Bomber
Weapons
Defense
Computer
Study

Final Engineering Report

INCREMENTAL COMPUTER LOGIC AND PROGRAMMING

Volume 4 October 1956

CONTRACT NUMBER
AF 33 (616) -2326

PROJECT NUMBER
UNIVAC 2052

Remington Rand Univac
DIVISION OF SPERRY RAND CORPORATION
1902 WEST MINNEHAHA AVE. ST. PAUL W4, MINNESOTA

FINAL ENGINEERING REPORT

VOLUME IV

INCREMENTAL COMPUTER LOGIC

AND PROGRAMMING

CONTRACT NUMBER
AF33(616)-2326

PX 56-4

PROJECT NUMBER
UNIVAC 2052

Remington Rand Univac
DIVISION OF SPERRY RAND CORPORATION
1902 WEST MINNEHAHA AVE. ST. PAUL W4, MINNESOTA

INTRODUCTION

This volume describes the incremental computer from the operational point of view. In brief, the computer accepts real-time analog inputs and continuously modifies pertinent functions of the variables so as to yield updated output functions. For example, the fire control problem has as inputs the vector position and vector velocity of an attacker and computes as outputs the azimuth and elevation lead angles. Only the change of the output function is computed, and this is added to the initial function value to give the updated value. The computer operates by performing one basic computation (whose formulation is the basic algorism discussed later) about 64 times every drum revolution. The quantities necessary for a computation are read off the drum serially and out of the random access memory during a minor cycle.

The detailed analysis of the mathematical operation of the computer and the points considered in programming and coding follow as outlined below.

A knowledge of Section I is not strictly necessary to the practical operation of the computer except perhaps for the section on errors. It is invaluable for a mathematical understanding of the computer action, however, and the special techniques discussed indicate interesting but less obvious possibilities of the computer.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

LIST OF TABLES

LIST OF FIGURES

1. MATHEMATICAL ANALYSIS OF OPERATIONS

a. INTRODUCTION. - The application of digital techniques to the methods of solving mathematical problems normally associated with analog computers leads to a hybrid computer that has some of the advantages and some of the limitations of both the analog computer and the general purpose digital computer. The ERA incremental computer, as presently designed, carries out numerical computations by acting on the small, discrete changes of the input variables to compute the small, discrete changes in the output variables. In the memory of the computer are stored the orders for a number of individual steps (called "minor cycles") of numerical computation. During one major cycle, all the variables in each minor cycle undergo changes of one increment, i.e., plus or minus one. At the end of each minor cycle, the output of that cycle is stored in the random access memory as a plus or minus one. This stored output can be used to modify any of the variables in any of the several steps including the step from which the output is obtained. In any minor cycle, those numbers whose changes are determined by other minor cycles (through the random access memory) or by an input quantity are called independent variables, and those that are changed by the step itself are called dependent variables.

An incremental computer is useful primarily because it is capable of doing fairly complicated operations within a small computer and it produces an up-to-date answer output once every major cycle, i.e., about once every 0.005 seconds.

The incremental computer can be made small because its arithmetic section only adds, subtracts, and complements. These can be combined to form the more complicated operations within a single step.

In principle, the incremental computer differs from a general purpose computer in that instead of performing a complete numerical operation at each step of each major cycle, it computes the change in that step's solution due to the

change in variables since the last major cycle. For example, instead of forming the sums $U_oQ_{n-1} + V_oT_{n-1}$, $U_oQ_n + V_oT_n$, $U_oQ_{n+1} + V_oT_{n+1}$, etc., directly in successive major cycles, the incremental computer forms

$$U_oQ_n + V_oT_n = U_oQ_{n-1} + V_oT_{n-1} + U_o(Q_n - Q_{n-1}) + V_o(T_n - T_{n-1})$$

$$\text{or } \Delta\left[U_oQ_n + V_oT_n\right] = U_o\Delta Q_n + V_o\Delta T_n,$$

$$\text{and } \Delta\left[U_oQ_{n+1} + V_oT_{n+1}\right] = U_o\Delta Q_{n+1} + V_o\Delta T_{n+1}, \text{ etc.}$$

By restricting the change of each variable to a +1 or -1 each major cycle, these operations are accomplished by addition or subtraction. Through the use of these simple operations, appropriately controlled, the incremental computer can form sums, differences, products, quotients, square roots, integrals, derivatives, integrals with an inverse integrand, logarithms, and exponentials. Any one of these arithmetic operations can be done in a single minor cycle.

It is basic to the use of an incremental computer that the error of one increment in any variable is small enough so that to neglect it would not cause serious errors in the answers. This implies that in five milliseconds the answers themselves change by a "negligible amount". This, in turn, implies that five milliseconds is a "negligible" period of time. In other words the incremental computer is useful for real time control of systems whose time constants are of the order of seconds or longer. In systems with relatively long time constants, the answers can be changed by large numbers of increments during a period of time short compared to the time constants. With longer time constants it is possible to represent the variables more accurately within the machine by making one increment a smaller part of the variable. The only practical limit on the accuracy of the numerical solutions of the mathematical problem programmed in the incremental computer is the length of time to effect the changes in the variables.

In direct analogy to analog computers, the incremental computer can be used for the solution of implicit equations. The pairs of operations multiply and divide, integrate and differentiate, squaring and extracting a square root, and generating the logarithm and the exponential, can be considered as explicit-implicit pairs in the incremental computer. Simultaneous and differential equations are solved by an implicit arrangement just as are similar problems in an analog computer. They are fundamentally different, however, because the incremental computer handles numbers digitally with the associated advantages numerically and electrically.

b. MACHINE ALGORISM. - The overall problem to be solved by the incremental computer must be divided into a sequence of arithmetic operations or steps. The steps are coupled by the increments stored in the random access memory; the output increments from one step are the input increments to other steps. In the presently designed equipment, the heart of the storage system is the magnetic drum. The commands and numbers required for each of the steps of the mathematical problem are stored sequentially on several tracks on the drum. Each step is acted on by the arithmetic section of the computer once each drum revolution, that is, one drum revolution corresponds to one major cycle.

Let us consider the operations performed by the arithmetic section of the computer during a single step. During each step four binary numbers are read from the magnetic drum by four reading heads in the arithmetic section of the computer. These numbers will be denoted as U_{i-1}, V_{i-1}, R_{i-1}, and S. Three of these numbers, U_{i-1}, V_{i-1}, and R_{i-1}, are processed in accordance with the commands from the control section of the computer and the new numbers are returned to replace the previously read numbers in their respective places. The number S is a constant scale factor which remains on the drum.

At each step the arithmetic section forms the incremtal sum:

$$R_i = R_{i-1} + U_i \Delta Q_i + V_{i-1} \Delta T_i + S \Delta P_i - S \Delta W_i \qquad (1)$$

To simplify discussion, the increment will be considered unity and all machine numbers will be considered integers. That is, the increments ΔT_i, ΔQ_i, ΔP_i, and ΔW_i, and the incremental changes in the variables U_i and V_i, namely ΔU_i and ΔV_i, can have the values +1 or -1. These incremental values are read from the random access memory from positions given by the commands interpreted by the control section. In many cases some of the increments are programmed to be zero for all major cycles of a particular step, i.e., they are not read from the random access memory.

By summing a single step over n major cycles, the values of the variables during the nth major cycle can be determined. That is,

$$\sum_{i=1}^{n} (R_i - R_{i-1}) = \sum_{i=1}^{n} U_i \Delta Q_i + \sum_{i=1}^{n} V_{i-1} \Delta T_i + S \sum_{i=1}^{n} \Delta P_i - S \sum_{i=1}^{n} \Delta W_i$$

$$\text{or} \quad R_n - R_o = \sum_{i=1}^{n} U_i \Delta Q_i + \sum_{i=1}^{n} V_{i-1} \Delta T_i + S(P_n - P_o) - S(W_n - W_o) \qquad (2)$$

In the computer $R_o = 0$ in every case. In all but special cases the incremental changes in the dependent variable are chosen to cause R_n to go toward zero. When R_n is sufficiently close to zero to be considered negligible, the step is "settled". When it has settled, the equation for the machine algorism for that is approximately:

$$0 = \sum_{i=1}^{n} U_i \Delta Q_i + \sum_{i=1}^{n} V_{i-1} \Delta T_i + S(P_n - P_o) - S(W_n - W_o) \qquad (3)$$

In each case the increments are considered small. If we consider them to be differentials, equation 3 suggests:

$$0 = \int_0^n U dQ + \int_0^n V dT + S\left[P_n - P_0\right] - S\left[W_n - W_0\right] \tag{4}$$

c. QUALITATIVE DESCRIPTION OF ALGORISMS. - In order to determine exactly the course of an arithmetic operation performed by a step of the incremental computer, equation 1 must be used with the mathematics of finite differences. It is useful to use equation 4 in a qualitative way to approximate the answers actually obtained in the incremental computer. In applications, the situation is reversed; the problem is to find an incremental program which will approximate the desired mathematics. It is hoped that the qualitative use of equation 4 will clarify the principles of the arithmetic algorisms.

Sum and differences: Set U and V constant and solve for W. Then from equation 4

$$0 = \int_0^n U_0 dQ + \int_0^n V_0 dT + S(P_n - P_0) - S(W_n - W_0)$$

$$W_n = \frac{U_0(Q_n - Q_0) + V_0(T_n - T_0)}{S} + P_n - P_0 + W_0$$

Let the initial conditions satisfy

$$W_0 = \frac{V_0 T_0 + U_0 Q_0}{S} + P_0$$

Then

$$W_n = \frac{U_0 Q_n + V_0 T_n}{S} + P_n \tag{5}$$

That is, through the use of the sum algorism up to three variables may be added in the form

$$x = x_1 + a_2 x_2 + a_3 x_3$$

where a_2 and a_3 must be rational numbers. a_2 and a_3 are the quotient of the positive or negative integers U_0 and V_0 divided by the integer S.

Product and quotient: Set $dT = dU$, $dQ = dV$, and solve for W. Then

$$0 = \int_o^n UdV + \int_o^n VdU + S(P_n - P_o) - S(W_n - W_o)$$

$$W_n = W_o + P_n - P_o + \frac{\int_o^n (UdV + VdU}{S}$$

$$= W_o + P_n - P_o + \frac{U_nV_n - U_oV_o}{S} = W_o - P_o + \frac{U_oV_o}{S} + P_n + \frac{V_nU_n}{s}$$

Let $W_o = \frac{U_oV_o}{S} + P_o$

Then $W_n = \frac{V_nU_n}{S} + P_n$, multiplication (6)

or $U_n = \frac{S(W_n - P_n)}{V_n}$, division (7)

Clearly, from equations 6 and 7, these algorisms should more properly be called "Product and Sum" and "Sum and Quotient" in general.

Square Root: The square root of a variable can be derived from the incremental algorism by letting $U = V = T = Q$. Then equation 4 becomes:

$$0 = \int_o^n VdV + \int_o^n VdV + S(P_n - P_o) - S(W_n - W_o)$$

$$0 = V^2 - V_o^2 - S(W_n - P_n - W_o + P_o)$$

$$V^2 = S(W - P) + V_o^2 - S(W_o - P_o)$$

Let $V_o^2 = S(W_o - P_o)$

$V^2 = S(W - P)$

and $V = \sqrt{S(W - P)}$ (8)

(1) INTEGRAL. - In equation 4 the availability of integration is obvious. However, in a step doing a single integration it is sometimes advantageous to set $U = V$, $X_n = X_o = 0$, and $\Delta W = \Delta T$. For more specific details refer to the quantitative description of integration.

(2) TIME DERIVATIVE. - Differentiation with respect to time can be treated as the inverse of integration. However, special care must be taken to make it stable. Differentiation is accomplished by letting the U variable be constant, the T variable be time, and dV = dQ and solve for V. Equation 4 becomes:

$$0 = \int_o^n U_o dV + \int_o^n V dt + S \quad P_n - P_o - W_n + W_o$$

Differentiating:

$$0 = U_o \frac{dV}{dt} + V + S\frac{dP}{dt} - S\frac{dW}{dt}$$

$$V = S\frac{d}{dt}(W - P) - U_o\frac{dV}{dt} \tag{9}$$

If U_o were zero, V would be proportional to the derivative of the function W - P. This algorism would properly be called "Addition and Differentiation". However, the last term with $U_o \neq 0$ is necessary to obtain a stable differentiation algorism. This stability term can be compensated for. The accuracy of the compensation is determined by the number of minor cycles used.

Integral with a Reciprocal Integrand.

Let U = V and solve for Q + T in equation 4:

$$0 = \int_o^n V dQ + \int_o^n V dT + S(P_n - P_o) - S(W_n - W_o)$$

Differentiating:

$$0 = Vd(Q + T) + Sd(P - W)$$

$$d(Q + T) = \frac{Sd(W - P)}{V}$$

Integrating:

$$Q + T = S\int \frac{d(W - P)}{V} \tag{10}$$

Usually Q = T and P = 0. Then equation 10 becomes

$$Q = \frac{S}{2}\int \frac{dW}{V} \tag{11}$$

(3) LOGARITHM. - Reciprocal integration can be used to generate the logarithm. If we let V = W, equation 11 becomes:

$$Q = \frac{S}{2}\int \frac{dW}{W}$$

$$\text{or} \quad Q = \frac{S}{2} \log_e W + C \tag{12}$$

(4) EXPONENTIAL. - The exponential can be obtained from the logarithm by solving for W. Then

$$\frac{2Q}{S} + C' = \log_e W$$

$$e^{\frac{2Q}{S} + C'} = e^{\log_e W}$$

$$W = e^{\frac{2Q}{S} + C'} \tag{13}$$

The foregoing has been a qualitative treatment of the use of the RRU incremental computer. The following discussion is a more formal presentation including the specific restrictive formulas for the step and the choice of the sign of the increment of the dependent variable to make the value of R_n go toward zero.

d. QUANTITATIVE DESCRIPTION OF INCREMENTAL ARITHMETIC OPERATIONS. - The algorisms previously qualitatively described are specifically described by the choices of the increments and initial values. In any step there are six increments to be specified (ΔU_i, ΔT_i, ΔV_i, ΔQ_i, ΔP_i, and ΔW_i), two initial values (U_o and V_o), and the scale factor (S). The initial value of R_i, that is, R_o, is always fixed at zero. The correct combination of the programmed initial values, the programmed selection of the six increments in each step, and the wired-in machine algorism represented by equation 1 cause the incremental computer to perform the arithmetic operations described above as well as some special operations described later. Those variables of a given step that have their increments determined by other than the step itself, e.g., by

other steps, are called independent variables. The increment of the dependent variable depends on the sign of the remainder, R_n, of the step in question (which may vary from major cycle to major cycle) and usually on the non-varying sign of other variables or constants as well. For convenience of notation the signum function will be used. It is defined by:

$$\text{Signum } x \equiv \text{sgn } x \equiv +1 \quad \text{for } x \geq 0$$
$$\equiv -1 \quad \text{for } x < 0$$

The programmer is free to choose the increments ΔU_i, ΔV_i, ΔP_i, ΔW_i, ΔQ_i, and ΔT_i from any position in the random access memory. Each increment stored in the random access memory is either sgn R_i, for one of the steps or sgn $(V_{input} - V_i)$ for one of the input quantities.

(1) SUM AND DIFFERENCE. - To perform the previously mentioned sum:

$$W_n = \frac{U_o Q_n + V_o T_n}{S} + P_n \tag{5}$$

$$\text{set } \Delta U_i = \Delta V_i = 0 \tag{14}$$

$$W_o = \frac{U_o Q_o + V_o T_o}{S} + P_o \tag{15}$$

$$\text{and } \Delta W_{i+1} = \text{sgn } R_i \text{ sgn } S \tag{16}$$

The sense of equation 16 is such that the increments of W tend to reduce the magnitude of R_i. Equation 1 becomes:

$$R_i - R_{i-1} = U_o \Delta Q_i + V_o \Delta T_i + S \Delta P_i - S \Delta W_i$$

By summing this equation for n major cycles one finds that:

$$\sum_{i=1}^{n} (R_i - R_{i-1}) = U_o \sum_{i=1}^{n} (Q_i - Q_{i-1}) + V_o \sum_{i=1}^{n} (T_i - T_{i-1}) + S \sum_{i=1}^{n} (P_i - P_{i-1})$$
$$- S \sum_{i=1}^{n} (W_i - W_{i-1})$$

$$R_n - R_o = U_o(Q_n - Q_o) + V_o(T_n - T_o) + S(P_n - P_o) - S(W_n - W_o) \tag{18}$$

Solving equation 18 for W_n:

$$W_n = \frac{U_oQ_n + V_oT_n}{S} + P_n - \frac{U_oQ_o + V_oT_o}{S} - P_o + W_o - \frac{R_n}{S} \tag{19}$$

Substituting equation 15 in equation 19 one has:

$$W_n = \frac{U_oQ_n + V_oT_n}{S} + P_n - \frac{R_n}{S} \tag{20}$$

Equation 20 is accurate as it stands, i.e., no approximations have been made. It is equivalent to equation 5 only if $\left|\frac{R_n}{S}\right|$ is negligible. Since equation 16 tends to reduce $|R_n|$, $|R_n|$ should eventually be sufficiently small, i.e., when the step has settled. When the step has settled, W is very nearly the desired sum.

(2) PRODUCT. - The product $W = \frac{UV}{S} + P$ can be formed approximately by setting:

$$\Delta Q_i = \Delta V_i \tag{21}$$

$$\Delta T_i = \Delta U_i \tag{22}$$

$$W_o = \frac{U_oV_o}{S} + P_o \tag{23}$$

$$\Delta W_i = \text{sgn } R_i \text{ sgn } S \tag{24}$$

Again the sense of equation 24 is to reduce $|R_n|$. Substituting into equation 1 we have:

$$\begin{aligned} R_i &= R_{i-1} + U_i\Delta V_i + V_{i-1}\Delta U_i + S\Delta P_i - S\Delta W_i \\ &= R_{i-1} + U_i\Delta V_i + V_{i-1}\Delta U_i + V_{i-1}U_{i-1} - U_{i-1}V_{i-1} + S\Delta P_i - S\Delta W_i \\ &= R_{i-1} + U_i\Delta V_i + V_{i-1}U_i - U_{i-1}V_{i-1} + S\Delta P_i - S\Delta W_i \\ &= R_{i-1} + U_iV_i - U_{i-1}V_{i-1} + S\Delta P_i - S\Delta W_i \\ &= R_{i-1} + \Delta (UV)_i + S\Delta P_i - S\Delta W_i \end{aligned}$$

Summing -

$$R_n - R_o = U_nV_n + SP_n - U_oV_o - SP_o + SW_o - SW_n$$

$$R_n = U_nV_n + SP_n - SW_n$$

$$W_n = \frac{U_nV_n}{S} + P_n - \frac{R_n}{S} \tag{25}$$

Equation 25 is exact; no approximations have been made in its derivation. If this step is settled, $\frac{R_n}{S}$ should be small enough that it can be neglected and W_n is approximately the desired product.

(3) QUOTIENT. - As mentioned above, division is effected implicitly from multiplication. To find the quotient $U = \frac{S(W - P)}{V}$, the product UV is compared to S(W - P) to determine whether U is too large or too small. This quotient can be formed by setting:

$$\Delta Q_i = \Delta V_i \tag{26}$$

$$\Delta T_i = \Delta U_i \tag{27}$$

$$U_o V_o = S(W_o - P_o) \tag{28}$$

$$\Delta U_{i+1} = -(\text{sgn } V_o)(\text{sgn } R_i) \tag{29}$$

Substituting into equation 1

$$R_i = R_{i-1} + U_i \Delta V_i + V_{i-1} \Delta U_i + S\Delta P_i - S\Delta W_i \tag{30}$$

which is identical to the corresponding equation for multiplication. As for multiplication, summing equation 30 yields:

$$R_n - R_o = U_n V_n + SP_n - U_o V_o - SP_o + SW_o - SW_n$$

$$R_n = U_n V_n + SP_n - SW_n$$

$$U_n = S \frac{W_n - P_n}{V_n} \quad \frac{R_n}{V_n} \tag{31}$$

No approximations have been made in the derivation of equation 31. Therefore, U_n is different from the desired quotient by the round off error $\frac{R_n}{V_n}$.

(4) SQUARE ROOT. - The square root algorism is effected by subtracting the product of the dependent variable times itself from the independent variable input. The sign of this difference indicates the next change in the dependent variable. The square root $U = \sqrt{S(W - P)}$ is formed by setting

$$U_o = V_o \tag{32}$$

$$\Delta T_i = \Delta Q_i = \Delta U_i = \Delta V_i \tag{33}$$

$$U_o^2 = SW_o - SP_o \tag{34}$$

$$\Delta U_{i+1} = -\text{sgn } R_i \text{ sgn } V_o \tag{35}$$

Substituting in equation 1,

$$R_i = R_{i+1} + U_i \Delta U_i + U_{i+1} \Delta U_i + S\Delta P_i - S\Delta W_i$$

$$= R_{i+1} + \Delta(U^2)_i + S\Delta P_i - S\Delta W_i$$

Summing

$$R_n = U_n^2 + SP_n - SW_n - U_o^2 - SP_o + SW_o$$

$$U_n^2 = SW_n - SP_n + R_n \tag{36}$$

When R_n settles to a negligible value, U_n^2 is approximately $S(W_n - P_n)$. Therefore, U_n must be close to the desired square root. Notice again the implicit approach to the extraction of the square root.

(5) INTEGRALS. - Integrals can be approximated in the incremental computer in a single step in several ways. One way is to form sums of the integrands. That is, let U, V, P, Q, and T be independent variables. Solve equation 1 for W. Then:

$$R_i = R_{i-1} + U_i\Delta Q_i + V_{i-1}\Delta T_i + S\Delta P_i - S\Delta W_i \tag{1}$$

$$\sum_{i=1}^{n} R_i - R_{i-1} = \sum_{i=1}^{n} \left\{U_i\Delta Q_i + V_{i-1}\Delta T_i\right\} + S \sum_{i=1}^{n} (\Delta P_i - \Delta W_i)$$

$$R_n = \sum_{i=1}^{n} U_i\Delta Q_i + \sum_{i=1}^{n} V_{i-1}\Delta T_i + SP_n - SW_n - SP_o + SW_o$$

Neglecting R_n

$$W_n = \frac{1}{S} \sum_{i=1}^{n} U_i\Delta Q_i + \frac{1}{S} \sum_{i=1}^{n} V_{i-1}\Delta T_i + P_n - P_o + W_o \tag{37}$$

Equation 37 is the Euler approximation to:

$$W_n = \frac{1}{S} \int_{Q_o}^{Q_n} UdQ + \frac{1}{S} \int_{T_o}^{T_n} VdT + P_n - P_o + W_o \tag{38}$$

A second way to integrate is the following:

Let $\Delta U_i = \Delta V_i$

$\Delta Q_i = \Delta T_i$

$U_o = V_o$

$\Delta W_{i+1} = \text{sgn } R_i \text{ sgn } S$

Then equation 1 becomes

$$R_i = R_{i-1} + U_i \Delta Q_i + U_{i-1} \Delta Q_i + S \Delta P_i - S \Delta W_i$$

$$\sum_{i=1}^{n} R_i - R_{i-1} = \sum_{i=1}^{n} (U_i + U_{i-1}) \Delta Q_i + S \sum_{i=1}^{n} (\Delta P_i - \Delta W_i)$$

$$R_n = \sum_{i=1}^{n} (U_i + U_{i-1}) \Delta Q_i + S(P_n - P_o - W_n + W_o) \tag{39}$$

Neglecting R_n

$$W_n = \frac{1}{S} \sum_{i=1}^{n} (U_i + U_{i-1}) \Delta Q_i + P_n - P_o + W_o \tag{40}$$

Equation 40 is the trapezoidal approximation to:

$$W_n = \frac{2}{S} \int_{Q_o}^{Q_n} U dQ + P_n - P_o + W_o \tag{41}$$

In those cases where the integrand U is an incrementally single valued function of the independent variable Q, the trapezoidal form is drift free, i.e., if the variable Q changes from its initial value and returns to its initial value, W_n will become W_o if U is a single valued function of Q. An example of a single valued function is a polynomial in Q. An example of a function that is not an incrementally single valued function is sin Q which is obtained by double integration rather than polynomial approximation.

(6) DERIVATIVES. - The incremental computer is capable of taking time derivatives in a single step if properly stabilized. The restrictions of the algorism are:

$$\Delta V_i = \Delta P_i = 0$$

$$\Delta Q_i = + 1$$

$$\Delta U_i = \Delta T_i = - \operatorname{sgn} R_i$$

Substituting in equation 1:

$$R_i = R_{i-1} + U_i + V_o \Delta U_i - S \Delta W_i$$

Summing

$$R_n - R_o = \sum_{i=1}^{n} U_i + V_o(U_n - U_o) - S(W_n - W_o)$$

Neglecting R_n

$$\sum_{i=1}^{n} U_i + V_o(U_n - U_o) = S(W_n - W_o)$$

The term $\sum_{i=1}^{n} U_i$ is approximately:

$$\sum_{i=1}^{n} U_i \approx \int_{\frac{1}{2}}^{t_n + \frac{1}{2}} U(\tau)\, d\tau$$

Then approximately:

$$\int_{\frac{1}{2}}^{t_n + \frac{1}{2}} U(\tau)\, d\tau + V_o(U_n - U_o) = S(W_n - W_o)$$

Differentiating:

$$U(t_n + \tfrac{1}{2}) + V_o \frac{dU(t_n)}{dt} = S \frac{dW(t_n)}{dt}$$

But $$U(t_n + \tfrac{1}{2}) = U(t_n) + \tfrac{1}{2} U'(t_n) + \tfrac{1}{8} U''(t_n) + \tfrac{1}{48} U'''(t_n) - - - -$$

So $$U(t_n) = S \frac{dW(t_n)}{dt} - (V_o P + \tfrac{1}{2}) \frac{dU(t_n)}{dt} - \tfrac{1}{8} U'' - \tfrac{1}{48} U''' - - -$$

$$S \frac{dW}{dt} = U(t_n) + (V_o + \tfrac{1}{2}) \frac{dU}{dt} + \frac{U''}{8} + \frac{U'''}{48} - - -$$

Now $$U(t_n + V_o + \tfrac{1}{2}) = U(t_n) + (V_o + \tfrac{1}{2}) U'(t_n) + \frac{(V_o + \frac{1}{2})^2}{2} U'' - - -$$

So $$S\frac{dW}{dt} = U(t_n + V_o + \tfrac{1}{2}) + \left[\frac{1}{8} - \frac{(V_o + \frac{1}{2})^2}{2}\right]\dot{U} - - -$$

$$S\frac{dW}{dt} = U(t_n + V_o + \tfrac{1}{2}) - \frac{V_o}{2}(V_o + 1)\,U$$

In other words U(t) is approximately equal to S times the derivative of W at time $(t + V_o + \frac{1}{2})$. The V_o which was necessary for stability introduces a time lag. To make this lag small, one must try to make V_o small. The size of V_o is determined by the nature of W. If V_o is too small and the second derivative of W is too high, U(t) will vary to both sides of the correct derivative value, eventually settling to the correct one. If V_o is too large, the delay will be unnecessarily long.

(7) INTEGRAL WITH A RECIPROCAL INTEGRAND. - A variation of the integral algorism is the form:

$$Q^{\cdot} = \frac{S}{2}\int\frac{dW}{U} \tag{42}$$

which is affected by letting:

$$\Delta T_i = \Delta Q_i \tag{43}$$

$$\Delta t_i = \Delta U_i \tag{44}$$

$$\Delta P_i = 0 \tag{45}$$

$$U_o = V_o \tag{46}$$

$$\Delta Q_{i+1} = -\operatorname{sgn} R_i \operatorname{sgn} U_o \tag{47}$$

Substituting in equation 1

$$R_i = R_{i-1} + U_i\Delta Q_i + U_{i-1}\Delta Q_i - S\Delta W_i$$

$$\Delta Q_i = \frac{R_i - R_{i-1} + S\Delta W_i}{U_i + U_{i-1}}$$

Summing over in cycles:

PX 56-4

$$Q_n - Q_o \simeq S \sum_{i=1}^{n} \frac{\Delta W_i}{U_i + U_{i-1}} + \sum_{i=1}^{n} \frac{R_i - R_{i-1}}{U_i + U_{i-1}} \quad (48)$$

If one neglects the contributions of the term containing the R's, equation 48 is approximately the integral equation 42.

