数理解析研究所講究録
第514巻 1984年 48-91

Simultaneous Computation of Functions, Partial Derivatives and Estimates of Rounding Errors ---- Complexity and Practicality ----

関数の値，そのすべての導関数の値，および関数値計算の際に生じる丸め誤差の大きさを同時に計算する方法———計算量と実用性

Masao IRI (University of Tokyo)

伊理 正夫　東京大学工学部

Abstract

A practical approach is proposed to simultaneously computing a function, its partial derivatives with respect to all the variables, and an estimate of the rounding error incurred in the computed value of the function. Theoretically, it has a complexity at most a constant times as large as that of evaluating the function alone, the constant being independent of the number of variables of the function, and is an alternative graphical interpretation of W. Baur and V. Strassen's results with some generalizations. Practically, it is stated in the form easily implementable as a computer program, so that it enables us to automatically compute the derivatives if only the program for computing the function is given. Remarks are added also on the cases of several functions, of higher derivatives and of nonstraight-line programs and on application to problems containing differential equations.

Introduction

The aim of this paper is multifold, but, in brief, it is to investigate the theoretical and the practical results which we can extract from the so-called computational graph for a function or functions.

In the case of a single rational function, W. Baur and V. Strassen proved a theorem [2] stating that the algebraic complexity of computing (i.e., total number of arithmetic operations needed to compute) the function of several variables and its partial derivatives with respect to all the variables is at most a constant (four, five, six or seven, depending on the manner of counting the number of operations, but independent of the number of variables) times as large as that of computing the function alone. Their theorem, when applied to the determinant of a square matrix of order n as a function of its n^2 entries, implies the seemingly "surprising" proposition that the computation of the determinant itself is already as difficult as the computation of the determinant and the adjoint matrix, hence as the computation of the determinant and the inverse matrix, since every entry of the adjoint matrix is the partial derivative of the determinant with respect to an entry of the matrix. This paper gives a more direct and more elementary proof to the theorem and that with some refinements and for a more general case where not only arithmetic operations but also any finitary operations are admitted.

The proof is systematic constructive, so that it gives us a quite practical method of calculating the partial derivatives of a function whose computational procedure is given in the form of a computational graph or something equivalent. In fact, the method can be implemented in a computer program [7]. Since we thus have a mechanical means by which to calculate the derivatives from the given computational procedure of a function, when we want to know the values of the partial derivatives of a fairly complicated function, there is no need, in principle at least, any more to describe the formulas for the derivatives, nor is "numerical differentiation" necessary any more either.

Furthermore, it is seen that a reasonable bound for the rounding error incurred in the final function value due to the finiteness of the precision of computation is ready to obtain once the function and the derivatives have been computed according to the procedure proposed in the following. This will enable us to incorporate a kind of automatic rounding error analysis in various fields of numerical computation.

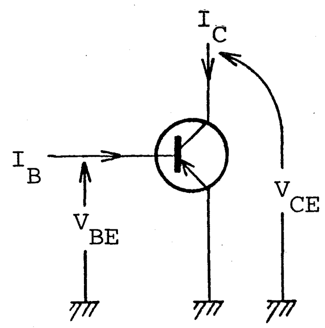
The basic idea as well as the technique is not sophisticated at all, but almost straightforward; all that is needed is the "first theorem of graph theory" concerning the degrees of vertices and the number of arcs of a graph, and the analogy between the partial derivative of a function with respect to one of its input variables and the shortest path from the vertex corresponding to the variable to the vertex corresponding to the function on the computational graph.

Since the problem to be considered here is the most fundamental in calculus and the mathematical techniques used are the most elementary, the results of this paper could readily be obtained by anybody who were so motivated, so that this paper might well be regarded as a kind of tutorial rather than an original contribution. In fact, some of the basic facts mentioned in this paper are found scattered here and there in the existing literature or have been talked about privately, although not completely [1, 10]. Nevertheless, the author would like to emphasize the importance of looking at the fundamental computational problem in calculus as such and to penetrate into it as we shall do in the following.

