AD-783 076

THE COMPUTER CAN DIFFERENTIATE TOO

Lee Litzler, et al

Texas University

Prepared for:

Office of Naval Research

April 1974

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D AD 783076

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Center for Cybernetic Studies
The University of Texas

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3. REPORT TITLE

The Computer Can Differentiate Too

4. DESCRIPTIVE NOTES *(Type of report and, inclusive dates)*

5. AUTHOR(S) *(First name, middle initial, last name)*

Lee Litzler
Darwin Klingman

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April 1974	21	9

8a. CONTRACT OR GRANT NO.

N00014-67-A-0126-0008;0009

b. PROJECT NO.

NR 047-021

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

Center for Cybernetic Studies
Research Report CS 176

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

Office of Naval Research (Code 434)
Washington, D. C.

13. ABSTRACT

For mathematical problems whose solution requires the evaluation of first or second partial derivatives of user-defined functions, symbolic differentiation is offered as a viable alternative to numerical differencing techniques and user-written subprograms. The paper describes an efficient user-oriented approach, equivalent to symbolic differentiation, for the generation of code capable of evaluating partial derivatives of user-specified functions. Computational results of an experimental implementation of the method are reported. The feasibility of this approach is additionally demonstrated by interfacint the experimental code with the Sequential Unconstrained Minimization Technique of Fiacco and McCormick for the solution of nonlinear programming problems and results are reported.

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Symbolic differentiation						
Analytic differentiation						
Numerical differencing						
User-oriented						
Nonlinear programming						
Code optimization						

Research Report
CS 176

THE COMPUTER CAN DIFFERENTIATE TOO

by

Lee Litzler
Darwin Klingman

April 1974

This research was partly supported by ONR Project NR 047-021, Contracts N00014-67-A-0126-0008 and N00014-67-A-0126-0009 with the Center for Cybernetic Studies, The University of Texas. Reproduction in whole or in part is permitted for any purpose of the United States Government.

CENTER FOR CYBERNETIC STUDIES
A. Charnes, Director
Business-Economics Building, 512
The University of Texas
Austin, Texas 78712

1.0 Introduction

For more than two decades, computers have been used to differentiate functions by applying the basic rules of calculus to the symbols appearing in the functions [1]. Symbolic or analytic differentiation is of interest because it appears to be the most efficient approach to evaluate the derivative of a function at a series of points. The symbolic expression for the derivative is found once, and that expression is then evaluated many times with little effort. Numeric differencing, the contrasting numerical method, requires that at each point the function be evaluated at least twice, and the derivative is then approximated by taking the slope of the line joining those points.

In 1962 Hanson, Caviness, and Joseph [2] surveyed the methods that had been published for symbolic differentiation and proposed a routine for analytic differentiation which included a wide class of allowable functions. By applying the transformation algorithm of Erchov [3], their routine eliminated the highly structured input requirements of earlier methods. The major detractor of Hanson's routine is that the first and second derivatives that the method produces are inefficient because derivatives tend to increase in complexity at an exponential rate.

The method of symbolic differentiation reported here retains the data structure of the ordered triple proposed by Hanson, et. al. as its skeleton, but introduces improved algorithms for the production and manipulation of the elements of that data structure. Implementation of this method has resulted in a routine which produces "optimized" code for partial derivatives comparable to special purpose subprograms written by users for evaluation of first and second partial derivatives. An experimental computer program has been developed on which computational results are reported. The routine has also been interfaced

with a nonlinear programming code and experimental results indicate that this method is a desirable alternative to other methods of evaluating first and second partial derivatives of user-defined functions.

1.1 Motivation

The authors' philosophy is that computers are intended to aid people in the solutions of their problems, not create new ones. An implication of this philosophy is that computer program execution time alone is inadequate for evaluating the cost of problem solutions. Time spent by humans in problem formulation, data preparation, and so on, frequently provides the major contribution to the costs of problem solution, regardless of whether such costs are itemized in dollar terms. Furthermore, the emphasis on and attention paid to such "human" aspects of scientific computation should and must increase heavily; the pace of solid-state technology is accelerating the increase in speed and capacity of computer hardware faster than the hardware costs are increasing. The human aspects, whose cost trend is just the opposite, will inevitably shoulder aside hardware considerations as the prime cost and time bottle-neck in obtaining problem solutions.

