A Computer Analytic Method for Solving Differential Equations

LEO HELLERMAN†

Introduction

In recent years, numeric analysis has been claiming an increasing share of overall mathematical research activity. The reason for this is apparently the need to have answers — numeric answers — to problems of modern technology, along with the development of the stored-program digital computer for carrying through the computations of numeric methods. But this emphasis on numbers is also an indication of the attitude of problem solvers: to use the computer, use numeric methods. And yet these methods are not always adequate. It may be more important to know how x depends on other variables than to know that $x = 3$.

The inadequacy of numeric techniques for the solution of differential equations is highlighted by the following engineering problem: In the design of a transistor switching circuit for a high-speed computer, we wish to know the output-current level at a particular time after the start of an input pulse. This information is contained in the solution of a nonlinear differential equation, which can be solved very nicely by numeric methods on a computer in, say, ten minutes. In evaluating the reliability of this circuit with respect to component deviation and drift, we want to know the statistical distribution of outputs. A simple method for finding this is synthetic sampling, or monte carlo[1]; but at ten minutes per solution, this may not be practical. However, monte carlo is known to be practical in estimating distributions associated with analytic expressions. This suggests that we first obtain the analytic solution of the differential equation, and then apply monte carlo to the solution. The methods are compared schematically in Fig. 1.

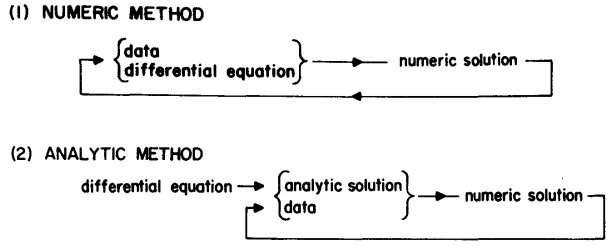

Fig. 1—Comparison of numeric and analytic methods of solving a differential equation many times.

† Product Development Laboratory, Data Systems Div. IBM, Poughkeepsie, N. Y.

[1] L. Hellerman and M. P. Racite, "Reliability Techniques for Electronic Circuit Design" *Trans. I.R.E. PGRQC*, Sept. 1958, pp. 9-16.

In the numeric method ((*1*) in Fig. 1), we must solve each case anew, starting with the data and differential equation. In the analytic method ((*2*) in Fig. 1), enough information is contained in the solution, so that we need solve the differential equation only once, and evaluate the solution for each case. Since a major portion of machine time is taken up with solving differential equations, there may be problems in which (*1*) is not practical and (*2*) is, provided (*2*) can be carried through by the computer.

The purpose of this paper is to call attention to a basic principle of analytic technique on a stored-program digital computer, and to illustrate this principle by a computer algorithm, and "address calculus," for finding solutions of ordinary differential equations by analysis. We also describe the implementation of this algorithm in an IBM 704 program. We see no reason why the same technique might not be applied to a host of other mathematical problems.

The Principle and General Approach

The principle of numeric computation in a stored-program digital computer is well known: numbers are represented by the contents of storage cells, and computation is accomplished by arithmetic manipulation on these contents. Functions are represented by a finite table of numeric values. The principle of analytic computation may be stated thus: algebraic symbols are represented by the *locations* of storage cells, and analysis is accomplished by manipulating addresses. Functions are represented by machine programs. An algorithm for the analytic solution of a problem is an assignment of the correspondence of algebraic symbols with addresses, and a description of the way the addresses are manipulated.

In the description of the following programs, in referring to the address of some location corresponding to some symbol S, we will say "address S." Address and symbol are equivalent, and we may use the symbol to designate the address. On the other hand "address *of* S" refers to some other location and address, say T, which has the address S as part of its contents. Thus the address T may be the address of S.

Our approach to the analytic solution of ordinary differential equations will be to develop the Taylor series expansion of the solution. If the differential equation is

$$y^{(k)}(x) = f(x; y^{(0)}(x), \ldots, y^{(k-1)}(x)) \qquad (1)$$

then the formal solution is

$$y(x) = y(0) + y^{(1)}(0)x + \frac{y^{(2)}(0)\,x^2}{2!} + \dots \quad (2)$$

where the $y^{(j)}(0)$ for $j = 0, \dots, (k-1)$ are assigned initial values, and for $j = k, k+1, \dots$ are determined from f and the derivatives of f.

