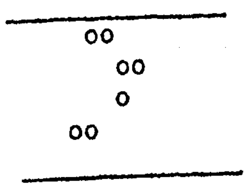
This representation could now be punched on the tape using the five hole code and prefixing each tetrad by a zero, i.e.

9864 = 01001,01000,00110,00100.

which would exactly fill our 20 digit memory space. It is evident, however, that this process is wasteful and that, by adopting the four hole, or octal, coding of (1) we could insert a five place decimal number in our available space. In A.R.C., where the accuracy is already rather low, we are certainly unable to sacrifice this extra decimal digit and consequently are forced to adopt a four hole coding of five rows.

A word is appropriate here on the manner in which we propose to store the sign digit of coded decimal numbers. The actual punching on the tape will consist of 6 rows of four hole groups. The first group will be either (0000) or (0001) according as the number is positive or negative. The remaining five groups will then contain the coded decimals as described above. In reading out this array into the machine, all of the first (sign) group of digits will be ignored except the units component, so that the number reaching the machine will be a single sign digit plus 5 groups of 4 digits, exactly filling the 21 memory spaces available. As an example of the appearance of our tape, we give below the punchings for the two numbers 9864 and -9864.


9864


-9864

This grouping of the digital input is not without effect on the disposition of orders. Inspection of the code table of p(30) shows that the total n number of orders is 26 i.e. < 32 ($= 2^5$) and they are thus representable by a single five hole punching. Again, since this is beyond the range of a four hole code it follows that we shall require two groups of four holes to represent our orders. This arrangement leaves 12 positions to be used for number locations in the memory and will be perfectly adequate for our high speed memory which contains only 256 ($= 2^8$) positions, but quite insufficient for the slower tape memory which should have a capacity of at least 10^5 words. In view of this, our tape reference order will be of the following form:

"Proceed to the word on the tape whose position is given by the number contained in high speed memory location (X)."

Although this is slightly slower than the direct order:

"Proceed to the word on the tape whose position is given by (X)."

it makes available a tape memory of 2^{20} or about 10^6 words.

In view of the slow speed of the tape memory (.25 sec/word) the hunting out of a number from an aggregate of 10^6, on a single tape, could take up to 70 hours. To reduce this time we propose having at least 8 separate tape feeds and, of course, only tapes holding data relevant to the problem in hand will be inserted into these feeds. Each tape will hold about 1000 words, so that the maximum hunting time will be 4-5 mins. To avoid any wait, a hunting order will start the tape moving towards the desired set of data, and then direct the control to proceed with the main sequence of operation. Whilst the machine is performing these the tape will be moving towards the required position and on arrival will stop. We assume that the machine is by now ready to use the tape data, an order then appears which says:

"Read data from tape into memory positions (X_i) to (X_{i+k})."

Of course, this arrangement will break down occasionally when the required position on the tape cannot be predicted in advance; so that, if the tape has not reached the required position, an interlock stops the machine until the tape is ready and then proceeds with the last order. To avoid delays of this kind the following procedure can be adopted. Suppose that sin(x) is required in the course of a calculation, but that (x) is initially unknown. The first tape order states:

"Move tape to position ($\sin \pi/4$)."

and, since the sin table will contain only 128 entries, the hunting time for the latter will be only 8 to 10 seconds which is much more reasonable.

Enough has been said to make clear the general properties of our code; its application will become evident from the specific problems contained in the body of this report.

To conclude, we give the exact make up of a coded order:

Digit	1------8	9------15	16------20
	MEMORY LOCATION	SEQUENCE	ORDER CODE

The position marked SEQUENCE is used only in the orders transfering data from tape to memory positions X_i to X_{i+k} and contains the number k. The partial substitution order A_L to M(X) replaces digits 1-8 of the order by those contained in the first eight places of the accumulator. Table I is a complete list of orders with code numbers.

1.3 The technique of coding.

We shall now give an outline of the form in which a coded example will appear, and discuss a particularly elegant method (due to Goldstine and von Neumann) of treating iterative processes.

The coding of a problem will be in three parts:

1) Mathematical formulation.
2) Schematic code.
3) Detailed code.

Of the first part, all that need be said is that the methods used must involve only arithmetic operations, so that, for example, a differential coefficient must be replaced by its finite difference approximation. We observe also, that all quantities in the calculation must be so adjusted that they remain less than unity.

The schematic code is much more complex. In a calculation which involves no iteration or decision on the part of the machine, the process is trivial since the code mearly duplicates those operations which a human computer would follow. We may represent the coding process, in this case, by a line:

$$\boxed{\text{IN}}\text{----------}\rightarrow\text{----------}\boxed{\text{OUT}}$$

OPERATIONS

FIG I

This simple arrangement becomes untrue as soon as any iterative process is attempted, for example, suppose that we wish to calculate the expression:

$$y = ax^2 + bx + c \,/\, dx + e. \qquad \text{-----(1)}$$

then if only one value of x is in question FigI represents the coding process. In FigII we give, in greater detail, the sequence which would be followed:

$$\text{IN} \text{---dx} + \text{e--}ax^2 + bx + c\text{---}\frac{ax^2+bx+c}{dx+e}\text{--OUT}$$

FIG II

Next let us suppose that we wish to perform the division, not by using our divide order, but by the iterative evaluation of 1/dx + o , followed by multiplication. Furthermore assume that n iterations are necessary. The schematic code now becomes:

$$\text{IN } dx+o\text{---}ax^2+bx+c\text{---It.1--It.2---It.n---}\frac{ax^2+bx+c}{dx+o}\text{- OUT}$$

FIG III

where we have inserted a separate set of orders for each iteration; a much more elegant method is, however, to use our conditional transfer order, programme one iteration and then arrange for the control to return to the start of this cycle n times; this can be simply arranged by placing the number n at the start of each cycle and subtracting unity from it before each iteration. For the first n subtractions this gives a result $\geqslant 0$, so that the conditional transfer Cc causes the control to repeat the cycle; after the nth. iteration, however, the result becomes negative and the conditional transfer order is inoperative so that the machine proceeds with the main sequence. This is shown in Fig IV

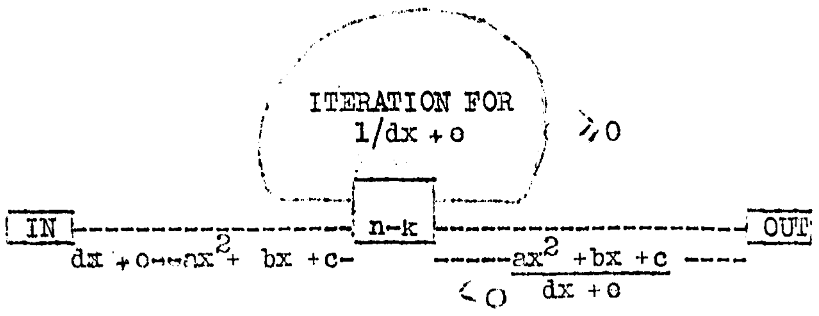


FIG IV

The portion of the diagram represented by $\overline{\text{n-k}}$ is called by Goldstine and von Neumann an "alternative box" and the looped portion of the diagram, a "simple induction loop". We next complicate the problem slightly by assuming that we wish to compute the value of (1) for $X = X_1$ to $X = X_1 + (N-1)X_0$, here we wish to add X_0 to X at each stage and repeat N times. The same iteration technique is available and the flow diagram now becomes:

