THE EVOLUTION OF SYNTHETIC CALCULUS: A MATHEMATICAL TECHNOLOGY FOR ADVANCED ARCHITECTURE

by

JOSEPH THAMES
THE AEROSPACE CORPORATION

ABSTRACT

Synthetic Calculus is a mature "instruction set architecture" that has evolved through many stages of software resynthesis over a period of about 20 years. It has been the foundation of a series of a very-high-level scientific programming languages incorporating built-in tools for mathematical optimization. Its basic mechanism is a fully dynamic process of differential arithmetic employing tagged data structures and generic operators. This "extended-value" arithmetic produces exact values of partial derivatives of the output with respect to designated input of computer programs as a by-product of their execution. As an arithmetic, rather than an algebra, this mechanism is capable of differentiating numerical algorithms in execution, producing exact partial derivatives of numerical solutions which have no formula expression amenable to analytic differentiation.

INTRODUCTION

The application of differential calculus, even in the era of digital computing, retains an overwhelming flavor of analytic or algebraic mathematics. There has been little emphasis on the synthetic or arithmetic side of calculus. This is unfortunate because the inability to partially differentiate large simulation models as they are executing is all that really stands in the way of routinely achieving solutions by closed-form induction. The word "induction" is used as a generic term for the iterative parameter searching processes that underlie algorithms for optimization, regression, differential correction, root finding, process identification, etc.

In spite of the ability of algebraic manipulators like MACSYMA (Refs. 6,7) to automatically derive differential formulas, partial differentiation remains a barrier in engineering involving large systems of cascaded simultaneous algebraic and/or differential equations that cannot be solved analytically, i.e., by symbol manipulation (see page 9). This scenario describes simulation programs that are commonly used for experimental (trial and error) induction with computers, to determine the design input parameters which drive the simulators. But closed-form parameter induction may be achieved by employing optimization as a mathematical balancing mechanism to find the combined values of these parameters which locate the roots of the system.

Since the late sixties, medium-sized problems in this class have been solved by very-high-level programming languages that appear similar to FORTRAN in nearly all of the syntactic statements making up programs. But these algebra-level statements only express the equations of simulation models which constitute the operand structure of calculus problems. The operation statements which control the closed-form induction processes represent a higher level of language than algebra, but this higher level is not very prominent in program syntax, since there may be only one such statement in a program. Yet the impact on language semantics is pervasive, because the arithmetic itself is extended to perform differentiation.

The full treatment of the computational theory of synthetic calculus is beyond the scope of this introductory paper. But its scope can be inferred by the fundamental mathematical tool that it renders for the first time as a built-in numerical operator, namely the multidimensional form of the Taylor series. As this basic operator is the design foundation for virtually all algorithms of numerical approximation, synthetic calculus, as its medium of application, applies to the entire field of mathematical computation.

The focus of this paper is on the foundations of this computational paradigm, namely the process of synthetic differentiation and the procedural mechanism through which it is applied for parametric induction. This procedural mechanism is a hierarchic exchange operation corresponding to a procedure call in a software hierarchy mechanized for nested differential induction.

1.0 Mathematical Architecture

Differential Induction Process - The general solution process of differential induction structurally corresponds to Newton's method of solving systems of implicit nonlinear equations. In essence this process involves three distinct steps:

(a) Generation of a linear system (matrix) of partial derivatives from the nonlinear equations at a point in parameter space,

(b) Solution of the linear system to estimate the unknowns of the nonlinear equations to find another point in the parameter space nearer to a "balance point",

(c) Iteration by repeating steps (a) and (b) until convergence to the balance point is achieved.

Step (a) in this process is the vital link between the algebra of a model and the methods of solving parameter induction problems. The overall efficiency of this process is governed by the speed of step (a) and the number of iterations, step (c), which in turn is governed by the accuracy of the derivatives produced in step (a).

The primary impediment to the routine solution of problems in this class is the necessity of computing partial derivatives of each output variable of a simulation model with respect to each input variable. This requirement greatly magnifies the arithmetic and storage load because each output variable produced must be accompanied by an array of output derivative values. But the rate of magnification in arithmetic is much smaller than with finite differencing, which is sometimes used to generate approximate partials. This is because differencing requires multiple function evaluations per differenced independent variable. Moreover, differencing schemes, although easy to program, fail to meet the accuracy requirement in general.

Induction Call Concept - The operation statements of synthetic calculus invoke a procedure context in which the output of each formula is differentiated with respect to its inputs, and the input derivatives are in turn the result of chain-rule propagated differentiation, so that the partial derivatives are ultimately with respect to the original input parameters of the simulation model. These parameters are identified in an induction call statement, such as:

FIND ⟨parameters⟩ IN ⟨model⟩ BY ⟨solver⟩
MATCHING ⟨equality constraint variables⟩
HOLDING ⟨inequality constraint variables⟩
TO MAXIMIZE ⟨objective variable⟩

which dynamically poses a mathematical programming problem. The ⟨parameters⟩ constitute the unknown independent variables of the equations contained in the ⟨model⟩ procedure, whose initial values are used to start the induction. Consequently, they are treated as read-only in the ⟨model⟩ context.

The library ⟨solver⟩ procedure is a numerical search algorithm which controls the induction. It is actually the element that is directly called. It first sets the required differentiation context for the ⟨model⟩ and executes the ⟨model⟩ producing the derivatives of all of its dependent variables with respect to the ⟨parameters⟩. The variables in the indented lines of the statement identify the dependent variables computed in the ⟨model⟩ whose defined conditions constitute the mathematical balance to be achieved by the ⟨solver⟩. The ⟨solver⟩ uses the derivatives of these variables to estimate and assign new values to the ⟨parameters⟩ and continues the iterative process until the balance point satisfying the balance conditions is found. After the return of the call, the problem connection has been broken, the derivatives destroyed, and the ⟨model⟩ may be subsequently executed as the operand of another induction call, or may be called as an ordinary algebraic subroutine.