Thus the heart of the problem is to develop analytic differentiation on a computer. In this connection we mention the work of H. Kahrimanian[2], and the LISP Programming System[3]. However, our approach is a bit different from both of these, being a close parallel to differentiation "by hand". A recently reviewed Soviet paper[4] appears to contain material quite similar to the work described here. The SHARE routine PE PARD[5] for differentiation and partial differentiation of rational functions is a prototype of our present program. Recall that the function to be differentiated is, in the computer, a stored program. PARD examines this program as a college sophomore examines a function to be differentiated, and when it finds it to be the sum of two parts, it applies the rule: the derivative of a sum is the sum of the derivatives. Or, if it finds a product, it uses $D(uv) = uDv + vDu$, and similarly for other differentiable combinations. Eventually, the derivative of a function is expressed in this way in terms of the derivatives of constants and the independent variable, and the differentiation process is complete. The problem in doing this on a computer is doing it in a uniform and orderly way, so that the method may be applied to arbitrary differentiable functions, and so that the results of the differentiation of each term can be combined in the end to one expression (program) for the derivative.

The Dendrite Nature of Functions

In this section, we examine a stored-program aspect of functions. Some notions will be defined which will facilitate the description of the differentiation algorithm.

A *binary operation* is an operation on two quantities. Addition, subtraction, multiplication, division and exponentiation are binary. *Unary operators* operate on a single quantity. Exp, log, sin, and cos are unary. Let the symbol $a * b$ have this meaning: * is a binary or unary operation. If * is binary it operates on a and b; if unary it operates on a, and b is ignored. Thus $a * b$ may be, for example, $a + b$, $a \div b$, a^b, or $\log a$, or $\sin a$. We will say a mathematical expression is a *finite dendrite* if it is composed by a finite number of binary and unary operations from a set of starting terms. We call a starting term an *elementary term*, or an *end:* it is not composed from other terms.

For example, consider the dendrite $y = (x + a)b - \sin x$. Its branching nature is shown in Fig. 2.

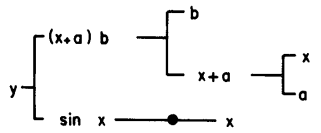

Fig. 2—The dendrite $y = (x + a)\,b - \sin x$.

The elementary terms are a, b, and x. The dendritic terms are, besides y itself, $(x + a)b$, $x + a$, and $\sin x$.

Note that the dendritic picture of y may serve as a flow chart for a program for its computation. First x is added to a, and the result is multiplied with b. Then x is operated on by some sine routine, and the result of this is combined by subtraction with $(x + a)b$ to give y. Thus a stored program for evaluating a function is essentially a sequence of binary and unary operations, starting with operations on elementary terms. That is, a program is a dendrite.

Blocks of *1*'s and *2*'s are a convenient notation for the branches of a dendrite. If $\alpha_1 \dots \alpha_n$ is such a block representing some dendritic term, ($\alpha_i = 1$ or 2), then, from the definition, $\alpha_1 \dots \alpha_n = \alpha_1 \dots \alpha_n 1 * \alpha_1 \dots \alpha_n 2$. The branch designations of the above example are shown in Fig. 3.

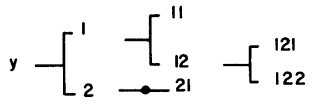

Fig. 3—Branch designations for the dendrite $y = (x + a)\,b - \sin x$.

A set of branches of the form

$$\alpha_1,\ \alpha_1\alpha_2,\ \dots,\ \alpha_1\alpha_2 \dots \alpha_n,$$

where the last branch is an elementary term, will be called a *chain*. All the chains of the example are

1, 11
1, 12, 121
1, 12, 122
2, 21

A finite dendrite has a finite number of chains.

The Differentiation Algorithm

Let us suppose we are given a y-program, that is, a stored program for computing y. We wish to extend

[2] H. G. Kahrimanian, "Analytical Differentiation by a Digital Computer," M. A. Thesis, Temple University, May 1953.

[3] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine" *Quarterly Progress Report* No. 53, Research Laboratory of Electronics, M.I.T., April 15, 1959.