The development of the symbolic differentiation routine reported here was motivated by the above philosophy and the fact that the authors were experiencing substantial time delays in using many codes for solving nonlinear programming (NLP) problems, since most solution methods require that a set of functions and their first and second partial derivatives be evaluated at arbitrary points. Such NLP computer programs require that the user provide subprograms which are able to evaluate the functions (often having many variables) and their partial derivatives at any point; consequently, a user is compelled to know a programming language such as FORTRAN, or know someone who does, in order to prepare a problem

for solution. Substantial time and effort are also required to code the needed routines and debug them once written. Even if standard numerical differencing techniques for first and second partials are used, considerable instability may occur depending on the interval size used and the function being evaluated [4, 5]. The purpose of this paper is to describe procedures used in developing standard routines which will eliminate these problems. Collectively these procedures will be referred to as a symbolic differentiation routine, SYDIF.

The results of the symbolic approach to differentiation and its computer implementation are not restricted to nonlinear programming problems. Any computer procedure which requires that users provide subprograms for evaluating function and their partial derivatives can be simplified by the symbolic differentiation approach. Some areas of application include graphical interactive routines which could be used to aid courses in calculus by displaying iso-curves of functions and their partial derivatives. The feasibility of using SYDIF in conjunction with introductory calculus courses taught in a Computer-Assisted Instruction mode, is also under consideration, since students' answers could be checked exactly by matching their expressions against the symbolic derivative produced. Other possible applications include areas such as numerical integration of systems of differential equations, solution of nonlinear equations by Newton's method, unconstrained optimization, guidance and trajectory systems, and applications of linear programming that iteratively solve non-linear problems.

2.0 Methodology

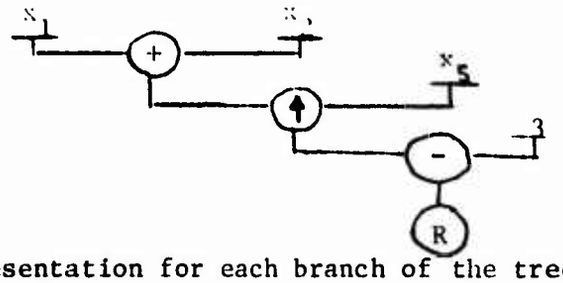
The symbolic differentiation algorithm follows the outline of such algorithms defined by Hanson [2] as "In step one, the mathematical expression is analyzed

and the order in which the appropriate rules of elementary differentiation are to be applied is determined. Step two consists of the actual application of these rules."

2.1 Data Structures

To accomplish step one, SYDIF transforms the input expression into an equivalent parenthesis-free expression represented by a tree-structure specifying the order of the operations. This data structure was chosen because it can be produced efficiently, evaluated easily, differentiated in a straight forward manner using the basic rules of calculus, and the resulting derivative can be optimized.

Each "branch" of the data structure represents a subexpression of the user's function as a binary operator and its two operands. The expression $(x_1 + x_2)^{x_5} - 3$ is represented by the following tree, where ↑ represents exponentiation:



The actual data representation for each branch of the tree is an ordered triple of a binary operator and its two operands. The table of triples representing the above tree is:

line	left operand	operator	right operand
1	x_1	+	x_2
2	(1)	↑	x_5
3 = R	(2)	-	3

where (1) in line 2 refers to the subexpression in the previous line 1. By sequentially evaluating the subexpressions beginning with the first one, the expression can be evaluated. The value at the root, R is the value of the expression. The above data structure is also equivalent to Polish (parenthesis-free) notation [7] which can be produced efficiently by parsing the original infix (normal algebraic) notation expression using the algorithm described below.

2.2 Parsing Algorithm

The parsing algorithm differs from the usual infix-to-postfix transformation algorithm in its manner of using an intermediate push-down stack. The parsing algorithm causes the input string to be scanned from left to right. Operands encountered are placed on an intermediate push-down stack. This is different from algorithms which only produce Polish notation since they move operands directly to the output string, but it is required since only complete subexpressions can be removed from the stack. The algorithm of Hanson, et. al. produces a similar data structure, but achieves it by using Erchov's right-to-left scan; consequently the entire expression must be input before processing. Furthermore, more elements must be kept in the stack since operators of equal precedence must be put onto the stack, hence a larger stack is required and more time is spent clearing the stack using Hanson's algorithm. The algorithm presented in this paper improves on the previous algorithm by using a left-to-right scan which takes advantage of the normal left-to-right execution of operators of equal precedence to achieve more efficient storage management.