As supplementary remarks, discussions are made on what will happen and how to do with it when there are several functions, when the computational procedure defining the function(s) contains recurrences and/or conditional branches, and when the same technique is applied to higher derivatives. Also illustrated are, as examples of application of the same principle to problems whose computational graphs are infinite, the problem of identifying the parameters of a physical system whose behaviour is described by a system of ordinary differential euqations, and a computational approach to the two-point boundary-value problem.

1. Motivation and Informal Presentation of the Problem

It may widely be recognized that, in the expressions of derivatives of functions, there appear many terms having factors similar to those appearing in the expressions of the functions. For example, the simplest mathematical model, the so-called Ebers-Moll model, of the relation between the voltages and the currents of a pnp-transistor, which is operating with the emitter grounded, is a couple of equations shown in Fig. 1.1.



$$
\begin{aligned}
I_B &= -(1-\alpha_F)\cdot I_{ES}\cdot[\exp(-q\cdot V_{BE}/k\cdot T)-1] \\
&\quad -(1-\alpha_R)\cdot I_{CS}\cdot[\exp(q\cdot(V_{CE}-V_{BE})/k\cdot T)-1], \\
I_C &= -\alpha_F\cdot I_{ES}\cdot[\exp(-q\cdot V_{BE}/k\cdot T)-1] \\
&\quad +I_{CS}\cdot[\exp(q\cdot(V_{CE}-V_{BE})/k\cdot T)-1].
\end{aligned}
$$

Fig. 1.1. Ebers-Moll model for a pnp-transistor

I_{ES}, I_{CS}: saturation currents for the emitter-base junction and for the collector-base junction

α_F, α_R : current transfer ratios

V_{BE}, V_{CE}: voltages with the emitter as the datum node

T : temperature

q : electric charge of an electron

k : Boltzmann constant

If we write down the derivatives of the base current I_B with respect to all the arguments appearing on the right-hand side of its expression, they will look as follows.

$$
\begin{aligned}
\partial I_B/\partial I_{ES} &= -(1-\alpha_F)\cdot[\exp(-qV_{BE}/kT)-1],\\
\partial I_B/\partial I_{CS} &= -(1-\alpha_R)\cdot[\exp(q(V_{CE}-V_{BE})/kT)-1],\\
\partial I_B/\partial \alpha_F &= I_{ES}\cdot[\exp(-qV_{BE}/kT)-1],\\
\partial I_B/\partial \alpha_R &= I_{CS}\cdot[\exp(q(V_{CE}-V_{BE})/kT)-1],\\
\partial I_B/\partial V_{BE} &= (q/kT)\cdot(1-\alpha_F)\cdot I_{ES}\cdot\exp(-qV_{BE}/kT)\\
&\qquad +(q/kT)\cdot(1-\alpha_R)\cdot I_{CS}\cdot\exp(q(V_{CE}-V_{BE})/kT),\\
\partial I_B/\partial V_{CE} &= -(q/kT)\cdot(1-\alpha_R)\cdot I_{CS}\cdot\exp(q(V_{CE}-V_{BE})/kT),\\
\partial I_B/\partial T &= (q/kT^2)\cdot[-(1-\alpha_F)\cdot I_{ES}\cdot V_{BE}\cdot\exp(-qV_{BE}/kT)\\
&\qquad +(1-\alpha_R)\cdot I_{CS}\cdot(V_{CE}-V_{BE})\cdot\exp(q(V_{CE}-V_{BE})/kT],\\
\partial I_B/\partial q &= (1/kT)\cdot[(1-\alpha_F)\cdot I_{ES}\cdot V_{BE}\cdot\exp(-qV_{BE}/kT)\\
&\qquad -(1-\alpha_R)\cdot I_{CS}\cdot(V_{CE}-V_{BE})\cdot\exp(q(V_{CE}-V_{BE})/kT)],\\
\partial I_B/\partial k &= (q/k^2T)\cdot[-(1-\alpha_F)\cdot I_{ES}\cdot V_{BE}\cdot\exp(-qV_{BE}/kT)\\
&\qquad +(1-\alpha_R)\cdot I_{CS}\cdot(V_{CE}-V_{BE})\cdot\exp(q(V_{CE}-V_{BE})/kT)].
\end{aligned}
\tag{1.1}
$$