The dependent variables in the ⟨model⟩ are dynamic linkages of values ordered hierarchically, i.e., typically consisting of a scalar containing the algebraic value of the variable, a vector containing the first partial derivatives of the variable with respect to the ⟨parameters⟩, and a symmetric triangular matrix containing the associated second partial derivatives. Thinking in geometric terms, it is useful to regard this extended value of a dependent variable as having "size" value and parametric "shape" value. Shape value is the tensorial differential value at the value point in vector parameter space, which infers the direction the parametric search must take to locate a stationary balance point on the function surface.

Extended Value Arithmetic - The differential arithmetic that produces the extended value of the variables is equivalent to the process of derivation that is familiar to us from our college days, with one important distinction. We are used to performing it with symbols, delaying numeric evaluation to the very end, seeking to perform analysis of mathematical structure. But, the tables in the back of any college calculus text are specifications for arithmetic as well as algebra.

When we derive a formula derivative of a function formula, we operate upon its binary terms. We replace the expression for the product of two variables by an expression for the sum of two products, the first times the derivative of the second plus the second times the derivative of the first. But we can, with

established values of the independent variables and accumulated storage of derivative values, perform the arithmetic products and the arithmetic sums on the fly, thereby producing a number immediately. Hence the process of derivation becomes a process of interpretive execution. This is the extended-value process of synthetic differentiation that is the foundation of synthetic calculus. It produces, as the extended value of each dependent variable, the pointwise values of its partial derivatives instead of symbolic formulas for them.

Synthetic calculus as an evolving paradigm of computing technology has been under continuous development for more than twenty years. During this software evolution, the foundation of exact differentiation progressed through phases of development similar to the evolution of programming systems from static compilation to dynamic interpretation.

Static Phases of Differentiation - Analytic differentiation as a phase of computation corresponds to the familiar process of compilation in that it is a phase of static translation. The result is a translated symbol system that may be executed by an algebra-level interpreter (e.g., a FORTRAN machine). The most static process of analytic differentiation corresponds to macro expansion without subroutines. It has the fastest execution because it utilizes the most primitive interpretation process. But it will result in the largest program. Heretofore, the use of this process has generally been limited by small memories to very small problems of few dimensions.

Differentiation Primitives - The beginnings of synthetic differentiation occurred when subroutines were employed to reduce the redundancy of the expanded code. It was recognized that the unary and binary differentiation formulas were a set of primitive subroutines that could be called repeatedly in any derived differentiation system, thereby reducing the large memory requirement. The "object" form of the differentiated system was a set of parsed subroutine calls. This compiler-interpreter sharing of the elements of differentiation was a half-way step between analytic and synthetic differentiation. The rules of differentiation still resided in the compiler and the interpreter was less primitive with the addition of the differentiation primitives. But, the contextual basis of differentiation, the set of independent coordinate variables that were differentiated with respect to, remained statically bound ("hard-wired") in the compiled calling sequences.

Dynamic Differentiation - The most procedurally dynamic form of synthetic differentiation occurs when the rules of differentiation are contained in the interpreter and the creation and pushdown of differentiation activation records is invoked by a call mechanism. In this case, no derived program exists as such. Instead, the original program is differentiated synthetically as it is executed.

The Architecture Barrier - The problem of calculus computing technology now centers upon the unsuitability of conventional computer architecture for calculus processes. The procedural dynamics of differential arithmetic, whose mathematical power is exemplified in the PROSE language (Refs. 20-29) cannot be fully compiled using analytic (algebraic) methods, because of the limitations of static binding discussed above. In order to effect the mathematical power of dynamically nested calculus contexts in languages like PROSE, a fully dynamic arithmetic of synthetic differentiation is necessary. This is the method that underlies PROSE's implementation, but it is achieved entirely via software interpretation. The major challenge to architecture design is the achievement of differentiation via hardware interpretation, exploiting the inherent vector parallelism involved in this type of arithmetic.

In a pure software implementation of synthetic differentiation, at least one order of magnitude in processing speed is sacrificed even in the "best case" situation. But an additional order of magnitude in overhead is inherent in the mismatch of conventional architectures to compute-bound processes involving large, complex data-structures (e.g. tensor partial-derivative values), thus magnifying the loss due to software interpretation. The net penalty is greater than 200 to 1 on conventional machines of the IBM 370 class. A principal aim of the design of the Aerospace Research Computer (ARC - Ref. 1) is to eliminate these penalties.

The experience of synthetic calculus has reinforced the major architectural concepts of the ARC design, since the present synthetic calculus implementations are software versions of tagged architecture with generic operators. This dynamic interpretive structure has evolved out of mathematical necessity from the more conventional "compiled" metacomputer structure. Variables in this architecture may change roles dynamically in that they may be independent, hence read-only, in one context and dependent in another. In contexts where no differentiation is in process, the same variables have no mathematical type, and may contain non-numerical data. Thus variables must be tagged with attribute fields to qualify their interpretation.

Likewise the semantics of the operators are context dependent. A multiply operator may perform scalar multiplication in one context or may perform differential multiplication of varying order with respect to arbitrary coordinate vectors in other contexts. Yet the procedure containing the operator instructions may be the same one in the several contexts. This necessitates that the operator semantics be changed dynamically by examination of the (dynamic) type of the operands.

2.0 Synthetic Differentiation

The procedure which has been the foundation of synthetic calculus was first introduced in the 1962-1964 period independently by R. E. Wengert of GE (Ref. 2) and M. W. Alford of TRW (Refs. 3, 4). In the early language development efforts, Alford's method was utilized. Later Wengert's method, being more purely interpretive, was adopted.

Since the objectives of synthetic differentiation, in contrast to analytic differentiation, is not the derivation of derivative formulas, nowhere in the associated model does an analytic differential formula necessarily exist. The differentiation operations are subsumed within the arithmetic operators. The operators are interpreted as ordinary arithmetic operators or as extended differential arithmetic operators depending upon the dynamic context within which the mathematical model is being executed. The synthetic process is performed by the chain rule of partial differentiation. In Alford's method this synthetic step is interleaved with analytically produced formulas of considerable size in "formula modules". In Wengert's method the chain rule is implicit in the primitive differential operators that are invoked by each arithmetic operator resulting in interleaved function and differential arithmetic. The derivative arithmetic may be extended to arbitrary orders of differentiation.