The SYDIF parsing algorithm is quite simple and can be described as follows. Operands are placed directly in the intermediate stack. Operators are compared using normal FORTRAN precedence with the operator nearest the top of the intermediate stack. If the operator from the input string has precedence greater than the precedence of the operator in the stack, the input operator is put on top of the stack. The "(" always goes to the stack. If the precedence of

the input operator is less or equal to that of the stack operator, the top three entries are moved from the stack to the table of triples, with the top operand moving to the right operand position. A new operand, representing the branch of the tree where the triple was placed, is put on the top of the stack. The input operator which caused the ejection from the stack is then compared with the top operator remaining on the stack, and the process repeats until the end of the expression is encountered. The end-of-expression symbol is of equal precedence to the begin-expression symbol, both the lowest possible, hence the stack is cleared. Errors in the syntax of the expression are easily recognized by the parsing algorithm.

2.3 Differentiation Algorithm

The expression can be differentiated after being transformed into a tree-structure because such a data structure specifies the order in which the operations are to be performed. The expression is differentiated in the same order that it is evaluated; beginning with the first triple and proceeding sequentially until the root is reached. The new "branches" of the tree representing the derivatives of the expression are attached after the root of the tree.

Subexpressions to be differentiated are classified into categories according to the types of the operands and operator. Each category of subexpression is treated as if it were an assembly language macro-call, the macro expansion being the partial derivative for that category of subexpressions. The parameters of the macro-call are the triple being differentiated, its operands and operator, the variable of differentiation, and pointers to the derivative of any reference to preceeding subexpressions. Such pointers are required for derivatives of some types of subexpressions because the chain rule of calculus requires that

the derivative of any line containing a reference to a preceeding subexpression, use the derivative of that subexpression in obtaining its own derivative.

This algorithm improves on the algorithm of Hanson, et. al. by recording the actual derivative instead of a pointer to the derivative, if that derivative is a single operand. By later referencing the actual derivative, approximately ten percent of the subexpressions produced for second partial derivatives are eliminated. Such a recording procedure also is a prerequisite for further optimization.

2.4 Optimization Algorithm

The underlying purpose of the symbolic differentiation routine is the production of efficient code for the evaluation of partial derivatives. Optimization actually starts with treating each line as a macro call, resulting in as few lines being created as possible. Later specific optimization techniques are applied, both as triples are being added to the tree, and again after the entire expression or partial derivative is created as described below.

As each line is added the transitive and commutative properties of the pairs of operators (+, -) and (*, /) are used to fold subexpressions such as $C*(R/C_2)$ into C_3*R_2, (where C_1 and C_2 are constants and R is a reference to a variable or previous triple and $C_3 = C_1/C_2$ is a new constant). C_3*R has a simpler derivative than $C_1*(R/C_2)$ because the latter requires differentiation of R/C_2. Thus folding shortens both the expression being created and later partial derivatives.

Redundant lines are removed after the entire expression or partial derivative is created. The SYDIF optimization algorithm compares each triple against previous triples; if a match is found, the duplicate is removed and all later

references to the removed triple are changed to point to the first occurence of the triple. The comparison process starts with the second triple being checked against the first and is completed when the subexpression at the root has been compared against all the preceeding ones. Triples which have operators + and * are commuted for the test; special checks are made for cases such as (R + R) = 2 * R or (0 - R) = -1 * R by also checking for the alternate form.

Dead code elimination, (the final phase of optimization) consists of purging superfluous subexpressions accumulated in creating expressions and their partials. The algorithm recursively eliminates all triples not referenced later, starting at the root of the tree. In the above example, once C_3*R is created, there may be no later references to the triple R/C_2, implying that the operation need not be present. Dead code elimination would remove the triple R/C_2 from the tree.