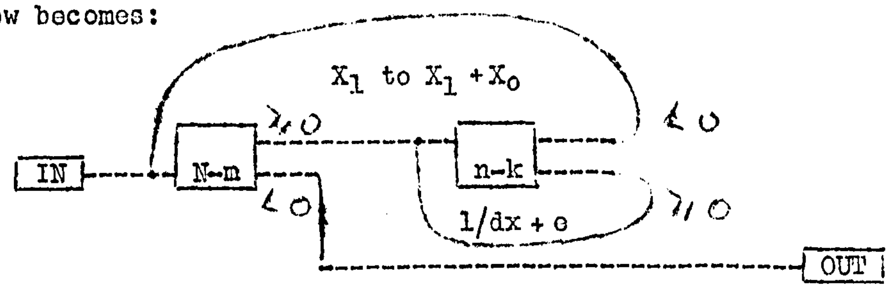


FIG V

Fig V is a simple example of a "multiple induction loop". More complicated patterns are, of course, encountered, but enough has been said to show how the method is useful in planning the programme. For minute details on flow diagramming reference can be made to the Goldstine-von Neumann report on "Planning and coding of problems for an electronic computing instrument."

When the flow diagram is complete the next stage is to write out the detailed code which will contain:

1) A full list of all control orders.
2) Detailed memory locations for orders.
3) Detailed memory locations for numbers and for any transient storage required during the computation.

We do not propose to lay down any inviolable rules of procedure in this matter. Our own technique will become apparent from the specific problems dealt with in the following sections, but we do not consider this permanant and are prepared to vary it in accord with the problem in hand and with increasing experience.

2.0 Coded problems.

2.01 A general example.

The first problem to be coded is that of calculating $(ax^2 + bx + c)$ for values of x varying from 0 to $n\Delta$ in steps of Δ. The fixed parameters a, b and c and the variable x will be stored in given memory locations. The value of x must be altered by Δ after the calculation of each term $(ax^2 + bx + c)$, and it will also be necessary to decide at each stage whether x has reached a value beyond the range of calculation. When this occurs the operation is, of course, complete. In the example chosen these two processes can be combined, with a consequent saving of orders in the code, if x is given the initial value $n\Delta$ and is decreased by Δ at each stage. The conditional transfer order Cc enables us to detect when x becomes negative and this determines the end of the operation.

It will not generally be possible to perform this trick, as it is only applicable when one of the bounding values of the variable is 0. The process for the more general case will become apparent in para.(2.04).

Suppose that $n = 64$, and that the calculated values are to be stored in the memory; it will be necessary to modify the order which directs this operation at each stage so that they are sent to 64 consequtive memory positions. This is done quite simply by adding 2^{-8} to the order. this advances the memory location indicated by the order by 1.

The constants 2^{-8} and Δ will therefore also be stored in fixed locations in the memory. The orders governing the calculation will be placed in the memory before parameters, constants and calculated values etc., and the latter can be given provisional locations only at first. Suppose that a, b, c, x, Δ, 2^{-8} are stored in locations M(k) to M(k+5), and that the table of results is at M(k+6) to M(k+69).

The coding is then as follows:-

Memory location.	Order.	Remarks.
(1)	M(k) to R	a
(2)	M(k+3) $\times$ R to cA	ax ($x = n\Delta$ at first stage.)
(3)	M(k+1) to A	ax + b
(4)	A to R	
(5)	M(k+3) . R to cA	$ax^2 + bx$
(6)	M(k+2) to A	$ax^2 + bx + c$
(7)	A to M(k+6)	
(8)	M(k+3) to cA	x
(9)	-M(k+4) to A	$x - \Delta$
(10)	Cc C to M(+2)	

< 0

(11) Operation complete.

≥ 0

(12)	A to M(k+3)	
(13)	M(8) to cA	order A to M(k+6)
(14)	M(k+5) to A	2^{-8}
(15)	A to M(8)	
(16)	C to M(+241)	

Since it is now apparent that 16 orders are needed, k takes the value 17, and the memory locations of a, b, etc. are given accordingly.

The allocation of memory locations to orders was simple in this example, since only one conditional transfer occured. In calculations where there are more than one the process will not be so easy, and it may be necessary to draw up a separate table of order numbers and memory locations. This process will be illustrated in suceeding examples.

2.02 Extraction of the square root.

As A.R.C. is not equipped with a square rooting unit this operation will be performed using the iteration formula for $\sqrt{b}$:

$$x_{n+1} = (x_n + b/x_n)/2 \qquad \text{-------(1)}$$

If necessary b must first be brought within the range of the machine by dividing by 2^p where p is the smallest integer such that $b.2^p < 1$. The value of the first approximation x_1 must now be decided. It might appear that the simplest value to use is 1, but a much better approximation is $(1+b)/2$, and as this value is easily formed it will be taken as a starting point.

Some re-arrangement of the formula is necessary as it is important that none of the quantities formed while calculating it should be > 1. It can be seen, that the term $b/x_n < 1$ since, if d is the error in the nth approximation, the error in the next approximation is $d^2/2\sqrt{b}$. It follows that $x_{n+1} > \sqrt{b} > b$, since $b < 1$, and the term b/x_n therefore lies in the range of the machine. The first term b/x_1 is no exception to this rule since $x_1 = (1+b)/2 > \sqrt{b}$. The term $(x_n + b/x_n)$ may, however, be > 1 but this difficulty can be overcome by calculating $(x_n - b/x_n)/2 + x_n$. This form is chosen rather than $x_n/2 + b/2x_n$, since the rounding off errors are less. Moreover, the value of $(x_n - b/x_n)/2$ can be used to decide whether the calculation has proceeded far enough since it is equal to $x_{n+1} - x_n$. As can be seen from (2), $x_n > x_{n+1}$, and the expression will therefore be negative as long as d lies within the range of the machine. When d becomes $< 2^{-20}$, however, this expression will be zero, and the Cc will indicate the completion of the iteration.

The only constants which have to be stored are b and 1/2, and these will be placed in memory locations M(k) and M(k+1) respectively.