The derivatives of the collector current I_C will look quite similar. All the terms on the right-hand sides of the expressions of (1.1) have a factor which appears as a factor (or a substantial part of it) of a term in the expression of I_B of Fig. 1.1. Therefore, when we want to compute I_B, I_C and all their derivatives, it is obviously inefficient to compute them according to the expressions in Fig. 1.1 and eq. (1.1), or the like.

Concerning this specific example, the I_B and I_C in Fig. 1.1 might most efficiently be computed according to the computational scheme of Scheme 1.1 with "intermediate variables", V_{CB}, E_O, E_E, E_C, X_E, X_C, Y_E, Y_C, W_E, W_C, A_F,

A_R, U_E, U_C, J_C, J_{1B} and J_{2B}. This scheme may either be written down explicitly by hand, or be automatically generated by a compiler from the expressions in Fig. 1.1. The computational scheme of Scheme 1.1 can equivalently (more precisely, in a little more general way) be expressed in another form, i.e., in the form of a "computational graph" [1] of Fig. 1.2,

1 $V_{CB} = V_{CE} - V_{BE}$

2 $E_O = k * T$

3 $E_E = -q * V_{BE}$

4 $E_C = q * V_{CB}$

5 $X_E = E_E / E_O$

6 $X_C = E_C / E_O$

7 $Y_E = \exp(X_E)$

8 $Y_C = \exp(X_C)$

9 $W_E = Y_E - 1$

10 $W_C = Y_C - 1$

11 $A_F = 1 - \alpha_F$

12 $A_R = 1 - \alpha_R$

13 $U_E = I_{ES} * W_E$

14 $U_C = I_{CS} * W_C$

15 $J_C = -\alpha_F * U_E$

16 $J_{1B} = -A_F * U_E$

17 $J_{2B} = -A_R * U_C$

18 $I_C = J_C + U_C$

19 $I_B = J_{1B} + J_{2B}$

Scheme 1.1. A computational scheme for the expressions of I_B and I_C in Fig. 1.