3.0 Experimental Program Implementation

An experimental implementation of SYDIF has been coded in FORTRAN and is being tested on the CDC 6600 at The University of Texas at Austin. This implementation was written to interface with the Sequential Unconstrained Minimization Technique (SUMT) of Fiacco and McCormick [6], but is equally applicable to any problem requiring partial derivatives or user-defined functions. The user of this routine is ultimately interested in two aspects of the program: how easy is it to use and how well does it work?

The authors' feel that users want a familiar, simple input form. The SYDIF implementation allows users to input their functions in FORTRAN notation with FORTRAN operators. Variables are X1 through X100. Real constants in E-notation are allowed, as are integers. Implied multiplication is allowed by inserting the multiplication operator (*) for missing operators. Many other options were

also included in order to provide users with a natural means for expressing functions.

The experimental program outputs optimized pseudo-machine code subprograms, equivalent to symbolic partials, for all first and second partial derivatives for each specified function. These representations can then be evaluated by interpretation or they can be translated directly into machine code for direct execution as a subprogram by a particular computer (see [7] pp. 342-347). The accuracy of the partial derivatives produced is limited only by the accuracy of the computer's floating point hardware and system functions. Computational experience with the experimental implementation of this code has exceeded the original high expectations. The optimization not only produces more efficient code for evaluation; but surprisingly, it also results in a faster differentiation phase by eliminating many subexpressions of the original expression and its first partial derivative that otherwise would have required differentiation. The optimization also reduces data structure storage requirements (by over 80% in some cases) that hamper Hanson's symbolic approach to differentiation, e.g., with no optimization, Hanson's algorithm for the differentiation of the expression $(X_1 + X_2)**X_5$ gives 132 subexpressions for all first and second partial derivatives (see Appendix A for this example). SYDIF with its optimization left only 24 subexpressions to be evaluated (see Appendix B).

The experimental SYDIF program takes 40,000 words of memory on the CDC 6600. Test runs on a problem having thirty functions (of varying complexity) took only two CPU seconds of processing to generate all first and second partial derivatives. Currently a machine-independent FORTRAN version is being developed and should be available soon.

4.0 Application of SYDIF in Non-Linear Mathematical Programming.

An enormous body of theoretical and computational research has been directed at achieving efficient solution procedures for nonlinear programming problems of the form:

$$\text{Minimize } F(x)$$
$$\text{subject to:}$$
$$g_i(x) \geq 0, \quad i = 1, \ldots, I.$$
$$\text{and} \quad h_k(x) = 0, \quad k = 1, \ldots, K.$$

Particularly powerful results have been obtained for the case in which the problem functions are once-or twice-differentiable and where the function F(x) and the set of points X satisfying the constraints are convex. This paper is not intended to provide a bibliography for nonlinear programming; see [8] for references. Chapter 8 of [6] discusses the Sequential Unconstrained Minimization Technique (SUMT) algorithm of Fiacco and McCormick, which is a procedure both elegant and known to be effective from a computational viewpoint. A computer program (in FORTRAN) implementing the SUMT algorithm is available from the SHARE Library under distribution number SDA 3189 (see also [9]).

As can be seen from (1), the statement of a general nonlinear programming problem requires one to specify the objective and constraint functions; there are I + K + 1 such functions. In addition, many of the procedures for solving (1), (e.g., SUMT, Zoutendijk, Newton, Davidon-Fletcher-Powell, etc.) use partial derivatives of these functions with respect to each of the variables as well as the Hessian (matrix of all possible mixed second partials) of each function; even the simplest direct search methods require at least the ability to calculate the values of the objective and constraint functions at arbitrary points.

SYDIF provides a user-oriented input capability and creates the tree structure for evaluating the original expression and all the partial derivatives.

The applications program "calls" evaluation programs, giving the routine a vector of values for the variables and which function or which partials are to be calculated. The triples for that expression or derivative are evaluated one by one, starting with the first one. As the value of each triple is calculated, that value is stored in an auxilary table. The entire process is simple and fast. Running in such an interpretive mode, the interfaced version of SUMT took only twice as long as when subroutines for partials derivative evaluation were coded by hand. The turn around time and the human interaction were shortened as expected; SYDIF took less than one tenth of the user's time.

Actual machine code for a particular computer could have been created, but the authors feel that the portability of the machine-independent approach is more important than complete efficiency, especially in the development stage. Later versions producing direct machine code will produce code that takes less execution time than is possible even with user-written routines.