The code is then as follows:

```
(1)    M(k) to cA                  b
(2)      R(1)                      b/2
(3)    M(k+ 1) to A                (b+1)/2 = x1
(4)    A to M(k+ 1)
(5)    M(k) to cA                   b
(6)    A ÷ M(k+ 1) to R             b/x1
(7)    R to cA
(8)    -M(k+ 1) to A                 x1
(9)     R I(1)                     (b/x1 - x1)/2
(10)      Cc     C to M( + 2)

          ≥0                              <0
(11)     Operation comp.      (12)   M(k+1) to A      (b/x1  + x1)/2 = x2
                              (13)   A to M(k+ 1)
                              (14)   C to M( + 248)
```

14 orders are therefore required, and b and 1/2 will be stored in M(15) and M(16), the final result appearing in M(16)

The number of iterations required depends entirely on the size of b, increasing as b decreases. In the worst possible case, when b = 0, 20 operations are needed; on the other hand when b is large (= .9 say), 3 iterations will give the result correct to 6 decimal (20 binary) places. It would be possible to scale b up, if small, to bring it into a more favourable range; usually, however, b will have been produced by a previous calculation of the machine, and the programming of this calculation should have been arranged to retain as many digits as possible. It seems, then, that the chances of our number being less than 2^{-10} are small, and the inclusion of extra orders in the code to modify the value in those cases is not justified.

2.03 Double length arithmetic.

It may be necessary, on occasions, to perform calculations to a greater accuracy than 20 binary places; for most purposes, however, 40 places will probably be sufficient, and we shall now give the coding of double length addition, subtraction and multiplication. It is, of course, possible to programme A.R.C. to calculate to any desired degree of accuracy.

One obvious use of double length arithmetic is in the calculation by the machine of tables for its own use, since these must be known correct to 20 binary places.

2.031 Addition and subtraction.

We can store these processes as one routine, the alternative sign being determined by substitution; it should be remarked here, that double length numbers will be stored with the sign attached only to the first half.

Suppose, now, that we wish to form the expression :

$$(a_1 + a_2) \pm (b_1 + b_2)$$

where a_1 and a_2 are respectively digits 1-20 and 21-40 of the double length number a and similarly for b. We have, therefore, to form the two factors $(a_1 \pm b_1)$ and $(a_2 \pm b_2)$. The only difference from ordinary arithmetic occurs when forming $(a_2 \pm b_2)$. Here a carry into the sign digit may occur indicating that 2^{-20} must be added into $(a_1 \pm b_1)$. In this case it is also necessary to restore the sign digit of $(a_2 \pm b_2)$ to zero.

The code is then as follows:

```
(1)   M(14) to cA          a2
(2)   ±M(16) to A          b2
(3)    Cc   C to M( +8)
     < 0                                      ≥ 0

(4)    -M(17) to A      -1          (11)    A to M(20)
(5)    A to M(20)                   (12)    C to M( +252)
(6)    ±M(18) to cA      2^-20
(7)    M(13) to A        a1
(8)    ±M(15) to A       ±b1
(9)    A to M(19)
(10)   Op. comp.
```

12 orders are required and storage for a_1, a_2, b_1, b_2, $1, 2^{-20}$ and the result of the calculation.

The allocation of memory positions is then as follows:

Memory Position.	Contents.
1 - 12	orders
13	a_1
14	a_2
15	b_1
16	b_2
17	1
18	2^{-20}
19	$a_1 \pm b_1 = c_1$
20	$a_2 \pm b_2 = c_2$

2.032 Multiplication.

$$(a_1 + a_2)(b_1 + b_2) = a_1b_1 + a_2b_1 + a_1b_2 = c_1 + c_2$$

We have to form the three partial products (a_1b_1), (a_1b_2), (a_2b_1). It is not necessary to form (a_2b_2) since this is $< 2^{-40}$, and therefore lies outside the range of calculation. (a_1b_1) will be entirely within the limits of accuracy required, and (a_2b_1) and (a_1b_2) will have their first 20 digits in the range 2^{-20} to 2^{-40}. When multiplying a_1 and b_1, therefore, the full 40 digits must be retained with no round off.

Next it should be noted, that we must correct the sign of (a_2b_1) and (a_1b_2) if necessary since if either a or b were negative one of these would appear with a negative sign. When all the multiplications have been performed, we must add a_1b_2, a_2b_1 and the second half of (a_1b_1), (ie $(a_1b_1)_2$) and decide whether two, one or no carries have occured beyond the binary point; the operation is then completed by adding the necessary carry (2^{-19}, 2^{-20} or 0) to $(a_1b_1)_1$.

The code is then as follows:

```
(1)     M(a₁) to R
(2)     R. M(b₂) to cA        a₁b₂
(3)       Cc   C to M( +2)
        < 0

(4)     -M(1) to A
(5)     A to M(40)
(6)     M(b₁) to R
(7)     R . M(a₂) to cA       a₂b₁
(8)       Cc   C to M( +2)

        < 0
(9)     -M(1) to A
(10)    A to M(41)
(11)    M(a₁) to R
(12)    R . M(b₁) to cA (no round off)
(13)    A to M(42)            (a₁b₁)₁
(14)    R to cA               (a₁b₁)₂
(15)    M(41) to A                      (a₁b₁)₂ + (a₂b₁)
                                                    cont.
```

```
(16)      Cc    C to M( + 14)

            < 0                                                              ≥ 0
(17)      M(40) to A      $(a_1b_1)_2 + (a_2b_1) + (a_1b_2)$      (30)    M(40) to A
(18)      Cc   C to M(+ 7)                                      (31)    Cc    C to M( +248)
            < 0                            ≥ 0                            < 0

(19)      -M(1) to A                 (25)   A to M(42)     (32)    C to M(+ 243)
(20)      A to M(42)                 (26)   M($(a_1b_1)_1$) to cA
(21)      M(41) to cA  $(a_1b_1)_1$    (27)   M($2^{-19}$) to A
(22)      M($2^{-20}$) to A          (28)   A to M(41)
(23)      A to M(41)                 (29)   Op. comp.
(24)      Cp. comp.
```

32 orders are required and storage space for a_1, a_2, b_1, b_2, $1, 2^{-19}, 2^{-20}$, (a_1b_2), (a_2b_1) and $(a_1b_1)_1$.

The memory allocations are then:

Memory position.	Contents.
1 - 32	orders
33	a_1
34	a_2
35	b_1
36	b_2
37	1
38	2^{-19}
39	2^{-20}
40	(a_1b_2)
41	$(a_1b_1)_1$ and $(a_1b_1)_1$ +carry = c_1
42	(a_2b_1) and $(a_1b_2 + a_2b_1 + (a_1b_1)_1 = c_1$

2.04 Tabulation of sin(x).

One useful function for which our machine is well suited is the calculation of tables. This will probably form a "spare time" occupation when the machine is not required for more urgent problems.

One obvious application is to programme A.R.C. to calculate the basic tables required for its own use. As mentioned before, this provides an example of the use of double length arithmetic as we shall have to work to 40 binary places in order to obtain tables correct to 20 places.

As an example we give the code for the calculation of sin(x) from 0 to $\pi/2$ at intervals of d. Sin(x) is chosen for tabulation rather than cos(x) since interpolation is thereby made slightly easier.