Differential Program Interpretation

An example of this process, as it is performed in the semantics of the PROSE language, is illustrated in the nine execution-state frames of Figure 1. A simple PROSE program is shown on the left. Each successive frame illustrates the results of the execution of respective lines. These steps are explained as follows:

(1) PROBLEM .SHOW - The program heading statement begins execution with all variables set to zero.

(2) X=1 Y=2 A=4 - These are ordinary assignment statements serving to initialize the as yet undesignated independent variables X and Y, and a constant A.

(3) FIND X, Y IN .TEST TO MATCH S - This is an induction call statement that creates a differentiation context basis in temporary storage and transmits control to a default induction algorithm, a Newton-Gauss-Gram-Schmidt routine (not-shown) designed to search for a smallest least-squares solution. The variables X and Y are established as independent, thus all arithmetic in the subprogram .TEST will be differentiated with respect to them each time it is called by the algorithm (only the first iteration is illustrated). Derivatives produced during each iteration will be added to a "stack" whose initial elements are vectors containing the values and the derivatives of the independent variables with respect to themselves. The width of the stack is NSIZE+1 in the case of first-order differentiation, where NSIZE is the number of independent variables.

(4) MODEL .TEST T=A+1 - The first subprogram assignment appears on this line but does not result in a differentiation since it does not depend upon either of the basis variables.

(5) Q=2*X**2+3*Y**3 - This is the first formula differentiated. It adds a vector to the temporary storage stack, and places a pointer to it in the original address of the dependent variable.

(6) R=A/Q**2 - The derivatives of the second formula differentiated are propagated via chaining from the previous derivatives in the stack.

(7) S=(3*R-Q)/T - Each new formula adds or updates (if repeated) a vector record in temporary storage and adds a tag indicating its status (i.e., whether it is a variable or a constant). This formula defines the induction criterion variable whose stationary point constitutes the mathematical balance of the induction.

(8) END - The termination of the model subprogram returns control to the induction algorithm which utilizes the derivatives $\partial S/\partial X$ and $\partial S/\partial Y$ to estimate the new values of X and Y and reinitialize the differential context basis.

(9) END - After the final (converged) iteration, the induction algorithm deactivates differentiation, moves the variable values back to their original addresses, releases the temporary storage stack, and completes the induction call by returning control to its following statement.

This example illustrates the dynamic nature of the synthetic differentiation process and the fact that the original program is differentiated without prior analytic differentiation. But it does not fully illustrate the arithmetic of differentiation. For a closer understanding, we must look at an arithmetic operator, as illustrated in Figure 2. This is a first-order multiplication operator which would be called by the interpreter executive whenever a binary multiplication is to be performed.

In the PROSE interpreter, all arithmetic operators are more complicated than this because