5.0 Summary

It was necessary to look at the area of creating routines for first and second partial derivatives because of the lack of a user-oriented interface between mathematical programming codes requiring derivatives and its users. Symbolic differentiation was chosen because of the instability/computer problems inherent in numerical differentiation techniques and the inability of numerical methods to overcome the human problem of providing the user with a desirable input format for his problem statement.

Having chosen the symbolic approach, parsing, differentiation, and code optimization procedures were devised in order to improve upon the computer time and storage requirements of early symbolic approaches [2]. These new procedures were then computational compared against early approaches and found to be substantially better in both time and space requirements.

A user-oriented front-end was added to the experimental symbolic differentiation routine. Next this routine was interfaced with SUMT in an interpretive mode and compared against SUMT solution time with hand-coded evaluation subroutines. Running in an interpretive mode the SUMT version took only twice the computer time and reduced the user interaction time 90%. If the user finds the increase in execution time to be prohibitive, a non-interpretive (or direct executing) version could easily be written for a particular computer.

REFERENCES

1. Nolan, J. F., Analytical Differentiation on a Digital Computer Mass. Inst. Technology (May, 1953).

2. Hanson, J. W., J. S. Caviness, and C. Joseph, "Analytic Differentiation by Computer" Comm. A.C.M. June 1962, pp. 349-355.

3. Erchov, G.P. Programming Program for the BESM Computer, Translated by M. Nadler, Pergamon Press, London, (1959).

4. Watson, Philipson, and Oates, Numerical Analysis, The Mathematics of Computing vol. 2, Ch. 5 pp. 102-114. Edward Arnold Ltd. London 1969.

5. Butler and Kerr, An Introduction to Numerical Methods, Ch. 5, pp.213-225, Sir Isaac Pitman and Sons, Ltd. London (1962).

6. Fiacco, A.V., and G. P. McCormick, Nonlinear Programming: Sequential Unconstrained Minimization Techniques, John Wiley and Sons, Inc. New York (1968).

7. Gries, David Compiler Construction for Digital Computers. John Wiley and Sons, Inc. New York (1971).

8. Himmilblau, David M. Applied Nonlinear Programming McGraw-Hill, Inc. New York (1972).

9. Mylander, W. C., R.L. Holmes, and G.P. McCormick, "A Guide to SUMT--Version 4" RAC-P-63, Research Analysis Corp. (1971).

Appendix A:

Example of code produced by basic differentiation algorithm without optimization

function: $(x_1 + x_2)$ ** x_5

note: L_k refers to the previous line numbered k

	T, the Table of Triples		D, the derivitive pointers
(1)	$x_1 + x_2$		L_3
(2)	L_1 **x_5		L_7
(3)	1.0 + 0.0	$\partial L_1/\partial x_1$	L_8
(4)	x_5 - 1.0		L_9
(5)	L_1 ** L_4	$\partial L_1/\partial x_1$	L_{17}
(6)	L_3 * X_5		L_{18}
(7)	L_6 * L_5		L_{21}
(8)	0.0 + 0.0	$\partial L_3/\partial x_1$	
(9)	0.0 + 0.0	$\partial L_4/\partial x_1$	
(10)	L_4 - 1.0		
(11)	L_1 ** L_{10}		
(12)	L_3 * L_4	$\partial L_5/\partial x_1$	
(13)	L_{12} * L_{11}		
(14)	LOG (L_1)		
(15)	L_9 * L_5		
(16)	L_{15} * L_{14}		
(17)	L_{16} + L_{13}		
(18)	L_8 * x_5	$\partial L_6/\partial x_1$	
(19)	L_{18} * L_5		
(20)	L_{17} * L_6	$\partial L_7/\partial x_1$	
(21)	L_{20} + L_{19}		

Line 7 = $\dfrac{\partial F}{\partial x_1}$ Line 21 = $\dfrac{\partial^2 F}{\partial x_1 x_2}$

$\frac{\partial F}{\partial X_1 X_2}$

Assuming $\frac{\partial^2 F}{\partial X_1 X_1}$ has been output, it begins at line 8