[4] L. V. Kantorovich, "On Carrying Out Numerical and Analytic Calculations on Machines with Programmed Control," *Izv. Akad. Nauk Armyan. SSR.*, Ser. Fiz.-Mat., Nauk 10 (1957), No. 2, 3–16. See review No. 4360 by J. W. Carr, III, in *Mathematical Review*, June 1959.

[5] M. R. Dispensa and L. Hellerman, "Differentiation and Partial Differentiation of Rational Functions" *PE PARD*, SHARE distributed Program D2-445, March 18, 1958.

this to a program for its derivative, a Dy-program, by adding additional instructions. The block of storage for the Dy-program will include the block for the y-program. Since y is a dendrite, $y = 1 * 2$ and

$$Dy = A_1 D(1) + A_2 D(2) \tag{3}$$

where A_1 and A_2 are functions determined by the operation $*$ and the branches 1 and 2. That is, $A_\alpha = A_\alpha(*, 1, 2)$ where $\alpha = 1, 2$. These functions are specified in Table I. It may happen that an A_α is the number 0, or 1, or some function which is known to exist in the y-program. This is the case for $y = u + v$ and $y = \exp u$, and in these cases it is unnecessary to place any instructions in the block reserved for the Dy-program. If an A_α is not of this type, say $A_\alpha = -u \div v^2$, then we do construct the program for this function and place it in the first available locations in the Dy-program block. Whether A_α is constructed or not, we save the addresses A_1 and A_2, in locations $L(1)$ and $L(2)$. Since we can only differentiate one term at a time, we also save the instruction to find $D(2)$, in a location $I(2)$.

TABLE I

A_1 AND A_2 IN $D(u * v) = A_1 Du + A_2 Dv$

$u * v$	A_1	A_2	I
$u + v$	1	1	v
$u - v$	1	-1	v
$u \cdot v$	v	u	v
$u \div v$	$1 \div v$	$-u \div v^2$	v
u^v, v constant	vu^{v-1}	0	v
$\exp u$	$\exp u$	0	0
$\ln u$	$1 \div u$	0	0
$\sin u$	$\cos u$	0	0
$\cos u$	$-\sin u$	0	0

We may now go on to find $D(1)$. Suppose 1 is dendritic and $1 = 11 * 12$. Then

$$D(1) = A_{11} D(11) + A_{12} D(12)$$

The functions A_{11} and A_{12} are constructed as A_1 and A_2, again by Table I, the addresses A_{11} and A_{12} are stored in $L(11)$ and $L(12)$, and after storing the instruction to find $D(12)$ in $I(12)$ we continue to find $D(11)$.

In general, if $\alpha_1 \ldots \alpha_n$ is dendritic, then

$$D(\alpha_1 \ldots \alpha_n) = A_{\alpha_1 \ldots \alpha_n 1} D(\alpha_1 \ldots \alpha_n 1) + A_{\alpha_1 \ldots \alpha_n 2} D(\alpha_1 \ldots \alpha_n 2) \tag{4}$$

The coefficients are constructed if necessary, and the addresses $A_{\alpha_1 \ldots \alpha_{n+1}}$ stored in $L(\alpha_1 \ldots \alpha_{n+1})$; the instruction to find $D(\alpha_1 \ldots \alpha_n 2)$ is saved in $I(\alpha_1 \ldots \alpha_n 2)$; then we go on to $D(\alpha_1 \ldots \alpha_n 1)$.

Eventually, since y is finite, we come to an elementary term, $11 \ldots 11$. This will be a constant, the independent variable, or an initial condition $y^{(j)}(0)$, $j = 0, \ldots, (k-1)$. Thus $D(11 \ldots 11) = A_{11 \ldots 111}$ is known, and we store this in $L(11 \ldots 111)$.

At this point we have traversed one chain of the dendrite y. We may now examine the I-cells for some deferred differentiation instruction, and proceed with the differentiation of this new term, until another end is reached. Continuing in this way, all chains will be completed, for there are a finite number.