We shall use the formulae:

$$\sin(x+d) = \sin(x)\cos(d) + \cos(x)\sin(d)$$
$$\cos(x+d) = \cos(x)\cos(d) - \sin(x)\sin(d)$$

to give sin(x) and cos(x) from 0 to $\pi/4$, this being, of course, equivalent to sin(x) from 0 to $\pi/2$. The routines for double length multiplication and addition must be placed in the memory for reference, and we shall assume that these are stored at the end of the programme, occupying 42 and 20 memory places respectively.

The initial data which must be inserted is the values of sin(0), cos(0), sin(d) and cos(d) and the constants 2^{-1} and 2^{-20}. Suffices indicate the first and second halves of the number under consideration; thus $\sin(d)_2$ means digits 21-40 of sin(d). The memory positions ($\times a_1$) and ($+ a_1$)etc. refer to locations in the double length arithmetic routines.

After rounding off to 20 digits the values of sin and cos are punched on two separate tapes. Finally the cos values are added to the end of the sin table in reversed order, thus giving all values from o to $\pi/2$.

The code is as follows:

(1)	T_i to $T_i +_{62}$ to M(88) toM(149)	Transfer of double length routines to memory.
(2)	cos d_1 to cA	
(3)	A to M($\times a_1$)	
(4)	cos d_2 to cA	
(5)	A to M($\times a_2$)	Transfer of cos(d)and sin(x_1) to D.L. routines.
(6)	sin x_1 to cA	
(7)	A to M($\times b_1$)	
(8)	sin x_2 to cA	
(9)	A to M($\times b_2$)	
(10)	M (13) to cA	This directs the control to the appropriate
(11)	A_L TO M(120)	order (13) at the end of the D.L.M. routine.
(12)	C to M(+76)	Control to D.L.M. routine.

DOUBLE LENGTH MULTIPLICATION ROUTINE GIVING SIN(X)COS(d)

(13)	M($\times c_1$) to cA	
(14)	A to M($+a_1$)	Transfer of products to D.L.Addition routine for
(15)	M($\times c_2$) to cA	the formation of sin(x+d).
(16)	A to M($+a_2$)	
(17)	M(cosx_1) to cA	
(18)	A to M($\times b_1$)	sin(x)is replaced by cos(x)in D.L.M.R.
(19)	M(cos x_2) to cA	
(20)	A to M($\times b_2$)	
(21)	M(24) to cA	
(22)	A_L TO M(120)	
(23)	C to M(+65)	

DOUBLE LENGTH MULTIPLICATION ROUTINE GIVING COS(X)COS(d)

(24)	M($\times c_1$) to cA	
(25)	A to M(160)	Transfer of cos(x)cos(d)to temporary storage.
(26)	M($\times c_2$) to cA	
(27)	A to M(161)	
(28)	M(sin d_1) to cA	
(29)	A to M($\times a_1$)	cos(d) is replaced by sin(d) in D.L.M.R.
(30)	M(sin d_2) to cA	
(31)	A to M($\times a_2$)	
(32)	M(35) to cA	
(33)	A_L to M(120)	
(34)	C to M(+54)	

DOUBLE LENGTH MULTIPLICATION ROUTINE GIVING COS(X)SIN(d)

(35)	M($\times c_1$) to cA	
(36)	A to M($+b_1$)	Transfer of cos(x)sin(d) to D.L.A.R.
(37)	M($\times c_2$) to cA	
(38)	A to M($+b_2$)	

(39)	M(42) to cA	Direction of control at end of D.L.A.R.
(40)	A_L to M(141)	
(41)	C to M(+ 89)	

DOUBLE LENGTH ADDITION GIVING SIN(X)COS(d) + COS(X)SIN(d).

(42)	M(+ c_1) to cA	
(43)	A to M(sin x_1)	sin(x + d) is stored ready for use in the
(44)	M(+ c_2) to cA	calculation of the next term.
(45)	A to M(sin x_2)	
(46)	M(2^{-1}) to A	
(47)	Cc < 0 C to (+ 4)	Rounding off to 20 binary places of sin(x + d).
(48)	M(2^{-20}) to cA	
(49)	M(sin x_1) to A	
(50)	A to M(+ c_1)	
(51)	M(+ c_1) to ${}_sT_i$	Storage on tape.
(52)	M(sin x_1) to cA	
(53)	A to M(× b_1)	cos(x) replaced by sin(x) in D.L.M.R.
(54)	M(sin x_2) to cA	
(55)	A to M(× b_2)	
(56)	M(59) to cA	
(57)	A_L to M(130)	
(58)	C to M(+30)	

DOUBLE LENGTH MULTIPLICATION GIVING SIN(X)SIN(d)

(59)	-M(× c_1) to cA	-sin(x)sin$(d)_1$ to D.L.A.R.
(60)	A to M(+a_1)	
(61)	-M(× c_2) to cA	-sin(x)sin$(d)_2$ to D.L.A.R. with sign erased.
(62)	M(-1) to A	The sign must be removed as no sign is
(63)	A to M(+ a_2)	attached to the second half of D.L. numbers.
(64)	M(160) to cA	
(65)	A to M(+b_1)	cos(x)cos(d) placed in D.L.A.R.
(66)	M(161) to cA	
(67)	A to M(+b_2)	
(68)	M(71) to cA	
(69)	A_L to M(141)	
(70)	C to M(+60)	

DOUBLE LENGTH ADDITION ROUTINE GIVING COS(X)COS(d) - SIN(X)SIN(d)

(71)	M(+c_1) to cA	
(72)	A to M(cos x_1)	cos(x + d) stored for use in calculation
(73)	M(+c_2) to cA	of next term.
(74)	A to M(cosx_2)	
(75)	M(2^{-1}) to A	
(76)	Cc < 0 C to (+4)	Round off of cos(x + d) to 20 places.
(77)	M(2^{-20}) to cA	
(78)	M(cos x_1) to A	
(79)	A to M(+ c_1)	
(80)	M(+ c_1) to ${}_cT_j$	Storage on tape.

(81) M(N) to cA

(82) $-M(2^{-P})$ to A — Record of number of terms calculated.

(83) $Cc \geqslant 0$, C to (+174)

< 0

(84) ${}_cT_{j+N-1 \text{ to } j}$ to M(k to k+N-1) — Transfer of cos table to end of sin table.

(85) M(k to k+N-1) to ${}_sT_{j+N+1 \text{ to } j+2N}$

(86) Op. comp.

86 orders are therefore needed for programming. A further 59 are required for the double length routines and memory positions are needed for sin(x), cos(x),(initially used for sin(0) and cos(0)) sin(d), cos(d),cos(x)cos(d), 2^{-1}, 2^{-20},N, 2^{-P}.