	T	D	
(1)	$X_1 + X_2$	L_8	
(2)	L_1 ** X_5	L_{12}	
(3)	1.0 + 0.0	L_{13}	
(4)	X_5 - 1.0	L_{14}	$\frac{\partial L_1}{\partial X_2}$ is required hence the original function must be differentiated with respect to X_2
(5)	L_1 ** L_4	L_{22}	
(6)	L_3 * X_5	L_{22}	
(7)	L_6 * L_5	L_{26}	

(8)	0.0 + 1	$\partial L_1/\partial x_2$
(9)	x_5 - 1.0	
(10)	L_1 ** L_9	
(11)	L_8 * x_5	$\partial L_2/\partial x_2$
(12)	L_{11} * L_{10}	
(13)	0.0 + 0.0	$\partial L_3/\partial x_2$
(14)	0.0 + 0.0	$\partial L_2/\partial x_2$
(15)	L_4 - 1.0	
(16)	L_1** L_{15}	
(17)	L_8 * L_4	
(18)	L_{17} * L_{16}	$\partial L_{10}/\partial x_2$
(19)	LOG (L_1)	
(20)	L_{14} * L_5	
(21)	L_{20} * L_{19}	
(22)	L_{21} *L_{18}	
(23)	L_{13} * x_5	$\partial L_8/\partial x_2$
(24)	L_{23} * L_5	
(25)	L_{22} * L_6	$\partial L_7/\partial x_2$
(26)	L_{25} + L_{24}	

The above code is sufficient for this example. The following summarizes the numbers of additional lines generated for all partial derivatives:

Function	Number of lines
F	2
$\frac{\partial F}{\partial X_1}$	5
$\frac{\partial^2 F}{\partial X_1 X_1}$	14
$\frac{\partial^2 F}{\partial X_1 X_2}$	18
$\frac{\partial^2 F}{\partial X_1 X_5}$	23
$\frac{\partial F}{\partial X_2}$	5
$\frac{\partial^2 F}{X_2 X_2}$	14
$\frac{\partial^2 F}{\partial X_2 X_5}$	23
$\frac{\partial F}{\partial X_5}$	8
$\frac{\partial^2 F}{\partial X_5 X_5}$	20
TOTAL	132 lines required

Appendix B

derivative code with optimization (All partials shown)

(1) $X_1 + X_2$

(2) L_1 ** X_5

(3) $X_5 - 1$
(4) L_1 ** L_3
(5) $L_4 * X_5$

$$\frac{\partial F}{\partial X_1} = X_5(X_1 + X_2)^{X_5 - 1}$$

(6) $X_5 + (-2.0)$
(7) L_1 ** L_6
(8) $L_7 * L_3$
(9) $X_5 * L_8$

$$\frac{\partial^2 F}{\partial X_1 X_1} = (X_1 + X_2)^{X_5 - 2}(X_5 - 1)X_5$$

$$\frac{\partial^2 F}{\partial X_1 X_2} = \text{value of } L_9$$

(10) LOG (L_1)
(11) $L_{10} * L_4$
(12) $X_5 * L_{11}$
(13) $L_4 + L_{12}$

$$\frac{\partial^2 F}{\partial X_1 X_5} = \text{Log } (X_1 + X_2)\,(X_1 + X_2)^{X_5 - 1} + \text{Log } (X_1 + X_2)^{X_5 - 1}$$

(14) $X_5 - 1.$
(15) L_1 ** L_{14}
(16) $L_{15} * X_5$

$$\frac{\partial F}{\partial X_2} = X_5(X_1 + X_2)^{X_5 - 1}$$

(17) $L_7 * L_{14}$
(18) $X_5 * L_{17}$

$$\frac{\partial^2 F}{\partial X_2 X_2}$$

(19) $L_{10} * L_{15}$
(20) $X_5 * L_{19}$
(21) $L_{15} + L_{20}$

$$\frac{\partial^2 F}{\partial X_2 X_5}$$

(22) Log (L_1)
(23) $L_2 * L_{22}$

$$\frac{\partial F}{\partial X_5} = \text{Log } (X_1 + X_2) \cdot (X_1 + X_2)^{X_5}$$

(24) $L_{22} * L_{23}$

$$\frac{\partial^2 F}{\partial X_5 X_5} = \text{Log } (X_1 + X_2)\ \text{Log } (X_1 + X_2)\ (X_1 + X_2)^{X_5}$$