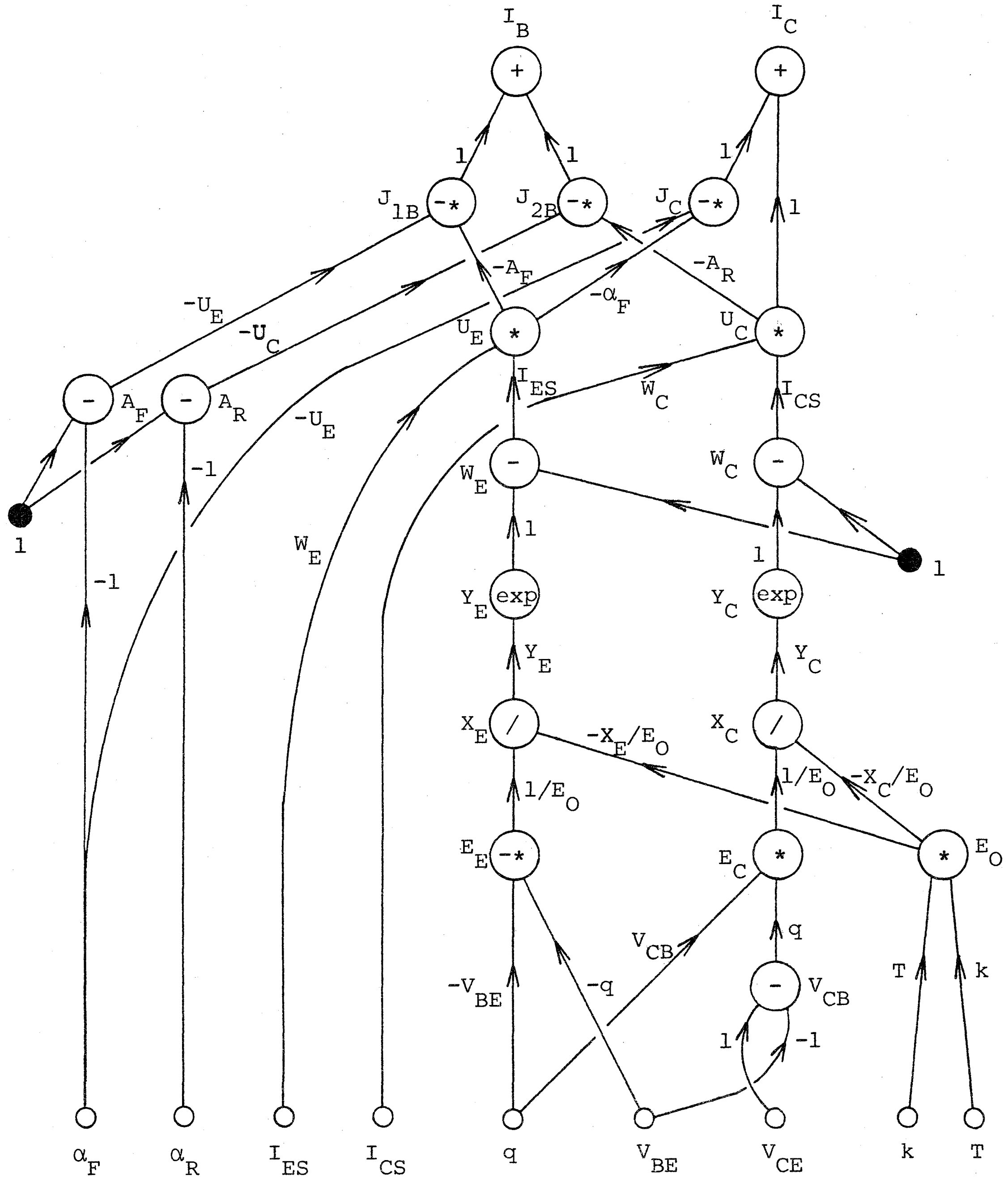


Fig. 1.2. The computational graph corresponding to Scheme 1.1, and the elementary partial derivatives (attached to arcs)

where it is clearly seen how many operations of what kinds are needed to compute the I_B and I_C for given values of input variables, α_F, α_R, I_{ES}, I_{CS}, q, V_{BE}, V_{CE}, k and T. The computational graph is acyclic [3,5,6], so that it determines a partial order on the vertex set. The order in which the intermediate variables (and the final functions I_B and I_C) are computed in Scheme 1.1 is one of the total orders which are consistent with that partial order.