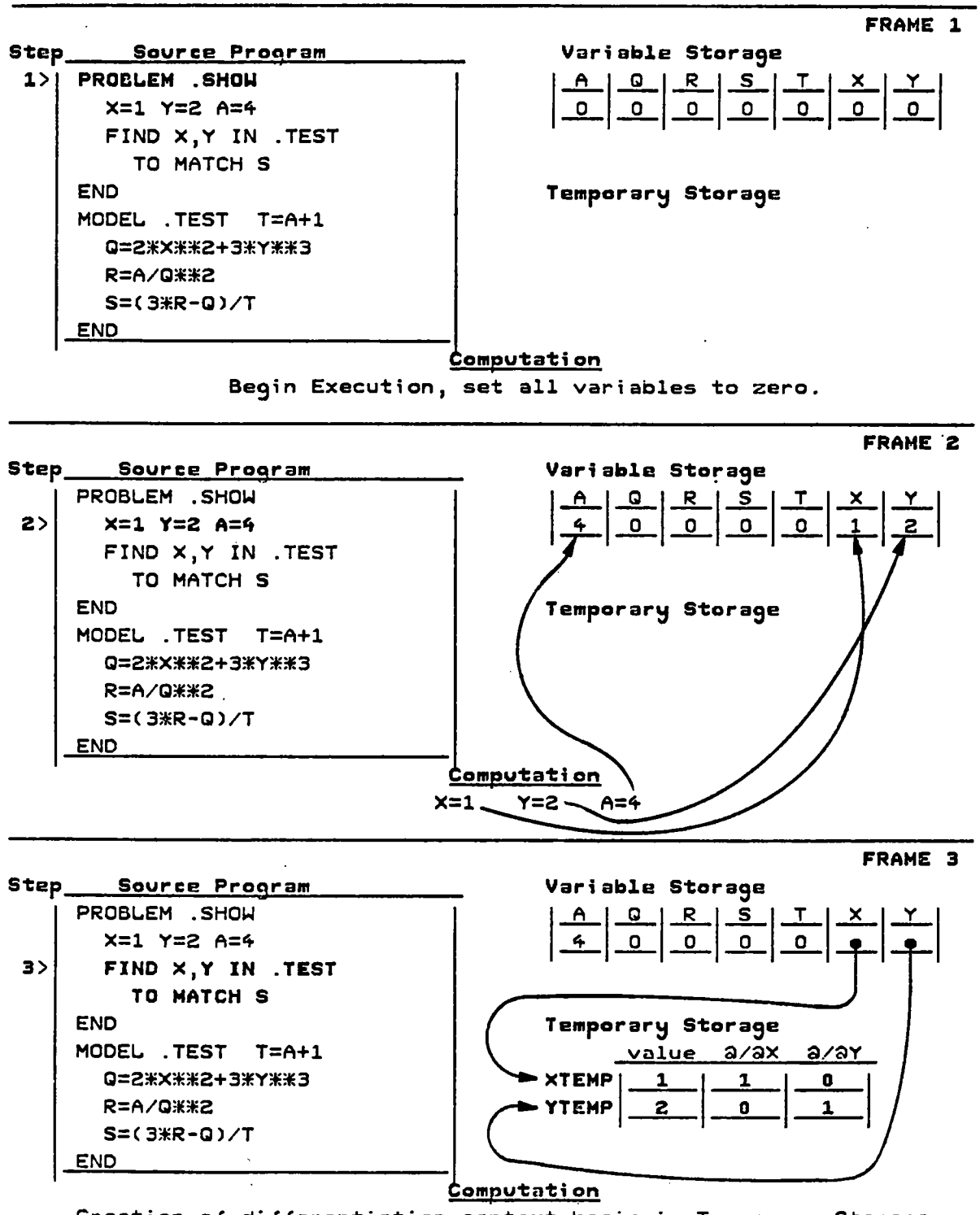


Figure 1. Synthetic Differentiation in PROSE Language Execution

FRAME 4

```
Step    Source Program
        PROBLEM .SHOW
          X=1 Y=2 A=4
          FIND X,Y IN .TEST
            TO MATCH S
        END
 4>     MODEL .TEST  T=A+1
          Q=2*X**2+3*Y**3
          R=A/Q**2
          S=(3*R-Q)/T
        END
```

Variable Storage

A	Q	R	S	T	X	Y
4	0	0	0	5	•	•

Temporary Storage

	value	∂/∂X	∂/∂Y
XTEMP	1	1	0
YTEMP	2	0	1

Computation

T=4+1=5

FRAME 5

```
Step    Source Program
        PROBLEM .SHOW
          X=1 Y=2 A=4
          FIND X,Y IN .TEST
            TO MATCH S
        END
        MODEL .TEST  T=A+1
 5>       Q=2*X**2+3*Y**3
          R=A/Q**2
          S=(3*R-Q)/T
        END
```

Variable Storage

A	Q	R	S	T	X	Y
4	•	0	0	5	•	•

Temporary Storage

	value	∂/∂X	∂/∂Y
XTEMP	1	1	0
YTEMP	2	0	1
QTEMP	26	4	36

Computation

Q = 2(1)(1) + 3(2)(2)(2) = 2 + 24 = 26

∂Q/∂X = 4x = 4(1) = 4

∂Q/∂Y = 9y(y) = 9(4) = 36

FRAME 6

```
Step    Source Program
        PROBLEM .SHOW
          X=1 Y=2 A=4
          FIND X,Y IN .TEST
            TO MATCH S
        END
        MODEL .TEST  T=A+1
          Q=2*X**2+3*Y**3
 6>       R=A/Q**2
          S=(3*R-Q)/T
        END
```

Variable Storage

A	Q	R	S	T	X	Y
4	•	•	0	5	•	•

Temporary Storage

	value	∂/∂X	∂/∂Y
XTEMP	1	1	0
YTEMP	2	0	1
QTEMP	26	4	36
RTEMP	.0059	-.002	-.017

Computation

R = 4/26/26 = .00592

∂R/∂X = (∂R/∂Q)(∂Q/∂X) = 4(-2A)/Q/Q/Q = 4(-8)/26/26/26 = -.00182

∂R/∂Y = (∂R/∂Q)(∂Q/∂Y) = 36(-8)/26/26/26 = -.01638

FRAME 7

Step — Source Program

```
   PROBLEM .SHOW
     X=1 Y=2 A=4
     FIND X,Y IN .TEST
       TO MATCH S
   END
   MODEL .TEST  T=A+1
     Q=2*X**2+3*Y**3
     R=A/Q**2
7>   S=(3*R-Q)/T
   END
```

Variable Storage

A	Q	R	S	T	X	Y
4	•	•	•	5	•	•

Temporary Storage

	value	∂/∂X	∂/∂Y
XTEMP	1	1	0
YTEMP	2	0	1
QTEMP	26	4	36
RTEMP	.0059	-.002	-.017
STEMP	-5.20	-.801	-7.05

Computation

$S = (3(.00572)-26)/5 = -25.983/5 = -5.1966$

$\partial S/\partial X = (\partial S/\partial R)(\partial R/\partial X)+(\partial S/\partial Q)(\partial Q/\partial X) = 3(-.00182)+(-1)(4)/5 = -0.8011$

$\partial S/\partial Y = (\partial S/\partial R)(\partial R/\partial Y)+(\partial S/\partial Q)(\partial Q/\partial Y) = 3(-.01638)+(-1)(36)/5 = -7.0496$

FRAME 8

Step — Source Program

```
   PROBLEM .SHOW
     X=1 Y=2 A=4
     FIND X,Y IN .TEST
       TO MATCH S
   END
   MODEL .TEST  T=A+1
     Q=2*X**2+3*Y**3
     R=A/Q**2
     S=(3*R-Q)/T
8> END
```

Variable Storage

A	Q	R	S	T	X	Y
4	26	.01	-5.	5	•	•

Temporary Storage

	value	∂/∂X	∂/∂Y
XTEMP	.9209	1	0
YTEMP	1.288	0	1

Induction Algorithm

Computation

Execution of induction algorithm resulting in new X and Y for next iteration and reinitialization of differentiation context stack.

FRAME 9

Step — Source Program

```
   PROBLEM .SHOW
     X=1 Y=2 A=4
     FIND X,Y IN .TEST
       TO MATCH S
9> END
   MODEL .TEST  T=A+1
     Q=2*X**2+3*Y**3
     R=A/Q**2
     S=(3*R-S)/T
   END
```

Variable Storage

A	Q	R	S	T	X	Y
4	2.3	.76	0	5	.76	.73

Temporary Storage

Computation

Convergence - values returned to variable storage, temporary released.