It is clear from (3) and (4) that Dy is simply the sum of all products of A's, where the subscripts of the factors of each product range over a complete chain. If $A_{\alpha_1 \ldots \alpha_j} = C_{ij}$, where i stands for the i-th chain, we may write

$$Dy = \sum_{i=1}^{s} \prod_{j=1}^{k(i)} C_{i1} C_{i2} \ldots C_{ij} \tag{5}$$

where i ranges over the set of chains of y, s in number, and where $C_{ik(i)}$ is the derivative of the end of the i-th chain.

Thus the Dy-program is completed by construction of a program for evaluating (5). This can be done because the addresses $A_{\alpha_1 \ldots \alpha_n}$ are at hand in the locations $L(\alpha_1 \ldots \alpha_n)$.

The algorithm as it stands requires excessive storage. To differentiate any function composed with n operations, we should allocate 2^n locations for storing the addresses $A_{\alpha_1 \ldots \alpha_n}$, for there are as many of these as n-blocks of 1's and 2's. But consider the situation when the first chain has been completed.

At that point we know

$$\begin{aligned} Dy = {} & A_1 A_{11} \ldots A_{11 \ldots 11} A_{11 \ldots 111} \\ & + A_2 D(2) \\ & + A_1 A_{12} D(12) \\ & + \ldots \\ & + A_1 A_{11} \ldots A_{11 \ldots 1} A_{11 \ldots 12} D(11 \ldots 12) \end{aligned} \tag{6}$$

The addresses of the A's and D's are consecutive cells in three blocks of storage, called the A_1-block, A_2-block, and I-block. The storage arrangement is shown schematically in Fig. 4. The lines in this figure indicate the formation of the products in (6).

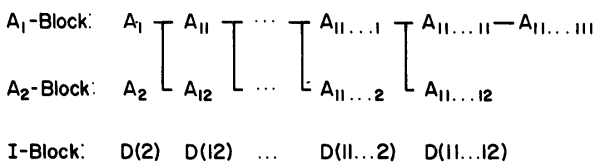

Fig. 4—Contents of A_1-Block, A_2-Block, and I-Block of storage, at completion of a chain.

Instead of continuing by completing another chain, construct the program for the product of terms of the first chain just completed, and place this program in the Dy-block. Then the addresses $A_{11 \ldots 11}$ and $A_{11 \ldots 111}$ are no longer needed. We may replace $A_{11 \ldots 11}$ in $L(11 \ldots 11)$ of the A_1-block by $A_{11 \ldots 12}$

and proceed to find $D(11 \ldots 12)$. The storage arrangement will now be as shown in Fig. 5.

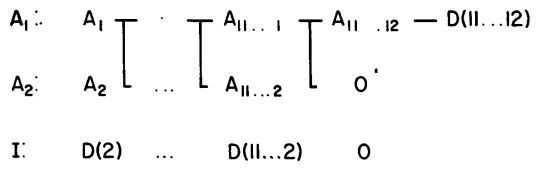

Fig. 5—Contents of storage blocks after down-dating of Fig. 4 arrangement.

The differentiation $D(11 \ldots 12)$ may generate further terms in the A_1, A_2, and I-blocks. Whenever an end is reached, we form the product for the completed chain, replace unnecessary A_1 terms by their corresponding A_2 terms, and then continue with differentiation of the next term, indicated by the last non-zero address in the I-block. This process of replacing terms no longer needed by terms of the next chain to be completed will be called *down-dating*. *Updating* refers to the process of adding new addresses to the A_1, A_2 and I-blocks, upon examination of each operation, as prescribed in Table I. These new addresses are stored in the appropriate blocks, and nowhere else. Since two cells are required for the storage of the A's from each operation, it is now only necessary to allocate $2n$ storage cells for saving the A's of any n-operation function.

An IBM 704 Program

A main feature of our IBM 704 implementation of the above algorithm is the pseudo-code used to specify functions. The function $A = B + C$ in 704 instructions would be

```
CLA  B
ADD  C
STO  A
```

But we do not actually need B in the accumulator, for we do not really intend to add numbers. The program is used only to recognize that B and C are composed by the binary operation addition, to form A. Thus a more compact code is possible, and desirable if the information we need is to be easily available. The code we use for $A = B + C$ is

$$A\text{: PZE } B, , C$$

That is, location A contains the addresses B and C in the address and decrement portion of the word, and the prefix PZE is used to indicate that these are composed with the binary operation of addition. The code for other operations is shown in Table II.