It is possible to define, for each step of computation of an intermediate variable (or final function) w in Scheme 1.1, i.e., for each vertex corresponding to variable w with operation ψ in Fig. 1.2, the partial derivatives of the w with respect to its operands. We shall call them the "elementary partial derivatives". They are naturally attached to the incoming arcs of the vertex corresponding to w in the computational graph. For instance if $w = \psi(u,v)$, then the elementary partial derivatives $\frac{\partial w}{\partial u} = \psi_u$ and $\frac{\partial w}{\partial v} = \psi_v$ are attached to the arcs connecting the vertices for the variables u and v to the vertex for w. The elementary partial derivatives for the computational scheme of Scheme 1.1, or for the computational graph of Fig. 1.2, are shown beside arcs in the same figure. As is readily seen from the chain rule for the derivatives of a composite function, the partial derivative of a function, say I_B, with respect to one of its variables, say V_{BE}, is obtained by first calculating the product of the elementary partial derivatives along each path from the vertex of the computational graph corresponding to V_{BE} to the vertex corresponding to I_B and then taking the sum of those products along all such paths. (This fact is remarked in many text books and papers; see, e.g., [1].) Here it should be noted that this sort of calculation is algebraically very similar to the calculation of the shortest path, where the elementary partial derivatives in the former correspond to the distances attached to arcs in the latter, the products to the sums, and the sum to the minimum, respectively. Thus, the calculation of the partial derivatives of I_B with respect to all its input variables is analogous to that of the shortest paths to a vertex of a directed graph from many other vertices. In the shortest-path problem it is well known that, if we want to find the shortest paths from many vertices to a single vertex, we had better start from the latter vertex and proceed to the former vertices instead of going from the former to the latter [4,6]. This know-how of ours in the shortest-path problem suggests us to make calculation of the partial derivatives of I_B not from the input variables to I_B but from I_B back towards the input variables. This can actually be done by introducing auxiliary quantities attached to the vertices of the computational graph, i.e., the partial derivatives of I_B with respect to the intermediate variables. In fact, as is illustrated in Fig. 1.3, the partial derivative $\frac{\partial I_B}{\partial w}$ of I_B with respect to an intermediate (or input) variable w is equal to the sum of the products $\frac{\partial I_B}{\partial u_i} \cdot \frac{\partial u_i}{\partial w}$ where u_i's are the intermediate variables (or the final function) of which w is an operand, and $\frac{\partial u_i}{\partial w}$'s are the corresponding elementary partial derivatives. In practice, we may initially set $\frac{\partial I_B}{\partial u} := 0$ for all $u \neq I_B$, then start from $\frac{\partial I_B}{\partial I_B} = 1$, and scan the computational graph from top to down as shown in Fig. 1.3 making the above-described calculation step by step. That is equivalent to making the computation according to Scheme 1.2. It is here quite obvious that Scheme 1.2 may be generated automatically, line by line, from Scheme 1.1. A similar scheme will be constructed for the partial derivatives of I_C.

The relation between the number of basic operations needed for computing a function alone and the number of additional operations for computing all its derivatives will intuitively be evident on the basis of the above considerations. Roughly speaking, the former number is equal to the number of vertices of the computational graph, whereas the latter number is nearly equal to the number of arcs (additions and multiplications) if the computations of elementary partial derivatives are ignored. As is seen from Fig. 1.2, we need little additional computation for elementary partial derivatives once we