Through the use of integration with a reciprocal integrand, the logarithm can be generated. If we let $U_i = W_i$ the form becomes:

$$Q = \frac{S}{2} \int \frac{dW}{W} \quad (49)$$

$$\text{or} \quad Q = \frac{S}{2} \log_e W \quad (50)$$

(8) EXPONENTIAL. - The exponential is found from the logarithm algorisms by letting Q be the independent variable and taking the increments of W from the step as the dependent variable. The algorism is identical to that of the logarithm except that instead of equation 47 the dependent variable is determined by:

$$\Delta W_{i+1} = + (\text{sgn } R)(\text{sgn } S)$$

e. SPECIAL ALGORISMS. - The following algorisms are special in the sense that they are not the usual arithmetic operations. They have been developed for some special purpose during the evolution or study of the fire control application. They are included here for completeness, as examples of the variation of the incremental computer algorism, and for possible application in servicing or other programs. This section can be skipped by the reader without losing the continuity of this presentation.

(1) Memory. Some time during each major cycle each position used in the random access memory has its old increment value removed and the updated increment entered. On some occasions it is desirable to have the previous value available. It is possible to do this by using a step in the following way:

Set $\Delta U_i = \Delta V_i = 0$

$U_i = V_i = 0$

$S = 1$

$\Delta W_{i+1} = \text{sgn } R_i$

That this is truly memory is proved as follows: With these restrictions equation 1 becomes:

$$R_i = R_{i-1} + \Delta P_i - \Delta W_i$$

If for $i = n$, $R_n = \Delta P_n - 1$, then

$$R_{n+1} = R_n + \Delta P_{n+1} - \Delta W_n$$
$$= \Delta P_n - 1 + \Delta P_{n+1} - \text{sgn } (P_n - 1)$$
$$= \Delta P_n - 1 + \Delta P_{n+1} - \Delta P_n$$
$$= \Delta P_{n+1} - 1$$

since $\text{sgn } (1 - 1) = \text{sgn } (0) = + 1$

and $\text{sgn } (-1 - 1) = \text{sgn } (-2) = - 1$

but initially

$$R_0 = 0, \text{ and sgn } R_0 = + 1 = \Delta W$$

$$R_1 = 0 + \Delta P_1 - 1 = \Delta P_1 - 1$$

Therefore, equation 50 is proven by mathematical induction. In this case sgn $(\Delta P_{n+1} - 1) = \Delta P_{n+1} = \Delta W_{n+2}$

or $\Delta W_n = \Delta P_{n-1}$

In other words, on the nth major cycle, if ΔP_n is desired, ΔP_n is addressed directly; if ΔP_{n-1} is desired, ΔW_n is addressed.

(1) SIGN. - Set

$U_o = V_o$, $\Delta U_i = \Delta V_i$, $\Delta T_i = - \Delta Q_i$

$\Delta P_i = 0$, $\Delta W_i = \text{sgn } R_{i-1}$, $S = 1$

Then $R_i = R_{i-1} + U_i \Delta Q_i - U_{i-1} \Delta Q_i - \Delta W_i$

$= R_{i-1} + \Delta U_i \Delta Q_i - \Delta W_i$

If for $i = n$

$$R_n = \Delta U_n \Delta Q_n - 1, \tag{52}$$

then $$\Delta W_{n+1} = \text{sgn}\,(\Delta U_n \Delta Q_n - 1) = \Delta U_n \Delta Q_n \tag{53}$$

and $$R_{n+1} = \Delta U_n \Delta Q_n - 1 + \Delta U_{n+1} \Delta Q_{n+1} - \Delta U_n \Delta Q_n$$

$$R_{n+1} = \Delta U_{n+1} \Delta Q_{n+1} - 1$$

and since $R_o = 0$, $R_1 = 0 + \Delta U_1 \Delta Q_1 - \Delta W_1 = \Delta U_1 \Delta Q_1 - 1$, equation 52 is proved by induction. Therefore, equation 53, giving the desired product, is proved.

(2) ABSOLUTE VALUE. - When a variable changes its magnitude by one, its absolute value changes by one. If the variable is positive, both changes are of the same sign; if negative, opposite sign. The change in the absolute value of a variable upon a change in the variable by one is given by:

$$\Delta_i |X| = \text{sgn}\, X_i \ \text{sgn}\ \Delta X_i \tag{54}$$

or $$\Delta_i |X_i| = \text{sgn}\, X_i \Delta X_i \tag{55}$$

One can form the product $X_i \Delta X_i$ by letting $U_i = X_i$ and $\Delta Q_i = \Delta X_i$. In order to sense the sign of this product, it should be put in a remainder alone. In other words a step which adds to the remainder the product $X_i \Delta X_i$ and subtracts the previous product $X_{i-1} \Delta X_{i-1}$ has as its remainder $X_i \Delta X_i$. The signum function of this remainder is the increment of the absolute value.

We have seen above that an increment can be stored for an extra major cycle through the use of memory. One might expect a memory step would be required to have available the ΔX_{i-1} required to form $X_{i-1} \Delta X_{i-1}$. In one special case this memory step can be avoided. This case is the following:

If a variable, X, is calculated in step j, the sign of its remainder is stored in the random access memory on the fifth digit period after the start of step j + 1. At any time prior to the fifth digit period the old value ΔX_{i-1} is available. After the fifth digit period the new value ΔX_i is available.

During the step j + 1 the increments for the calculation of step j + 2 are being selected. Step j + 2 can be used to calculate the absolute value of X_i without a memory step by letting:

$$\Delta U_i = \Delta V_i = \Delta Q_i = \Delta X_i \tag{56}$$

$$\Delta T_i = -\Delta X_{i-1} \tag{57}$$

$$U_o = V_o = +1 \tag{58}$$

$$S = 0 \tag{59}$$

The accumulation of the sign of the remainder of step j + 2 is the absolute value of X. This step does not require servoing.

To prove that sgn R_i is $\Delta_i|X|$, consider the following:

If equations 56, 57, and 58 are substituted in equation 1, it becomes:

$$R_i = R_{i-1} + X_i \Delta X_i - X_{i-1} \Delta X_{i-1}$$

Summing

$$R_n - R_o = X_n \Delta X_n - X_o \Delta X_o$$

$$R_n = X_n \Delta X_n - 1$$

If $|X_n| \geq 2$, sgn R_n = sgn $X_n \Delta X_n$

and sgn $X_n \Delta X_n = \Delta_n|X|$ since if X changes by one with the same sign as X itself, its absolute value increases by one. If ΔX and X are of opposite sign, the absolute value decreases.

If $X_n = +1$, sgn $R_n = \Delta X_n$, since sgn 0 = + 1. If $X_n = +1$, ΔX_n is $\Delta_n|X|$.

If $X_n = 0$, $|X_n| = 0$, and $\Delta_n|X| = -1$ since $|-1| = |1| = +1$. In this case sgn R_n = sgn - 1 = - 1 = $\Delta_n|X|$.

Lastly, if $X_n = -1$ and $\Delta_n X = +1$, sgn R_n = sgn (-1-1) = - 1 = $\Delta_n|X|$. If $X_n = -1$ and $\Delta X_n = -1$, sgn R_n = sgn (+ 1 - 1) = sgn 0 = + 1 = $\Delta_n|X|$.

Therefore, for every value of X_n, sgn $R_n = \Delta_n|X|$. The accumulation of these increments is the absolute value of X.

NOTE: The restriction that the calculation of the absolute value should occur two minor cycles subsequent to the calculation of the variable does not detract from its usefulness since this timing is usually the most desired.

(3) INDEPENDENT VARIABLE (TIME) CONTROL. - In some servicing problems it is desirable to have an independent variable change from one value to another and remain within one increment of that value indefinitely. That is, the change in the variable will consist of N + 1's followed by an indefinite sequence of alternately - 1 and + 1. In order to perform this function three core memory positions are used.

One memory position is used as a constant + 1. Any position not otherwise used can so serve. A second memory position should be a "step function", i.e., + 1 for the first major cycle and - 1 indefinitely thereafter. This latter function can be obtained by letting

$$U_o = V_o = 1$$

$$\Delta U_i = \Delta V_i = \Delta P_i = 0 = S$$

$$\Delta Q_i = - 1$$

$$\Delta T_{i+1} = - \operatorname{sgn} R_i$$

Then $R_1 = - 1 - 1 = - 2$

$R_2 = - 2 - 1 + 1 = - 2$

$R_n = R_{n-1} - 1 + 1 = R_{n-1} = - 2$

That is, after the first major cycle, $\operatorname{sgn} R_n = - 1$ indefinitely.

Given these two functions, the independent variable can be controlled in a single step by letting:

$$U_o = V_o = N$$

$$\Delta U_i = \Delta V_i = \Delta P_i = 0$$

$$\Delta Q_i = + 1$$

$$\Delta W_i = \operatorname{sgn} R_{i-1}$$

$S = 2$

and ΔT_i = + sgn remainder of the step function.

Then $R_1 = N(+1) + N(+1) - 2 = +2N - 2$

$R_2 = +2N - 2 + N(+1) + N(-1) - 2 = +2N - 4$

and if $n \leq N + 1$

$R_n = R_{n-2} + N - N - 2 = R_{n-2} - 2 = +2N - 2n$

$R_N = 0$

$R_{N+1} = -2$

$R_{N+2} = -2 + N - N + 2 = 0$

$R_{N+2i} = -2 + N - N + 2 = 0$

$R_{N+2i+1} = 0 + N - N - 2 = -2$

Therefore, the sign of this remainder gives the desired function.

(4) GATING. - In some service problems it is desirable to gate certain variables. By "gating" is meant either allowing a sequence of increments to be duplicated exactly in a certain memory position until a certain time after which that position will have alternately plus ones and minus ones unconditionally, or causing a memory position to have alternate plus and minus ones until a certain time after which a given sequence of increments is duplicated. These gates will be referred to as "initially open" or "initially closed" respectively.

Either of these gates requires a gate timing pulse. Incrementally the gate timing step memory position has alternately plus one and minus one until the switching time. At the switching time there will be one extra plus one and the alternate sequence will continue thereafter. To effect such a function the algorism is:

$U_o = 0$

$V_o = 3$

$S = 2M$

$$\Delta U_i = -\Delta V_i = \Delta W_i = \text{sgn } R_{i-1}$$

$$\Delta P_i = 0$$

$$\Delta T_i = -\Delta Q_i$$

We wish to have this timing sequence switch when $T_N = M$. Let T be a variable such that $T_0 = 0$ and $T > 0$. To demonstrate that this gate timing step produces the desired sequence, substitute in equation 1. Then

$$R_1 = 0 + 1\ (-\Delta T_1) + 3\Delta T_1 - 2M = 2\Delta T_1 - 2M < 0$$

$$R_2 = 2\Delta T_1 - 2M - 0\Delta T_2 + 2\Delta T_2 + 2M = 2\ (\Delta T_1 + \Delta T_2) = 2\ (T_2 - T_0) \geq 0$$

As long as the sequence is alternate U_i will be + 1 for odd major cycles and 0 for even major cycles. Likewise, V_{i-1}, will be + 3 for odd and + 2 for even major cycles. If we assume that $R_{2i} \geq 0$ where i is an integer, then

$$R_{2i+1} = R_{2i} - \Delta T_{2i+1} + 3\Delta T_{2i+1} - 2N = R_{2i} - 2M + 2\Delta T_{2i+1}$$

and if $R_{2i} - 2M + 2\Delta T_{2i+1} < 0$

$$R_{2i+2} = R_{2i} - 2M + 2\Delta T_{2i+1} - 0 + 2\Delta T_{2i+2} = R_{2i} + 2(T_{2i+2} - T_{2i})$$

If we assume the sequence of alternate increments, the general values of the remainders are:

$$R_{2n} = 2(T_{2n} - T_0) = 2T_{2n}$$

$$R_{2n+1} = 2T_{2n+1} - 2M$$

Conversely, the sequence of sgn R_i will be alternate as long as sgn R_{2n+1} is - 1 where $i \leq 2n + 1$. The smallest value of i for which this condition does not hold is found as follows:

$$2T_{2n+1} - 2M \geq 0$$

$$T_{2n+1} \geq M$$

In other words, the alternate sequence will continue until the first T on an odd major cycle is greater or equal to M, the predetermined switching point. Assume that switching occurs on the Nth major cycle, i.e., $T_N \geq M$

Then $R_N = 2T_{N-1} + 2\Delta T_N - 2M = 0$ or 1

But sgn 0 = sgn 1 = + 1. The sequence sgn R_i is no longer alternate but contains a + 1 for the Nth major cycle as well as for the N-1th. Then $U_{N+1} = 2$. The next remainder is:

$$R_{N+1} = R_N - 2\Delta T_{N+1} + 2\Delta T_{N+1} - 2M = -2M \text{ or } 1 - 2M$$

$V_{N+1} = 1$. Since $1 - 2M < 0$, $U_{N+2} = 1$. Using these values

$$R_{N+2} = R_N - 2M - \Delta T_{N+2} + \Delta T_{N+2} + 2M = R_N = 0 \text{ or } 1$$

After the N + 2th major cycle $U_{N+2} = 1$, $V_{N+2} = 2$, and $R_{N+2} = 0$. These are precisely the values obtained after the Nth major cycle. In other words, the step has become cyclic with a period of two major cycles. The remainder of this step is alternately 0 and - 2M (or 1 and 1 - 2M). In either case sgn R_i will be an infinitely long alternating sequence. Therefore, this step yields the desired gate timing pulse. This step is not reversible; the sequence will remain alternating regardless of subsequent values of T.

A single gate timing step can supply timing to open and/or close any number of gates at some particular time. Both the initially open and the initially closed gates will be considered here.

The initially opened gate is formed by the following algorism:

$U_0 = 1$

$V_0 = 2$

$S = 2$

$\Delta Q_i = -\Delta T_i$

$\Delta U_i = -\Delta V_i = \text{sgn } R_i$ in the gate timing step.

$\Delta P_i = 0$

$\Delta W_i = \text{sgn } R_{i-1}$

Prior to switching the U_i will be 1 on even major cycles and 0 on odd. V_{i-1} will be 3 on even major cycles and 2 on odd. In every major cycle prior to switching $V_{i-1} - U_i = 2$. Substituting in equation 1 we have:

$$R_i = R_{i-1} + U_i(-\Delta T_i) + V_{i-1}(\Delta T_i) - 2\Delta W_i$$

$$= R_{i-1} + 2\Delta T_i - 2\Delta W_i$$

$$R_1 = 0 + 2\Delta T_1 - 2$$

But $\quad \text{sgn } R_1 = \Delta T_1$

So $\quad R_2 = 2\Delta T_1 - 2 + 2\Delta T_2 - 2\Delta T_1 = 2\Delta T_2 - 2$

In general

$$R_n = 2\Delta T_n - 2$$

and $\quad \text{sgn } R_n = \Delta T_n$

prior to the gate switching. If the gate switching pulse occurs on the Nth major cycle where N is odd, the values of U and V will have the following values in the neighborhood of the Nth major cycle.

Major Cycle ≡ i	U_i	V_i	V_{i-1}	$V_{i-1} - U_i$
N - 3	1	2	3	2
N - 2	0	3	2	2
N - 1	1	2	3	2
N	2	1	2	0
N + 1	1	2	1	0
N + 2	2	1	2	0

After switching:

$$R_N = 2\Delta T_{N-1} - 2 - 2\Delta T_N + 2\Delta T_N - 2\Delta T_{N-1} = -2$$

$$R_{N+1} = -2 + 2 = 0$$

On all subsequent major cycles ΔT_i has no effect because $V_{i-1} - U_i = 0$ and the sgn R_i will alternate. Therefore, if $k \geq N$, $\sum_{i=1}^{k} \text{sgn } R_i = T_{N-1} - T_o$ k even

$$= T_{n-1} - T_o - 1 \text{ k odd.}$$

The initially closed gate is formed by letting the algorism be:

$U_0 = 0$

$V_0 = 1$

$S = 2$

$\Delta Q_i = -\Delta T_i$

$\Delta V_i = -\Delta U_i = \text{sgn } R_i$ in the gate timing step

$\Delta P = 0$

$\Delta W_i = \text{sgn } R_{i-1}$

Prior to switching both the U_i and the V_{i-1} will be 1 on even major cycles and 0 on the odd. The difference $U_i - V_{i-1} = 0$. Substituting in equation 1:

$$R_i = R_{i-1} + U_i \Delta Q_i - V_{i-1} \Delta Q_i - 2\Delta W_i$$

$$= R_{i-1} + 0\Delta Q_i - 2\Delta W_i = R_{i-1} - 2\Delta W_i$$

$R_1 = 0 - 2 = -2$

$R_2 = -2 + 2 = 0$

$R_{2n} = 0$

$R_{2n+1} = -2$

If the gate switching pulse occurs on the Nth major cycle where N is odd, the values of U and V will be the following in the neighborhood of the Nth major cycle:

Major cycle $\equiv i$	U_i	V_i	V_{i-1}	$U_i - V_{i-1}$
N - 3	1	0	1	0
N - 2	0	1	0	0
N - 1	1	0	1	0
N	2	-1	0	2
N + 1	1	0	-1	2
N + 2	2	-1	0	2

$R_N = 0 + 2\Delta Q_N - 2$

$\text{Sgn } R_N = \Delta Q_N$

$$R_{N+1} = 2\Delta Q_N - 2 + 2\Delta Q_{N+1} - 2\Delta Q_N = 2\Delta Q_{N+1} - 2$$

and $\operatorname{sgn} R_{N+1} = \Delta Q_{N+1}$

In general

$$\operatorname{sgn} R_{N+j} = Q_{N+j}$$

The function

$$\sum_{i=1}^{k} \operatorname{sgn} R_i = Q_k - Q_N \text{ where } k \geq N$$

which is the desired gated function.

(5) ERRORS IN INCREMENTAL ALGORISMS. - In performing the incremental algorism the computer can come up with only one possible answer. Since all the operations are digital, there is not the slightest ambiguity about the answer. This answer will be the same whether the incremental algorism is performed by a small, specially built computer or a large general purpose computer. However, the function generated by the incremental computer may not coincide with the function for which its program was designed. The difference between the desired answer and the answer the incremental computer will yield with a given program is called the program error.

In any digital computer there exists round off error when a number is handled whose significance exceeds the ability of the machine to represent the number as an integer. For example, a ten binary digit number can represent any integer from one to 1024. If ten binary digits are used to represent numbers from0.01 to 10.24, the number 7.8382 would have to be rounded off to 7.84 with a round off error of 0.0018 or 0.01.

In an incremental computer the round off error is usually no more than plus or minus one increment on input quantities and slightly more on output quantities, assuming that the computer can keep up with their changes. In order to keep up with the changes in the variables, the increments must be at least as

large as the maximum changes of the variables that can occur in one major cycle. For example, the error of a variable input or output due to round off is about equal to the maximum change the variable can make in 0.005 seconds in the case of real time application. In many applications this amount of error is small.

Round off error in an incremental computer corresponds to the error made in an analog computer due to the difference between the analog representation of a number and the number itself. To decrease the round off error by a factor of two in an analog machine may be a very difficult engineering operation. To decrease the round off error by a factor of two in an incremental computer not used for real time control, one more binary digit is required for the numbers and twice as much time is required for the computation.

The round-off error for the outputs from addition, subtraction, multiplication, and integration steps is equal to the term $\frac{R}{S}$. This term is usually less than one, but it can be as high as three even when the numerical process has "settled". In order to keep the fractional error small in addition, subtraction, multiplication and integration, the answer should be kept large as compared to the possible error. That is, the magnitude of the quantities $\frac{U_0Q + V_0T}{S} + P$ for addition and subtraction, $\frac{UV}{S} + P$ for multiplication, and $\frac{2}{S}\int UdT$ for integration must be kept as large as is consistent with the requirement that the change in the quantity shall not exceed one in one major cycle. The only quantity which may be adjusted to accomplish this is the scale factor. Therefore, to minimize round-off error in addition, subtraction, multiplication and integration, S should be just large enough that the arithmetic process can keep up with the changes that can occur. In division and differentiation the quantities to be kept large are: $\frac{S(W - P)}{V}$ and $S\frac{dW}{dt}$. In these cases S should be made as large as the requirement that the step must keep up will allow. Round-off is the only error that occurs in addition, subtraction, multiplication and

division. The round-off term is not discarded as in normal round-off, but is retained, preventing the error from accumulating. The error in the answer is due only to the final rounding off. If the independent variables are returned to their initial values, the dependent variable will return to its initial value.

Round-off error is not the only error that occurs in incremental integration and arithmetic processes which use integration. More accurately, integration accumulates the round-off error which may grow systematically or statistically. The integrated round-off error is called drift. To investigate drift, consider the polygonal function:

$$F(T) = F(T_i) + (T - T_i)\left[\frac{F(T_i + \Delta T_{i+1}) - F(T_i)}{\Delta T_{i+1}}\right] \tag{60}$$

$$\text{for } T_i \le T \le T_{i+1} \quad \text{if } \Delta T_{i+1} = +1$$

$$T_i \ge T \ge T_{i+1} \quad \text{if } \Delta T_{i+1} = -1$$

$$\text{and } T_{i+1} = T_i + \Delta T_{i+1}$$

Then

$$\int_{T_o}^{T_n} F(T)dT = \int_{T_o}^{T_1} \left\{F(T_o) + \left[T - T_o\right]\left[\frac{F(T_1) - F(T_o)}{T_1 - T_o}\right]\right\}dT$$

$$+ \int_{T_1}^{T_2} \left\{F(T_1) + \left[T - T_1\right]\left[\frac{F(T_2) - F(T_1)}{T_2 - T_1}\right]\right\}dT + \cdots + \int_{T_{n-1}}^{T_n} \left\{F(T_{n-1})\right.$$

$$\left. + (T - T_{n-1})\left[\frac{F(T_n) - F(T_{n-1})}{T_n - T_{n-1}}\right]\right\} dT$$

$$= F(T_o)\,(T_1 - T_o) + \left[\frac{F(T_1) - F(T_o)}{T_1 - T_2}\right]\left[\frac{T^2}{2} - TT_o\right]\Bigg|_{T_o}^{T_1} + \cdots$$

$$+ F(T_{n-1})(T_n - T_{n-1}) + \left[\frac{F(T_n) - F(T_{n-1})}{T_n - T_{n-1}}\right]\left[\frac{T^2}{2} - TT_{n-1}\right]_{T_{n-1}}^{T_n}$$

PX 56-4

$$= F(T_0)(T_1 - T_0) + \left[F(T_1) - F(T_0)\right]\left[\frac{\frac{T_1^2}{2} - T_1T_0 + \frac{T_0^2}{2}}{T_1 - T_0}\right] + \ldots$$

$$+ F(T_{n-1})(T_n - T_{n-1}) + \left[F(T_n) - F(T_{n-1})\right]\frac{\left[\frac{T_n^2}{2} - T_nT_{n-1} + \frac{T_{n-1}^2}{2}\right]}{T_n - T_{n-1}}$$

$$\pm F(T_0)(T_1 - T_0) + (T_1 - T_0)\frac{\left[F(T_1) - F(T_0)\right]}{2} + \ldots$$

$$+ (T_n - T_{n-1})\,F(T_{n-1}) + \frac{T_n - T_{n-1}}{2}\left[F(T_n) - F(T_{n-1})\right]$$

$$= (T_1 - T_0)\left[\frac{F(T_1) + F(T_0)}{2}\right] + \ldots + (T_n - T_{n-1})\left[\frac{F(T_n) + F(T_{n-1})}{2}\right]$$

$$= \sum_{i=1}^{n} \frac{F(T_i) + F(T_{i-1})}{2}\,\Delta T_i$$

That is:

$$\int_{T_0}^{T_n} F(T)\,dT = \sum_{i=1}^{n} \frac{F(T_i) + F(T_{i-1})}{2}\,\Delta T_i \tag{61}$$

This form is suggestive of the incremental integration formula, equation 39. Substituting equation 61 into equation 39,

$$W_n - W_o = \frac{2}{S}\int_{T_o}^{T_n} U\,dT - \frac{R_n}{S} \tag{62}$$

If U is the polygonal function:

$$U(T) = U(T_i) + (T - T_i)\left[\frac{U(T_{i+1}) - U(T_i)}{\Delta T_{i+1}}\right] \tag{63}$$

$$\text{for } T_i \le T \le T_{i+1} \qquad \text{if } \Delta T_{i+1} = +1$$

$$T_i \ge T \ge T_{i+1} \qquad \text{if } \Delta T_{i+1} = -1$$

If the process has settled in n major cycles, $\frac{R_n}{S}$ can be neglected and

$$W_n - W_o \approx \frac{2}{S}\int_{T_o}^{T_n} U\,dT \tag{64}$$

Equation 64 is accurate except for the round-off error $\frac{R_n}{S}$ for the polygonal function defined by equation 63. In many applications, however, the function

to be integrated is not of this form. Let the function to be integrated be G(T). Let the changes of G from one major cycle to the next have a magnitude no greater than one. The polygonal function U(T) can be made to approximate G(T) within one increment. Define this difference as:

$$S(T) = U(T) - G(T) \tag{65}$$

S(T) is the input round-off error.

Then: $$\frac{2}{S}\int_{T_o}^{T_n} G(T)\,dT = \frac{2}{S}\int_{T_o}^{T_n} U(T)\,dT + \frac{2}{S}\int_{T_o}^{T_n} S(T)\,dT \tag{66}$$

The term $\frac{2}{S}\int_{T_o}^{T_n} S(T)\,dT$ represents the error that would be caused by the approximation $G(T) \approx U(T)$. Since $|S(T)| \leq 1$, $\left|\int_{T_o}^{T_n} S(T)\,dT\right| \leq n$. Moreover, the actual error is usually much less than n because the average value of S(T) is approximately one-half in most cases instead of one, and the sign of S(T) tends to alternate. If one assumes that this error accumulated statistically and that the probable error for the round off error for one major cycle is σ, then the probable error for n major cycles is $\sigma\sqrt{n}$. Since the probable error σ is the probable error of the term $\frac{2}{S}\int_{T_i}^{T_{i+1}} S(T)\,dT$, its value for most applications is about 0.0005. Under these circumstances an accumulated probable error will be equal to one increment in 4,000,000 major cycles (5.6 hours of computer operation).

To show that T is not a single valued function of W, it is sufficient to show that for two different major cycles with the same value of W there are different values for T. Consider a sequence of ΔW's such that they are + 1 until W_n is reached, and are - 1 immediately thereafter. In this case . . . . $\Delta W_n = 1$, $\Delta W_{n+1} = -1$. Then $W_{n+1} = W_{n-1} + \Delta W_n + \Delta W_{n+1} = W_{n-1}$. Without loss of generality assume that

$$0 < R_{n-2} < S + 2. \tag{67}$$

Assuming that W is positive for this problem, equation 57

$$\Delta T_{i+1} = - \operatorname{sgn} W \operatorname{sgn} R_i \tag{57}$$

indicates that $\Delta T_{n-1} = - 1$. Substituting in equation 41

$$R_{n-1} = R_{n-2} + (W_{n-1} + W_{n-2})\ \Delta T_{n-1} - S\Delta W_{n-1}$$

$$= R_{n-2} - 2W_{n-2} - 1 - S < 0$$

Therefore,

$$\Delta T_n = + 1.$$

Then $R_n = R_{n-1} + (W_n + W_{n-1})\ \Delta T_n - S\,\Delta W_n$

$$= R_{n-2} - 2W_{n-2} - 1 - S + 2W_{n-2} + 3 - S$$

$$R_n = R_{n-2} + 2 - 2S < 0$$

Therefore,

$$T_{n+1} = + 1.$$

Then $T_{n+1} = T_{n-1} + \Delta T_n + \Delta T_{n+1} = T_{n-1} + 2 \neq T_{n-1}$

Although $W_{n+1} = W_{n-1}$, $T_{n+1} \neq T_{n-1}$. T is not a single valued function of W. Therefore, the drift in the logarithm is not reversible.