Table II also shows the detailed 704 version of Table I. When a function $K = u * v$ is differentiated, the construction of A_1 and A_2 and the updating of certain blocks of storage is specified by this table. After $K = u * v$ is differentiated, the next step is to find Du. In the flow chart of Fig. 6 "new K" refers to u.

TABLE II

Up-Dating of *Dy*-Program, A_1-Block, A_2-Block, and I-Block

Function $K =$	Code at Location K	*Dy*-Program New Terms for Construction of A_1 and A_2	A_1	A_2	I
$u + v$	PZE u , , v	None	1	1	v
$u - v$	PON u , , v	None	1	-1	v
$u \cdot v$	PTW u , , v	None	v	u	v
$u \div v$	PTH u , , v	$L + 0$: MON v , , *MFL1*‡ $L + 1$: PTW K , , L	L	$-(L + 1)$	v
u^v	MON u , , v	$L + 0$: PON v , , *FL1*‡ $L + 1$: MON u , , L $L + 2$: PTW v , , $L + 1$	$L + 2$	0	0
exp u	MZE u , , 1	None	K	0	0
ln u	MZE u , , 2	L: MON u , , *MFL1*‡	L	0	0
sin u	MZE u , , 8	L: MZE u , , 16	L	0	0
cos u	MZE u , , 16	L: MZE u , , 8	$-L$	0	0

‡*MFL1* is the address of -1; *FL1* is the address of 1.

The 704 flow chart is clarified by a description of the roles played by certain blocks of storage.

(*a*) *Constants block.* All constants are given addresses of storage locations in this block.

(*b*) *Initial conditions block.* This contains the locations $y^{(j)}(0)$, $j \leqq k - 1$, as consecutive storage cells. When an end $y^{(j)}(0)$, $j < k - 1$, is recognized, its derivative is the address $y^{(j+1)}(0)$.

(*c*) *Variable of differentiation.* This is a single storage cell.

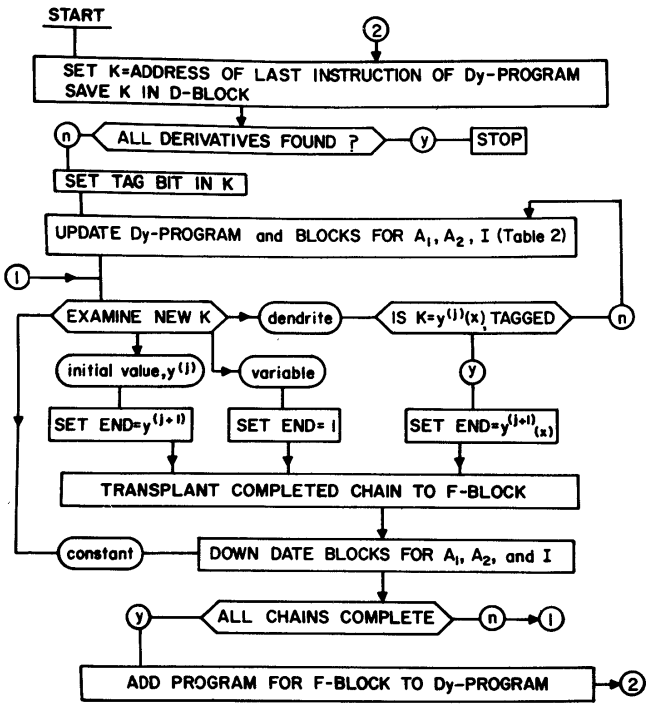

Fig. 6—Flow chart for successive differentiation of $y^{(k)}(x) = f(x; y^{(0)}(x), \ldots, y^{(k-1)}(x))$

(*d*) *Function program block.* This contains the sequence of pseudo-instructions defining the function to be differentiated. The last pseudo-instruction is in $y^{(k)}(0)$. New terms for the construction of A_1 and A_2, as shown in Table II, are placed in the first available locations following $y^{(k)}(0)$, as needed. The program of pseudo instructions for formula (5) is also stored here, when all its terms have been constructed.