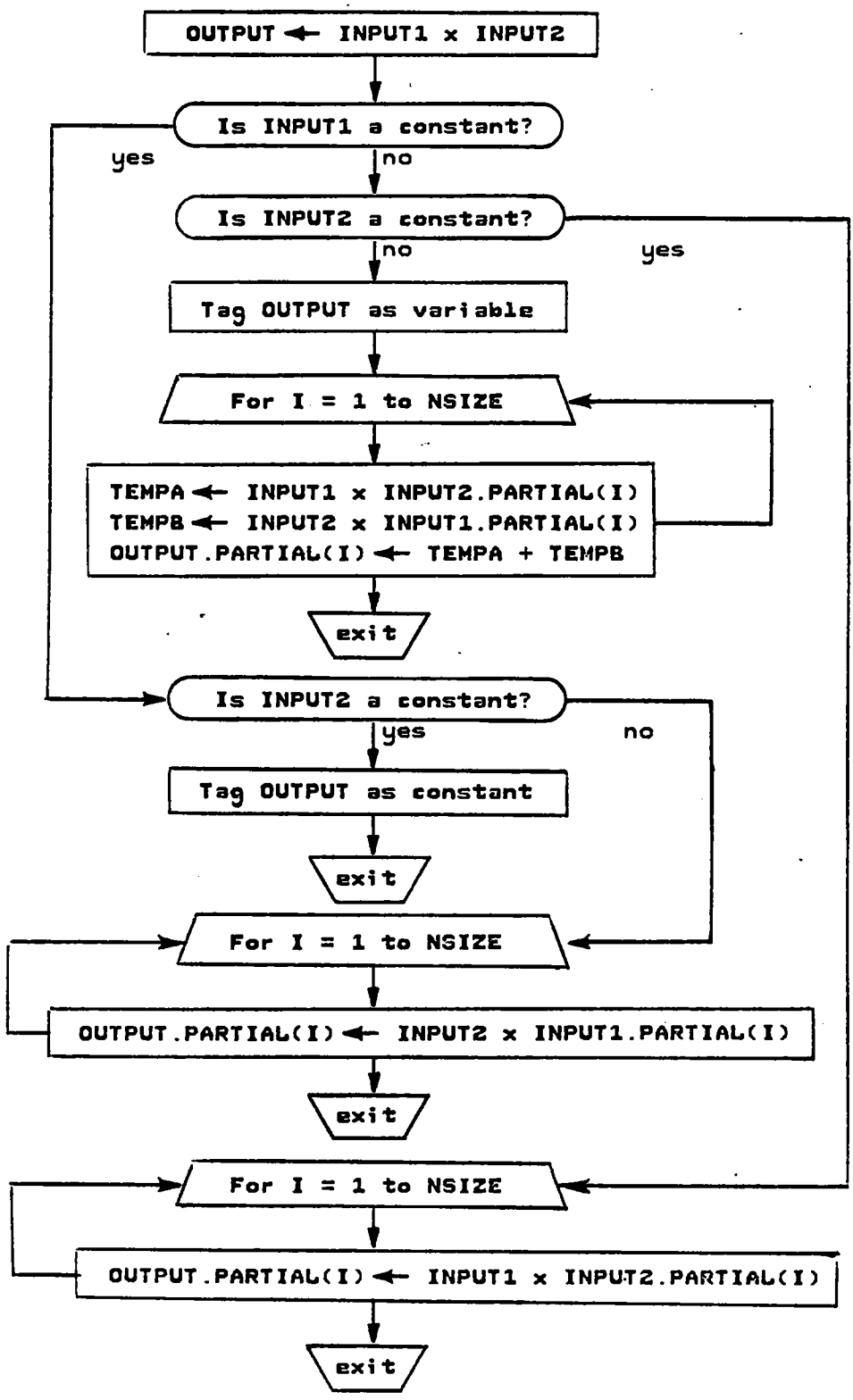


Figure 2. First Order Multiply Operator

they contain explicit code for computing second-order partials (the Hessian matrices) as well as first-order partials (the gradient vectors). However, in future implementations of synthetic calculus in the Aerospace Research Computer, only first-order operators will be necessary. With a recursion-oriented vector-of-vectors data structure, these operators can be employed recursively to produce mixed-indefinite-order "fluxion tensors". A fluxion tensor is a tagged sparse data structure whose mixed-order of differential elements is in proportion to their nonlinear significance. For example, if a dependent variable is quadratic in its dependence on one independent variable and linear with respect to another, it would have first and second-order partials with respect to the former but only first order partials with respect to the latter. This example serves to illustrate the importance of dynamic data storage allocation and flexible tagged architectures in the further development of synthetic calculus.

Differential Metacomputer Evolution

The fully dynamic mode of differential interpretation has evolved from the static (analytic) mode in a series of stages of software resynthesis. The distinctive character of the evolution reinforces an emergent pattern of economical logic design that is as important in application development as it is in the development of basic machines. This is the natural emergence of programmable tools or "metacomputers."

Submergence of Invariant Program Structure - The original programs used for differential correction (Ref. 8) were written in assembly language, which greatly limited their flexibility for change. However, it was soon recognized that only certain portions of the programs were subject to frequent change -- the engineering simulation models. The remaining mathematical algorithms were much more stable and could be regarded as invariant. Consequently, the first suggestion of two distinct software levels emerged, i.e., engineering models and numerical solution algorithms.

Coupling of Extended-Value Semantics - The separation of levels presented little difficulty for simulation, where the mathematical coupling between the two levels was relatively simple. However, a much greater difficulty was encountered for inductive differential correction because partial-derivative values were required from the system model by the differential-correction algorithms. This requirement led to the development of synthetic methods for computing derivative values from large systems of model formulas.

Building Blocks of a Modeling Alphabet - In order to minimize software interpretation, modeling alphabets of very-high-level arithmetic primitives were developed. Each primitive consisted of formula-level (multiterm) and multi-formula-level arithmetic, rather than term-level arithmetic. This enabled the use of a greater concentration of direct-machine code in each primitive formula module (See Figure 3).