(6) SUMMARY. - The algorisms of addition, subtraction, multiplication, and division are subject only to round-off error. These errors are small, e.g., with full scale numbers equal to 1000 the round-off error is about 0.1 percent. The algorisms involving integration are subject to drift as well as round-off errors. Drift is either reversible or irreversible and it is either systematic or random. (Actually, there are no random processes in an incremental computer. By random is meant that which cannot be predicted analytically from the difference equations describing the process.) The random drift is very small and can usually be neglected. The systematic drift can be anticipated analytically, and perhaps eliminated by a suitable modification of the algorism. Reversible drift occurs when a single valued function is integrated. Such drift is zero

when the independent variable is returned to its initial value. The algorism for differentiation is unstable, but can be used with suitable feedback.

By using the smallest value of a scale factor that allows the step to keep up in the algorisms for addition, subtraction, multiplication, and integration and the largest scale factor for division and differentiation, the round-off errors can be kept to a minimum.

f. APPLICATIONS OF THE INCREMENTAL TECHNIQUES USING SEVERAL STEPS. - In the following discussion the steps for the variables are indicated by a superscript. As before, the number of the major cycle is indicated by a subscript. Independent variables will be indicated by letters without superscripts.

(1) FILTERS. - A single RC low pass filter satisfies the following differential equation:

$$e_j - e_o = RC \frac{de_o}{dt}$$

or
$$RCe_o = \int_x^T (e_j - e_o)dt \tag{68}$$

By using a single integration step with two simultaneous point slope (or Euler) integrations, a single stage RC filter represented by equation 68 can be realized. The algorism for this step is:

$$U_o = (e_j)_o \qquad \Delta U_i = (\Delta e_j)_i$$

$$\Delta Q_i = -\Delta T_i = +1 \qquad \Delta P_i = 0$$

$$V_o = 0 \qquad \Delta V_i = \Delta W_i = \text{sgn } R_{i-1}$$

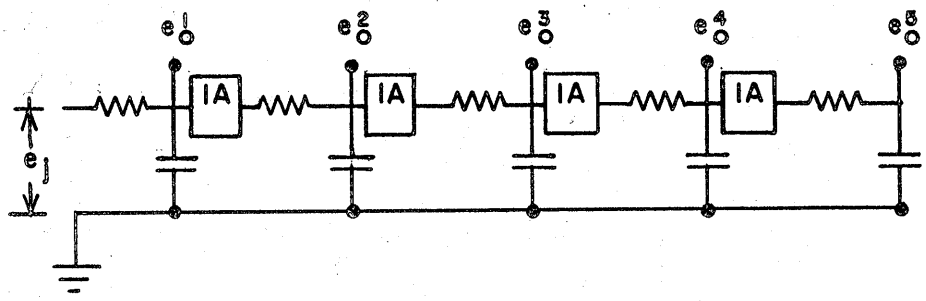
$$S = RC + 2$$

This algorism corresponds to the integral equation:

$$\int_{t_{n-\frac{1}{2}}}^{t_{n+\frac{1}{2}}} e_j d\tau - \int_{t_{n-\frac{1}{2}}}^{t_{n+\frac{1}{2}}} e_o d = (RC + 2)e_o$$

or
$$\int_{t_{n-\frac{1}{2}}}^{t_{n+\frac{1}{2}}} e_j d - \int_{t_{n-\frac{1}{2}}}^{t_{n+\frac{1}{2}}} e_o d = RCe_o \tag{69}$$

which is the filter differential equation 68. Therefore, when equation 69 is satisfied $W_i^{(1)}$ is very nearly e_o. These filters can be cascaded to give rather sharp attenuation. By cascading five sections, the mathematics simulates five electrical RC filters cascaded with unity gain isolation amplifiers between sections. The circuit would look like:



This program was run numerically on a simulation routine in an 1103 large scale computer to obtain the frequency response of the system and the five outputs agreed in amplitude and phase shift with that expected from filter theory.

(2) POLYNOMIAL. - Since multiplication can be performed in a single step, a quadratic can befformed in one step. If it is desired to form $Ax^2 + Bx + C$, form the product

$$W_n - W_o = Ax(x + \frac{B}{A}) \text{ and let } W_o = C.$$

Then W_n is the desired quadratic. In specific incremental notation let $U_o = x_o$, $V_o = x_o + \frac{B}{A}$, $\Delta P_i = 0$, $\Delta U_i = \Delta V_i = \Delta T_i = \Delta Q_i = \Delta x_i$, and $\Delta W_{i+1} =$ signum R_i. Then $U = x$, $V = x + \frac{B}{A}$, and

$$W_n - W_o = \frac{x(x + \frac{B}{A})}{S} \quad \text{when the step has settled down.}$$

The scale factor, S, is chosen as a compromise as indicated by the following discussion. For the proper choice of the initial value W_o, W_n is proportional to the desired quadratic. A cubic can be generated in two steps. Probably the simplest way of generating the cubic $Ax^3 + Bx^2 + Cx + D$ is to generate the

quadratic $Ax^2 + Bx + C$ in one step, and multiply it by x and add D in the second step. A more general way is to factor the cubic into

$$(x + a)\left\{Ax^2 + (B - aA)x + (C + a^2A - Ba)\right\} + D - a^3A + a^2B - ac \quad (70)$$

The quadratic indicated is generated in the first step and the quadratic is multiplied by x + a and the constant term is added to it in the second step. Factoring the quadratic in this manner is helpful because the usual limitation on the accuracy of such a computation is due to the scaling of the quadratic which is necessary in the first step to allow the step to keep up with changes that occur in the independent variable. This rate of change is given by:

$$\frac{d}{dt}\left\{Ax^2 + (B - aA)x + (C + a^2A - Ba)\right\} = (2Ax + B - aA)\left|\frac{dx}{dt}\right| \quad (71)$$

where $\left|\frac{dx}{dt}\right|$ is the maximum absolute value of the rate of change of x at this value of x. The constant "a" is chosen such that the maximum rate at which the quadratic can change at all values of x is as small as possible. If $\left|\frac{dx}{dt}\right|$ is constant throughout the range of x, equation 71 is linear in x and $a = \frac{B}{A} + x_{min} + x_{max}$. For any choice of a, if the scale factor S is equal to the maximum rate of change of the quadratic, the quadratic will keep up with the changes of x throughout its range. For this choice of a, the quadratic will keep up with the minimum of round-off error. In the case of a constant $\left|\frac{dx}{dt}\right|$, $S = A(x_{max} - x_{min})\left|\frac{dx}{dt}\right|$.

Quartics and higher degree polynomials can be generated as well. In general a polynomial of a degree n can be generated in n-1 steps. Variation in the details of generation can be made allowing the programmer some flexibility in adapting the computer to a particular polynomial. For example, a sixth degree polynomial can be formed by the product of two cubics, the product of a quartic and a quadratic, the product of three quadratics, or the five steps can produce the sequence of quadratic, cubic, quartic, fifth degree, and sixth degree. One of these systems would be the best for a given sixth degree polynomial. The

best system would be the one which can keep up with the change in the independent variable with the minimum of error in the polynomial caused by round-off of all the steps. To obtain the various constants required for the optimum system, criteria of the type suggested for the cubic should be used. These criteria are too complicated to be included in a paper of this scope.

(3) SINE AND COSINE. - The differential equation $\frac{d^2y}{dx^2} + \omega^2 y = 0$ (72) defines the trigonometric sine and cosine. This equation can be solved by the incremental computer either through the use of the differentiation routine or through the integration routine. Inasmuch as differentiation requires that the independent variable increments always have the same sign, integration is the more useful. Integrating equation 72 twice we have:

$$y(x) = -\omega^2 \int_0^t dt \left\{ \int_0^t d\eta\, y(\eta) + C_1 \right\} + C_2 \tag{73}$$

The general solution to either equation 72 or equation 73 is:

$$y = A \cos \omega x + B \sin \omega x \tag{74}$$

If we wish to find the cos ωx, the initial conditions are $y(o) = A$. Then $y = A \cos \omega x$. The first integration of $A \cos \omega x$ yields $A \sin \omega x$. The second integration yields $-A \cos \omega x$. The sum of $A \cos \omega x$ and $-A \cos \omega x$ should equal 0. The sign of the sum of the two calculated terms determines the sign of the correction to be applied to $A \cos \omega x$.

The sine and cosine can be generated by the direct application of two steps of integration. This algorism is:

$$\Delta P_i^{(1)} = \Delta P_i^{(2)} = 0 \tag{75}$$

$$\Delta T_i^{(1)} = \Delta Q_i^{(1)} = -\Delta T_i^{(2)} = -\Delta Q_i^{(2)} = \Delta x_i \tag{76}$$

$$U_o^{(1)} = V_o^{(1)} = A \cos \omega x_o \tag{77}$$

$$U_o^{(2)} = V_o^{(2)} = A \sin \omega x_o + 1 \tag{78}$$

$$\Delta U_i^{(1)} = \Delta V_i^{(1)} = \Delta W_i^{(2)} = \operatorname{sgn} R_{i-1}^{(2)} \tag{79}$$

$$\Delta U_i^{(2)} = \Delta V_i^{(2)} = \Delta W_{i+1}^{(1)} = \text{sgn } R_i^{(1)} \tag{80}$$

$$S^{(1)} = S^{(2)} = \frac{2}{\omega} \geq 2A \tag{81}$$

One incremental change in x represents an angular change of $\frac{2}{S}$ radians. With this algorism the sine and cosine are generated very well under favorable circumstances. This program was tried with several values of A and S to hcheck on its usefulness. For example, it was found that with A = 1000 and S = 2000, that after 6,283 cycles representing 2 π radians of angular change, the error in the function A cos 2 π was between two and three increments, about 0.25 percent.

If, for some reason, it is known that the angle for which the sine and/or cosine is needed is limited by the problem, it is advantageous to change the scale factors so that they are no longer equal and are just small enough to allow both steps of the algorism to keep up with the change of the independent variable. If, for example, it is known that $-20^0 \leq \omega x \leq +20^0$, we know that

$$\left|\frac{d}{dx}\cos \omega x\right| = \left|\omega \sin \omega x\right| \leq \omega \cdot \quad 342$$

If $\omega = 0.001$, we can generate 2924 cos ω x in the second step instead of 1000 cos ω x by letting the scale factor in the second step be 684 instead of 2000 and the scale factor in the first step be 5848 instead of 2000. The variable 1000 sin ω x is computed in the first step just as before. By adjusting these scale factors the rounding off error is reduced for the cosine in this application causing less final error in both the sine and cosine.

By looking more closely at the difference equation actually solved by the algorism, the source of this small error can be found. Notice that in equation 79, $\Delta U_i^{(1)} = \Delta V_i^{(1)} = \text{sgn } R_{i-1}^{(2)}$. In this case there is a lag, i.e.,

$$\Delta_i \sin \omega x = \frac{\cos \omega x_{i-2} + \cos \omega x_{i-1}}{S} \Delta x_i$$

whereas $\Delta_i \cos \omega x = \frac{\sin \omega x_{i-1} + \sin \omega x_i}{S} \Delta x_i$.

The latter represents trapezoidal integration. The difference equation actually being solved is:

$$y_i - y_o = - \sum_{k=1}^{i-1} \left\{ \frac{\sum_{j=1}^{k} [y_j + y_{j-1}] \Delta T_j + \sum_{i=1}^{k-1} [y_j + y_{j-1}] \Delta T_j}{S^2} \right\} \Delta T_k \quad (82)$$

which corresponds to the integral equation:

$$y(x) = -\omega^2 \int_0^{x-1} d\tau \left[\int_0^{\tau} d\eta\, y(\eta) + C_1 \right] + C_2 \quad (83)$$

or to the differential equation:

$$\frac{d^2y(x)}{dx^2} + \omega^2 y(x-1) = 0 \quad (84)$$

Expanding equation 84 in a Taylor's series,

$$\frac{S^2}{4} \frac{d^2y}{dx^2} + y(x) - \frac{dy(x)}{dx} + \frac{1}{2} \frac{d^2y(x)}{dx^2} - \frac{1}{6} \frac{d^3y(x)}{dx^3} + \ldots = 0 \quad (85)$$

This series is strongly convergent. An extremely small error is made by rounding off equation 85 to:

$$\frac{S^2}{4} y'' + y - y' + \frac{y''}{2} = 0 \quad (86)$$

Equation 86 has the general solution:

$$y(x) = e^{\frac{2x}{S^2+2}} \left[A \sin \frac{2\sqrt{S^2+1}}{2 + S^2} x + B \cos \frac{2\sqrt{S^2+1}}{2 + S^2} x \right] \quad (87)$$

In the usual case, S is of the order of 1000. With such an S, it is justified to approximate the solution by:

$$y(x) = e^{\frac{2x}{S^2}} \left[A \sin \frac{2}{S} x + B \cos \frac{2}{S} x \right] \quad (88)$$

Equation 88 represents the solution to the difference equation 82 actually being solved by the algorism of equations 75 through equation 81. For many applications this solution is close enough to the sine and cosine to be used. In

particular, in the usual case where the independent variable changes relatively slowly, the scale factor can be made large, and the exponential multiplier in equation 88 changes from one slowly. In the cases where S cannot be made large and the computation is to be carried on for a number of seconds, this algorism produces considerable error.

For a numerical check an incremental algorism of this kind was computed on an 1103 large scale digital computer. All of the steps were performed to simulate the incremental operation. The initial values were A = 150, B = 0, and $x_0 = 0$. In order to show how the algorism would differ from the sine in a reasonable number of cycles, a small scale factor was required. The scale factor was 400. After 23,562 major cycles (corresponding to two minutes of incremental computer running time), y was found to be -200.9. From equation 88,

$e^{\frac{2 \cdot 23,562}{160,000}} \cdot 150 \cdot (-0.9987) = -201.1$; in very good agreement. In this case, after a long time (two minutes), and under poor conditions (small S and no changes in ΔT), a 35 percent error was produced. Compare this result with the more realistic result above.

In the derivation of difference equation 82 it was assumed that ΔT was + 1. Since x is an independent variable, it is conceivable that ΔT can be both signs. During each major cycle that ΔT is different from the previous major cycle, the difference equation 82 does not apply, and there is no drift. Defining:

$N \equiv$ no. of major cycles,

$n \equiv$ no. of changes in T,

the equation 88 becomes:

$$y(x) = e^{\frac{2(N-n)}{S^2}} \left[A \sin \frac{2}{S} x + B \cos \frac{2}{S} x \right] \qquad (89)$$

PX 56-4

The variable n can be large when x is varying slowly causing the changes in x to be alternately plus and minus one.

It is possible to alter the algorism to generate the trigonometric functions more accurately.

In some applications x is monotonic, that is, $\Delta T = \Delta x = +1$ for every major cycle. Such would be the case, for example, if a sine function of a certain frequency (relative to the speed of the drum) were desired. The sine-cosine algorism for monotonic independent variable can be formed by modifying the algorism to read:

$$\Delta T_i^{(1)} = \Delta T_i^{(2)} = \Delta P_i^{(1)} = \Delta P_i^{(2)} = 0$$

$$\Delta Q_i^{(1)} = -\Delta Q_i^{(2)} = \Delta x_i = +1$$

$$U_o^{(1)} = A \cos \omega x_o$$

$$U_o^{(2)} = A \sin \omega_o (x_o + \tfrac{1}{2}) + 1$$

$$S^{(1)} = S^{(2)} = \frac{1}{\omega} \geq A$$

$$\Delta U_i^{(1)} = \Delta W_i^{(2)} = \operatorname{sgn} R_{i-1}^{(2)}$$

$$\Delta U_i^{(2)} = \Delta W_{i+1}^{(1)} = \operatorname{sgn} R_i^{(1)}$$

This modified algorism, using point slope instead of trapezoidal integration, corresponds to the incremental equation:

$$U_n^{(1)} = -\sum_{i=0}^{n-1} \frac{U_i^{(2)} \Delta x_i}{S^{(2)}} = -\sum_{i=0}^{n-1} \frac{\Delta x_i}{S^{(2)}} \sum_{j=0}^{i} \frac{\Delta x_j}{S^{(1)}} U_j^{(1)}$$

$$U_n^{(1)} = -\omega^2 \sum_{i=0}^{n-1} \sum_{j=0}^{i} U_j^{(1)}$$

In general $y(t+\frac{1}{2}) - y(t-\frac{1}{2}) = y(t) + \frac{1}{2}y'(t) + \frac{1}{8}y''' + \frac{1}{48}y''' + \frac{1}{384}y''' + \cdots$

$$- y(t) - \frac{1}{2}y'(t) + \frac{1}{8}y''(t) - \frac{1}{48}y''' + \frac{1}{384}y''' + \cdots$$

$$= y'(t) = \frac{1}{24}y''' + \frac{1}{1930}y^{v} + \cdots$$

PX 56-4

$$\overset{\prime}{y}(t) = y(t+\tfrac{1}{2}) - y(t-\tfrac{1}{2}) - \tfrac{1}{24}\overset{\prime\prime\prime}{y} - \tfrac{1}{1920}\overset{v}{y} + \cdots\cdots$$

$$\sum_{t=a}^{b} \overset{\prime}{y}(t) = y(b+\tfrac{1}{2}) - y(a-\tfrac{1}{2}) - \sum_{t=a}^{b} (\tfrac{1}{24}\overset{\prime\prime\prime}{y} + \tfrac{1}{1920}\overset{v}{y} + \cdots\cdots)$$

or approximately

$$\sum_{t=a}^{b} y'(t) = \int_{b-\frac{1}{2}}^{b+\frac{1}{2}} y'(\tau)d\tau - \tfrac{1}{24}\int_{a-\frac{1}{2}}^{b+\frac{1}{2}} \overset{\prime\prime\prime}{y}d\tau - \tfrac{1}{1920}\int_{a-\frac{1}{2}}^{b+\frac{1}{2}} \overset{v}{y}d\tau + \cdots,$$

$$= \int_{a-\frac{1}{2}}^{b+\frac{1}{2}} y'(\tau)d\tau - \frac{\overset{\prime\prime}{y}(b+\frac{1}{2}) - \overset{\prime\prime}{y}(a-\frac{1}{2})}{24} - \frac{\overset{\prime\prime\prime}{y}(b+\frac{1}{2}) - \overset{\prime\prime\prime}{y}(b+\frac{1}{2})}{1920} + \cdots\cdots$$

Then

$$\sum_{j=0}^{i} U_j^{(1)} = \int_{a-\frac{1}{2}}^{t(i+\frac{1}{2})} U(\tau)d\tau - \omega\frac{U'(i+\frac{1}{2})}{24} - \omega\frac{U'''(i+\frac{1}{2})}{1920} + \cdots + \text{Const.}$$

So

$$\omega^2 \sum_{i=0}^{n-1}\sum_{j=0}^{i} U_j^{(1)} = \omega^2 \int_{-1}^{n-1+1} d\eta \int_{-1}^{n} d\tau\, U(\tau) - \omega^4\frac{U}{24} - \frac{\omega^4 U''}{1920} + \cdots + \text{Const.}$$

That is,

$$-U_n^{(1)} = \omega^2 \int_{o}^{t_n} dt \int_{o}^{t} d\tau\, U_n^{(1)} - \omega^4 U_n^{(1)} - \cdots$$

Differentiating twice

$$(1 - \omega^4)\, U_n^{(1)} + \omega^2\, U_n^{(1)} = 0$$

or $$U_n^{(1)} + \frac{\omega^2}{1-\omega^4} U_n^{(1)} = 0$$

Let $$\Omega^2 = \frac{\omega^2}{1-\omega^4} \qquad (90)$$

Substituting in equation 90

$$y''(x) + \Omega^2 y(x) = 0 \qquad (91)$$

PX 56-4

This differential equation is of the form of equation 72. Equation 91 has the general solution $y(x) = A \sin \Omega x + B \cos \Omega x$. The approximations necessary to derive equation 91 from the integral equation describing the effect of the modified algorism are much less severe than those necessary for the unmodified algorism. The modified algorism is much more accurate than the unmodified, virtually eliminating the exponential term of equation 88 found for the unmodified algorism. The only other errors of this algorism are the accumulation of the integral of the difference between the sine and cosine functions and their polygonal representations and the final round-off error. It must be remembered, however, that this modified algorism is most accurate when the independent variable is monotonic. With non-monotone variables, this method is more accurate than the method using trapezoidal integration.

To demonstrate these methods, simulations were carried out. In the case of a monotone variable, the two methods were used to generate 120 $\sin \omega t$. The simulations were carried out for 13,000 major cycles (65 seconds of real time and 13 complete 360° rotations of the input angle). Here again the scale factor was chosen low in order to show the errors in a bad case. The results were; the first method (trapezoidal integration) yielded 157.4 and the second method (point slope integration) yielded 120.38. Their relative errors were 31 percent and 0.3 percent in this extreme case.

A second simulation was done where the input variable was a sinusoidal function of time instead of monotone. The errors made by the second method were about one-third that of the first.

In any of the systems discussed, if ω is very large, S must be made small in the integration steps and the amplitude, A, will have to be small. This combination tends to yield more drift due to the differences between the desired trigonometric function and its polygonal representation. Both of these systems

are subject to this drift to the same extent. Since polynomials are not subject to drift error, a polynomial approximation to the sine or cosine can be made which will give sufficient accuracy with final round off being the only computation error. If the independent variable is restricted to a suitable range, e.g., $-\pi \leq \omega \quad x \leq +\pi$, the approximation can be made directly by Legendre or Tchebycheff polynomials. If, on the other hand, the independent variable has an extended range, the polynomial should be chosen to approximate the trigonometric function from $\frac{-\pi}{2} \leq \theta \leq \frac{+\pi}{2}$, and let $\theta = \omega x + 2n\pi$ where n is a positive or negative integer. The polynomial approximation to the sine:

$$\sin x \approx .986x - .143x^3$$

is in error no more than 0.006 at any value of x between $\frac{-\pi}{2}$ and $\frac{+\pi}{2}$. This cubic approximation should be close enough for those cases where the use of the polynomial is called for. In the extreme case, where a very accurate sine is needed after a very large number of major cycles, it may be necessary to use a fifth degree approximation.

(4) CONCLUSION. - To generate the sine and cosine, the simplest and generally most useful method is to use two steps for double integration. This method lends itself to a wide range of applications. In some cases this method may lead to a drift error that is excessive.

A second method has been devised which greatly reduces this drift error in those cases where the independent variable is monotone. In every case of a monotonic independent variable this method should be used since it, too, requires only two steps of integration, but yields greater accuracy than the first method.

In some cases, generation of the sine and cosine by double integration may not be satisfactory. Suitable approximations can be made with polynomials

either with one polynomial for the whole range or with a polynomial approximation from $\frac{-\pi}{2}$ to $\frac{+\pi}{2}$ repeated over the entire range.

2. PROGRAMMING THE INCREMENTAL COMPUTER

a. INTRODUCTION. - This section is concerned with the preparation of a problem for computation on an incremental computer. This preparation may be arbitrarily divided into a series of interrelated operations. It is assumed that, as is usually the case, the problem is to generate some output variable, which is a function of the input variables. The first step is to determine a series of operations which will generate the desired function. This procedure is known as "programming". The selection of the proper operations and intermediate functions is subject to various criteria to be discussed below. Further, the optimum sequence of operations must be determined.

The second step is to determine coefficients for each of the intermediate functions, a procedure known as "scaling". The scaling of a function determines the precision to which it is known in the computer and also the rate at which the computer quantity can vary. The accuracy of the output function is quite dependent upon this scaling operation. As the incremental computer computes a function relative to its initial value, the initial values of all the functions must be computed and the appropriate quantities inserted in each register. In cases where the inputs start out at values different from the initial values in the computer, it takes a period of time for the functions in the computer to equate themselves with the inputs. This period, known as "settling time" is kept to a minimum by proper choice of initial values.

The various steps listed separately above are actually interrelated. In practice this means that the changes in one step affect the others so that it is often necessary to iterate the steps.

Finally, the operations must be coded for the computer, and the input tapes must be cut.

b. DEFINITIONS. - At this point some of the terms common to this work will be defined.

Coefficient - This is a constant, which a function is multiplied by, in the computer. A function x is represented by Cx increments in the computer, where C_x is the coefficient of the function.

A direct operation is defined as one in which ΔW is the dependent variable.

An inverse operation is one in which ΔW is an independent variable.

The output, or result of step, refers to either the sign of the remainder of the step or else the dependent variable generated by the step and defined by the equation

$$F_n = F_o + \sum_{i=1}^{n} \text{sgn } R_i$$

Overloading- A step is said to be overloaded if the output function changes more rapidly than one increment per cycle. This condition is characterized by large values on the R-line. It is the opposite of keeping up.

Scaling is the operation of determining scale factors (and/or coefficients) for the steps of a program.

Scaling down is to decrease the coefficient of the output of a step, scaling up to increase it.

Settling - Immediately after a computation has started, many steps and inputs will be overloaded. When no more steps or inputs are overloaded, they are all settled. The settling time is the time from the start of a computation until all steps and inputs are settled.

Flipping a step is to reverse the sign of its dependent variable.

c. PROGRAMMING

(1) CHOICE OF FUNCTIONS. - In preparing a problem for an incremental computer the first operation is to write the program. This consists of a

sequence of intermediate functions which bridge the gap from the input functions to the output functions. In order to obtain the greatest possible accuracy in the output, it is necessary to choose terms of the sequence which introduce a minimum amount of error, and, also, to arrange them in an optimum order.

In choosing a set of intermediate functions, it is desirable to select functions which fall in the general category of analytically "well-behaved" functions; that is to say that they should be continuous and reasonably bounded. Furthermore, they should be relatively insensitive to absolute error and should tend to minimize its propogation. For example, the relative error in the difference between two almost equal terms is quite sensitive to variations in either one of them. Or, to cite another case, the square root of a function becomes highly sensitive to absolute error in the function as it approaches zero.

In addition to the restrictions placed on the function from the general standpoint of error, the incremental computer itself has properties which should be taken into account. Due to the representation of a function which is used in the computer, absolute rather than relative error becomes the most applicable type. In the many cases where a certain relative error is aimed for it is usually desirable to select functions with as limited a range as possible so that the two types of error may be as similar as possible. This limiting of range serves other useful functions as well, for as the maximum rate of change of any computer variable is one increment per cycle, a limited range allows a much greater total number of increments to be used to represent the function, other things being equal, for the same settling time. Scaling of a restricted function is usually more satisfactory as wide ranges tend to allow scaling to be too high at one end and too low at the other. This restriction of functions applies not only to the functions themselves but to their rates as

well. Here, even more clearly, the accuracy with which a function may be represented is inversely proportional to its maximum rate. This may be seen from the fact that the round-off error and the maximum rate are both proportional to increment size.

The principles which make inverse pairs, e.g., multiplication and division, so easily obtainable in the incremental computer also make it possible to set up larger loops by which implicit functions may be computed. Larger loops, however, are not as foolproof and may become unstable, either oscillating or else changing as rapidly as possible to some value other than the desired one.

The fact that the incremental computer is rate-limited becomes quite important when dealing with loops. This limiting may restrict changes in such a manner so as to diminish overshoot. On the other hand, there have been cases where it limited negative feedback so as to allow instability. In general, however, the usual analytical methods developed for feedback loops should apply to incremental loops as well.

Once a sequence has been selected, the next step is to determine incremental methods for generating it. In addition to the specified operations this generation may involve such techniques as polynomial approximation and double integration for sine and cosine.

On long programs, especially, it is desirable to pack as much on to the drum as possible. This means performing as many operations in each step as possible. Though usually only one operation can be performed at a time, there are a few cases where two or more operations can be done at once. One of the most obvious cases of this is the use of the P-input. By using it, a term may be added at the same time another operations is being performed. The chief limitation to the P-input is that it must be scaled the same as W. This usually is not a serious problem. Where it is, however, it may be necessary to use a separate

addition step. Luckily, addition steps usually have short word lengths so that they are not as costly space-wise as most other steps. In some cases addition can be accomplished using only the Q- and P-inputs, thus leaving the V-line available for comparisons. As a final remark it might be mentioned that repetition should be kept to a minimum with no function being generated more than once if at all possible. It should be pointed out that it is less costly as a rule to add in a function already generated than to regenerate it.

(2) SEQUENCING. - After the programmer has determined the set of functions to be used, the next step is to arrange them in the best possible sequence. In this operation the programmer aims (1) to have minimum lag between input and output, i.e., minimum computation time, and (2) to efficiently utilize the RAM (random access memory) so as to have adequate memory positions available.