(*e*) *Derivative block, D.* The derivative of the initial condition $y^{(k-1)}(0)$ is $y^{(k)}(0)$, which is not an initial condition but an address in the function program block. The *D*-block cells contain addresses of $y^{(j)}(0)$, $j \geqq k$, stored in order, so that these may be treated in a manner similar to initial conditions.

(*f*) *A_1-block, A_2-block, and Instruction block I.* The roles played by these blocks are as described in connection with Figs. 4 and 5. In up-dating we add new terms as prescribed by Table II. In down-dating we eliminate terms that are no longer needed.

(*g*) *Factor block, F.* This saves all completed non-trivial (no zero factors) A_1 chains. In transplanting an A_1 chain into the *F*-block, all ones and minus ones are boiled down to a single sign for the entire product. All ones are omitted from the *F*-block, unless the particular product contains nothing but a single one.

In the flow chart of the 704 program, K stands for the address of some pseudo-order of the y-program currently under examination. The program starts with examination of the last K, $y^{(k)}(0)$. A tag bit in K will indicate that $Dy^{(k)}$ has been found, and is in the *D*-block, so that it need not be found over again when constructing higher derivatives.

The series construction, which involves multiplying each derivative by the appropriate power of $x - x_0$ and dividing by factorials, is straightforward, and is not shown in the flow chart.

The Taylor series solution of the differential equation which is finally obtained is in the form of a sequence of pseudo-instructions. It is always possible to convert these and print them on paper using familiar mathematical symbols, but we do not do this and will hardly ever want to. If a differential equation is sufficiently complicated to warrant using the program, the chances are that any significant information in the expression for the solution will be hidden in its complexity. If w is some complicated function of x, y, and z, $w = f(x, y, z)$, and we want to find out how w depends on x, it will do no good to inspect the expression f. Instead, we picture w versus x by evaluating f for a range of x values. Similarly, if we wish to study w versus y, we evaluate f with a range of y values. The point is, we need find the program for f only once. We may then evaluate it numerically as many times as we wish, illuminating the dependence on any desired variables.

To evaluate the solution obtained, it is necessary to convert the pseudo-code program to a regular machine-language code, and to supply numeric data. This is done by interpretive and output routines. The flow of information is shown in Fig. 7.

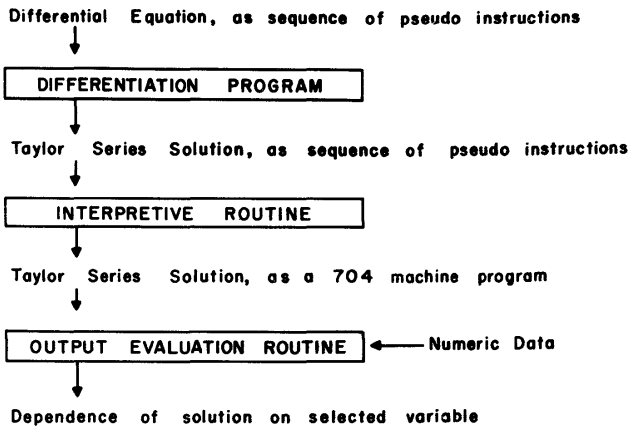

Fig. 7—Data flow for solution of differential equation.

An example of the solution of a differential equation, showing the kind of information that can be obtained from these solutions, is shown in Figs. 8, 9, and 10.

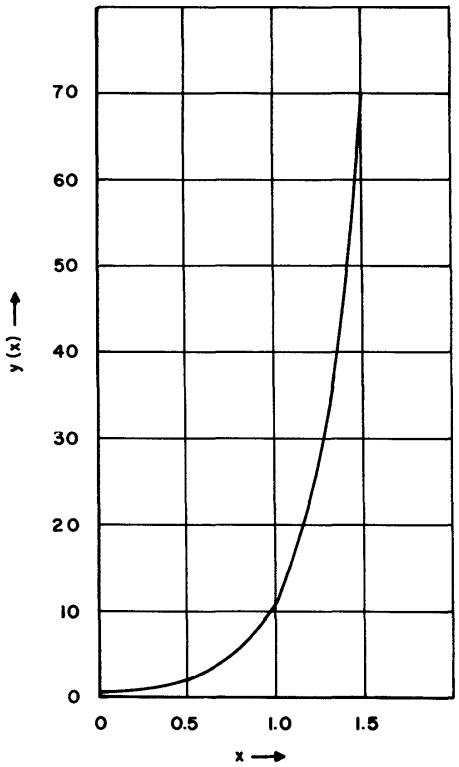

Fig. 8—Eight terms of series solution of $\frac{dy}{dx} = y^a + x$, $y(0) = 1$, $a = 2$.