Ubiquitous Plug Structure - Since the analytic derivative formulas produced derivatives that were local to each module, a chain rule mechanism for synthetic differentiation was interleaved between module calls in order to synthesize the global derivatives of the entire model. This mechanism, illustrated in Figure 4, was a simple matrix multiplication that updated a global array of derivatives values to propagate the influence of the local values of each module. With this mechanism, the synthetic differentiation process became ubiquitous, since an entire model (sequence of formula modules) could be treated as a module in a synthetic hierarchy of models, ad infinitum.

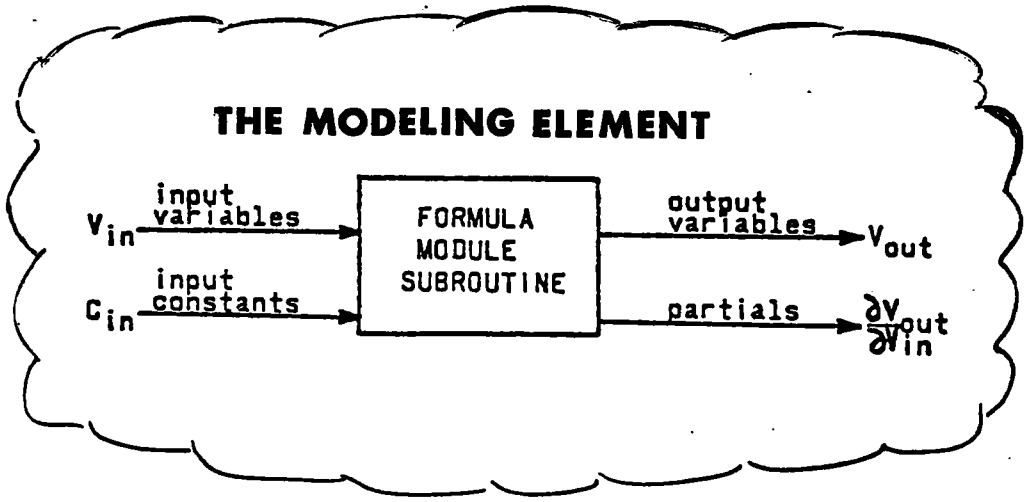
This aspect of synthetic differentiation overcame a subtle but very serious limitation of analytic differentiation that will ultimately turn out to have been the "dead-end" of analytic calculus -- the necessity of having an output function formula to differentiate. In all but the simplest of engineering problems, it is necessary to "differentiate a black box", where for example the black box might contain a process of numerical integration or the solution of a set of implicit equations. The output of the black box is a number or set of numbers, representing a discrete value of the function, as likewise is its input. There is no formula to operate on analytically. But if the black box is a programmed algorithm, whose primitives are arithmetic, then its derivatives may be produced as a by-product of its execution, via extended-value arithmetic.

Programmable Metacomputers - With the development of libraries of formula modules came the beginnings of a new level of programming by module assembly. A series of computer programs were developed which were themselves programmable. With the mechanism of synthetic differentiation built into the semantics of the model op-codes, the engineering metaprograms, defined in "model assembly language" could be executed with different orders of differentiation. One could choose to evaluate first derivatives for least-squares correlation or nonlinear equation solving or first and second derivatives for nonlinear optimization or optimal control. As a result, some of the more general metacomputer codes were capable of operating at three different levels of arithmetic corresponding to:

(1) Simulation no derivatives (zero-order arithmetic)

(2) Correlation first derivatives (first-order arithmetic)

(3) Optimization second derivatives (second-order arithmetic)



Typical Formula Module Subroutine

Identification: P30052

Name: Ideal Velocity Gain Module

Inputs: X - Specific Impulse (ISP)
Y - Mass Ratio (MR)

Output: Z - Velocity Increment (DELTAV)

Arithmetic:

Syntax Level:	Z=32.174*X*LOGN(Y)
Context Level 1:	ZX=32.174*LOGN(Y)
(1st Partials)	ZY=32.174*X/Y
Context Level 2:	ZXX=0
(2nd Partials)	ZXY=32.174/Y
	ZYY=-32.174*X/Y**2