In the present computer, due to (1) the fact that the result of a computation is not stored until the step following, and (2) the fact that the increments must be read in the step preceding the computation, a result computed in cycle n cannot be used until cycle n + 2. In arranging the sequence of steps, to obtain the minimum computation lag, mathematically consecutive operations should therefore fall on alternate cycles. The (n + 1)th cycle would use the result of the previous major cycle, thus causing a major cycle delay.

The simplest means of spacing the operations as described above is the 2-interlace. With the 2-interlace, consecutive operations are written in alternate drum positions with the first half of the program occupying, say, the even numbered positions and the second half occupying the odd-numbered positions. With this arrangement there is a two drum revolution delay between the beginning and end of the program, though all of the quantities are computed twice within this period. (In the first revolution the first half of the program is computed. In the second revolution the results of the first half are used as a

basis for the computations of the second half while the first half is being simultaneously recomputed.) The effect of the two-interlace is to reduce the computation delay from one cycle per step to two cycles over-all. It has the advantage of being simple to use with a program written out in consecutive fashion. It is possible, however, to reduce the delay still further in many cases. It often happens that there are sequences of operations which can take place simultaneously. It is possible to interlace these sequences so that both are performed in the same time it would take for one using the overall two-interlace mentioned above. It is often possible to divide the steps in a program into two categories: those rapidly varying and those slowly varying. If the rapidly varying functions are placed on alternate steps, and the slowly varying functions placed in between, the result is essentially a one-cycle time delay.

To obtain the shortest possible time delay it is usually necessary to use a combination of the above methods, because a method as simple as the two-interlace will reduce delay to two cycles (for a completely consecutive program), more elaborate techniques may often be unnecessary. In some cases it is possible to further compensate for delay by extrapolating ahead a period equal to the delay. A one-cycle delay occurs when an input is operated upon during the same cycle in which it is taken in. This is due to the fact that the increments for an operation are drawn from the RAM in the step preceding and that the results of a comparison are not available until the cycle following. To obtain the shortest possible input delay, the input should be first used during the second step after the comparison.

Generally speaking, one RAM position is needed for each step and for each input. As the number of cores is limited (in this computer to 64), RAM capacity becomes a limiting factor in long programs. To expand the capacity of the

computer in this respect, a second head was installed on the R-line to allow the following command:

$$W_i = + \text{ sgn } R_{i-1}$$

This enables the programmer in many cases to obtain ΔW without referencing the RAM. Without this command most memory positions would be always occupied since sgn R formed by a step is needed as ΔW in the step on the next revolution. With it the position usually need be occupied only until the last other step calling for sgn R. As in a typical program sgn R may be used only during a few steps immediately following the step in which it is generated, the same core may be used to store several values of sgn R in the course of a revolution; one after the other. In long programs, the programmer may have to sequence the steps so as to make maximum use of this feature. In most cases, however, it would appear the capacity of the RAM is adequate so that no special rearrangement is necessary. In a typical problem that uses the entire drum capacity, about half of the RAM is needed.

d. SCALING. - The aim of the scaling operation is to generate the sequence of functions selected in the programming operation, with the minimum of error. There are two principal sources of error to be dealt with: 1) round-off error and 2) overloading error. The former is due to the fact that a function cannot be expressed any more accurately, on the average, than to the nearest whole increment, while the latter is due to the error which is introduced when a function changes more rapidly than one increment per cycle so that the step generating it cannot keep up.

Unfortunately, these two errors are related in such a way that to reduce one usually increases the other. When a step is scaled up, which is the only way to decrease round-off error for the step, then the maximum rate of the output of the step is decreased so that overloading becomes more likely. Similarly,

scaling down a step to decrease the likelihood of overloading increases the round-off error for the step. The two types of error are different in nature; whereas round-off error occurs all the time, overloading occurs only in certain cases, for example, during settling, or when a variable is changing at a high rate. Thus, it is a case of comparing an error which is always present with the probability of another error being present. In cases where scaling is critical, a compromise normally has to be made.

In scaling programs where the problem is completely defined and the values of the functions throughout the computation are always known, a straight analytical approach may be used, scaling the steps so that the rate of their output is never greater than one increment per cycle unless overloading is deliberately allowed to occur on occasion for the sake of reducing round-off error.

In actual practice, however, it usually happens that the system is not completely known; with only general information on the functions being available. Two examples of this would be: 1) where the inputs can vary over wide ranges and are different each time, and 2) where the system is so complex that involved analysis makes scaling exceedingly time-consuming. One solution to these problems lies in simulation, or in trial computation, whereby a representative set of examples can be run with tentative scaling and the final scaling based on these. A second approach to these problems involves the use of approximations by which the analysis can be simplified. An example of such procedures is conservative scaling.

(1) CONSERVATIVE SCALING. - The ideal case of scaling is when all functions are always varying at the rate of one increment per cycle without any steps ever overloading. As the rate of a function in an actual case normally varies, a function usually varies at less than one increment per cycle on some occasions and at more than this rate (i.e., overloads the step) on others.

Though round-off error is less than one increment, overloading may cause an error of many increments. It might seem desirable, therefore, to scale a step so that it will never overload under any conditions. This is known as conservative scaling. It has the advantage of being relatively easy to do, and, provided the round-off error is tolerable, yields satisfactory results. It has the additional advantage of providing the shortest possible settling time for a given input to a step. The assumption for conservative scaling is that the input to each step can vary at the maximum possible rate, i.e., one increment per cycle, and in the direction causing greatest output rate. For this method of scaling, the ranges of the functions need to be known.

Conservative scaling usually applies to steps where the Δ P-input is not used. In all direct operations the assumption that P can vary at one increment per cycle automatically means that W must be able to vary at the same rate, without even considering the effect of the other inputs. The Δ P-input, therefore, will be assumed zero in the analysis of the individual operations which follow.*

Addition

$$S\Delta W_i \geqq U_o\Delta Q_i + V_o\Delta T_i$$

$$S \geqq |U_o| + |V_o| \quad .$$

Multiplication

$$S\Delta W \geqq U_i\Delta V_i + V_{i-1}\Delta U_i$$

$$S \geqq |U|_{max} + |V|_{max}$$

Integration

$$S\Delta W \geqq U_i\Delta T + U_{i-1}\Delta T$$

$$S \geqq 2|U|_{max}$$

Division

$$V_{i-1}\Delta U_i \geqq U_i\Delta V_i + S\Delta W_i$$

*S will be considered as positive unless otherwise stated.

$$|V|_{min} \geq |U|_{max} + S$$

$$|S| \leq |V|_{min} - |U|_{max}$$

Note that for division the minimum value of V is the limiting factor. It is possible to have cases where no S exists which will satisfy the equation.

Square Root

$$2U \Delta U \geq S\Delta W$$

$$S \leq 2|U|_{min}$$

Exponential

$$2U\Delta T \leq S\Delta W$$

$$S \geq 2|U|_{max}$$

Logarithm

$$2U\Delta T \geq S\Delta W$$

$$S \leq 2|U|_{min}$$

In conservative scaling the assumption is made that all functions vary at their maximum rate (i.e., one increment per cycle). Though this may often be the case during the settling period, during other times functions are usually changing at some rate less than maximum, and often this rate has some definite limit below one increment per cycle. To give improved scaling, a higher order method has been evolved. This modification assigns to each function an e (efficiency) factor defined as its maximum rate in increments per cycle. Scale factors are then calculated as above with the exception that appropriate terms are multiplied by their e-factors.

With this type of scaling the effect of settling time becomes more important. As was mentioned above, in conservative scaling a step will settle as fast as the inputs to it. In a completely conservatively scaled program, for example, the program will have settled as soon as the inputs have settled so that the overall settling time is merely the maximum input settling time. (In

the present computer, with an input range of 1000 increments, the overall settling time would be 500 cycles maximum for a conservatively scaled program, assuming all of the initial values to be at midscale.) When it is assumed, however, that the maximum rates are less than one increment per cycle, then some steps are going to overload during settling when rates are one increment per cycle. Settling will then take an additional period of time after the inputs have settled, during which the accumulated values on the R-lines in the various steps will be reduced to zero.

At this point a brief discussion of settling will be introduced. The settling period for a step may be arbitrarily divided into two parts: input limited and output limited. Input limiting is said to occur when the inputs are changing in such a way as to cause the output to vary at less than one increment per cycle. Output limiting is said to occur when a step is overloaded. The total settling time for a step may be calculated by first determining over which periods it is input and output limited. The times for the two types of settling are calculated separately and then added together. As an example, the case of multiplication will be considered. The factors U and V will be assumed to start from zero and proceed at one increment per cycle to the positive values U_{max} and V_{max} which are greater than S. Input limiting will occur for S/2 cycles, at which time the product will be S/4 increments. Output limiting will then occur until the value UV/S is reached. The total settling time is therefore: $\left\{S/2 + \frac{UV - S2/4}{S}\right\} = \frac{UV + S2/4}{S} = \frac{U \cdot V_{max}}{S} + \frac{3}{4}$ cycles. In this particular case the delay caused by the input limiting was equal to one-half the period over which input limiting occurred.

An output limited step tends to accumulate large values on the R-line, thus necessitating longer word length. In general, a step should be scaled up until it either takes too long to settle or else cannot keep up. If settling were

not a limitation, then all outputs would presumably be scaled to change at a maximum rate so that conservative scaling and its modification would be identical for all steps following input steps. As it is, settling time often appears to be a limiting factor so that the modification is of value.

Conservative scaling is inefficient in that to insure that no step ever overloads, the steps are usually scaled to run appreciably under full capacity. This situation can be improved by the simple expedient of scaling the steps upward to a value, say 30 percent, above the conservative value.

Because of its advantage, i.e., that all steps always keep up, conservative scaling should be used in cases where the accuracy is adequate. It provides minimum settling time and minimum word length. Its use is essentially mandatory in steps involving integration, for if an integrand does not always correspond to the function it is being integrated with respect to, integration error will be introduced. In the incremental computer this error is cumulative, and remains until computation is re-initiated. This applies also to operations implicitly using integration, such as generation of the logarithm and the exponential.

(2) WORD LENGTH CONSIDERATIONS. - When a program is so long that it necessitates use of the whole drum, the word length of each step must be taken into account. A step which is output-limited to any great extent accumulates large values on the R-line. This action is most likely to occur during the settling period just after the computer is turned on. One of the advantages of conservative scaling is that output limiting never occurs, hence the number on the R-line is never greater than 2S.

In cases when conservative scaling is not used some criterion other than the magnitude of S must be used to assign word length. One method is to simulate a series of extreme cases as a basis for picking word length. A straight analytical approach may also be used. It should be emphasized that while

temporary overloading is usually permissible, overflowing of the R-line must never be allowed to occur; for, if the R-line ever overflows, permanent, almost invariably serious, errors are introduced, making any additional computation useless.

(3) INPUTS - RANGE EXPANSION. - Inputs which vary over only a fraction of the range from zero to full scale may often be scaled up so that this region occupies an entire comparison interval. This is permissible because of the fact that the digital-to-analog converter converts only the first ten low order digits of a number, ignoring the others. The new regions are equal to the original (12 to 1012), plus some multiple of 1024.

(4) EXAMPLE: SCALING x^2. - As an example consider the function x^2, where $-5 < x < +3$ and $|\dot{x}| \leq .02$/cycle. First x is scaled. Assume, as is the case with the present computer, that the input range is 0 ± 500 increments. As the maximum value of $|x|$ is 5 it would appear possible to give x a coefficient of 100, thus, representing it over the range from -500 to +300 increments. To do this, however, would give the computer function a maximum rate of 1/C or .01/cycle. This is one-half the maximum rate of the external function. Now, while this condition might be tolerable, e.g., when the peak rate occurred only rarely, in general it is desirable to have the computer function always keep up. This is accomplished by specifying C = 50. The range of the function in increments is now from -250 to +150, a total range of 400. If it is assumed that x may be at any value when computation is initiated, then the logical initial value for x would be -50. If more is known about the behavior of x, a more suitable choice may be made.

Having scaled the input to the computer, the next step is to scale the function x^2 itself. First of all conservative scaling will be used. Referring to the algorism for squaring it will be noted that the quantity x appears in both

the U and V registers and that it is essentially added to R, every time x increases by one increment. The largest value which appears in these registers is -250, corresponding to x = -5. In the worst possible case, therefore, R changes by 500 every cycle, due to this action. To compensate for this we simply set S = 500. The coefficient of the output C_W is $\frac{(C_u)^2}{S} = \frac{(50)^2}{500} = 5$ $x^2_{max} = (5)^2 = 25$, 5(25) = 125 increments. Using a different form: $\frac{(-250)(-250)}{(500)} = 125$.

In this case the same result may be obtained using the rate of the function.

$$\dot{x^2} = 2x\dot{x}, \quad \dot{x^2}_{max} = 10(.02) = .2, \quad 1/.2 = 5$$

As a further illustration assume that the coefficient of 5 is not large enough to provide the desired accuracy and that for that reason C was raised to 10, (S = 250). In this case the maximum value of Cx^2 equals 250. If x varies at 0.02 per cycle when $|x| > 2.5$, however, x^2 will not keep up. For all values less than this (which in some cases may be most of the time) x^2 will keep up. If it is known that $\dot{x}$ is small whenever $|x|$ is large, then this type of scaling would keep up all of the time.

Consider now the settling time of x^2 under the two types of scaling, assuming x to be at its maximum. For conservative scaling (C = 5), x^2 will settle just as soon as x will; where $x_0 = -1$, in 200 cycles. In the second case, however, this is not true. Assuming that $x_0 = -1$ as above, x^2 will keep up for 125-50 = 75 cycles at which time $x^2 = 6.25$. x^2 will then change at its maximum rate (.1/cycle) until it reaches the final value of 25. The total settling time is 75 + 10(25 - 6.25) = 263 cycles.

e. COEFFICIENT RELATIONSHIPS. - This section deals with the algebraic equations and methods for computing scale factors and coefficients for the various operations.

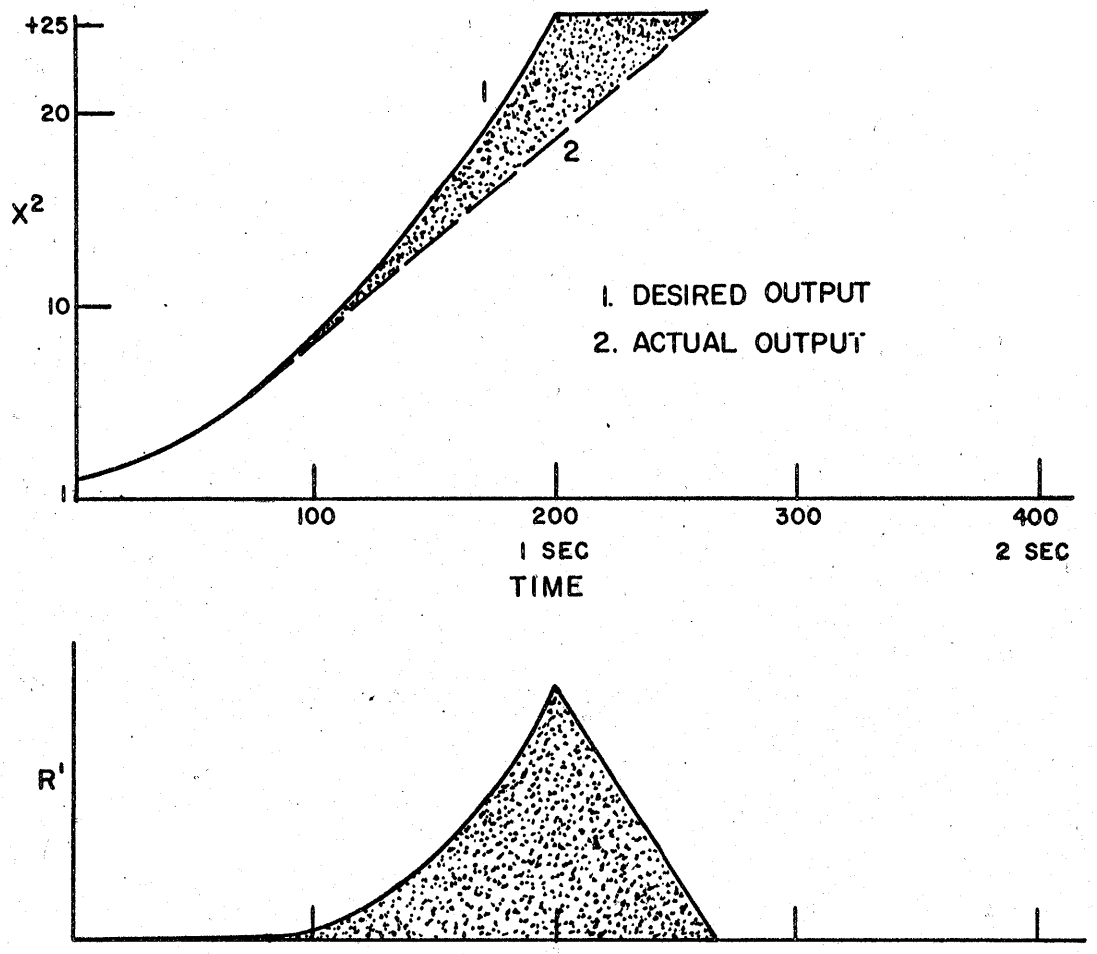


Figure 1. Settling For x^2

PX 56-4

(1) CONVENTIONS. - Whenever a function f is represented in the computer it has two functions associated with it: the function F and the constant, or coefficient, C_f. These three functions are related by the equation

$$F = C_f \cdot f \tag{1}$$

F is the computer representation of f and is defined as the number of increments representing f. The coefficient, C_f, relating the two has units of increments per unit. As an example, if a distance x is represented in the computer by 10 increments per foot, then when x = 7 ft., X = 70 increments. $C_x = 10$. It should be emphasized here that the basic algorism and all equations associated with it deal with the function F, not f.

(2) BASIC ALGORISM AND ASSOCIATED EQUATIONS. - In the addition operation the increments ΔU and ΔV are programmed to be zero so that U and V remain at their initial values U_o and V_o and act as scale factors. It is not the actual values of U, V, and S that are important, but rather their ratios. This is fairly evident from the basic addition algorism 2.

$$W = \frac{U_o Q + V_o T}{S} + P = \frac{U_o}{S} Q + \frac{V_o}{S} T + P \tag{2}$$

To obtain the coefficient relationships begin with the general equation 3.

$$w = kq + mt + p \tag{3}$$

By definition

$$P = C_p p,\ U = C_u u,\ V = C_v v,\ W = C_w w,\ Q = C_q q,\ \text{and}\ T = C_t t \tag{4}$$

Substituting equation 4 in equation 3

$$\frac{W}{C_w} = k \frac{Q}{C_q} + m \frac{T}{C_t} + \frac{P}{C_p}$$

or

$$W = k \frac{C_w}{C_q} Q + m \frac{C_w}{C_t} T + \frac{C_w}{C_p} P \tag{5}$$

From equation 2 and equation 5

$$\frac{U_o}{S} = k \frac{C_w}{C_q} \qquad \frac{V_o}{S} = m \frac{C_w}{C_t},\ \text{and}\ C_w = C_p \tag{6}$$

or $$C_w = \frac{U_o}{S}\frac{C_q}{k} = \frac{V_o}{S}\frac{C_t}{m} = C_p$$

or $$S = \frac{U_o}{k}\frac{C_q}{C_w} = \frac{V_o}{m}\frac{C_t}{C_w}$$

The usual procedure for picking U_o, V_o, and S is to pick the smallest integers yielding the desired accuracy for the ratio. Small integers are picked to keep the remainder as low as possible. As for all direct operations, the P-term must have the same coefficient as W.

Product

Taking the multiplication algorism equation 7 and substituting equation 4 into the general equation for multiplication we obtain

$$W \doteq \frac{UV}{S} + P \tag{7}$$

$$w = kuv + p \tag{8}$$

$$\frac{W}{C_w} = k\,\frac{U}{C_u}\,\frac{V}{C_v} + \frac{P}{C_p}$$

$$W = \frac{k\,C_w}{C_u C_v}\,UV + \frac{C_w}{C_p}\,P$$

let $$C_w = C_p$$

$$S = \frac{C_u C_v}{k\,C_w} \tag{9}$$

or $$C_w = C_p = \frac{C_u C_v}{k\,S}$$

Quotient

Proceeding as for multiplication

$$U = \frac{S(W - P)}{V} \tag{10}$$

$$u \doteq \frac{k(w - p)}{v} \tag{11}$$

$$\frac{U}{C_u} = k\left(\frac{W}{C_w} - \frac{P}{C_p}\right) \Big/ \left(\frac{V}{C_v}\right)$$

let $$C_w = C_p$$

$$U = \frac{k\ C_u C_v}{C_w}\left(\frac{W - P}{V}\right),\quad S = \frac{k\ C_u C_v}{C_w},\quad C_u = \frac{S\ C_w}{k\ C_u C_v} \tag{12}$$

Square Root

$$u = k\sqrt{w - p},\quad u^2 = k^2(w - p) \tag{13}$$

Substituting equation 4 in equation 13

$$\frac{U^2}{C_u{}^2} = k^2\left(\frac{W}{C_w} - \frac{P}{C_p}\right)$$

let

$$C_w = C_p$$

$$\frac{U^2}{C_u{}^2} = k^2\left(\frac{W - P}{C_w}\right) \tag{14}$$

from the square root algorism

$$U^2 = S(W - P) \tag{15}$$

from equation 14 and equation 15

$$S = \frac{k^2 C_u{}^2}{C_w} = \frac{(k\ C_u)^2}{C_w},\quad C_u = \frac{\sqrt{S\ C_w}}{k} \tag{16}$$

Integration - Trapezoidal

$$W = \frac{2}{S}\int U dT \tag{17}$$

$$w = k\int u dt \tag{18}$$

Substituting equation 4 in equation 18

$$\frac{W}{C_w} = k\int \frac{U}{C_u}\, d\,\frac{T}{C_t} = \frac{k}{C_u C_t}\int U dT$$

or $$W = \frac{k\ C_w}{C_u C_t}\int U dT \tag{19}$$

from equation 17 and equation 19

$$\frac{2}{S} = \frac{k\ C_w}{C_u C_t},\quad S = \frac{2C_u C_t}{k\ C_w},\quad C_w = \frac{2C_u C_t}{k\ S} \tag{20}$$

Integration - Point Slope

$$W = \frac{1}{S}\int U dT \tag{21}$$

PX 56-4

Using equation 21 in place of equation 17

$$\frac{1}{S} = \frac{k\ C_w}{C_u C_t},\quad S = \frac{C_u C_t}{k\ C_w},\quad C_w = \frac{C_u C_t}{k\ S} \tag{22}$$

Integration - Reciprocal Integrand

$$Q = \frac{S}{2} \int \frac{dW}{U} \tag{23}$$

$$q = k \int \frac{dw}{u} \tag{24}$$

Substituting equation 4 in equation 24

$$\frac{Q}{C_q} = k \int \frac{dW/C_w}{U/C_u} = \frac{k\ C_u}{C_w} \int \frac{dW}{U}$$

or $$Q = \frac{k\ C_q C_u}{C_w} \int \frac{dW}{U} \tag{25}$$

From equations 23 and 25

$$\frac{S}{2} = \frac{k\ C_q C_u}{C_w},\quad S = \frac{2k\ C_q C_u}{C_w},\quad C_q = \frac{S\ C_w}{2k\ C_u} \tag{26}$$

Differentiation

Using the formula without the stabilizing term

$$U = \frac{S}{2} \frac{dW}{dT} \tag{27}$$

$$u = k \frac{dw}{dt} \tag{28}$$

Substituting equation 4 in equation 28

$$\frac{U}{C_u} = k \frac{dW/C_w}{dt/C_t} = \frac{k\ C_t}{C_w} \frac{dW}{dT} \tag{29}$$

$$U = \frac{k\ C_u C_t}{C_w} \frac{dW}{dT}$$

From equations 27 and 29

$$\frac{S}{2} = \frac{k\ C_u C_t}{C_w},\quad S = \frac{2k\ C_u C_t}{C_w},\quad C_w = \frac{2k\ C_u C_t}{S} \tag{30}$$

Exponential

$$W = e^{2T/S} \tag{31}$$

This is obtained from

$$kw = \int kudt \qquad \text{where } w = u = e^t \tag{32}$$

Substituting equation 3 in equation 32

$$\frac{W}{C_w} = \int \frac{U}{C_u}\frac{dT}{C_T} = \frac{1}{C_uC_T}\int UdT \tag{33}$$

In order to have $\Delta W = \Delta U$, $C_w = C_u$, therefore $W = U$

$$W = \frac{1}{C_T}\int UdT \tag{34}$$

From equations 17 and 34

$$\frac{2}{S} = \frac{1}{C_t},\ S = 2C_t,\ C_t = \frac{S}{2} \tag{35}$$

From the above it should be noted that C_w is independent of C_t, the only requirement for consistency being that $C_w = C_u$. The coefficient of w is determined from the relationship

$$W = C_w\, kw \tag{36}$$

Logarithm

$$T = \frac{S}{2}\log_e W \tag{37}$$

The logarithm is generated as the inverse of the exponential, namely, from the equation

$$w = \int ud(kt) \tag{38}$$

where $w = u = x$ and $kt = \log_e x$ (39)

substituting equation 4 in equation 38

$$\frac{W}{C_w} = k\int \frac{U}{C_u}\, d\,\frac{T}{C_t} = \frac{k}{C_uC_t}\int UdT \tag{40}$$

dividing by equation 17

$$\frac{2}{S} = \frac{k\,C_w}{C_uC_t} \tag{41}$$

But as $C_u = C_w$ as with the exponential

$$\frac{2}{CS} = \frac{k}{C_t},\ C_t = \frac{k\,S}{2},\ S = \frac{2C_t}{k} \tag{42}$$

Once again C_t is independent of C_w or C_u and is determined only by the scaling.

PX56-4

f. INITIAL VALUES

(1) Introduction. - At the time when incremental computation is begun, all U and V registers must be set to a consistent set of values, which are called initial values. This section deals with the theory of computing these register values once the initial point has been determined. Re-examination of the basic algorism reveals that there is one term (usually $S\Delta W$) which is affected by the previous sign of the remainder. At the beginning, however, there should be no previous sign of the remainder to consider; therefore, the initial values should be set so as to cancel out any effects of this term for the first cycle.

The following helps to explain the effect of this term. Assume all independent increments are zero. The increment for the dependent variable (determined on the previous cycle), is defined as plus one for the first cycle. This increment multiplied by some register quantity is added into the R register, causing a second increment to be generated opposite in sign to the first. This second increment on the next cycle will counteract the effects of the first increment, thus leaving the R-line at or near zero. All steps following this one will have a -1 for the result of this step, whereas it will be first a +1 and then a -1 for all steps preceding it. In order to compensate for this difference the registers of all steps following the step are set to an initial value one higher than normal. The initial output for these steps is calculated using this higher value. After the first cycle, compensation occurs and the output is equal to the unraised value.

(2) COMPARISON. - V is updated prior to comparison. During the first cycle $\Delta V = +1$ so that $(V_0 + 1)$ is compared. For this reason the V register should be set to a value one less than the desired initial value.

If the register values in successive steps are each raised by an increment as specified anove, the values in the final step could be considerably different from the values based on the unraised functions. After the computer is turned on these steps will settle to the unraised values as expected, but only after several cycles, as the raised value is several increments different from the unraised value. This difference can be minimized by choosing the sign of the functions generated in the various steps in such a way as to have successive offsets compensate for each other. For example, instead of generating x, x^2, and $x^2 + y$, one could generate x, $-x^2$, and $x^2 + y$. In this case the second step would be said to be <u>flipped</u> (from $+ x^2$ to $- x^2$). Flipping is most conveniently done by changing the signs of increments in such a way as to leave the register values unchanged.

For arithmetic operations, flipping does not affect the final answer, but merely eliminates the initial delay described above. In the case of integration and related operations, however, this initial delay can cause integration error which is permanent. For this reason flipping should be used in steps affecting an integral. In other cases the procedure is desirable though not necessarily worth the effort.

g. METHODS

(1) INTRODUCTION. - The theory of program preparation has been dealt with above. The following is a presentation of the techniques and conventions which have been used in the task of program preparation. While no claim is made that this system is the best one, it has worked satisfactorily and is the result of modification of several prior systems with which trouble was experienced over the past months.