Conclusion

We have described an analytic method for finding a series solution of ordinary differential equations on

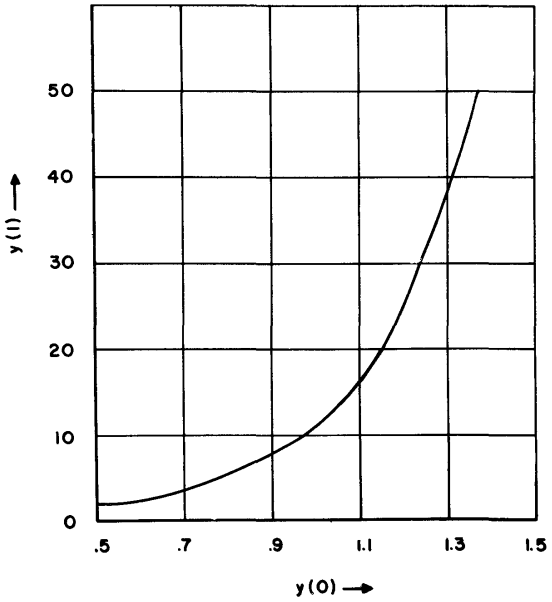

Fig. 9—Eight terms of series solution of $\frac{dy}{dx} = y^a + x$, $x = 1$, $a = 2$.

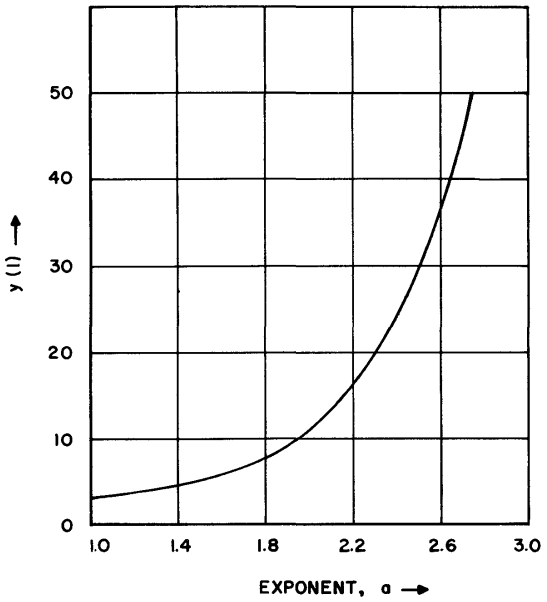

Fig. 10—Eight terms of series solution of $\frac{dy}{dx} = y^a + x$, $y(0) = 1$, $x = 1$.

a stored-program digital computer. Note, however, that the present IBM 704 program for implementing this method has room for improvement. Indeed, in the present program little attempt is made to simplify the generated derivative expression. This is a severe waste of storage capacity, and unduly limits the number of series terms that can be found. Further, the unsimplified expression, containing redundant and irrelevant terms, increases the machine time for evaluating a solution. For this reason, we cannot now obtain a significant estimate of the merit of the analytic method in comparison to conventional numeric techniques.

The method should be useful in illuminating local properties of solutions. It also appears to lend itself to extending solutions by analytic continuation, but this is a problem that has not yet been attacked.

Another needed improvement, if we are to handle the differential equations of electrical engineering practice, is the capability of handling simultaneous equations. The obvious modification to do this is to provide a separate function program and *D*-block for each differential equation of the system.

Acknowledgment

The research reported in this paper has been sponsored by the Electronics Research Directorate of the Air Force Cambridge Research Center, Air Research and Development Command, under contract AF 19(604)-4152.

The idea of using the computer to develop the series solution of a differential equation occurred in conversation with Ramon Alonzo.

The author also wishes to express his gratitude to Albert G. Engelhardt for substantial help in the planning and development of his work.