The allocation of memory positions is then as follows:

Memory position.	Contents.
1 to 86	Code for tabulation of sin(x).
87 to 128	Code of D.L.M.
129 to 148	Code of D.L.A.
149, 150	sin x_1, sin x_2
151, 152	cos x_1, cos x_2
153, 154	sin d
155, 156	cos d
157	2^{-1}
158	2^{-20}
159, 160	cos(x)cos(d)
161	N
162	2^{-P}

2.05 Use of tables and interpolation.

One of the most important features of a computer is its ability to extract data from tables, and to interpolate if necessary. A.R.C. will be provided with a library of tapes containing such useful tables as sin, e^x, log(x) etc., and we shall now give the coding for a typical interpolation, taking as the function, sin(y).

Our table here will consist of sin(y) tabulated from 0 to $\pi/2$, or of sin(y) and cos(y) tabulated from 0 to $\pi/4$. It has been found that the coding is simplified if a sin table from 0 to $\pi/2$ is used since the determination of the sign is made easier. The code can, of course, immediately be adopted for cosines since $\cos(y) = \sin|\pi/2 - y|$. We shall assume that this table is stored in the high speed memory; interpolation directly from the tape is almost identical.

Some preliminary adjustment is necessary to bring y within the range 0 to $\pi/2$ and to determine the correct sign of the final result. First, it is obvious that the addition of 2π makes no difference to the value of the sin, while the addition of π changes the sign. It is probable that y will have been calculated by the machine, and if we arrange this calculation so that when y reaches the value π, a 1 is carried over into the sign digit, the sign of our result will be given correctly. Larger values of y will also be given correctly, since multiples of 2π

will cause a carry past the sign digit which will be lost. Thus, suppose $2\pi < y < 3\pi$, two digits will have carried past the binary point leaving 0 in the sign digit; this is correct, since $\sin(y) > 0$ in this range. On the other hand, if $\pi < y < 2\pi$, one digit will have carried over indicating that $\sin(y)$ is negative.

Having decided the sign of our result, it is next necessary to determine whether y (or $y - \pi$ if $y > \pi$) lies within the first or second quadrant. This is done by adding $\pi/2$ (= .1 in binary scale) to y and observing whether a carry into the sign digit occurs. We can thus decide whether we require $\sin(y)$ or $\sin(\pi - y)$.

The coding for the discrimination is as follows:

```
     (1)   M(y) to cA
     (2)    Cc    C to ( + 8)                             ≥ 0
< 0  (3)   M(1.0) to A      giving (y - π)     (10)   M(.1) to A   giving (y+π/2)
     (4)   A to M(y)                            (11)    Cc    C to ( + 8) cont.
     (5)   O(  ) to cA   Change order which                   with interpolation
     (6)   M(k) to A     determines sign of
     (7)   A to M(0)     sin y. This will       (12)   M(1.0) to cA
     (8)   M(y) to cA    occur in a subsequent  (13)   -M(y) to A     (π -y)
     (9)   C to ( + 1)   code normally.         (14)   C to ( + 5)  Continue
                                                                with interpolation
```

14 orders are needed for this discrimination, and we must also store the constants 1.0, 0.1, k and the variable y. 18 memory positions will be occupied in all.

We can now consider the actual process of interpolation. Suppose that $y = x + nd$, where x is the nearest smaller tabulated value to y, and d is the interval of tabulation.

Then, using central differences :-

$$\sin(x + nd) = \sin(x) + nD_1 + n^2D_2/2! + n(n^2 - 1)D_3/3! + \ldots \quad \text{----}(1)$$

where:

$$D_1 = \frac{\sin(x - d) - \sin(x + d)}{2}$$

$$D_2 = \sin(x - d) - 2\sin(x) + \sin(x + d) \quad \text{etc.}$$

We require to calculate sin(y) correct to 6 decimal places, and the number of terms of (1) which must be included depends, of course, on the value of d. Reference to the table will be simplified if the interval is $\pi/2^m$, and it can be shown, that, if $m = 8$, second differences will give $\sin(x + nd)$ to the required accuracy.

We therefore have to evaluate the function :

$$\sin(x) + \frac{n}{2}(\sin(x - d) - \sin(x + d)) + \frac{n^2}{2}(\sin(x - d) - 2\sin(x) + \sin(x + d)) \quad \text{---------}(2)$$

It is first necessary to locate the section of the table from which we have to interpolate. This is done by directing the control to the memory position given by the first 8 digits of y. The sin table will have been stored so that the memory location of any term is the same as the first 8 digits of the number whose sin is stored there (plus a possible constant K)

We can thereby pick out the location and value of sin(x); sin(x + d) and sin(x − d) are then obtained by instructing the control to advance and go back one position in the memory. The proportionate part, n, must also be calculated, but as this is equal to the remaining 12 digits of $y \times 2^8$, this is simple.

The coding is then as follows:

(19)	T_i to T_{i+128} to M(k) to M(k+ 128)	Transfer of sin table to memory.
(20)	M(y) to cA	
(21)	M(k) to A	
(22)	A_L to M(40)	
(23)	A_L to M(46)	Insertion of the location of sin(x), sin(x+ d) and sin(x − d) in the appropriate orders.
(24)	M(2^{-8}) to A	
(25)	A_L to M(30)	
(26)	−M(2^{-7}) to A	
(27)	A_L to M(38)	
(28)	S_L (8)	Giving n.
(29)	A to M(51)	
(30)	−M(sin(x + d)) to cA	
(31)	A to M(53)	sin(x +d) must be stored for use again.
(32)	M(sin(x − d)) to A	
(33)	S_R(1)	(sin(x − d) − sin(x+ d))/2.
(34)	A to M(54)	
(35)	A to R	
(36)	R . M(n) to cA	n(sin(x − d) − sin(x+ d))/2.
(37)	A to M(52)	
(38)	M(54) to cA	
(39)	sin(x +d) to A	
(40)	−sin(x) to A	(sin(x − d) − 2sin(x) + sin(x + d))/2.
(41)	A to R	
(42)	(n) . R to cA	
(43)	A to R	
(44)	(n) . R to cA	n^2(sin(x − d) − 2sin(x) + sin(x + d))/2.
(45)	M(52) to A	
(46)	sin(x) to A	giving |sin(y)|
(47)	Op. comp.	

30 orders are therefore needed, and, in addition, storage for 2^{-8}, 2^{-7}, n, sin(x + d), (sin(x − d) − sin(x + d))/2, $\frac{n}{2}$(sin(x−d) − sin(x +d)) k and sin(y), but some of these quantities can be stored in the same position.

The whole process of discrimination and interpolation therefore occupies 54 memory positions as follows:

Memory location.	Contents.
1 -- 14	Discrimination.
15	y
16	1.0
17	Constant for order 6 in discrim.
18	0.1
19 - 47	Interpolation.
48	2^{-8}
49	2^{-7}
50	k
51	n
52	$n(\sin(x - d) - \sin(x + d))/2$
53	$-\sin(x + d)$
54	$(\sin(x - d) - \sin(x + d))/2$
55 - 183	sin table.