The results of the preparation routine are expressed in the form of three tabulations: (1) an algebraic listing of the dependent variables in each step

accompanied by a sign column, step number, scale factor, and coefficient; (2) a program tabulation giving the signs and addresses of each of the increments in each step, and (3) a constants tabulation listing the values of U_o, V_o, and S for each step. Tabulation 1 is essentially for checking and is the basis for tabulations 2 and 3.

The first step of programming is to determine a set of intermediate functions to be used in computing the desired outputs. These are arranged in sequence in accordance with the principles described in the section on sequencing and listed on Tabulation 1, the sign column being left blank. On Tabulation 1, the quantities appearing on the V-line are singly underlined. Those steps which have inputs or outputs have the V-line quantity doubly underlined.

The scaling operation is then begun, using whichever method is desired. Work in general is done consecutively from step to step in sequence. By this time the initial point has usually been decided upon so that settling time can be taken into consideration. If the coefficients are being primarily determined rather than the scale factors, then the scale factors must be calculated, too, rounded off to the nearest integer, and used as a basis for calculating exact coefficients. In this way the round-off error for the scale factors can usually be eliminated. It often pays to go first through the program and set approximate scale factors or coefficients, performing the exact calculation later. In the case of conservative scaling it is often desirable to calculate W_{max} in a step and use this value as a basis for scaling succeeding steps. This is particularly convenient in cases where the functions in many steps maximize under the same conditions.

The next operation is to calculate the initial values. Presumably by this time the initial point has been determined from input and settling considerations. When flipping is to be done, it is desirable at this time to calculate the

complete set of scaled initial functions and to tabulate them. Then, when the initial values are calculated, the sign of the dependent variable is taken in such a way as to make the values of the U and V registers holding this function as close to the calculated value as possible. As an example, consider the function x^2 where $x_0 = 5$, $C_x = 100$, and $C_{x2} = 10$. The input to step is $Cx_0 + 1 = 501$, and the output is $\frac{(501)^2}{200} = 251$. The step is then flipped to give $- x^2$ so that $250 = 251 - 1$ rather than 252 may be entered into the U or V register of the succeeding step using x^2. Whenever a step is flipped a minus sign is entered in the sign column of Tabulation 1. Initial values are most conveniently calculated using the scaled values of the functions as shown above. For, using the scaled values it is possible to go from one step to the next, using only the previous values along with the scale factors in the computations. If these scaled values are checked with the initial function values mentioned above, a double-check on the scaling operation is provided. For convenience, the flipping of a step is done in such a way as to leave the quantities in the U and V registers for the step unaltered. Flipping a direct operation consists of reversing the sign of ΔP, ΔQ, and ΔT. Flipping an inverse operation consists of reversing the sign of ΔW and the dependent increment.

(2) SIGN CONVENTION. - The dependent variable of a step is defined as the function listed on Tabulation 1, neglecting the sign column. It is this value which appears in subsequent U and V registers. The function sgn R multiplied by the sign in the sign column is taken as a positive increment. Inverse operations, i.e., division, square root, differentiation, and logarithm, are considered to have a negative sign if $\Delta f = -$ sgn R, which is the convention used in the section on theory.

In setting up the convention it was attempted to have a system whereby flipping could be done with a minimum of after affects. Too, it was desired to have a rigid convention in which errors were readily apparent.

Again it should be stated that though various operations are stated separately, the programmer must take all into account simultaneously to do the best job of scaling. Usually the operations are repeated, at least to some extent, in an attempt to converge to the best possible program.

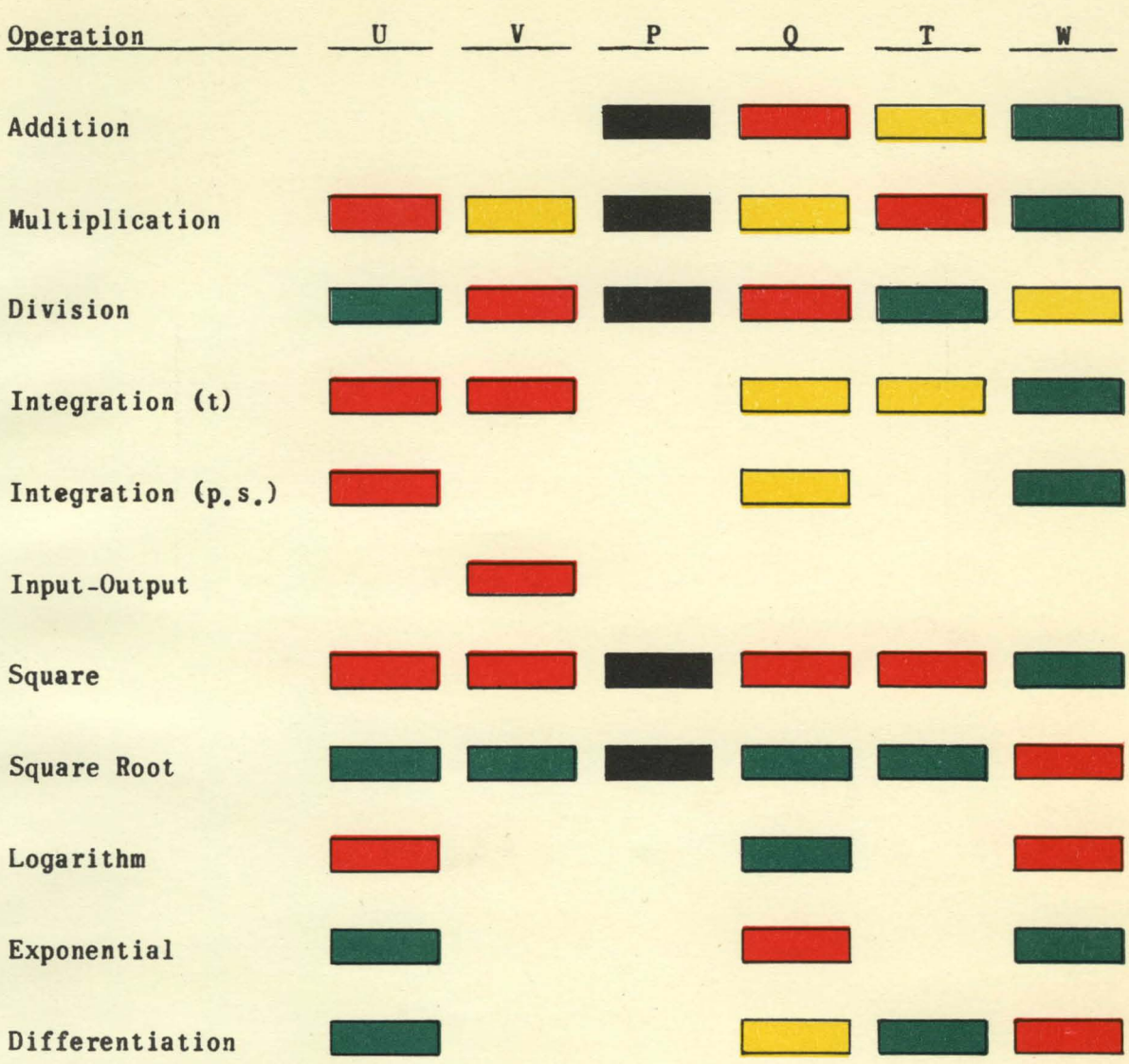
h. CHECKING OF COMPUTED PROGRAM

(1) INTRODUCTION. - Tabulation 2 provides a convenient intermediate form for a program. This is primarily because a punched tape of the tabulation can be fed into the Univac Scientific Computer for simulation. This form also has the advantage that it is convenient for checking errors. Due to the standard form and convention adopted a great many of the common clerical errors can be either avoided or else quickly spotted. The checks which are applied are 1) address checks, and 2) sign checks. A careful check of Tabulation 2 can almost eliminate programming errors. In the checking operations involving either Tabulation 2 or 3, Tabulation 1 is used as the standard.

The following are the checks which are applied to Tabulation 2. The incremental addresses are checked against Tabulation 1 to determine: 1) whether the right addresses were used, and 2) whether they were addressed to the appropriate increments. Knowledge of the normal layout of the operations (see Figure 2) makes the latter simple to perform.

Due to the convention used, the signs of the register values are always for the positive function (that is, for +f, even though f may be negative). They are unaffected by flipping the step in which they occur. The signs of ΔU and ΔV may therefore be checked to see that they generate + U and + V. The signs of the other increments may then be checked to see that they generate a function with the same sign as specified in the sign column of Tabulation 1.

As the U and V registers contain the operands as listed on Tabulation 1, they may be checked against the initial value tabulations. A fairly convenient



NOTE: Each color stands for one incremental address. The dependent increment is green.

PX 56-4

Figure 2. Tabulation No. 2 Check

check on the initial values is obtained by calculating them first using the unscaled inputs and the set of equations and then recalculating them just on the basis of the U, V, and S register values alone using the algorisms.

(2) SIMULATION. - The above checks allow one to be reasonably certain as to the correctness of his program. The next step is usually simulation on the Univac Scientific Computer. This is convenient at this time because Tabulations 2 and 3 may be punched on paper tape and fed directly into the computer once the SIMIC (SIMulation, Incremental Computer) control tape has been run in. While the SIMIC routine allows the use of only static points as inputs, it, nevertheless, has been very successful for troubleshooting purposes.

The usual procedure is to pick a static point at which all of the inputs have changed. The point is run until all of the computer values are expected to have settled, with periodic dumps taken during the simulation as well as at the end. The values for the computer functions are calculated for this second point and compared with the values obtained with the dumps. The comparison shows up any program errors which have not been found earlier. If errors are found, it is possible to check for errors in steps following the erroneous one by determining whether or not the later values are consistent with the first error. Once the program has been checked out in this manner it is possible to run any other simulation which is desired, including one with dynamic inputs (by means of the DYSIMIC and POLYSIMIC routines). When the program has proved satisfactory it may be coded up for the incremental computer itself.

i. SIMULATION

(1) INTRODUCTION. - There are three simulation routines which may be used: SIMIC, POLYSIMIC, and DYSIMIC. The difference between the three involves (1) different incremental input procedures, and (2) different monitoring features. In the SIMIC program the inputs (equivalent to the analog inputs for the real

incremental computer) remain constant unless programmed to a different value. While this limited type of input is very useful for eliminating program errors, it does not permit simulation under dynamic input conditions which normally occur in practice. In some cases the incremental program itself may be used to provide the dynamic inputs. This is done by incrementally programming functions to provide the desired inputs. A POLYSIMIC program is a SIMIC program in which incrementally-computed polynomials in time are used for inputs. The third possibility is DYSIMIC. This routine, independent of SIMIC, takes values for the inputs, tabulated for regular intervals over the proposed simulation period and uses cubic interpolation to provide values to the computer every major cycle. Independent computer routines may be used to provide the input tabulations if desired.

(2) SIMIC. - The SIMIC input program is composed of several sections, each introduced by a code word. These code words prepare the computer to act appropriately upon the data in the section. The sequence of these sections in an actual program is important only in that the data conditions for any desired simulation must be given before the COMPUTE command is given. The program is all in Flex-code.

(3) PROGRAM. - The section is introduced by the word PROGRAM preceded by a shift up following a carriage return. It contains all of the incremental addresses from Tabulation 2. The information consists of one line for each step of the program, each line being composed of a 3-digit octal step number followed by six signed 3-digit octal addresses. The computer ignores all material following the code word until it reads in a carriage return followed by 3 octal digits. During simulation the sign of the remainder of each step is considered to be stored in a register bearing the step number as an address. The addresses, in order from left to right, are: ΔU, ΔV, ΔP, ΔQ, ΔT, and ΔW. Any increment

may be programmed zero by using "n" as an address. The sequence of steps within the program, constants, and input sections is immaterial; the step numbers determine sequence. The results of a comparison in a step are considered to be stored in a register with the same units and tens digit as the step, but with 3 for the hundreds digit. There can be comparisons only up through step 077. There must be a line beginning with the number of the last step in the program plus one followed by the words "end of program". This causes the computer to begin the next major cycle. Finally, there must be a line beginning with the number 400 followed by six unsigned 3-digit octal step numbers. The presence of a step number causes the V-line of the step to be printed out. The addition of 300 (octal) to the step number causes the R-line of the step to be printed out. The program may contain up to 277 (octal) steps. A stop code (anywhere in the program section) signifies the end of the section and causes the computer to begin looking for the next code word. However, a carriage return at the end of each line (including the last) is necessary to cause that line to be stored.

(4) CONSTANTS. - This section, introduced by the sequence: carriage return, shift up, CONSTANTS, contains all of the initial values and scale factors from Tabulation 3. There is one line for each step. Each line begins with a step number followed by 3 signed 6-digit decimal numbers: U_o, V_o, and S, in that order. Each number, though integral, may be punctuated by a period between any digits. The section is ended by a stop code. Again, each line (including the last) must be followed by a carriage return.

(5) OUTPUT HEADING. - This section is introduced by the sequence: OUTPUT, space, HEADING, carriage return. All material following the carriage return is stored character by character. It is punched out before each section of results. A stop code following a carriage return signifies the end of this material. It is limited to 376 (octal) characters, including spaces, shifts up or

down, punctuation, etc. The output heading may be eliminated by programming an output heading with no characters in it.

(6) INPUT. - This section is introduced by the sequence: INPUT. All material is ignored until a carriage return followed by a step number is read. Each line consists of a step number followed by signed, 6-digit decimal number with a period between the third and fourth digits. There must be an input line for each step containing a comparison. The section ends with a stop code following a carriage return.

(7) CONTROL. - The above sections are used for loading information into the computer. The control of the simulation is essentially performed by the words "type (one space) spacing", "compute", "dump", and "restore". The word "type spacing", preceded by a carriage return, is followed by a signed, 6-digit, decimal number with a period between the third and fourth digits. During the simulation, this number gives the number of major cycles between print outs (actually punched). The word "cycles" may optionally be added after the number; it has no effect. The number, once set, remains the same until changed. The word "compute", occurs with the same format as "type spacing". The number in this case, however, specifies the number of major cycles to be simulated; it is normally a multiple of the "type spacing" number. The "compute" command must be repeated for each computation desired.

A "dump" is a print-out of all the simulated registers in a program. It may be called for at any time by inserting the word "dump", preceded by a carriage return and followed by a space. This instruction by itself will print out the contents of all the R-lines. If the space is followed by a "V", all of the V- and R-lines will be printed out. If the space is followed by a "U", all of the U-, V-, and R-lines will be printed out. This print-out, as is the normal print-out, is in flex-coded decimal. A special tape may be added to the

SIMIC master program which causes octal rather than decimal print-out. The code symbol " = " will cause six inches of leader to be punched. This is often useful for identifying the various sections of a tape.

Often it is desired to simulate several programs in succession without reloading the master tape; the "restore" command allows this to be done by setting all of the R-lines to zero and by setting all of the increments to their initial value, which is plus one. Restoration is caused by the word "restore" preceded by a carriage return and followed by a carriage return and then a stop code.

As mentioned above, stop codes signify the end of certain sections (any section of variable length) and cause the computer to search for the next code word. Any uncalled for stop codes will cause the computer to stop and may be used for this purpose. The computer may be started again by pushing the START button.

(8) COMPUTATION TIME. - As would be expected, the computation time is a function of: (1) number of steps, (2) number of major cycles, and (3) number of print-outs. As a rough guide, the computation rate may be taken as four cycles per second for a one-hundred octal-step program. Print-outs require approximately one second per line. As this program operates almost exclusively off the drum, it is essential that an 8-interlace be used for most rapid computation. However, there is little loss in speed when a 16-interlace is used. Use of 4-interlace will increase the computation time prohibitively.

(9) DYSIMIC. - For convenience, DYSIMIC has been designed to use essentially the same program and constants sections as SIMIC. It is more restricted with respect to format, however. The computer recognizes the "PROGRA" of "PROGRAM", and then ignores all else until it reads a carriage return followed by three zeros. It then takes in the following information, ignoring step numbers. It assumes the steps to be consecutive and increasing. This process

stops on the recognition of the "e" from "end of program". Search is then begun for the 400-line which may contain the addresses of up to six V addresses, i.e., step numbers. The following carriage return triggers the search for the 500-line into which the control section is condensed. The "500" is followed by four octal number in order: (1) total major cycles, (2) major cycles between input references [note: (2) = (1) for no references], (3) major cycles between print-outs, and (4) steps per major cycle [(4) = 76 (octal)].

The carriage return following the 500-line triggers the search for the "CONSTA" of "CONSTANTS". After a carriage return followed by three zeros the constants are stored. Once again the step numbers are ignored and the steps are assumed to be in consecutive, increasing order. U_0, V_0, and S must be present in that order. The numbers are signed, 6-digit and decimal. The period used in SIMIC is ignored and thus need not be present. The "c" from "end of constants" is sensed and stops the computer, ready for the actual simulation.

j. DEMONSTRATION PROGRAM. - An example is included below as an illustration of programming. In it, the functions ΔA and ΔE are computed. (The Δ here does not indicate an increment.) In the following pages the equations, program, and sample calculations are given. In addition, a SIMIC input tape is reproduced.

This program uses double point-slope integration to generate sine and cosine functions. The use of this type of integration leaves the V-line free for input/output. The initial values of the inputs were chosen and the corresponding T and R_f calculated. The values used are:

$$R = 5400 \text{ ft.}, \quad -\dot{R} = + 400 \text{ ft./sec.},$$

$$(A - \pi) = E = \dot{E} = \dot{\tau} = 0$$

$$a = .94, \quad \rho_0 = .6, \quad W = 800 \text{ ft./sec.}, \quad (V_0 - 1800) = 451 \text{ ft./sec.}$$

for these: $T = 2.3264$ sec.

$R_f = 4469.4$ ft.

$\Delta E = 7.3$ milliradians

$\Delta A = 0$

TABLE NO. 1 - EQUATIONS

$$\bar{V} = V_o - 10^{-4} \rho_o R_f (W \cos A \cos E + 2030)$$

$$R_f = \dot{R} + \dot{R}T + \frac{R^2T^2 (\dot{E}^2 + \dot{\tau}^2)}{2(R + \dot{R}T)}$$

$$T = R_f/\bar{V}$$

$$Q = 14T^2$$

$$\Delta A = \frac{\dot{\tau} T}{\cos E} + \frac{(.060\ a + .133)\ \rho_o W T \sin A}{\bar{V}} + \frac{W \cos A \cos E \sin E}{4.4 \times 10^4}$$

$$\Delta E = \dot{E}T + \frac{Q \cos E}{R_f} + \frac{(.060\ a + .133)\ \rho_o W T \sin E \cos A}{\bar{V}} - \frac{W \sin A}{5.58 \times 10^4}$$

PX 56-4

TABLE NO. 2 - INPUTS

Input	Description	Unit	Coefficient	Range Minimum	Range Maximum
R	sight range	ft	.16667	0	6000
$(A - \pi)$	sight azimuth	rad	500	-1	+1
E	sight elevation	rad	750	$-\frac{2}{3}$	$+\frac{2}{3}$
$-\dot{R}$	range rate	ft/sec	1.25	0	800
τ *	traverse rate	rad/sec	1750.	-.3	+.3
$\dot{E}$	elevation rate	rad/sec	1750.	-.3	-.3
W	gun platform velocity	ft/sec	1.	0	1000
$(V_0 = 1800)$	muzzle velocity	ft/sec	1.	1800	2800
ρ_0	relative air density	---	1000	0	1.0
(a - .7833)	relative air temperature	---	4615	.5666	1.0000

* $\dot{\tau} = \dot{A} \cos E$

PX 56-4

TABLE NO. 3 - TABULATION NO. 1

Step	Sign	Function	Inputs
000	+	RT	$\underline{\underline{R}}$
001	+	$\underline{\sin E}\ \cos A$	
002	+	$R + \underline{\dot{R}T}$	$\underline{\underline{-\dot{R}}}$
003	+	$\underline{R^2T^2}$	
004	+	$\underline{\dot{E}^2}$	
005	+	$Q = 14\underline{T^2}$	
006	+	$\dot{E}^2 + \underline{\underline{\dot{\tau}}}^2$	$\underline{\underline{\dot{\tau}}}$
007	+	$\cos A\ \underline{\cos E}$	
010	+	$(\underline{\underline{a}} - .7833)\ T$	$\underline{\underline{a}}$
011	+	$\underline{Q \cos E/R_f}$	
012	+	$\underline{\underline{\rho}}_T\ (.060\ a + .133)$	ρ
013	+	$\underline{R^2T^2}\ (\dot{\tau}^2 + \dot{E}^2)$	
014	+	$\rho\ \underline{\underline{TW}}\ (.060\ a + .133)$	W
015	+	$\underline{R^2T^2\ (\dot{\tau}^2 + \dot{E}^2)/2(R + \dot{R}T)}$	
016	+	$\underline{W \cos A}\ \cos E$	
017	+	$R_f = R + \dot{R}T + \dfrac{R^2T^2(\dot{\tau}^2 + \dot{E}^2)}{2(R + \dot{R}T)}$	$\underline{\underline{E}}$
020	+	$\dot{\tau}\ \underline{T}$	
021	+	$\sin E = \int \cos E dE$	$\underline{\underline{(V_o - 1800)}}$
022	+	$\underline{R_f}\ (W \cos A \cos E + 2030)$	
023	+	$\underline{\dot{\tau}\ T/\cos E}$	
024	+	$\underline{\underline{\dot{E}}T} + \underline{Q \cos E/R_f}$	$\underline{\underline{\dot{E}}}$
025	+	$\bar{V} = V_o - 10^{-4}\ \rho\ \underline{R_f(W \cos A \cos E + 2030)}$	
026	+	$\cos E = -\int \sin E dE$	$\underline{\underline{(A - \pi)}}$
027	+	$\underline{\rho\ TW(.060\ a + .133)/\bar{V}}$	

Note: Single underline indicates V-operand; double underline indicates input or output.

PX 56-4

TABLE NO. 3 - TABULATION NO. 1 - (cont.)

Step	Sign	Function	Inputs
030	+	$R_f / \underline{\bar{V}} = T$	
031	+	ρ TW $\underline{\sin A}$ (.060 a + .133)/ $\bar{V}$ + $\dot{\tau}$ T/cos E = ΔA	
032	+	ρ TW $\underline{\sin E \cos A}$ (.060 a + .133)/ $\bar{V}$ + Q cos E/R_f + $\dot{E}T = \Delta E$	
033	+	$\sin A = \int \cos A dA$	$\underline{\underline{\Delta A \text{ out}}}$
034	+	$(.060\ a + .133)T = .06\ \left[(a - .7833)T + \underline{\underline{3T}}\right]$	
035	+	$\cos A = -\int \sin A dA$	$\underline{\underline{\Delta E \text{ out}}}$
036	+	$\underline{Q} \cos E$	

PX 56-4

Note: Single underline indicates V-operand, double underline indicates input or output.

TABLE NO. 4 - SAMPLE SCALING CALCULATIONS

$W \cos A \cos E$

$$S = \frac{(800.89)(1.)}{(1.)} \equiv 800$$

$$C = \frac{800.84}{800} = 1.00105$$

$\dot{\tau} T$

$$S = \frac{(1750)(300)}{(2000)} \equiv 262$$

$$C = 2003.8$$

$R_f / \bar{V} = T$

$$S = \frac{(300)(1.)}{(1/6)} \equiv 1800$$

$$C = (1800)(1/6) = 300.00$$

$R_f (W \cos A \cos E + 2030)$

$$C = \frac{(1.00105)(1/6)}{S \equiv (2000)} = 83.4208 \times 10^{-6}$$

$(\dot{E})(T) + Q \cos E/R_f$

$$S = \frac{(1750)(300)}{2000} \equiv 262$$

$$C = \frac{(1750)(300)}{(262)} = 2003.8$$

$\dot{\tau} T/\cos E$

$$S = \frac{(1214.90)(500)}{(2003.8)} \equiv 303$$

$$C = \frac{(303)(2003.8)}{(1214.90)} = 499.75$$

PX 56-4

TABLE NO. 5 - SAMPLE INITIAL VALUE CALCULATIONS

#00 RT

$$W = \frac{(699)(901)}{(1500)} + 1 = 421$$

$$= 420$$

#01 sin E cos A

$$W = \frac{(-594)(+1)}{(892)} + 1 = 0$$

$$= -1$$

#02 $R + \dot{R}T$

$$W = \frac{(699)(-499)}{(2250)} + 900 + 1 = 746$$

$$= -155$$

#03 R^2T^2

$$W = \frac{(421)^2}{667} + 1 = 267$$

$$= 266$$

#04 $\dot{E}^2$

$$W = \frac{(+1)^2}{(875)} + 1 = +1$$

$$= 0$$

#05 $Q = 14T^2$

$$W = \frac{(699)^2}{(3214)} + 1 = 153$$

$$= 152$$

PX 56-4

TABLE NO. 6 - INCREMENTAL COMPUTER PROGRAM - TABULATION NO. 2

Program No. 2032-B Date: 9/10/56 Programmer: GAC

Title: Demonstration Program Page 1 of 2

Step	Δ U		Δ V		Δ P		Δ Q		Δ T		Δ W		Remarks
000	+	030	+	300		n	+	300	+	030	+	000	
001	+	035	+	021		n	+	021	+	035	+	001	
002	+	030	+	302	+	300	-	302	-	030	+	002	
003	+	000	+	000		n	+	000	+	000	+	003	
004	+	324	+	324		n	+	324	+	324	+	004	
005	+	030	+	030		n	+	030	+	030	+	005	
006	+	306	+	306	+	004	+	306	+	306	+	006	
007	+	035	+	026		n	+	026	+	035	+	007	
010	+	030	+	310		n	+	310	+	030	+	010	
011	+	017	+	011		n	-	011	-	017	-	036	
012	+	034	+	312		n	+	312	+	034	+	012	
013	+	006	+	003		n	+	003	+	006	+	013	
014	+	012	+	314		n	+	314	+	012	+	014	
015	+	002	+	015		n	-	015	-	002	-	013	
016	+	007	+	314		n	+	314	+	007	+	016	
017		n	+	317	+	002	+	015		n	+	017	
020	+	306	+	030		n	+	030	+	306	+	020	
021	+	026	+	321		n	+	317		n	+	021	
022	+	016	+	017		n	+	017	+	016	+	022	
023	+	026	+	023		n	-	023	-	026	-	020	
024	+	030	+	324	+	011	+	324	+	030	+	024	

PX 56-4

TABLE NO. 6 - INCREMENTAL COMPUTER PROGRAM - TABULATION NO. 2 (cont.)

Program No. 2032-B Date: 9/10/56 Programmer: GAC

Title: Demonstration Program Page 2 of 2

Step	Δ U		Δ V		Δ P		Δ Q		Δ T		Δ W		Remarks
025	+	312	+	022	+	321	-	022	-	312	+	025	
026	+	021	+	326		n	-	317		n	+	026	
027	+	025	+	027		n	-	027	-	025	-	014	
030	+	030	+	025		n	-	025	-	030	-	017	
031	+	027	+	033	+	023	+	033	+	027	+	031	
032	+	027	+	001	+	011	+	001	+	027	+	032	
033	+	035	+	031		n	+	326		n	+	033	
034		n		n		n	+	030	+	010	+	034	
035	+	033	P	032		n	-	326		n	+	035	
036	+	026	+	005		n	+	005	+	026	+	036	
037		end of program											

PX 56-4

TABLE NO. 7 - INCREMENTAL COMPUTER PROGRAM - TABULATION NO. 3

Program No. 2032-B Date: 9/10/56 Programmer: GAC

Title: Demonstration Program Page 1 of 2

Step		U_0		V_0		S	C	
000	+	000.699	+	000.899	+	001.500	.03333	
001	-	000.594	+	000.001	+	000.892	500.48	
002	+	000.699	+	000.499	+	002.250	.16667	
003	+	000.421	+	000.421	+	000.667	1.6666×10^{-6}	
004	+	000.001	+	000.001	+	000.875	3500	
005	+	000.699	+	000.699	+	003.214	2.0002	
006	+	000.001	-	000.001	+	000.875	3500	
007	-	000.594	+	001.216	+	000.903	800.84	
010	+	000.699	+	000.722	+	001.153	1200.78	
001	+	000.746	+	000.035	+	000.167	2003.8	
012	+	000.679	+	000.599	+	001.100	1399.5	
013	+	000.001	+	000.267	+	000.250	2.3332×10^{-5}	
014	+	000.371	+	000.799	+	000.700	2	
015	+	000.746	+	000.001	+	000.596	.16688	
016	-	000.800	+	000.801	+	000.800	1.00105	
017	+	000.001	-	000.001	+	000.001	.16667	
020	+	000.001	+	000.699	+	000.262	2003.8	
021	+	001.215	+	000.450	+	001.215	750.00	
022	+	001.230	+	000.751	+	002.000	83.421×10^{-6}	
023	+	001.216	+	000.001	+	000.303	499.75	
024	+	000.699	+	000.001	+	000.262	2003.8	

PX 56-4

TABLE NO. 7 - INCREMENTAL COMPUTER PROGRAM - TABULATION NO. 3 (cont.)