Figure 3. Very-High-Level Differential Arithmetic Primitives

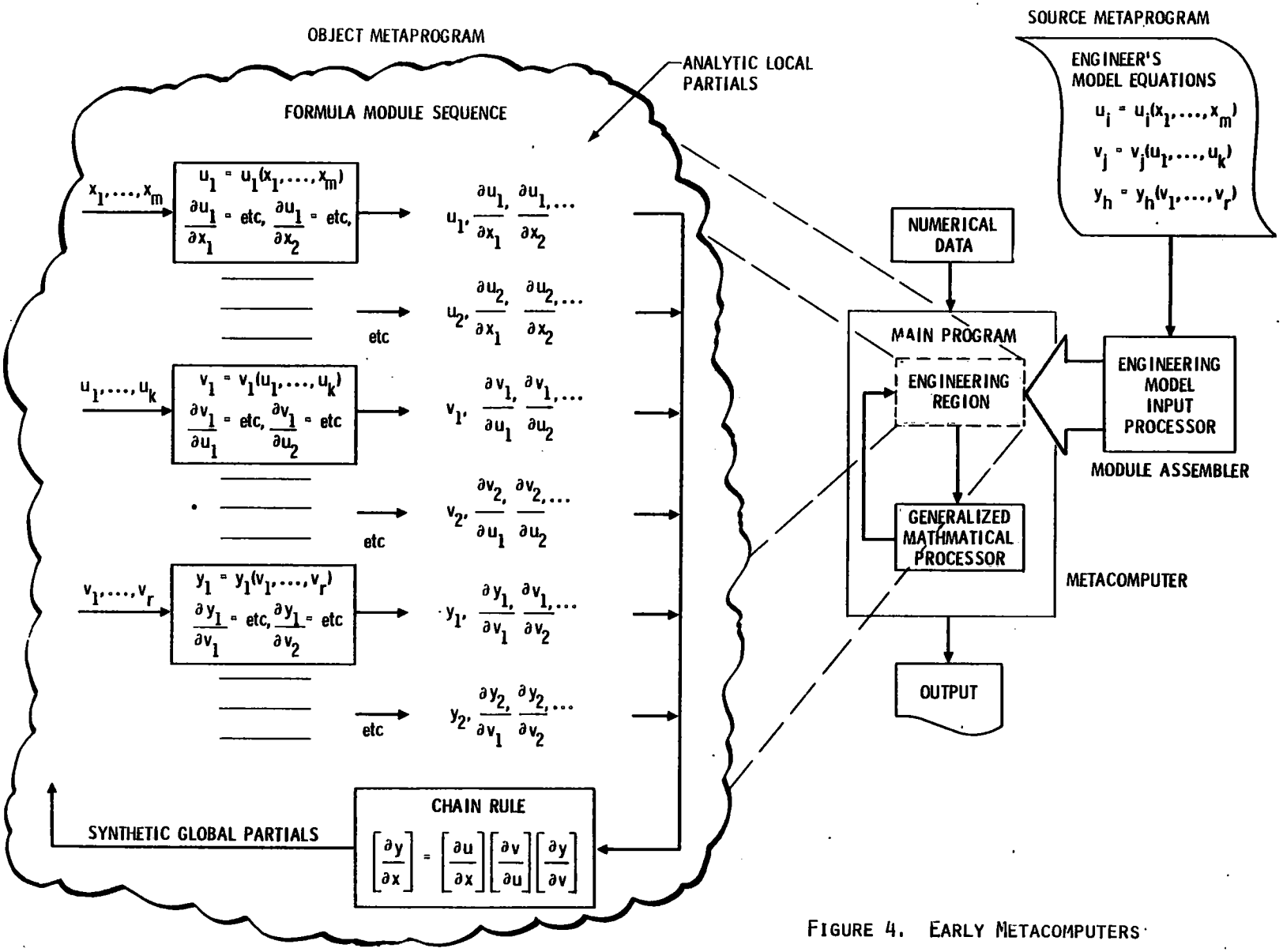


FIGURE 4. EARLY METACOMPUTERS

Each level resulted in higher "band-width" of data to be stored in the computer. At level (1) the band-width included only the output value of each modeled formula. At level (2) it expanded to include the gradient vector. At level (3) it expanded to include the upper-triangular Hessian matrix as well as the gradient vector.

Evolution of Metaprogramming Language - Following a period of extensive usage which illuminated its limitations of fixed metacomputer structure (Refs. 9-14), this strategy of program design was broadened to provide greater applicability. A new universal class of special-purpose metacomputers were evolved which separated naturally into tools for solving specific problems in the general problem classes of simulation, correlation, and optimization. These tools were used to solve numerous problems by teams of systems engineers over a period of several years. From this experience, a greater perspective of computational theory evolved, resulting in the notion of a general metaprogramming language that would be a resynthesis of all that had been learned.

The objective of this resynthesis was to disassemble all of the monolithic structure of the metacomputers and place the constituent tool modules in a library. A general-purpose metaprogramming language was to be developed that would permit engineers to assemble ad hoc metacomputers for a very large class of system problems. Over the period between 1967 through 1976 a series of such languages evolved. The initial languages, MC (NASA, Ref. 13) and MODTRAN (TRW, Ref. 15), were syntactic copies of FORTRAN with additional operators added for calculus operations.

Calculus-Level Languages - In 1968, a macro processor ML/I (Ref. 16) was employed to explore syntactic forms of "problem-structured" programming at the calculus level of expression. The result was SLANG (Refs. 17-19). In 1974, PROSE (Refs. 20-29), a more comprehensive metaprogramming language was introduced into the commercial market place on service-bureau computers. In 1975 a time-sharing version was introduced, and further algorithmic enhancements were made until 1976. In 1978 a user group was formed with charter members representing many of the Fortune 500 companies and a few universities.

3.0 Summary and Future Strategy

In a twenty-year history of almost continuous evolution of synthetic calculus, the limit of what was possible with prevalent architecture has been reached. Further progress cannot be made without the development of metacomputer-oriented architecture capable of high-speed movement of dynamic array structures (e.g. partial derivatives) and arithmetic for synthetic differentiation. Figure 5 is a scoreboard of capabilities of the various metacomputer systems that evolved in the synthetic calculus paradigm. None of the systems that were ever implemented completely covered the scope of this field.

Based upon this twenty years of intensive research, synthetic calculus has earned the status of proven technology. Perhaps the most significant lesson that has been learned is that by submerging the complexity and tedium of mathematics into the invisible part of the computer, we can greatly enhance the use of mathematics in engineering and science, not to mention education.

Closing the Mathematical Computing Gap - An unfortunate trend of the last two decades has been a divergence of computer-architecture design from the challenge of advanced mathematics. Mathematical computing has been relegated to the upper strata of programming, while the major work of computer science has been devoted to operating systems and mathematically-limited programming systems. While the application needs for the use of more powerful and more unified mathematical tools have become acute, scientific computing has been hobbled and enmired in prevailing computing environments. Synthetic calculus provides an avenue of escape from this web of complexity.

At a time when software development costs dwarf computer costs, it is time for a reassessment and a redirection of computing investments. Synthetic calculus presents an opportunity to rebalance the scale of computing cost by building mathematics (as opposed to only arithmetic) into the machines of the future and simplifying the process of programming. Wholesale application of this technology can result in general and substantial cost reductions in all engineering projects which depend upon mathematical computing.

REFERENCES

1. Speckhard, A.E., and Fleming, R.C., "The Aerospace Research Computer, A Reconfigurable High-Level Language Machine", Proceedings of the Workshop on High-Level Language Computer Architecture, Los Angeles, CA, October 7,8 and 9, 1981.

2. Wengert, R.E., "A Simple Automatic Derivative Evaluation Program", Communication of the ACM 7,8 (August 1964), pp 463-464.

3. Alford, M.W., "Mathematical Aspects of the Systems Optimization Programs", TRW Report 68.4711.4-90, July 15, 1968.

4. Alford, M.W. "Mathematical Aspects of the Flight Analysis Computer Programs", AIAA Paper 68.583, Fourth Propulsion Joint Specialist Conference, Cleveland, Ohio, June 10-14, 1968.