2.06 Three dimensional Fourier summation.

Among the computations for which A.R.C. is likely to be used, are those arising in X-ray crystallography. In particular, there is the problem of effecting Fourier summations of the type:

$$\sum_{-H}^{H} \sum_{-K}^{K} \sum_{-L}^{L} F\cos(\ 2\pi(hx+ ky+ lz) + \alpha\ )$$

where x,y,z are fixed, h,k,l vary between the limits shown, and F and α are given for any particular (h,k,l).

The programme of such a summation is also of interest as it provides an example of the use of the interpolation routine given above.

A typical problem of this sort might involve 1,000 terms, and it is obviously impossible to store all the initial data in the high speed memory. We shall assume, then, that the values of (h,k,l), F and α are stored on tape; (x,y,z) can be placed in the high speed memory, since they are constant throughout the calculation. As will be seen later, the most convenient disposal of tape data is as follows:

$$h_1,k_1,l_1,\overset{\alpha_1}{F_1},h_2,k_2,l_2,\overset{\alpha_2}{F_2},\ldots\ldots\text{etc.}$$

It may be remarked here, that tape storage is particularly convenient in problems where a large amount of data is used in sequence, since, when a number has been read off, the control automatically moves to the next position on the tape. It follows that, by disposing data on the tape carefully, the machine can read off values in sequence, thus saving time and memory space in the code.

Storage space will be required in the high speed memory for (x,y,z), for the partial sum of the series, for the constant N giving the number of terms in the series, and the constant 2^{-p} which will be subtracted from N at each stage to keep track of the number of terms computed. We shall, in addition, require 129 positions for the table of sins and 54 for the interpolation routine.

A word here is needed on the subject of scale factors. It will be necessary to reduce (h,k,l) by suitable factors to bring them within the range of the machine. Usually, (h,k,l) will not exceed 32, and it will be sufficient to divide all values by 2^5 before forming hx, ky etc.. We must, of course, correct for this scale factor before taking the cosine; it is convenient, too, to effect the multiplication by 2π at this stage, since this consists merely of one left-shift.

The code is then as follows:

(1)	T_i to T_{i+128} to M(k) to M(k + 128)		Transfer of sin table.
(2)	T_j to T_{j+58} to M(169) to M(222)		Transfer of interpolation routine.
(3)	k to cA		Insertion of k in interpolation routine.
(4)	A to M(k)		
(5)	T_i to M(223)	$h \cdot 2^{-5}$	
(6)	M(x) to R		
(7)	R . M(223) to cA		
(8)	$S_L(6)$	$2\pi hx$	
(9)	A_L to M(223)		
(10)	T_{i+1} to M(224)	$k \cdot 2^{-5}$	
(11)	M(y) to R		
(12)	R · M(224) to cA		
(13)	$S_L(6)$	$2\pi ky$	
(14)	M(223) to A	$2\pi(hx + ky)$	
(15)	A to M(223)		
(16)	T_{i+2} to M(224)	$1 \cdot 2^{-5}$	
(17)	M(z) to R		
(18)	R · M(224) to cA		
(19)	$S_L(6)$	$2\pi lz$	
(20)	M(223) to A	$2\pi(hx+ky+lz)$	
(21)	T_{i+3} to M(224)		
(22)	M(224) to A	$2\pi(hx+ky+lz) + \alpha = \Theta$	
(23)	-M(0,1) to A	giving $\Theta - \pi/2$. This is necessary since we require cos Θ and sin Θ is tabulated.	
(24)	A to M(183)	Θ is inserted in interpolation routine.	
(25)	M(30) to cA	Location of order disposing of cos(Θ) after interpolation is inserted in interp. code	
(26)	A_L to M(173)		
(27)	M(30) to cA	Last order of interpolation code is modified to continue with the summation code.	
(28)	A_L to M(222)		
(29)	C to (+ 149)	Control to first order of interpolation.	

INTERPOLATION.

(30)	A to R	cos(Θ).
(31)	T_{i+4} to M(224)	F.

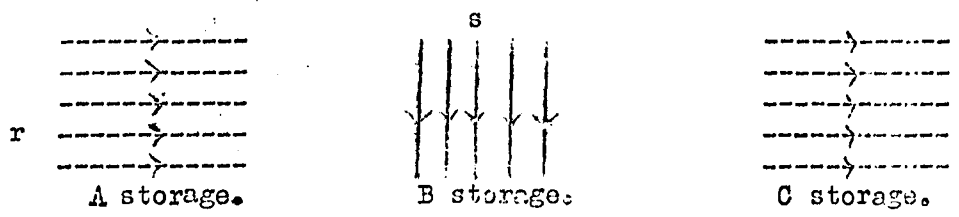
```
(32)   R . M(224) to cA          F cos(Θ).
(33)   M(225) to A              Previously calculated terms added in.
(34)   A to M(225)
(35)   M(N) to cA
(36)   -M(2^-P) to A            Record of no. of terms calculated.
(37)   A to M(N)
(38)     Cc    ,   C to ( + 219)

       < 0
(39)   Op. comp.
```

The allocation of memory positions is as follows:

<u>Memory position.</u>	<u>Contents.</u>
1 - 39	Code for summation.
40 - 168	Sin table.
169 - 222	Interpolation routine.
223	
224	Storage positions.
225	
226 - 228	(x,y,z)
229	N
230	2^{-P}
231	0.1
232	k = 41

2.07 <u>Matrix multiplication.</u>

An operation of common occurence in numerical mathematics is that of matrix multiplication. We shall consider two methods of performing this using A.R.C.. In the first, the whole of each matrix and of the product is stored in the high speed memory; naturally, this restricts each to a size less than 256/3 terms, and, since the programme has also to be included, this means in practice 8 . 8 element matrices. The second method stores one row at a time of the first matrix in the high speed memory, and takes the elements of the second, in order, from the tape. The advantage of this arrangement lies in the fact that matrices of size up to about 200 × 200 could be readily operated upon. The diagrams show how the elements of the two matrices are stored in linear sequence in the memory, and the simple, triply iterative, process is clear from the following codes.



$$C = AB$$
$$C_{rs} = \sum_i a_{ri} b_{is}$$

1) <u>Both matrices in high speed memory.</u>

(1) M(7) to cA
(2) + 2^{-P} to A — Advance of $a_{r,i}$ to $a_{r,i+1}$
(3) A to M(7)
(4) M(8) to cA — and $b_{i,s}$ to $b_{i+1,s}$.
(5) + 2^{-P} to A
(6) A to M(8)
(7) M($a_{r,i}$) to R
(8) M($b_{i,s}$) . R to cA — Formation of single term $a_{r,i}b_{i,s}$ and
(9) + M(ab) to A
(10) A to M(ab) — partial sums of these.
(11) M(n) to cA
(12) -2^{-P} to A
(13) A to M(n)
(14) Cc C to (+ 243)
< 0

(15) M(ab) to cA
(16) A to M(π i) — $\sum_i a_{r,i}b_{i,s} = c_{r,s}$
(17) M(16) to cA
(18) + 2^{-P} to A — Advance of disposal position from π_i to π_{i+1}
(19) A to M(16)
(20) M(N) to cA — Restoration of iteration index in (11).
(21) A_L to M(n)

(22) M(n') to cA — Formation of iteration index for completion of rth. row of c_{rs}.
(23) -2^{-P} to A
(24) A to M(n')
(25) Cc C to (+10)
< 0 $\geqslant 0$

(26) M($b_{0,0}$) to cA — Return of b_{rn} to $b_{0,0}$.
(27) A_L to M(8)
(28) M(N) to cA — Restoration of iteration index in (22).
(29) A_L to M(n')
(30) M(n'') to cA — Formation of iteration index for rows of a (ie. [illegible])
(31) -2^{-P} to A
(32) A to M(n'')
(33) Cc C to (+224) ($a_{r,i}$ advances to $a_{r+1,i}$ after (3).)
< 0
(34) Cp. comp.