Program No. 2032-B Date: 9/10/56 Programmer: GAC

Title: Demonstration Program Page 2 of 2

Step		U_o		V_o		S	C	
025	+	000.601	+	000.460	+	000.834	1.00024	
026	+	000.001	-	000.001	+	000.463	1214.90	
027	+	001.922	+	000.283	+	001.500	3000.0	
030	+	000.699	+	001.922	+	001.800	300.00	
031	+	000.283	+	000.001	+	000.500	3000.0	
032	+	000.283	+	000.001	+	000.500	3000.0	
033	-	000.595	-	000.001	+	000.595	500.00	
034	+	000.001	+	000/012	+	000.013	1539.45	
035	+	000.001	+	000.021	+	000.420	595.24	
036	+	001.216	+	000.153	+	001.215	2.0000	

PX 56-4

TABLE NO. 8 - DEMONSTRATION PROGRAM UNIVAC SIMIC INPUT TAPE

Program No. 2032-B Date: 9/10/56 Programmer: GTU GAC

Title: Demonstration Program Page 1 of 5

Step	Δ U		Δ V		Δ P		Δ Q		Δ T		Δ W		
000	+	030	+	300		n	+	300	+	030	+	000	
001	+	035	+	021		n	+	021	+	035	+	001	
002	+	030	+	302	+	300	-	302	-	030	+	002	
003	+	000	+	000		n	+	000	+	000	+	003	
004	+	324	+	324		n	+	324	+	324	+	004	
005	+	030	+	030		n	+	030	+	030	+	005	
006	+	306	+	306	+	004	+	306	+	306	+	006	
007	+	035	+	026		n	+	026	+	035	+	007	
010	+	030	+	310		n	+	310	+	030	+	010	
011	+	017	+	011		n	-	011	-	017	-	036	
012	+	034	+	312		n	+	312	+	034	+	012	
013	+	006	+	003		n	+	003	+	006	+	013	
014	+	012	+	314		n	+	314	+	012	+	014	
015	+	002	+	015		n	-	015	-	002	-	013	
016	+	007	+	314		n	+	314	+	007	+	016	
017		n	+	317	+	002	+	015		n	+	017	
020	+	306	+	030		n	+	030	+	306	+	020	
021	+	026	+	321		n	+	317		n	+	021	
022	+	016	+	017		n	+	017	+	016	+	022	
023	+	026	+	023		n	-	023	-	026	-	020	
024	+	030	+	324	+	011	+	324	+	030	+	024	

PX 56-4

TABLE NO. 8 - DEMONSTRATION PROGRAM UNIVAC SIMIC INPUT TAPE (cont.)

Program No. 2032-B Date: 9/10/56 Programmer: GTU GAC

Title: Demonstration Program Page 2 of 5

Step	Δ U		Δ V		Δ P		Δ Q		Δ T		Δ W		
025	+	312	+	022	+	321	-	022	-	312	+	025	
026	+	021	+	326		n	-	317		n	+	026	
027	+	025	+	027		n	-	027	-	025	-	014	
030	+	030	+	025		n	-	025	-	030	-	017	
031	+	027	+	033	+	023	+	033	+	027	+	031	
032	+	027	+	001	+	011	+	001	+	027	+	032	
033	+	035	+	031		n	+	326		n	+	033	
034		n		n		n	+	030	+	010	+	034	
035	+	033	+	032		n	-	325		n	+	035	
036	+	026	+	005		n	+	005	+	026	+	036	
037		end of program											
400		033		035		020		030		022		036	
(STOP)													

PX 56-4

TABLE NO. 8 - DEMONSTRATION PROGRAM UNIVAC SIMIC INPUT TAPE (cont.)

Program No. 2032-B Date: 9/10/56 Programmer: GTU GAC

Title: Demonstration Program Page 3 of 5

CONSTANTS

Step		U_0		V_0		S	
000	+	000.699	+	000.899	+	001.500	
001	-	000.594	+	000.001	+	000.892	
002	+	000.699	+	000.499	+	002.250	
003	+	000.421	+	000.421	+	000.667	
004	+	000.001	+	000.001	+	000.875	
005	+	000.699	+	000.699	+	003.214	
006	+	000.001	-	000.001	+	000.875	
007	-	000.594	+	001.216	+	000.903	
010	+	000.699	+	000.722	+	001.153	
011	+	000.746	+	000.035	+	000.167	
012	+	000.679	+	000.599	+	001.100	
013	+	000.001	+	000.267	+	000.250	
014	+	000.371	+	000.799	+	000.700	
015	+	000.746	+	000.001	+	000.596	
016	-	000.800	+	000.801	+	000.800	
017	+	000.001	-	000.001	+	000.001	
020	+	000.001	+	000.699	+	000.262	
021	+	001.215	+	000.450	+	001.215	
022	+	001.230	+	000.751	+	002.000	
023	+	001.216	+	000.001	+	000.303	
024	+	000.699	+	000.001	+	000.262	

PX 56-4

TABLE NO. 8 - DEMONSTRATION PROGRAM UNIVAC SIMIC INPUT TAPE (cont.)

Program No. 2032-B Date: 9/10/56 Programmer: GTU GAC

Title: Demonstration Program Page 4 of 5

CONSTANTS

Step		U_0		V_0		S	
025	+	000.601	+	000.460	+	000.834	
026	+	000.001	-	000.001	+	000.463	
027	+	001.922	+	000.283	+	001.500	
030	+	000.699	+	001.922	+	001.800	
031	+	000.283	+	000.001	+	000.500	
032	+	000.283	+	000.001	+	000.500	
033	-	000.595	-	000.001	+	000.595	
034	+	000.001	+	000.012	+	000.013	
035	+	000.001	+	000.021	+	000.420	
036	+	001.216	+	000.153	+	001.215	
(STOP)							

PX 56-4

TABLE NO. 8 - DEMONSTRATION PROGRAM UNIVAC SIMIC INPUT TAPE (cont.)

Program No. 2032-B Date: 9/10/56 Programmer: GTU GAC

Title: Demonstration Program Page 5 of 5

INPUT

000 + 000.582

002 + 000.645

006 + 000.036

010 + 000.723

012 + 000.600

014 + 000.800

017 + 000.327

021 + 000.451

024 - 000.092

026 - 000.470

(STOP)

OUTPUT HEADING

RESULTS

Program 2032-b Demonstration Program GTU GAC 9/10/56

CYCLE dA dE T $\bar{V}$ Rf Q

(STOP)

type spacing 000.050

compute 000.800 cycles

dump u

(STOP)

PX 56-4

3. CODING

a. INTRODUCTION. - The solution to a mathematical problem is expressed as a series of equations. In order for a computer to solve a problem, it must be given instructions so that it will solve these equations. The translation from equations to machine instructions is called coding.

All of the information necessary to code for the incremental computer is contained in the command code sheet (Figure 3) and the algorisms. The following paragraphs will explain and illustrate their use.

b. PROCEDURE IN CODING. - The equations are broken down into a series of simpler equations, called steps. Each step must be capable of being handled by one of the special algorisms. For example, the multiplication algorisms is $W = \frac{UV}{S} + P$, so two quantities could be multiplied and a third added to the product in one step.

Using the special algorism, identify the variables of each step with the corresponding variable in the algorism. For example, let us compute the equation $Z = XY$. From the multiplication algorism we are given $W = \frac{UV}{S} + P$. To change our equation to look like this we can write $Z = XY + 0$. It is now plain that $P = 0$, $X = U$, $Y = V$, $Z = W$.

We have now identified the variables in our simple equation with the variables in the algorism. Using the restrictions for the multiplication algorism we have

$$\Delta Q_i = \Delta V_i, \quad \Delta T_i = \Delta U_i, \quad \Delta W_{i+1} = \begin{cases} +1 \text{ for } R_i \text{ positive} \\ -1 \text{ for } R_i \text{ negative} \end{cases}$$

So we instruct the machine to read $+\Delta Q$ and $+\Delta V$ from the address where ΔY is stored, and $+\Delta T$ and $+\Delta U$ from the address where ΔX is stored. We can make $P = 0$ by not reading it.

When R_i is positive, the output increment of the step read by ΔW is positive, and when R_i is negative the increment is negative. Therefore, if we

PX 56-4

COMMAND CODES

P_3	P_2	P_1	
0			D. S. Address
2	0	0	No Action
2	2		Initiate Comparison Non-complement . . . . .
2	3		Initiate Comparison Complement . . . . .
2	4		End pulse. . . . .
2	5		End pulse. . . . .
2	6		Initiate step . . . . .
2	7		Initiate step . . . . .
2	e	1	Read + Δ P from last indicated address . . . . .
2	e	2	Read + Δ W from last indicated address . . . . .
2	e	3	Read + Δ V from last indicated address . . . . .
2	e	4	Read + Δ Q from last indicated address . . . . .
2	e	5	Read + Δ T from last indicated address . . . . .
2	e	6	Read + Δ U from last indicated address . . . . .
2	e	7	Read + Δ W from Sign of R_{i-1} . . . . .
2	d	1	Read - Δ P from last indicated address . . . . .
2	d	2	Read - Δ W from last indicated address . . . . .
2	d	3	Read - Δ V from last indicated address . . . . .
2	d	4	Read - Δ Q from last indicated address . . . . .
2	d	5	Read - Δ T from last indicated address . . . . .
2	d	6	Read - Δ U from last indicated address . . . . .
2	d	7	Initiate cycle . . . . .
	0	$\overline{2^2}$	0 Comparator address . . . . .
	0	2^2	1
	1	$\overline{2^2}$	2
	1	2^2	3
	⋮	⋮	⋮
	7	$\overline{2^2}$	15
	7	2^2	16 Comparator address . . . . .

Figure 3. Command Codes, Incremental Computer

COMMAND CODES

P_3	P_2	P_1		COMMAND POSITION
0			D. S. Address	
2	0	0	No Action	
2	2		Initiate Comparison Non-complement	09 of x
2	3		Initiate Comparison Complement	09 of x
2	4		End pulse.	12th digit after Initiate Comparison
2	5		End pulse.	12th digit after Initiate Comparison
2	6		Initiate step	00 of x
2	7		Initiate step	00 of x
2	e	1	Read + Δ P from last indicated address	02 of x - 1 through last digit of x - 1
2	e	2	Read + Δ W from last indicated address	01 of x - 1 through 00 of x
2	e	3	Read + Δ V from last indicated address	02 of x - 1 through 00 of x
2	e	4	Read + Δ Q from last indicated address	04 of x - 1 through 02 of x
2	e	5	Read + Δ T from last indicated address	03 of x - 1 through 00 of x
2	e	6	Read + Δ U from last indicated address	03 of x - 1 through 01 of x
2	e	7	Read + Δ W from Sign of R_{i-1}	p - 20 of x - 1 where p = sum of the number of digits in x - 1 and x. $p \geq 21$.
2	d	1	Read - Δ P from last indicated address	02 of x - 1 through last digit of x - 1
2	d	2	Read - Δ W from last indicated address	01 of x - 1 through 00 of x
2	d	3	Read - Δ V from last indicated address	02 of x - 1 through 00 of x
2	d	4	Read - Δ Q from last indicated address	04 of x - 1 through 02 of x
2	d	5	Read - Δ T from last indicated address	03 of x - 1 through 00 of x
2	d	6	Read - Δ U from last indicated address	03 of x - 1 through 01 of x
2	d	7	Initiate cycle	digit period preceding 00 of x
	0	$\overline{2^2}$	0 Comparator address	digit period following Initiate Comparison
	0	2^2	1	
	1	$\overline{2^2}$	2	
	1	2^2	3	
	'	'	'	
	'	'	'	
	7	$\overline{2^2}$	15	
	7	2^2	16 Comparator address	digit period following Initiate Comparison

NOTES

1. Command Position refers to the allowable digit positions for commands for the xth step.
2. Address for storing sign of R_i . . . . . 05 of x + 1
3. Address for storing results of the comparison . . . . 4th digit following end pulse
4. e - even numbers
   d - odd numbers
5. Information from a digit storage address remains available until another D. S. address is programmed.
6. Where command codes permit, combinations of commands may be programmed simultaneously, and comparator addresses may be programmed with commands or D.S. addresses if their common parts can be made identical.
7. $\overline{2^2}$ = 0, 1, 2 or 3 in this position

   2^2 = 4, 5, 6, or 7 in this position
8. DS - Digit Storage
9. M.S.D. of the U_0 and U_1 preceding initiate cycle must be 0.
10. First two M.S.D.'s of the R_1 preceding initiate cycle must be 0.

program ΔW to be plus (+)ΔW, the required case is satisfied. To make $\Delta W = -1$ when R_i is positive, we would read $-\Delta W$. To do this in the case of multiplication would be incorrect, however, as the value for R_i would change away from rather than toward zero.

The command code sheet gives a code number for every command and the allowed digit positions for that command to be given. For example, let us read one of the variables in the above equation; take $+\Delta V$. The code number to read $+\Delta V$ is given as 2e3. The e can be any even number 0 through 6. The allowed positions are "02 of x-1 through 00 of x". The meaning of this is made clear in note No. 1. The "x" refers to the step number with which we are concerned. The 02 and 00 refer to digit positions within a step. The digit positions are numbered starting with the initiate step command as zero; this is necessarily so, since the digit position for initiate step (IS) command is 00 of x. Thus, we see that $+\Delta V$ must be read in the step before the step in which it is to be used in computation. The same is true of the other variables with minor differences in initial digit positions allowed.

After all of the variables have been read from their respective addresses, computation takes place automatically. All that remains is to store the sign of the remainder for use in the next major cycle. According to the sheet this is to be done at 05 of x + 1, the fifth digit position of the following step.

Let us illustrate with a typical equation and how it is coded.

$Z = (A/d) \log A - b$

1. Break into steps

   Step No. 1 A/d

   Step No. 2 log A

   Step No. 3 ----

   Step No. 4 (Step No. 2)(Step No. 1) - b

Each step can be generated by the use of only one special algorism. Note that step No. 3 is blank. This is because the next step after step No. 2 uses the results of step No. 2. The results of step No. 2 are not available until after they are stored in 05 of step No. 3; if read out as soon as possible in step No. 3 it will not be available for computation until step No. 4. Now let us look at the program for this equation and the step-by-step analysis of the program. This is a general format that all programs follow. Let us assume that ΔA is found at address 10, Δb at 011, Δd at 012.

	Command Code	Remarks	Digit Period
Step No. 0	2d7	Initiate cycle (IC)	
	260	Initiate step (IS) No. 0	(0)
	010	ΔA address	(1)
	212	Read - W from last indicated address	(2)
	012	Δd address	(3)
Prepare Step No. 1	203	read $+\Delta V$ from last indicated address	(4)
	214	read $-\Delta Q$ from last indicated address	(5)
	001	A/d address	(6)
	206	read $+\Delta U$ from last indicated address	(7)
	215	read $-\Delta T$ from last indicated address	(8)
	260	IS No. 1	(0)
	010	A address	(1)
	202	$+\Delta W$	(2)
	203	$+\Delta V$	(3)
Compute Step No. 1 Prepare Step No. 2	206	$+\Delta U$	(4)
	200	no action	(5)
	002	log A address	(6)
	204	$+\Delta Q$	(7)
	205	$+\Delta T$	(8)

Store Step No. 1 Compute Step No. 2 Prepare Step No. 3	260	IS No. 2	(0)
	200	no action	(1)
	200	no action	(2)
	200	no action	(3)
	200	no action	(4)
	001	store sign R_1	(5)
Store Step No. 2 Prepare Step No. 4	260	IS No. 3	(0)
	004	A/d $\left[\log A\right]$ - b address	(1)
	202	$+\Delta W$	(2)
	011	Δb	(3)
	211	$-\Delta P$	(4)
	002	Store sign R_2 (log A)	(5)
	203	$+\Delta V$	(6)
	204	$+\Delta Q$	(7)
	001	A/d	(8)
	206	$+\Delta U$	(9)
	205	$+\Delta T$	(10)
Compute Step No. 4	260	IS No. 4	(0)
	200	No action	(1)
	200	No action	(2)
	200	No action	(3)
	200	No action	(4)
Store Step No. 4	260	IS No. 5	(0)
	200		(1)
	200		(2)
	200		(3)
	200		(4)
	004		(5)

PX 56-4

Explanation

Step No. 0 2d7 Initiate cycle. This command begins operation. The d can be any odd number 1 through 7. "d" will hereafter be called 1.

digit period 00 26__ initiate step command goes at the beginning of every step. Any number can be put in the space. It will be called 0 from here on.

digit period 01, 02 In the division algorism we have $U = \frac{W - P}{VS}$. To get our equation

$P = 0$

$\Delta W = \Delta A$

$\Delta V = \Delta Q = \Delta d$

$\Delta U = \Delta T = \Delta(A/d)$ which is the answer or output of the step.

also

$$U_{i+1} = \begin{cases} +1 \text{ for } R_i \text{ positive and } V_i \text{ negative} \\ +1 \text{ for } R_i \text{ negative and } V_i \text{ positive} \\ -1 \text{ for } R_i \text{ positive and } V_i \text{ positive} \\ -1 \text{ for } R_i \text{ negative and } V_i \text{ negative} \end{cases}$$

This will be the result if $-\Delta W$, $-\Delta Q$, $-\Delta T$ are read.

According to the command code we can get $-\Delta A$ on ΔW by giving first the address of $\Delta A(10)$ and then the command "Read $-\Delta W$ from the last indicated address" in digit period (dp) 01 or after. The address 10 was therefore put in dp 01 and the command "read $-\Delta W$" in dp 02.

dp 03, 04, 05 Similarly we read $+\Delta V$ and $+\Delta Q$ from address 12 where Δd is stored. Notice it is not necessary to repeat the address since the command reads "from last indicated address".

PX 56-4

dp 06, 07, 08 + ΔU and + ΔT are the output of the step, but must be read for use in computation. The place where the output of a step is stored is arbitrary, but the customary place is in the address whose number is the same as the step number in which it is computed; in this case 01.

Step No. 1 dp 00 I.S. No. 1. Step 1 is automatically computed; step No. 2 must be prepared.

dp 01, 02, 03, 04 From the algorism we have

$$Q = \frac{1}{2}S \log W$$

$$\Delta V = \Delta U = \Delta W = \Delta A$$

$$\Delta T = \Delta Q = \Delta \log A$$

ΔA is of course at address 10, and the following commands read it out. The address of log A is 002 as given by the convention mentioned in step 00. Digit period 05 contains a no action command. If there were an address in this position the contents of the R line of step No. 0 would be stored here and would erase anything previously stored.

Step No. 2 dp 00 I.S.

dp 01, 02, 03, 04 There is nothing to prepare for step No. 3, so the no action command is given.

dp 05 The sign of the remainder of a step is stored at 05 of x + 1 according to the command code, so the remainder of step No. 1 is stored at the address given here.

Step No. 3 dp 00 I.S.

dp 01, 02 To prepare for step No. 4, the algorism gives $W = \frac{UV}{S} + P$ To form our equation

$\Delta(A/d) = \Delta U = \Delta T$, $\Delta P = -\Delta b$, $\Delta \log A = \Delta V = \Delta Q$

The output of step No. 4 is stored at 04, so this is read by $+\Delta W$ in 01, 02. We want $\Delta P = -\Delta b$, so these are read in 03, 04. The sign of R_2 is stored in dp 05. Log A is read from 02 by $+\Delta V$ and $+\Delta Q$. Notice it is not necessary to repeat the address. A/d is read from 01 where it was stored by $+\Delta U$ and $+\Delta T$.

Step No. 4 dp 00 I.S. There is nothing to be prepared or stored, and computation is automatic. The length of this step is somewhat arbitrary, but is set by factors which will be mentioned later.

Step No. 5 The result of step No. 4 is stored in 05 of step No. 5.

(1) CONSTANTS. - The U, V, and S constants are put into the computer in binary notation. A typical coding form might look like this:

Constants			Octal Program and Constants				Remarks
S	V_0	U_0	P4	P3	P2	P1	
				2	6	0	Initiate cycle
1			4	-	-	-	
0	1		2	-	-	-	
1	1	0	6	-	-	-	
0	0	0	0	-	-	-	
1	0	1	5	-	-	-	
0	1	0	2	-	-	-	
1	1	1	7				
0	0	1	1				
1	0	0	4				
		1	1				

The constants are staggered. This staggering causes S to reach the arithmetic section first, and forms $S\Delta P$ and $-S\Delta W$. Next V arrives and $V\Delta T$ is formed and added to the rest, and so on. If we call the line containing initiate cycle, 00; the least significant digit of S begins on 01, of V on 02, and of U on 03.

Then the rows of digits are taken as octal numbers and the result is put in the P4 column, as illustrated above. Of course there must be enough digit positions in a step so that the constants can be put in.

If a constant is a negative number, the 2's complement of the positive constant is used. All complemented numbers have a series of 1's for the most significant digits. These 1's should be continued up to the constants of the following step. For example

S	V	U	P4	P3	P2	P1	
1	1	1	7				
1	1	1	7				
1	1	1	7				
1	1	1	7	2	6	0	Initiate step
0	1	1	3				
1	0	1	5				
0	1	1	3				
1	0	0	4				
0	1	0	2				
1	1	1	7				
	etc.						

Thus, we see that each column is continued up to the following step.

(2) INPUT/OUTPUT. - Comparisons.

Quantities are introduced into the incremental computer by means of comparisons. In the process of comparing, the digital quantity in the computer on the V-line is converted to an analog current by means of a digital-to-analog converter. This current and the input current are run in opposition through windings of a magnetic modulator. The output of the modulator is fed to a detector which generates an increment of either + 1 or - 1 depending on the sign of the current difference. In the case of computer outputs, the detector output is fed to a holding circuit which adjusts the value of the output current so that it approaches in magnitude the current from the digital-to-analog converter.

The input circuitry is such that it can only accept currents in the range from zero to plus full scale. In order that variables which are both positive

and negative could be handled, a special command, complement comparison, was introduced. The inputs are biased externally so that zero for the variable corresponds to a half-scale current. Within the computer the highest order digit is complemented. The reason for this can be seen by considering the behavior of the highest order digit. Normally this digit is zero for numbers less than half-scale and one for numbers greater than half-scale. Using complement notation, however, the highest order digit(s) is zero for positive numbers and one for negative numbers. If we adopt the convention that positive numbers be represented by currents greater than half scale, then it can be seen that the highest order digit must be complemented.

The necessary commands for a comparison are:

P3	P2	P1	Command
2	2	-	Initiate comparison non-complement
-	-	-	comparator address of modulator where variable appears
2	4	-	end pulse
0	-	-	address for storing results of comparison

The following is a typical input step:

Step (x - 1)	2 6 0	Initiate step (x-1)	
		quantity appears on V-line	
	2 6 0	Initiate step x	0
			1
			2
			3
			4
			5
			6
			7
			8
	2 2 0	Initiate comparison	9
	0 - -	comparator address	10 1
			2
			3
			.
			.
			.

2 6 0	Initiate Step (x + 1)	:
		11
		12
2 4 0	end pulse - - - - - - - - - -	13
		14
		15
		16
0 - -	address for storing results of comparison	17

Explanation of Format.

As previously mentioned, the quantity to be compared must appear on the V-line of the step during which it is compared. Therefore, ΔV is read during the step before, i.e., in x - 1. At 09 of x, the command "comparison non-complement" (or complement as the case may be) is given. The very next digit period, dp 10 of x, the address of the modulator which holds this variable is given. The modulator addresses are given at the bottom of the command code sheet. For example, if our variable, A, were stored in modulator 3, we would give, in 10 of x, the command -14, -15, -16, or -17; they all have the same effect here according to the command code sheet. Thirteen digit periods later the end pulse is given; this stops the comparison. Four digit periods later the address at which the result of the comparison is stored is given. This is the same as the address from which ΔV was read. In our case, A was stored at 010. This gives the ΔA needed in the computations and up-dates the value of A.

Output.

Output is exactly like input except the result of the comparison is not stored. As mentioned above, the result of the comparison is fed into a detector and ultimately adjusts the output current.

At this time it is thought that there should be a minimum of approximately 35 digit periods between initiate comparison commands.

(3) COMBINING COMMANDS. - According to the command sheet, commands may be programmed either simultaneously or with comparator addresses if their common

parts can be made identical. An example of this is a situation where the twelfth digit after initiate comparison came at the digit position where "read + ΔP" is. Both commands could be given by 241.

(4) WORD LENGTH. - The number of digit positions in a step, also known as word length, must be at least as large as the largest of the following:

1) The number of commands and addresses

2) The number of digits on the R-line when it is at its maximum.

If the R-line does not overload, its maximum is about 3S (assuming $S > U$, $S > V$); if it does overload, its value will be higher. If the word length is not long enough to accommodate the R-line, some of the most significant digits are lost and large errors introduced.

(5) USE OF DOUBLE HEAD ON R-LINE. - Since there are 100 (octal) cores for storage of the output of a step or result of a comparison, the number of steps plus the number of comparisons would be limited to 100 octal (64 decimal) except for a special command. This special command allows the sign of a remainder to be read from a special track, thus freeing a core for some other use. The command necessary for this operation is "Read + ΔW from sign of R_{i-1}", and the allowable digit position is P-20 of x - 1 where P = sum of digits in x - 1 and x; $P \geq 21$.

In normal practice this special command is used as often as possible if there are more than 64 steps plus comparisons so that core storage is available as needed during coding without re-arranging the command structure.

When this special command is used practically the only limitation on the length of a program is the total number of digit positions. If the double head on the R-line is used, the maximum word length is less than or equal to 20, and the sum of any two consecutive words must be greater than or equal to 21.

(6) **STABILITY COMPARISONS.** - In order to stabilize the modulators, provision must be made in every program for zero and full scale comparisons. This is done in the following way.

Zero Comparison.

In some steps where the V-line is free, make the initial constant on the V-line equal to zero and do not program ΔV. This will keep V at zero. Then make an output comparison in the usual manner. At the time of this writing the modulator used for zero comparison is modulator 2, so the corresponding comparator's address would be put in the program.

Full Scale Comparison.

In some other step where the V-line is free, make the 10 least significant digits of the initial constant on the V-line equal 1. Then do not program ΔV and proceed as in zero comparison. The present full scale modulator is No. 1.

4. SIMULATION

In order to test the behavior of the incremental computer, and to acquire facility in programming and scaling for the incremental computer, a program which simulates the logic of the incremental computer was written for the Remington Rand Univac Division's 1103 large scale computer. This section describes this program. The commands which comprise the program are listed at the end of this section. Reference is made to the following publications, obtainable from Remington Rand Univac Division, which describe the programming of the 1103.

1) "Notes on the Logic of the ERA 1103 Computer"

2) "The ERA 1103 Computer System", PX 71920, Vol. IV, Section 6, Programming.

This simulation program was given the name SIMIC for SIMulation Incremental Computer. It requires an input tape, the preparation and format of which have been previously described. Since this input tape contains constant inputs (R, A, E, etc.) SIMIC gives a static simulation of the Incremental Computer. A simulation program to receive a new set of inputs every major cycle is being prepared and has been given the name DYSIMIC, for DYnamic SIMulation, Incremental Computer.