SYNTHETIC CALCULUS LANGUAGES	MODTRAN-ROP	MODTRAN-MAFIA	MODTRAN-GOP	SLANG I (Refs. 17-18)	SLANG II (Ref. 15)	PROSE 1.0 (Ref. 22)	TSPROSE (Refs. 20,23-26)	PROSE 2.0 (Refs. 24-26)	PROSE-EPOC (Refs. 27,28)	PROSE 3.0 (Ref. 29)
SIMULATION										
Static	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX
Dynamic		XX	XX	XX	XX	XX	XX	XX	XX	XX
Continuous		XX	XX	XX	XX	XX	XX	XX	XX	XX
Unconstrained		XX	XX	XX	XX	XX	XX	XX	XX	XX
Constrained							XX	XX	XX	XX
Discrete										XX
CORRELATION										
Static	XX		XX	XX	XX	XX	XX	XX	XX	XX
Unconstrained	XX		XX	XX	XX	XX	XX	XX	XX	XX
Constrained							XX	XX	XX	XX
Dynamic		XX	XX							XX
Batch		XX	XX							XX
Sequential		XX	XX							XX
OPTIMIZATION										
Static	XX		XX	XX	XX	XX	XX	XX	XX	XX
Unconstrained	XX		XX	XX	XX	XX	XX	XX	XX	XX
Constrained	XX		XX	XX	XX	XX	XX	XX	XX	XX
Equality	XX		XX	XX	XX	XX	XX	XX	XX	XX
Inequality						XX	XX	XX	XX	XX
Continuous						XX	XX	XX	XX	XX
Discrete							XX	XX	XX	XX
Dynamic			XX						XX	XX
Continuous			XX						XX	XX
Discrete									XX	XX

Figure 5. Mathematical Scoreboard of Synthetic Calculus Tools

5. Alford, M.W. and Lear, C.W. "A Computational Method for the Optimization of Multistage Ballistic Missile Systems", AIAA Journal, Vol. 4, No. 9, September 1966.

6. MACSYMA Reference Manual, MIT Laboratory for Computer Science, 12/77.

7. Pavelle, R. Rothstein, M., and Fitch, J. "Computer Algebra" Scientific American, Vol. 245, No. 6. (Dec 1981), p 136.

8. Thames, J.M. "Flight Analysis of the Apollo Propulsion Systems", NASA Program Apollo Working Paper 1196, March 8, 1966.

9. Alford, M.W., "A Computationally Stable Noise-in-the-State Filtering Algorithm" AIAA Paper 68-887, presented at the AIAA Guidance, Control, and Flight Dynamics Conference, Pasadena, California, August 12-14, 1968.

10. Lear, C.W., "A Method for Estimating Missile Propulsion Performance from Flight Data", Transactions of the Eighth Symposium on Ballistic Missile and Space Technology, Volume I.

11. Lear, C.W. and Powers, C.S., "Flight Instrumentation Data Comparison with Static Test Data", Presented and Third Annual Meeting of the Static Test Working Group of the Chemical Propulsion Information Agency.

12. Norris, J.D., and Vernon, D.W., "Apollo Propulsion System Performance Evaluation", AIAA Paper 68-685, Fourth Propulsion Joint Specialist Conference, Cleveland Ohio, June 10-14, 1968.

13. Hooper, J.C., "Performance Analysis of the Ascent Propulsion Subsystem of the Apollo Spacecraft", NASA Program Apollo Working Paper MSC-03408.

14. Conine, R.D. "Transportation System Optimization Program: Demonstration Problem",. TRW Report 06818-6021-r000, December 11, 1967.

15. McCully, J.D., "The Q Approach to Problem Solving", Proceedings, FJCC 69, AFIPS 1969, pp. 691-699.

16. Brown, P.J., "The ML/I Macro Processor", Communications of the ACM (October 1967).

17. Adamson, D.S., "Introduction to SLANG", TRW Doc. 99900-672-r0-00, 1968.

18. Thames, J.M., "SLANG, A Problem-Solving Language for Continuous-Model Simulation and Optimization", Proceedings, ACM 24th National Conf. (December 1969) pp. 23-41.

19. Adamson, D.S., and Winant, C.W. "A SLANG Simulation of an Initially Strong Shock Wave Downstream of an Infinite Area Change", Proc. Conf. on Applic. of Continuous-System Simulation Languages (June 1969) pp. 231-240.

20. Thames, J.M., "Computing in Calculus", Research/Development, 26,5 (May 1975) pp. 24-30.

21. PROSE General Information Manual, United Computing Systems, Kansas CIty, Mo 64108.

22. PROSE - A General Purpose Higher Level Language, Batch System Guide, Control Data Corp. Pub. No. 840031000 (Jan 1974).

23. PROSE Time-Sharing System Guide, United Computing Systems, Kansas City, Mo 64108.

24. PROSE - A General Purpose Higher Level Language, Procedure Manual, Control Data Corp. Cybernet Services, Pub. No. 84003000 Rev B (Jan 1977).

25. PROSE - A General Purpose Higher Level Language, Calculus Operations Manual, Control Data Corp. Cybernet Services, Pub. No. 84000170 Rev. A (Jan 1977).

26. PROSE - A General Purpose Higher Level Language, Calculus Applications Guide, Control Data Corp. Cybernet Services, Pub. No. 84000170 Rev. A (Jan 1977).

27. McDonough, J.M., and Park, D.E., "A Discrete Maximum Principle Solution to an Optimal Control Formulation of Timberland Management Problems", Presented at the Western Forest Economics' Conference, Wemme, Oregon, May 1975.

28. McDonough, J.M., and Park, D.E., "Nonlinear Optimal Control Approach to Interregional Management of Timber Production and Distribution", Proceedings, Systems Analysis Workshop, Society of American Foresters, University of Georgia, Aug. 11-13, 1975.

29. Robinson, M.N., "PROSE Simulation External Specification", PROSE, Inc. Technical Specification, 5 March 1974.