(35) M(7) to cA — Restoration of $a_{r,i}$ to $a_{r,1}$.
(36) -M(N) to A
(37) A to M(7)
(38) C to (+ 222)

<u>Memory location.</u>	<u>Contents.</u>
1 - 38	code.
39	2^{-P}
40	M(ab)
41	M(n)
42	M(N)
43	M(n')

cont.

<u>Memory location.</u>	<u>Contents.</u>
44	$M(n'')$
$45 - 44 + n^2$	$M(a_{ri})$
$45 + n^2 - 44 + 2n^2$	$M(b_{is})$
$45 + 2n^2 - 44 + 3n^2$	$M(\Pi_i)$

2) <u>Using tape as secondary medium.</u>

Read one <u>row</u> of A into M and one <u>term</u> of B.

```
(1)   $a^T{}_{i,i+n}$ to M(j) to (j + n)      One row of A.

(2)   $b^T{}_{i'}$ to $M(b_{i,s})$              One element of B.

(3)   M(6) to cA
(4)   $2^{-p}$ to A
(5)   A to M(6)
(6)   $M(a_{ri})$ to R
(7)   $M(b_{is}) \cdot R$ to cA                 Formation of partial sum $\sum a_{ri}b_{is}$.
(8)   + M(ab) to A
(9)   A to M(ab)
(10)  M(n) to cA
(11)  $-2^{-p}$ to A
(12)  A to M(n)
(13)    Cc    C to (+ 245)

      $< 0$
(14)  M(ab) to $T_0$
(15)  M(N) to cA     Restoration of
(16)  $A_L$ to M(n)    index.
(17)  M(n') to cA    Iteration for
(18)  $-2^{-p}$ to A     row end.
(19)  A to M(n')
(20)    Cc    C to (+ 9)  $\geq 0$     (29)  M(6) to cA
                                  (30)  -M(N) to A
        $< 0$                         (31)  A to M(6)
(21)  $_bT$ to $_bT(b_{11})$                (32)  C to (+ 227)
(22)  M(N) to cA
(23)  $A_L$ to M(n')

(24)  M(n'') to cA
(25)  $-2^{-p}$ to A
(26)  A to M(n'')
(27)    Cc    C to (+ 230)
(28)  Op. comp.
```

<u>Memory locations.</u>	<u>Contents.</u>
1 - 32	Code.
33	$M(b_{is})$
34	2^{-p}
35	M(ab)
36	M(n)

Memory location.	Contents.
37	M(N)
38	M(n')
39	M(n'')
40 – (39 + n)	$M(a_{ri})$

2.08 Solution of a differential equation.

To show the power and flexibility of our machine we shall now code the solution of the ~~non~~-linear differential equation:

$$\frac{d^2y}{dx^2} + f(x).\frac{dy}{dx} + F(x) = 0 \qquad \text{-------(1)}$$

This is really a very simple equation from the point of view of the machine, and we propose to use only the crudest method for its solution; it should be emphasized that, in practice, more sophisticated difference techniques would be used.

We assume that the boundary conditions are :

$$x = 0, \quad y = q, \quad dy/dx = p. \qquad \text{-------(2)}$$

and that we may represent the differential cooffs. by the difference equns.:

$$(dy/dx)_n = y_{n+1} - y_{n-1} \;/2a$$
$$(d^2y/dx^2)_n = y_{n+1} - 2y_n + y_{n-1} \;/a^2 \qquad \text{-------(3)}$$

where a is the interval of differencing.

Using (3), (1) may be written :

$$y_{n+1} - 2y_n + y_{n-1} + \frac{a}{2}(y_{n+1} - y_{n-1})f(na) + a^2F(na) = 0 \qquad \text{-------(4)}$$

or:

$$y_{n+1} = (\; y_n + (\frac{a}{4}f(na) - \frac{1}{2})y_{n-1} - \frac{a^2}{2}F(na) \;)/\; (\frac{1}{2} + \frac{a}{4}\, f(na)\;)$$

with the boundary conditions:

$$y_0 = q, \; y_1 - y_{-1} = 2ap. \qquad \text{-------(5)}$$

Now, from (4), with n = 0:

$$y_1(1 + \frac{a}{2}f(0)\;) = 2y_0 + y_{-1}(\frac{a}{2}f(0) - 1) - a^2\,F(0),$$

or, using (5) :

$$y_1 = q + ap + \frac{1}{2}a^2(\,F(0) - pf(0)\;) \qquad \text{-------(6)}$$

From this point, a straightforward application of (4) with n = 1,2 etc. will give the values of y at all points nx.

We now give the simple iterative code required to obtain all values of y_n up to n = N. It is assumed that tables of values of f(na) and F(na) have already been calculated and are stored in the memory, and that y_0, y_1 and various constants such as $-a^2/2$, $+ a/4$ have been obtained in the preliminary mathematical discussion of the problem. Another assumption is that the calculations have been so arranged that $|y_n| < 1$; if this is not true suitable scale factors would have to be inserted.

(1)	M(17) to cA	
(2)	$+2^{-p}$ to A	
(3)	A to M(17)	
(4)	M(18) to cA	
(5)	$+2^{-p}$ to A	
(6)	A to M(18)	
(7)	M(21) to cA	
(8)	$+2^{-p}$ to A	
(9)	A to M(21)	
(10)	M(26) to cA	
(11)	$+2^{-p}$ to A	
(12)	A to M(26)	
(13)	M(29) to cA	
(14)	$+2^{-p}$ to A	
(15)	A to M(29)	
(16)	$-a^2/2$ to R	
(17)	F(na) . R to cA	
(18)	y_n to A	$y_n - a^2F(na)/2$
(19)	A to M(40)	
(20)	$+a/4$ to R	
(21)	f(na) . R to cA	
(22)	$+1/2$ to A	$1/2 + a\, f(na)/4$
(23)	A to M(41)	
(24)	-1 to A	$-1/2 + a\, f(na)/4$
(25)	A to R	
(26)	y_{n-1} . R to cA	
(27)	M(40) to A	$y_n + (a\, f(na)/4 - 1/2)y_{n-1} - a^2F(na)/2.$
(28)	A $\div$ M(41) to R	y_{n+1}
(29)	R to M( )	
(30)	M(42) to cA	$N \cdot 2^p$
(31)	-2^{-p} to A	
(32)	A to M(42)	
(33)	Cc C to (+ 224)	
	< 0	
(34)	Cp. comp.	