The SIMIC program contains room for more computation steps (minor cycles) than the Incremental Computer itself. Consequently, a program to generate variable inputs to the program may be included in SIMIC ahead of the program to be simulated. When used in this way, it is called POLYSIMIC since the variable inputs are approximated by polynomials. This has been used primarily because DYSIMIC was not yet ready. When it becomes ready, POLYSIMIC will no longer be used.

When a simulation is to be made, a bioctal punched tape containing the SIMIC program is loaded into the 1103 computer by means of the Ferranti loading routine

in the Service Library. It goes on the drum in addresses 40000 through 44177. Drum addresses 44200 through 50000 plus 40x, where x is the number of minor cycles in the simulated program, are used for storage in the course of simulation. The input tape is then inserted in the tape reader, PAK is set to 40000 and the machine started. The simulation proceeds as per instructions on the input tape until the end of the input tape. If the code word "restore" appears at the start of a second input tape, this second simulation can be run without re-loading the SIMIC program tape.

The SIMIC program can be divided into three tasks which the program must perform:

1) Read input tape; recognize code words; store constants and addresses
2) Carry out the simulation
3) Convert to decimal, flex-code, and punch out the results when called for by the input tape.

The SIMIC program works from high-speed storage (HSS) in accomplishing tasks 1 and 3 while 2 is done on the drum.

Starting at 40000 the program jumps to 41776 which is the start of a block transfer. 40000 through 41777 is transferred to HSS and control passes to 00010 (which now contains the instruction in 40010). This and the instructions following cause one character to be read from the input tape. This character is examined to see if it is a stop code (43) or carriage return (45). If it is a stop, the program jumps to 00007 which is an unconditional stop, with 40000 as NI. If a carriage return is found, the program jumps back to 00010 to read another character. If the character so examined is neither stop nor carriage return, it is stored in the scratch pad (00017 through 00037) and the program jumps to 00100. As the characters are read in, they are stored in order in the scratch pad. Starting with 00100 the program examines the appropriate number

of characters in the scratch pad to determine the presence of one of the code words. If a given code word is not found, the program jumps to a section which seeks to identify the next code word. If a code word is found, the program jumps to the section which handles, in an appropriate fashion, the characters which follow that code word on the input tape. If none of the code words are found, the program returns to 00010, reads another character from the input tape, and again looks for one of the code words among the last n characters read, (n depends on the number of characters that comprise a code word). For example, the section beginning with 00100 seeks to identify the code word "program". In this case n = 9 for the code word to be recognized includes "carriage return" and "shift up" characters as well as the seven letters of the word "program". The following table lists the code words, together with the first address of the section in which they are recognized, and where the program goes if the code word is or is not recognized.

Address	Code Word	If recognized jump to:	If not recognized jump to:
00100	cr-su-p-r-o-g-r-a-m	42000	00140
00140	cr-su-c-o-n-s-t-a-n-t-s	00150	00200
00200	o-u-t-p-u-t-sp.-h-e-a-d-i-n-g	01000	00240
00240	t-y-p-e-sp.-s-p-a-c-i-n-g	01100	00300
00300	c-o-m-p-u-t-e	01200	00340
00340	i-n-p-u-t	00350	01024
01024	d-u-m-p-sp.	01034	01223
01223	r-e-s-t-o-r-e	01233	01252
01252	=	01255	00010

where cr means carriage return code (45)
su means shift up code (47)
sp. means space code (04)
= means equal sign code (44)

If none of the code words is found, control is returned to 00010 which reads in another character and initiates the above sequence again. This continues until a code word is found.

When a code word is found, the program jumps to the indicated instruction where appropriate action is taken. When this action has been completed, the program jumps back to 40000 and, after the block transfer to HSS, begins to look for a "stop" or the next code word. A "stop" code at the end of the input tape (the incremental program being simulated) causes the computer to stop with 40000 in PAK.

The following paragraphs will sketch what happens when the various code words are recognized. The reader should have in mind the format of the input tape as well as the Incremental Computer algorism which this program is designed to simulate.

When "program" is recognized, the SIMIC program jumps to 42000. This initiates a block transfer of 42000 through 42777 into HSS, and jump to 00100. A character is read from the tape and checked to see if it is a relevant symbol, such as a number, carriage return, stop code, letters "e" (from "end of program") or "n", plus or minus sign, period, space, shift up, shift down, back space, delete, or tab. Any symbols other than these will cause the program to read the tape, ignoring everything, until a carriage return is found. When a number is found, it is converted from flex-code symbols into binary and the program jumps to 00140. Beginning at 00140 is a series of index jumps which cause the numbers to be organized into groups of three as they come in (see incremental tape format). There will be seven groups of three digits each, representing step, U, V, P, Q, T, and W. After each group is completed and stored, the program goes to 00204. Here, and in the steps which follow, these addresses are manipulated in such a way as to produce the proper modifications in the steps

from 00400 to 00437. This block of steps is the nucleus of the whole SIMIC program, for it is here that the Incremental Computer's algorism is simulated. It may be indexed as follows:

400 - 404 adds $-S\Delta W_i$ to R_{i-1}

405 - 411 adds $V_{i-1}\Delta T_i$

412 - 416 updates U: $U_i = U_{i-1} + \Delta U_i$

417 - 422 adds $U_i\Delta Q_i$

423 - 427 adds $S\Delta P_i$ to form R_i

430 - 434 updates V: $V_i = V_{i-1} + \Delta V_i$

435 - 437 compares V_i with input and stores the difference as the ΔV_i to be used in this step at the next major cycle.

After this block of instructions has been set up to carry out the algorism according to addresses given in, say, step 000, the carriage return at the end of the line signals that this task is completed. This causes this block to be transferred to the drum at 50000 through 50037 (for step 000) and then to get ready for the next line of addresses by again block transferring from 42000 through 42777 into HSS. The next line is handled in the same way except that the algorism block (400 - 437) is transferred into 50040 - 50077 on the drum (for step 001). Thus, the algorism for each step is set up in accordance with the input tape and transferred to the drum. The instructions for step x_8 appear on the drum beginning at $(50000_8 + 40_8x_8)$, the subscript 8 denoting an octal number. When carrying out the simulation, the computation will start at 50000 and run in order to (50000 + 40y) where jump instruction has been placed by the program after finding the code word e-n-d (part of "end of program") opposite, say, step y. This is a jump to 00040. At the time the computation is carried out, HSS will contain instructions which decide after each major cycle, if it is time to stop and whether or not it is time to punch out results. This comes from

the "compute" and "type spacing" numbers on the input tape, and will be taken up later.

After "end of program", the input tape contains a line beginning with 400 in the "step" column. (The six 3-digit octal numbers of this line are the step numbers (of the incremental program) of those steps whose V-line values are to be punched out as output of the program.) When "400" is sensed, the SIMIC program jumps to 00500 where the instructions at 44005 through 44012 are changed to give the desired print-out.

The "stop" code at the end of the "program" section returns control to 40000, and the machine begins to look for the next code word.

Normally the next code word encountered is constants. The program jumps to 00150, which now corresponds to 40150. Here, indexes at 43041 through 43043 are restored and the program jumps to 43000. As before, there is a block transfer. The HSS now contains a duplicate of the instructions that are on the drum from 43000 through 43777. Starting again at 00100 the SIMIC program reads and decodes the step numbers and constants U_0, V_0, and S listed on the input tape, and stores these in their proper positions. These positions are listed in the table below. The table also includes, for convenient reference, the locations of other quantities used in the simulation.

Quantity Stored	Location on Drum	Ultimate Location in HSS at Time of Computation
U_{0_i}'s	44200 - 44477	00200 - 00477
V_{0_i}'s	44500 - 44777	00500 - 00777
R_i's	45000 - 45277	01000 - 01277
Results of Comparisons	45300 - 45377	01300 - 01377
S's	45400 - 45677	01400 - 01677
Inputs	45700 - 45777	01700 - 01777

The location of a particular U_0, V_0, or S, within the range indicated above, depends of course upon the step number associated with it. Thus, SIMIC has room for up to 300 (octal) steps in the program to be simulated. SIMIC uses these locations as U-lines, V-lines, and R-lines are used in the Incremental Computer, i.e., the values in these registers are changed in the course of the computation. To allow the initial values to be brought back, instructions located at 43200 - 43224 mirror the U_0's and V_0's at 46000 through 46577. Later, if desired, the code word "restore" will clear the R-lines and return the U_0's and V_0's to 44200 through 44477.

After the constants have been stored, stop code causes the program to go back to 40000 as before, and to look for the next code word. Normally this will be "Input", causing a jump to 00350 (40350). Here 43041 - 43043 are cleared and we jump to 43000. After the block transfer to HSS the program jumps to 00100. Because the indices at 00041 - 00043 are now cleared, the program goes through the index jumps at 141, 142, and 143 (after each step number has been read in) and goes (via 00144) to 00210. Here, and in subsequent instructions, the input values are decoded, converted to binary, and stored in 45700 - 45777, depending on the step number. Again, the stop code at the end of the section causes a jump to 40000 and initiates the search for the next code word.

The code word "output heading" causes a jump to 01000 (41000). The effect of these instructions is simply to store the flexowriter symbols, as read from the input tape, in locations starting with 41400. There is room for 376 (octal) symbols in the output heading, since any more than this would write over some of the instructions effecting a block transfer which is still needed. These instructions are located at 41775 - 41777. After storing the heading, control returns to 40000 and a search for the next code word is initiated.

The code word "type spacing" causes a jump to 01100 (41100) where the number following this code word is decoded, converted to binary, and stored in 44003. This time a stop code is not required to cause the program to return to 40000 and seek another code word.

The next code word is "compute". This causes a jump to 01200 (41200) where the number following the code word is stored in 44002. Then the stored output headings are punched out and the program jumps to 44000.

The instructions starting at 44000 control the functions. First 44000 - 45177 is block-transferred to HSS and control jumps to 00040. HSS now contains all the data needed by the computation which is on the drum starting with 50000, as well as the instructions which decide when to punch-out the outputs (type spacing) and when to stop the computation ("compute" number). Starting with 00040, the program sets up the output registers, punches out the first line of outputs (for cycle 000), and jumps to 50000. Upon reaching 50000 + 40x (octal), where x is the step number corresponding to "end of program" on the input tape, one major cycle has been simulated. The jump to 00040 is made, and the cycle counter (at 00013) is compared with the "type spacing" and "compute" numbers. Whether or not an output is punched now, the program jumps to 50000 if the number of cycles computed is less than the "compute" number. The program continues thus until these numbers are equal, when it again jumps to 40000 and looks for another code word. Usually this is a stop code, and the simulation stops.

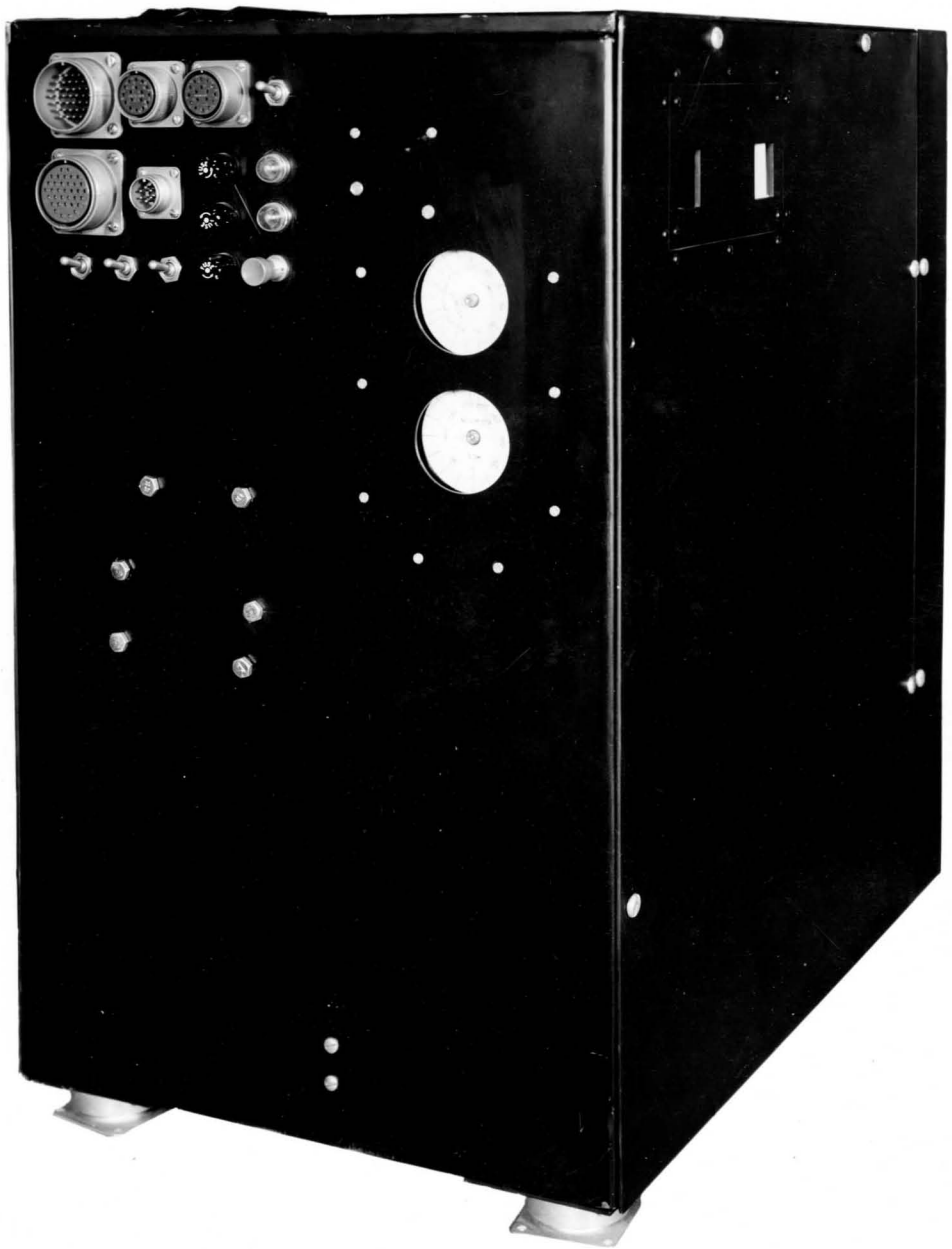
The remaining code words in the foregoing table are simply convenience features which have been added to SIMIC since its original inception. "Restore" has been described. Use of this word at the beginning of an input tape enables it to be fed into SIMIC after another simulation without the necessity of reading in the (large) SIMIC tape again. Used within an input program it enables the simulated program to be brought back to its initial values of U_0 and V_0 and

run again with new input values. Thus, a dynamic simulation could be achieved by listing a new set of input values for each major cycle of computation. This is rather impractical, however, as it would mean a very long list of inputs. A "stop" code after "restore" is required to effect a return to 40000 and a search for the next code word.

The "=" code word causes approximately six inches of leader to be punched out. This is convenient in identifying portions of the output tape.

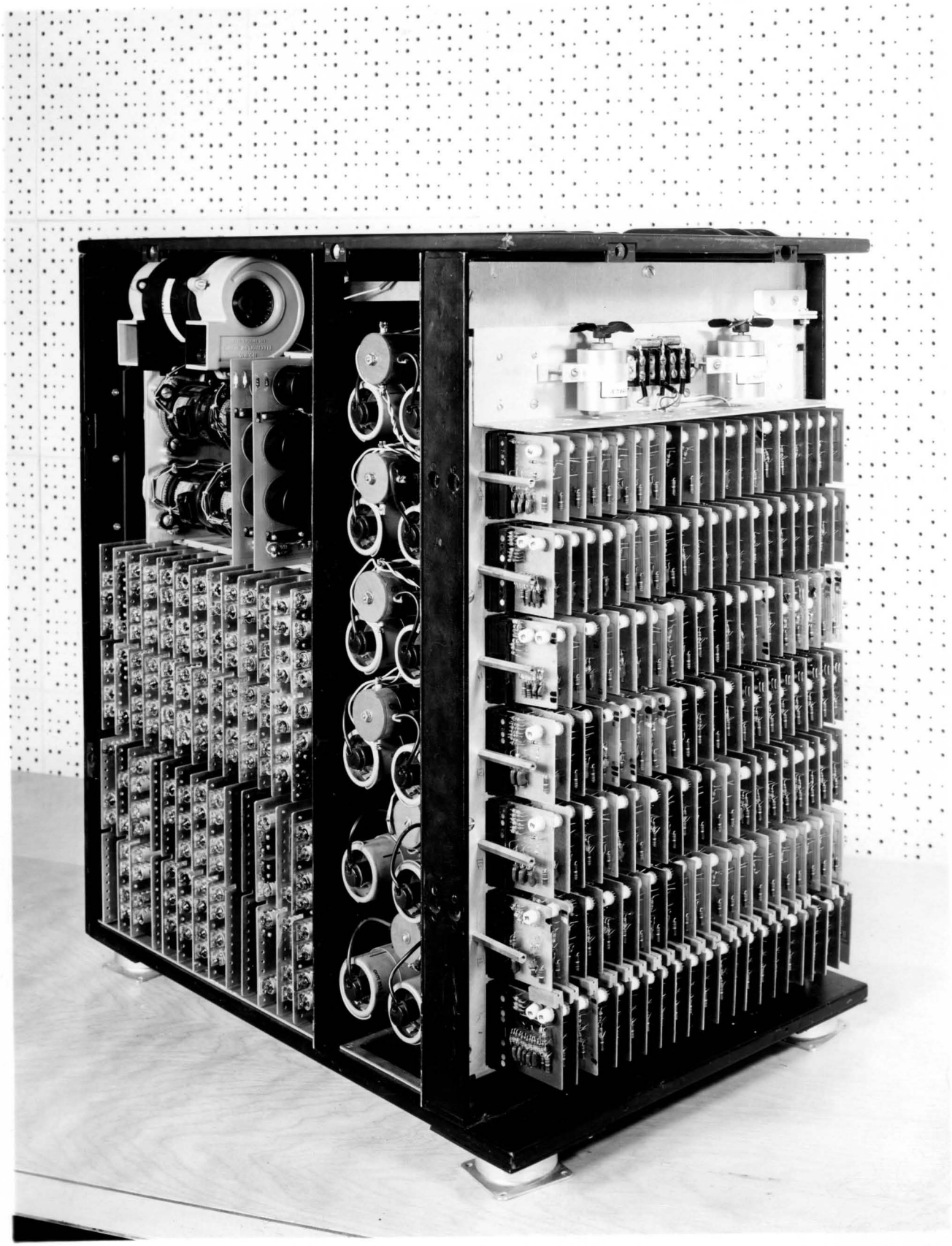
The code word "d-u-m-p-space" causes SIMIC to convert to decimal and flexcode, the contents of all the R's in order. When immediately followed by "V", the program will punch out all the V's and all the R's. When followed by "U" (i.e., d-u-m-p-space-u) it will punch out the U's, V's, and R's. These dumps are invaluable in analyzing a proposed incremental computer program, especially in deciding upon scaling and word length which are governed in part by the build-up of the R-line.

It is planned to add a "monitor" function to SIMIC also, but this has not been completely debugged as of the date of this report. Monitoring would compare the R-line of each step with three times that step's scale factor and punch out the cycle and the R-line when 35 is exceeded. It would do this each time an output is called for ("type spacing" number). The primary reason for adding this to SIMIC is that DYSIMIC, which has this useful feature, is not yet ready. It is planned to test Incremental Computer programs on POLYSIMIC until DYSIMIC is ready for use.



OB 9581

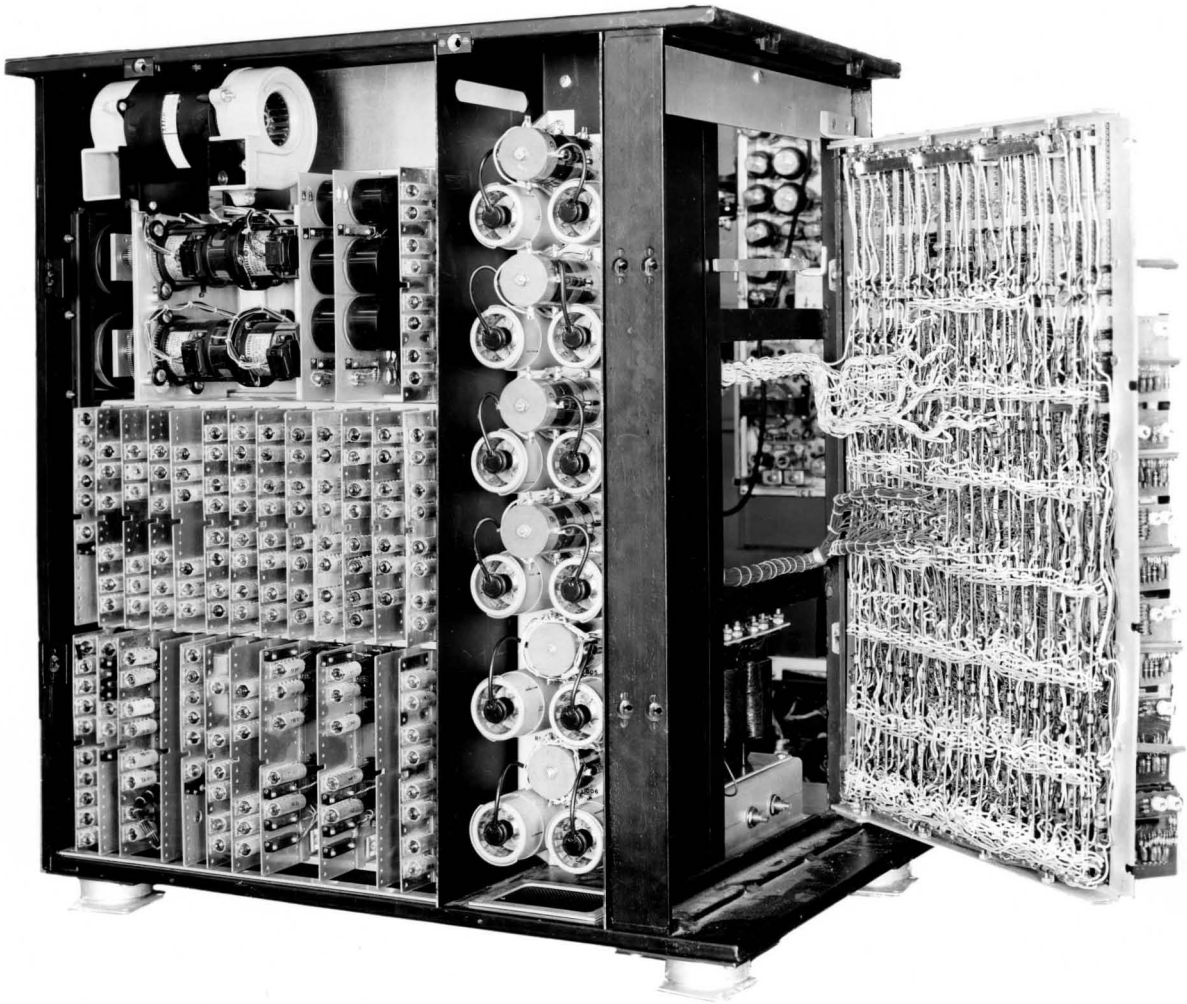
Digital Computer for Fire Control



Computer - Sides Off

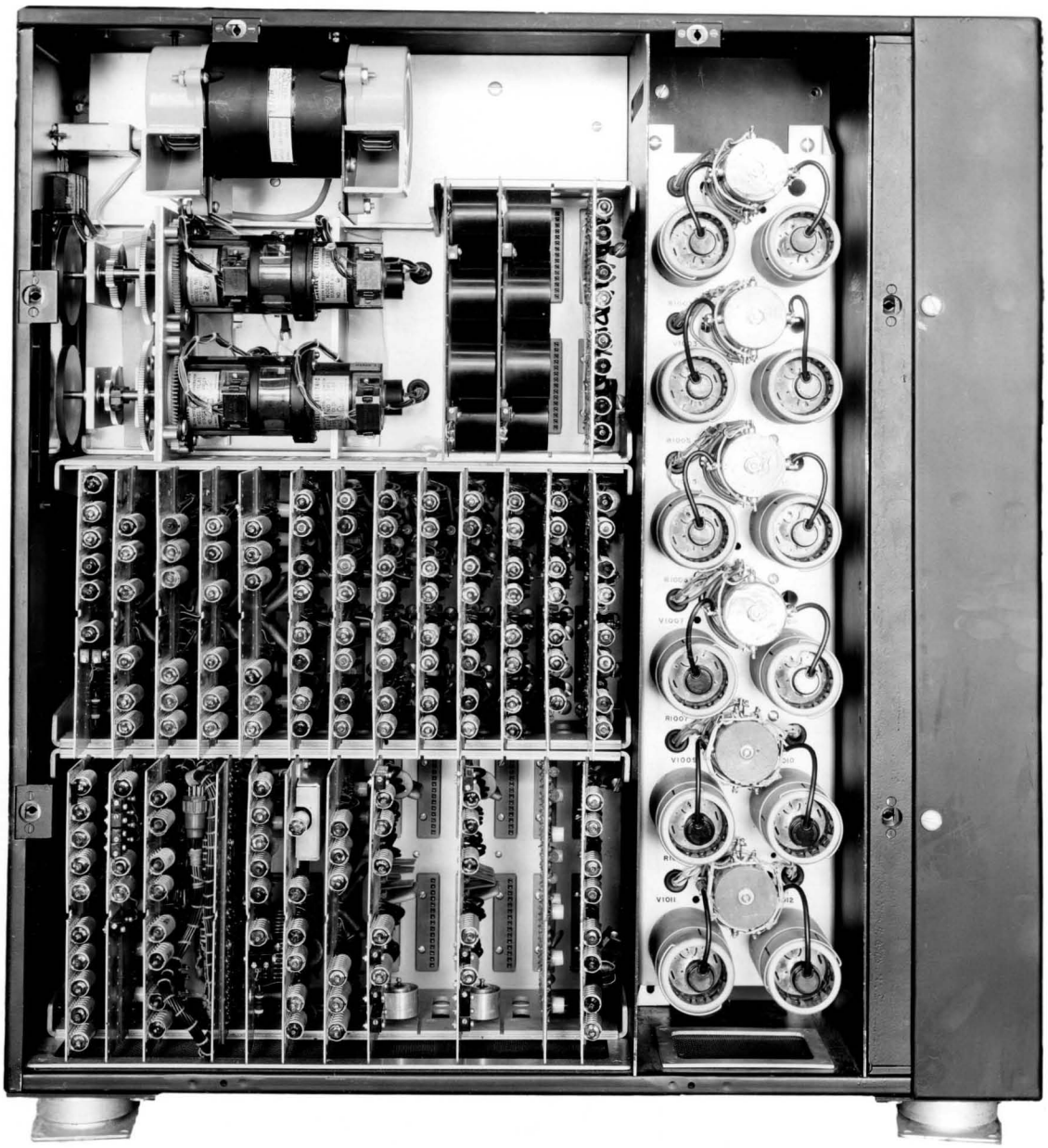
OB 9565

Magnetic Storage Drums - Incremental
Computer and Univac Scientific



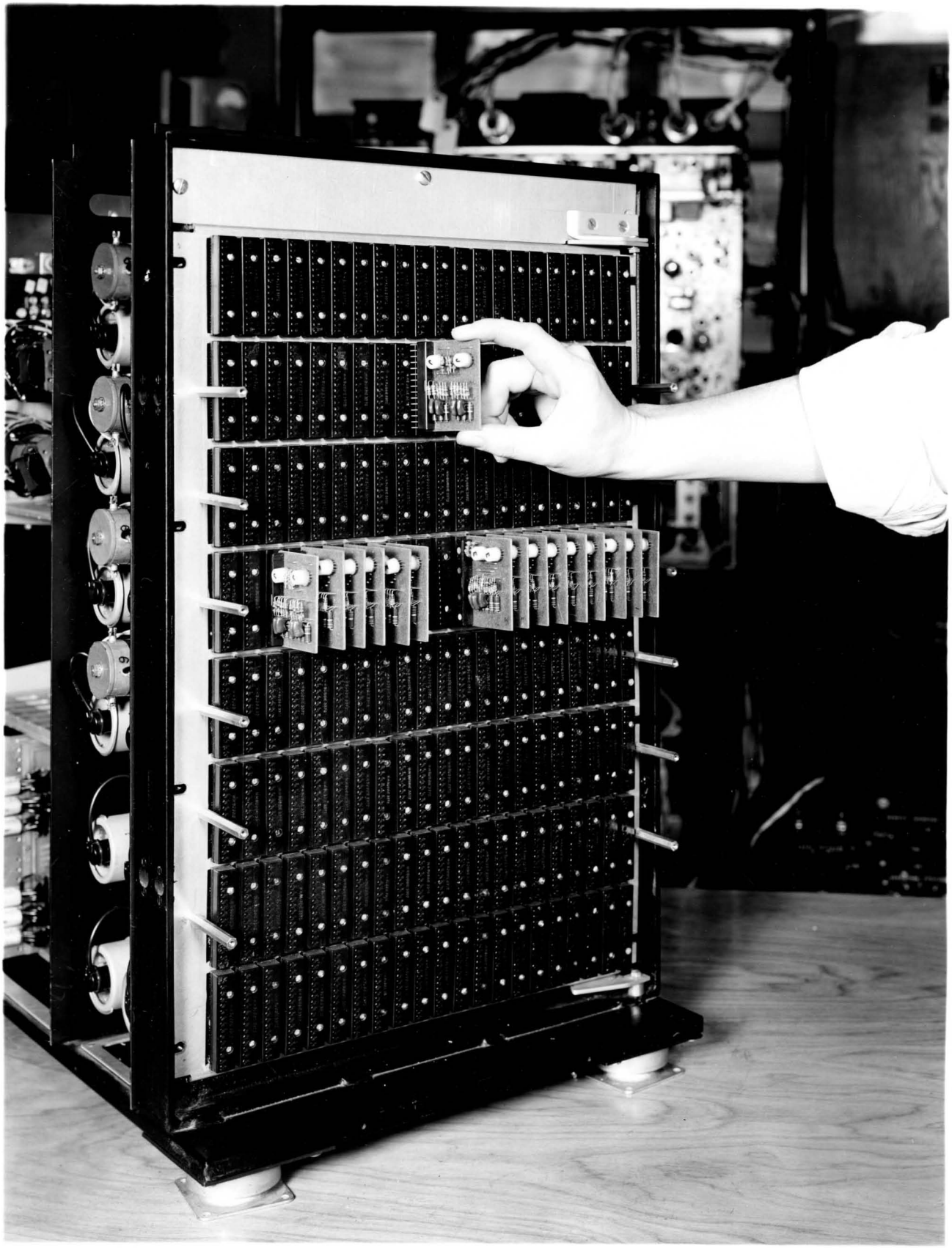
OB 9576

Computer - Magnetic Switch Wiring Exposed



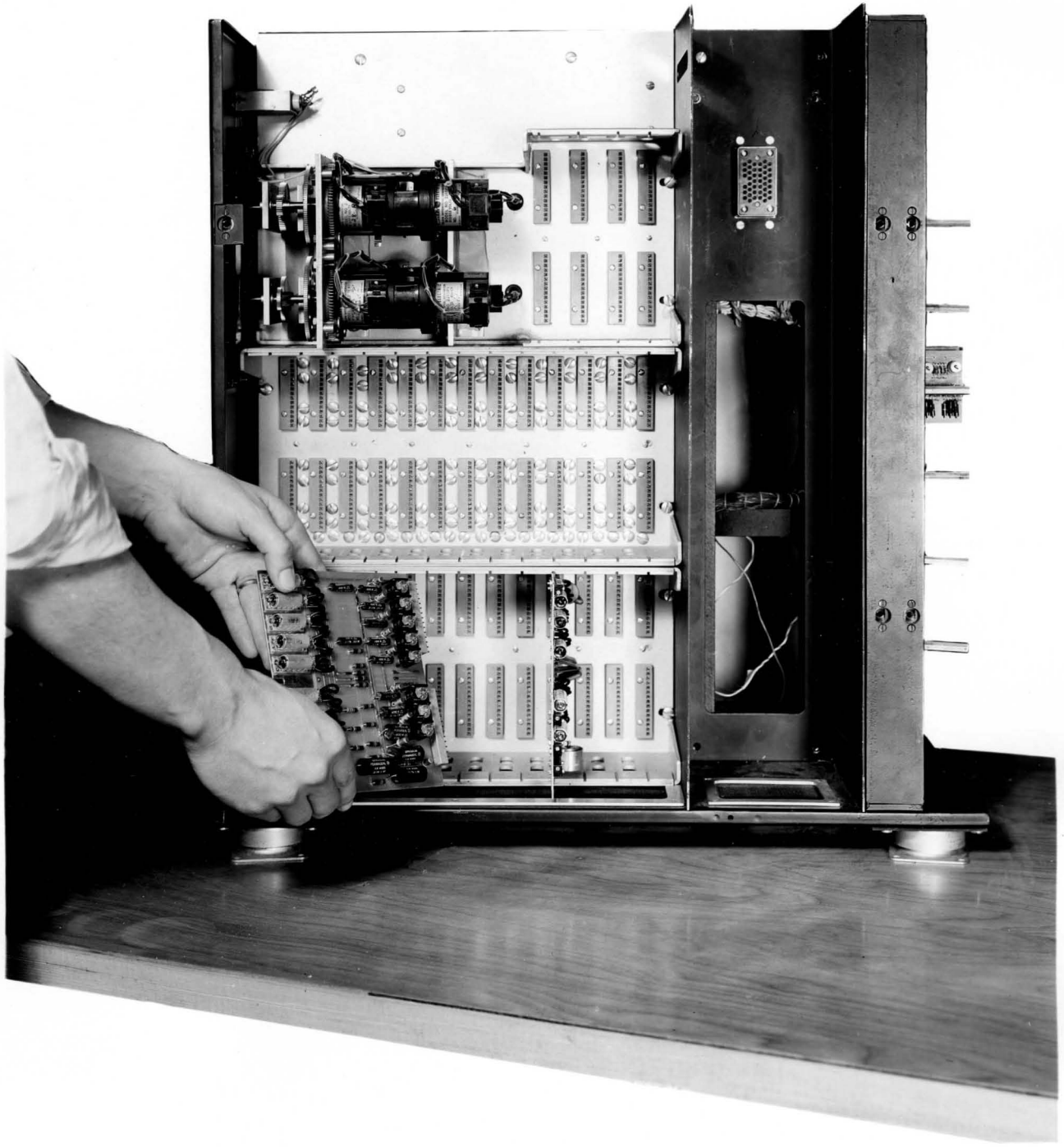
Subminiature Tube Classes and Driver Units in Computer

Computer - Drum and Power Supply



Insertion of Magnetic Switch

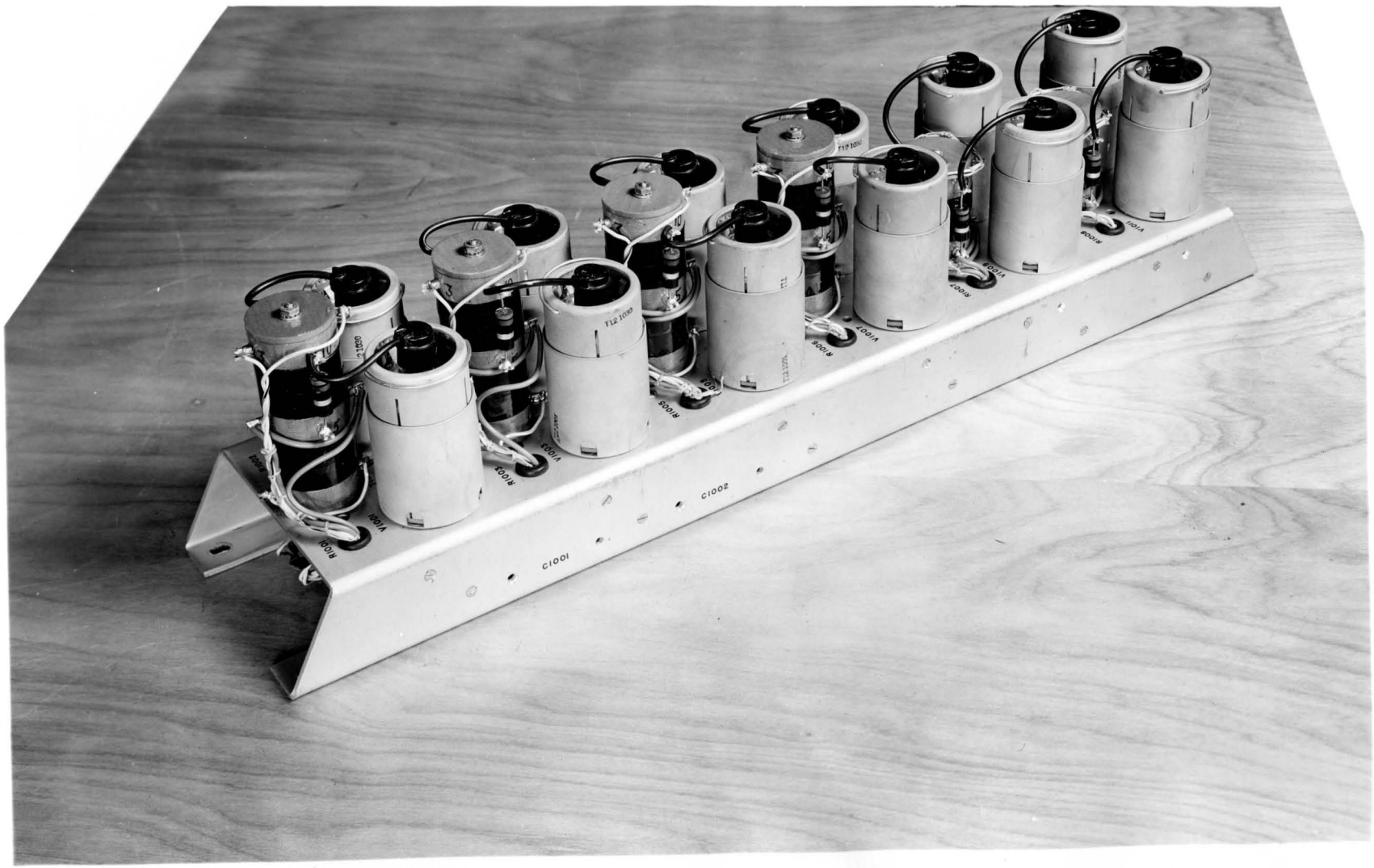
OA 9590



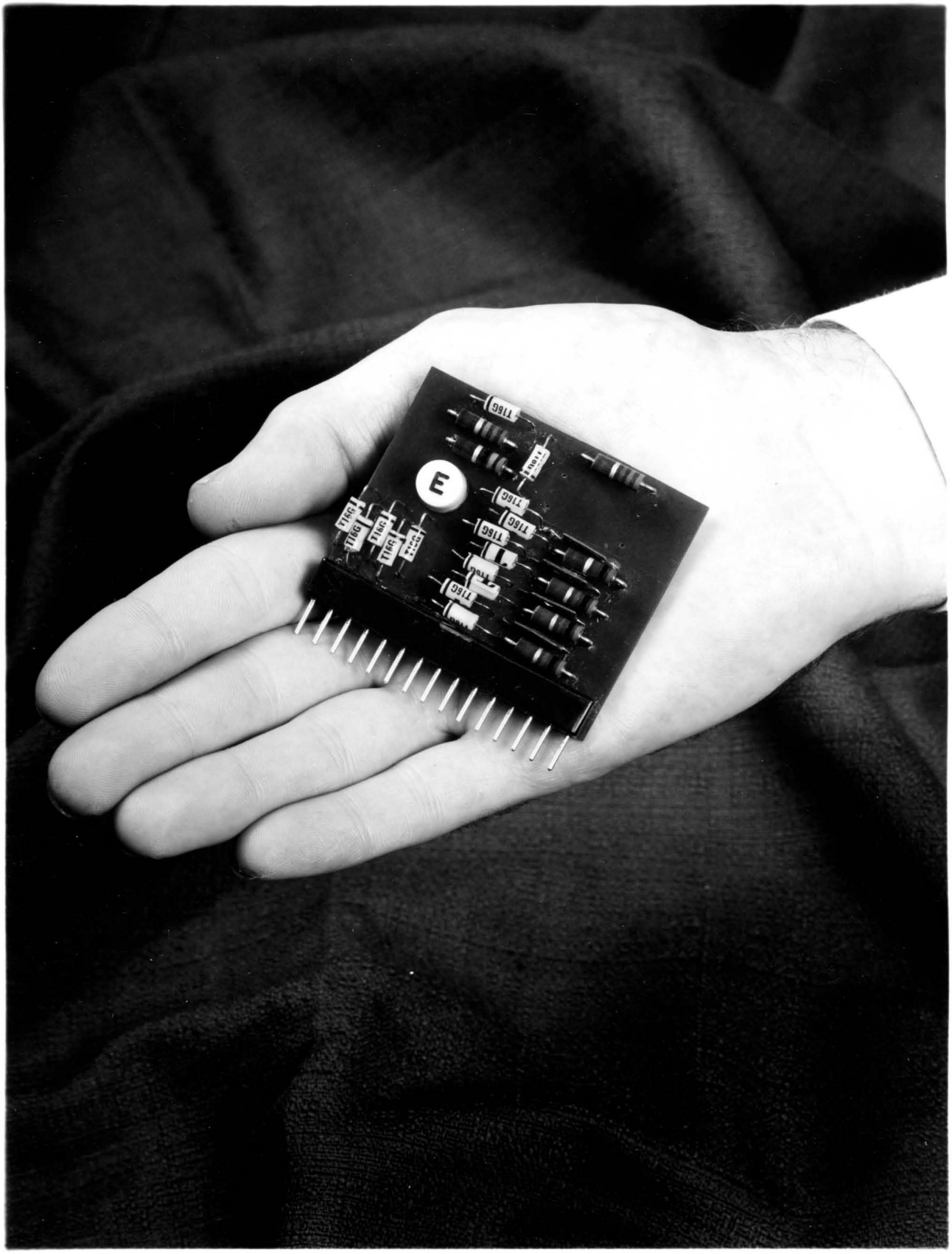
OA 9592

Computer - Insertion of Subminiature Tube Classes

OB 9597

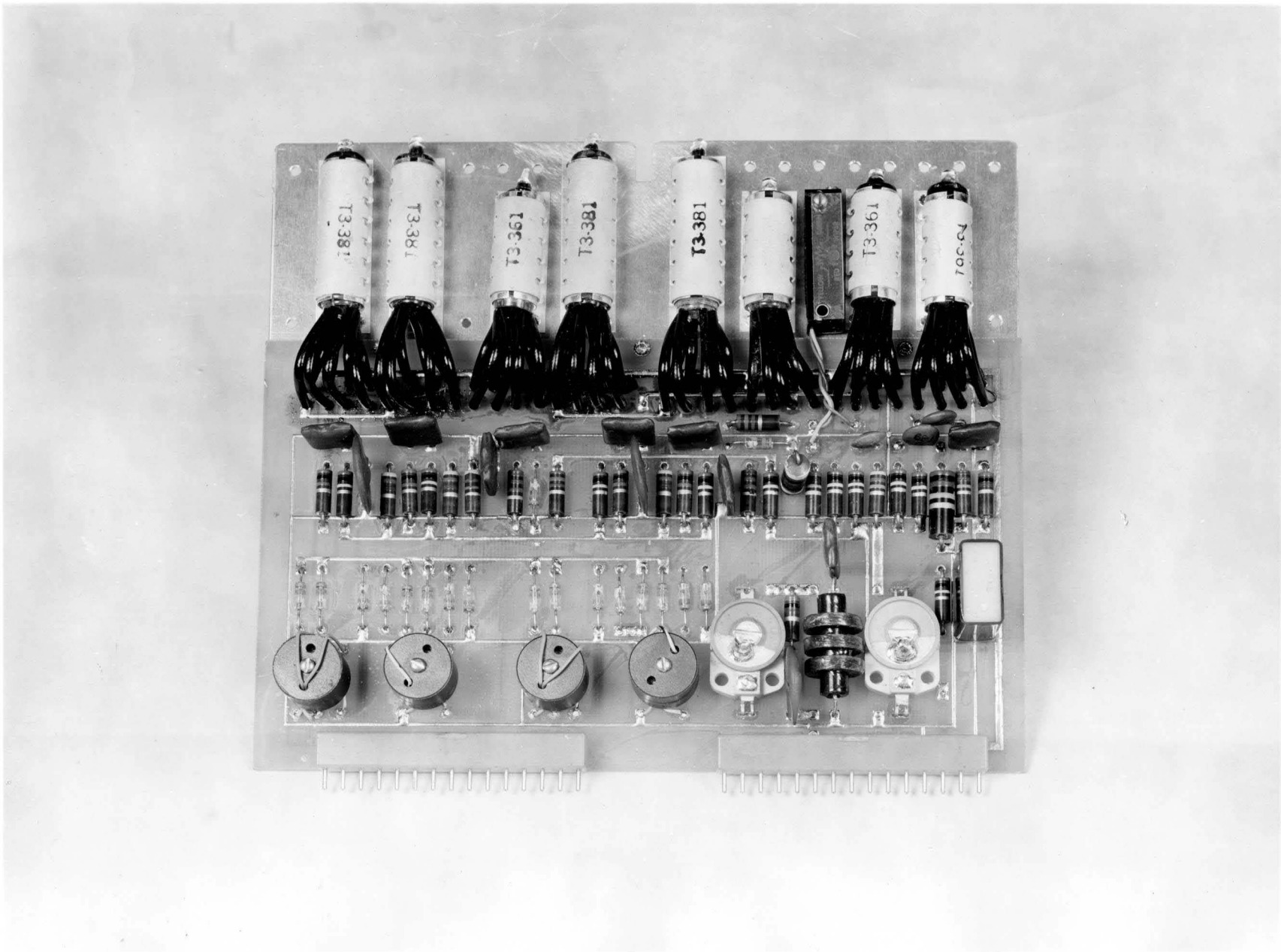


Pulse Driver Unit



Magnetic Switch

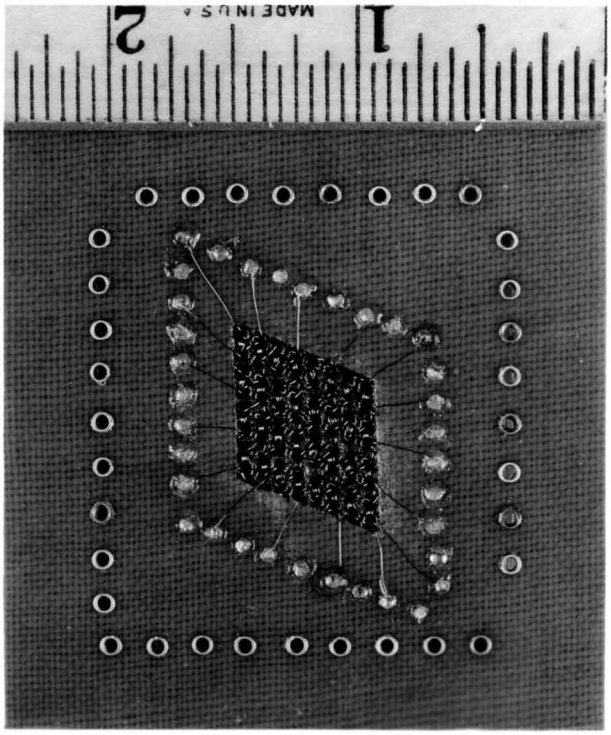
OA 9111

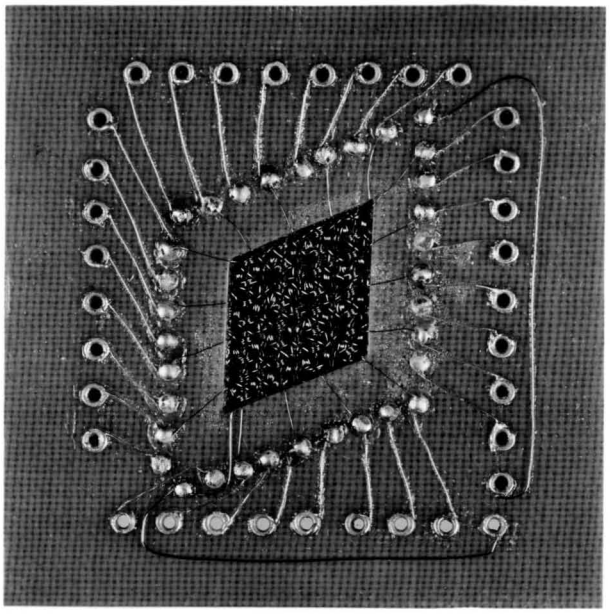


Subminiature Tube Unit

Magnetic Storage Drum - Incremental Computer

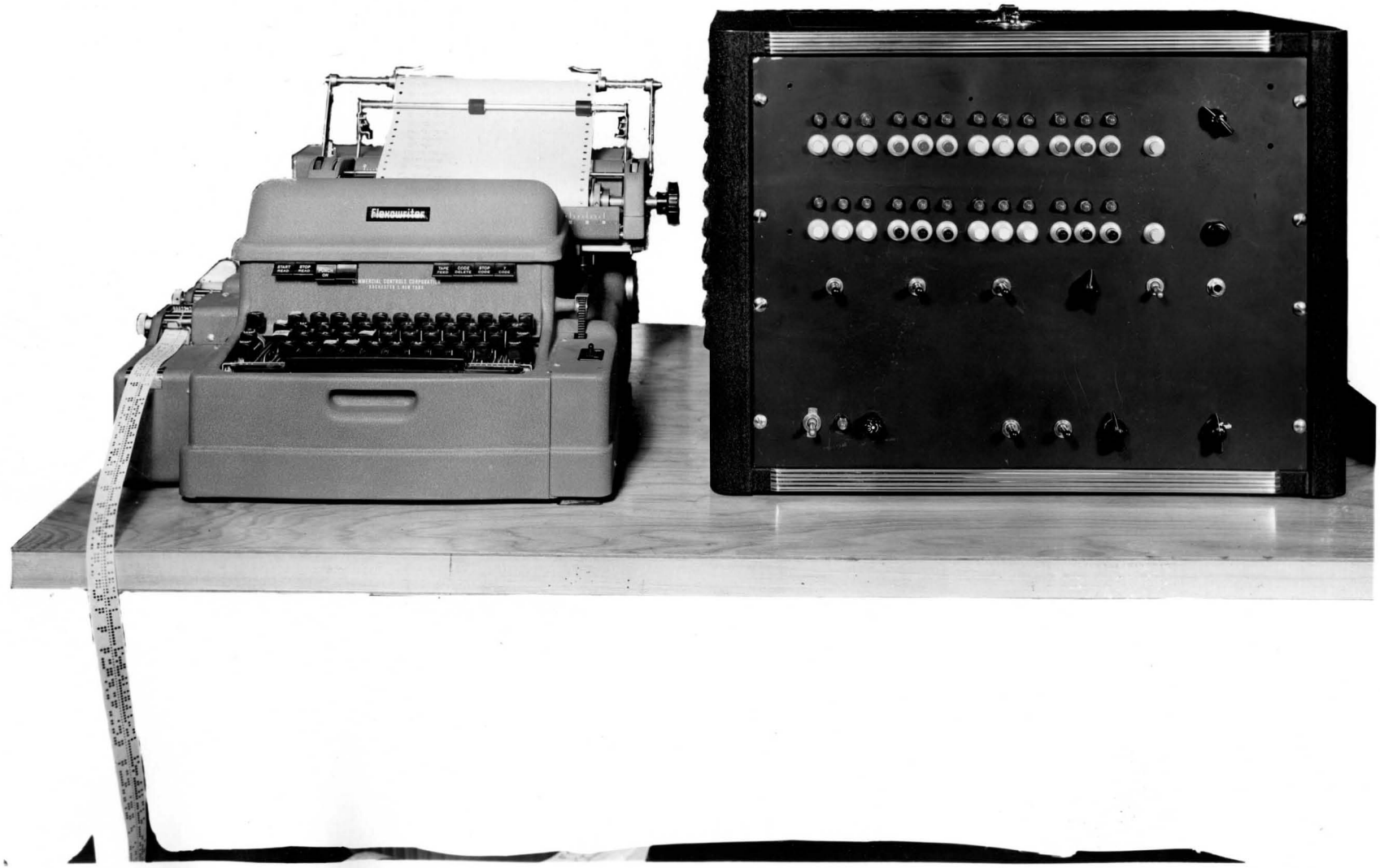
OB 9458





OA 8921

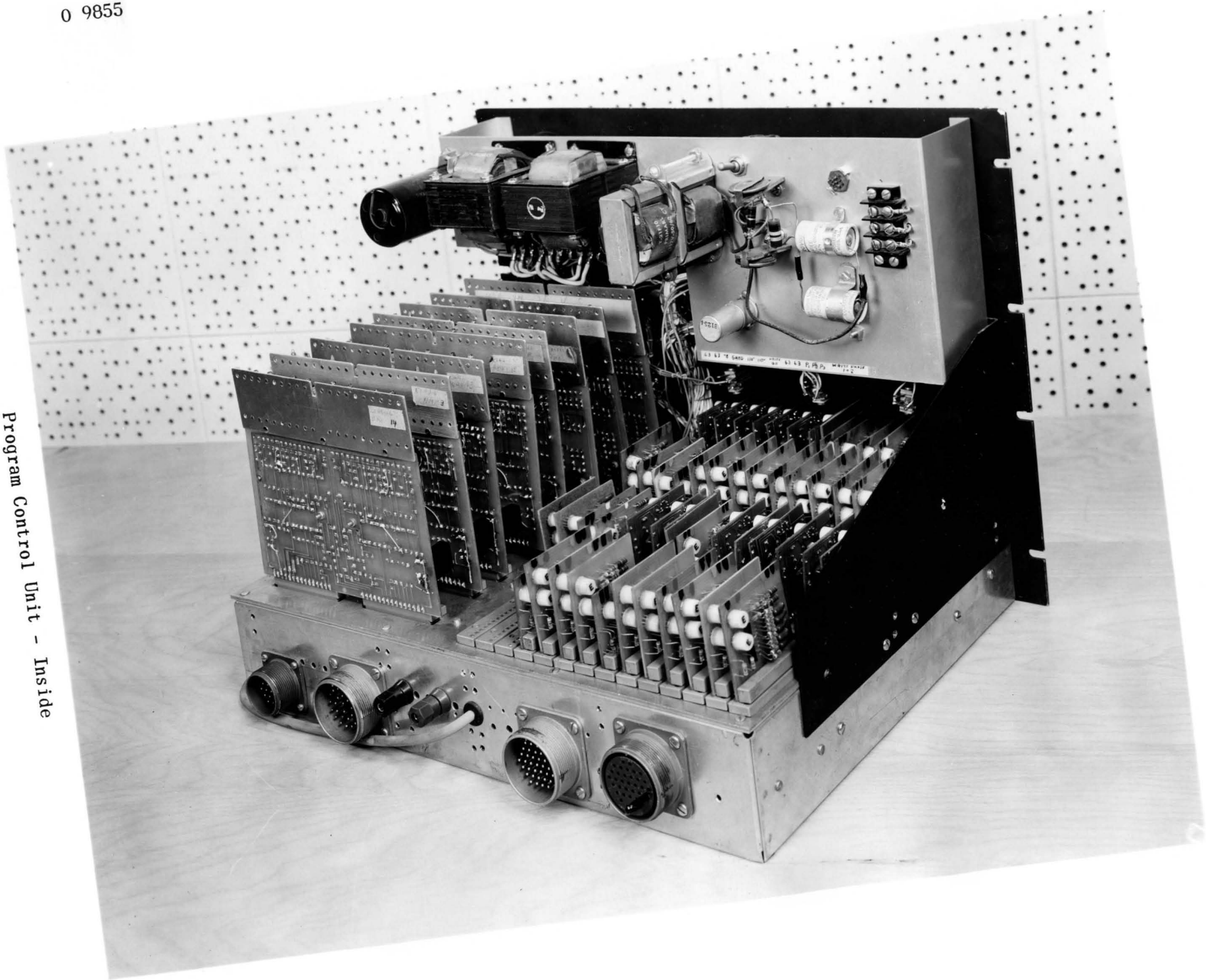
Core Matrix - Random Access Storage



Programing Equipment

Program Control Unit - Front

0 9855



Program Control Unit - Inside