The memory positions for constants are as follows:

Memory position.	Contents.
1 - 34	code.
35	$-a^2/2$
36	$a/4$
37	$1/2$
38	-1
39	2^{-p}
40	Storage for (19).

Memory position.	Contents.
41	Storage for (22).
42	Storage for iteration index (30).
43 to 43 + N	Storage for f(0) to f(Na).
44 + N to 44 + 2N	Storage for F(0) to F(Na).
45 + 2N to 45 + 3N	Storage for y_0 to y_n.

2.09 Conversions to and from binary scale.

As mentioned earlier, A.R.C. will perform its own conversions from decimal to binary scale, and vice-versa; these are the last problems to be considered.

2.091 Decimal-binary conversion.

The normal arithmetic process of decimal-binary conversion of a decimal number is as follows: the number is successively multiplied by 2, 1 or zero is recorded in the corresponding binary number according as there is or is not a carry past the decimal point. Unfortunately this method is not very suitable for a binary machine, since decimal numbers cannot be inserted as such, but have to be represented by a binary code. Thus, the decimal number 9 would be inserted in the machine as 1001. The conversion process outlined above becomes very complicated if we have to operate on a decimal number represented by binaries, as carries between the digits have to be adjusted at each multiplication.

We shall, accordingly, adopt the less direct method to be described now. It is best illustrated by an example; thus, suppose we have to convert .341 to binary scale. This can be written:

$$10^{-2}((3.10 + 4).10 + 1)$$

We therefore isolate the first digit of the number and multiply by 10 (1010 in binary scale). The next digit is now added in, and the sum again multiplied by 10. This process is repeated until all the digits have been used. We now have to multiply by a scaling factor $2^p/10^x$ to produce the correct result; as numbers will be added in to the end of the accumulator, p will have the value 20. In practice we shall be dealing with 5 digit decimal numbers and the scale factor will be $2^{20}/10^5$.

The only question remaining is that of the sign of the number. We have mentioned previously, that data will be inserted on punched tele-type tape, using binary code for decimal numbers. ARC is restricted to 5 decimal numbers and the input will be in the form of 5 cyphers representing the decimal digits and a sixth giving the sign as described in para.(1.2). We require to operate with the modulus of the number in both conversions, and if our decimal number is negative it is therefore necessary to adjust the corresponding binary number obtained by complementing and affixing a negative sign.

Details of the code are as follows:

(1)	M(D) to cA	Decimal number.
(2)	$S_L(4)$	This shifts the first digit of the number into the register.
(3)	A to M(T_1)	
(4)	R to cA	

(5) M(B) to A — Result of previous digits added in.
(6) A to M(B)
(7) $S_L(2)$ — Multiplication by 10.
(8) M(B) to A
(9) $S_L(1)$
(10) A to M(B)
(11) M(T0) to cA
(12) -2^{-20} to A — Record of number of digits converted.
(13) A to M(T0)
(14) Cc C to M(+ 243)

< 0

(15) M(B) to R

(16) $2^{20}/10^5$. R to cA — Scale factor.
(17) A to M(B)
(19) Cc C to (+ 5)

< 0

(20) -M(B) to cA
(21) 1.0 to A — B complemented and given correct sign.
(22) 1.0 to A
(23) A to M(B)
(24) Op. comp.

<u>Memory positions.</u>	<u>Contents.</u>
1 - 24	Code.
25	T_1
26	B
27	D
28	2^{-17}
29	2^{-20}
30	$2^{20}/10^5$
31	1.0

2.092 Binary-decimal conversion.

Binary-decimal conversion is slightly easier than the reverse process as the straightforward arithmetic method can be used. The process is to multoply the binary number sucessively by 1010 (ie.10). The corresponding decimal digits are given in their coded binary form by the carry past the binary point. The modulus of the binary number must be used during conversion, and a negative sign attached to the result if appropriate.

The coding is as follows:

(1) |M(B)| to R
(2) R . 0.1010 to cA
(3) $M(T_1)$ to R — Portion of number already converted.
(4) $S_L(4)$ — Carry past binary point which would have occured on mult. by 1010 shifted to R.

(5) R to $M(T_1)$
(6) A to $M(T_2)$
(7) M(19) to cA

(8) -2^{-20} to A
(9) A to M(19)
(10) Cc C to (+247)

< 0

(11) M(B) to cA
(12) Cc C to (+8)

< 0

(13) $M(T_1)$ to cA Sign adjusted.
(14) 1.0 to A
(15) A to $M(T_1)$
(16) $M(T_1)$ to T
(17) On. comp.

Memory positions.	Contents.
1 - 17	Code
18	0.1010
19	2^{-17} (for order (7))
20	2^{-20}
21	1.0

Table I

No.	Code	Symbol	Description
1	0,0000	T_1 to M	Fill high speed memory from input tape,
2	0,0001	T to T_i	Start tape moving to position T_i and proceed with next order,
3	0,0010	${}_nT_i$ to ${}_{i+j}$ to M_1 to ${}_{1+j}$	Read material on nth tape between i and i j into M.
4	0,0011	M_1 to ${}_{1+j}$ to T_n	Punch material from memory location 1 to 1 j onto nth tape.
5	0,0100	C to M	Shift control to order located at M(x).
6	0,0101	Cc to M	If A $\geq$ 0 shift control as in 5.
7	0,0110	L(N)	Shift contents of A and R, N places to left. So that eg. A(0)--A(20);R(0)--R(20) to A(0),A(2)-A(20),0;R(1)--R(20), A(1).
8	0,0111	R(N)	Shift contents of A, N places to right so that eg. A(0),A(1)-A(20) to A(0),0,A(1)--A(19).
9	0,1000	M to cA	
10	0,1001	|M| to cA	
11	0,1010	-M to cA	
12	0,1011	-|M| to cA	
13	0,1100	M to A	
14	0,1101	|M| to A	
15	0,1110	-M to A	
16	0,1111	-|M| to A	
17	1,0000	M . R to cA	Clear A. Multiply M by R, place 1st 20 digits and sign in A after adding unity to 21st digit. Leave last 20 digits in R.
18	1,0001	M . R to cA(N.R.)	Perform mult. without round off.
19	1,0010	A ÷ M to cR	Divide A by M, place quotient in R with last digit unity. Leave remainder in A.
20	1,0011	M to cR	
21	1,0100	R to cA	

22	1,0101	R to M	
23	1,0110	A to M	
24	1,0111	A_L to M	Substitute digits 1-8 of A in order located at M(x).
25	1,1000	A to cR	
26	1,1001	E	Signal completion of operation.