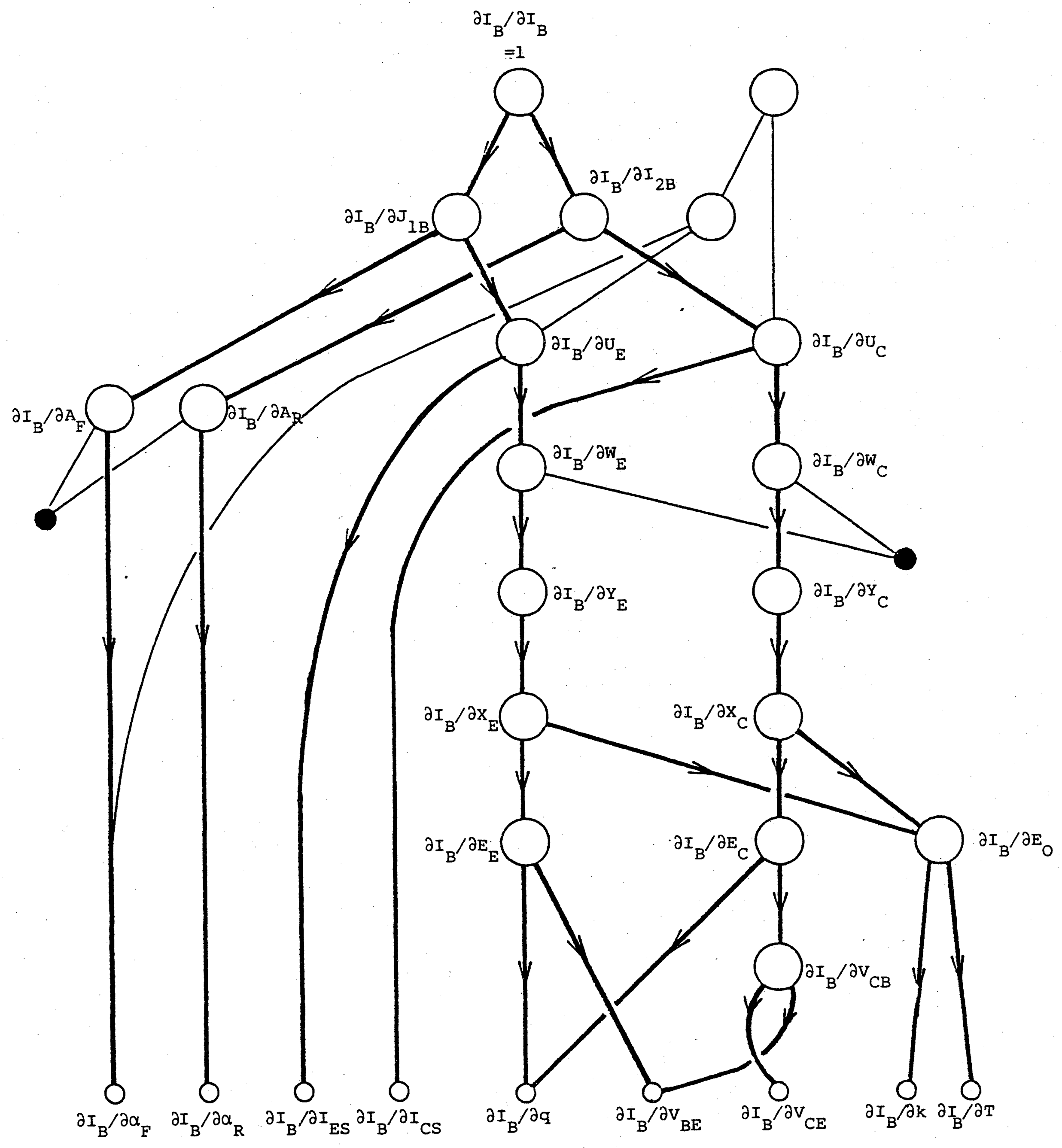

$\partial I_B/\partial I_B$
$=1$
$\partial I_B/\partial I_{2B}$
$\partial I_B/\partial J_{1B}$
$\partial I_B/\partial U_E$
$\partial I_B/\partial U_C$
$\partial I_B/\partial A_F$
$\partial I_B/\partial A_R$
$\partial I_B/\partial W_E$
$\partial I_B/\partial W_C$
$\partial I_B/\partial Y_E$
$\partial I_B/\partial Y_C$
$\partial I_B/\partial X_E$
$\partial I_B/\partial X_C$
$\partial I_B/\partial E_E$
$\partial I_B/\partial E_C$
$\partial I_B/\partial E_O$
$\partial I_B/\partial V_{CB}$
$\partial I_B/\partial \alpha_F$
$\partial I_B/\partial \alpha_R$
$\partial I_B/\partial I_{ES}$
$\partial I_B/\partial I_{CS}$
$\partial I_B/\partial q$
$\partial I_B/\partial V_{BE}$
$\partial I_B/\partial V_{CE}$
$\partial I_B/\partial k$
$\partial I_B/\partial T$

Fig. 1.3. Partial derivatives of I_B (defined on vertices through bold-line arcs)

have computed the function. (Specifically, only three divisions were needed in the case of the computational graph of Fig. 1.2.) Therefore, we can conclude intuitively that the complexity of computing a function and that of

1	$\partial I_B/\partial V_{CE} := \partial I_B/\partial V_{CE} + \partial I_B/\partial V_{CB} * 1$;	$\partial I_B/\partial V_{BE} := \partial I_B/\partial V_{BE} + \partial I_B/\partial V_{CB} * (-1)$
2	$\partial I_B/\partial k := \partial I_B/\partial k + \partial I_B/\partial E_0 * T$;	$\partial I_B/\partial T := \partial I_B/\partial T + \partial I_B/\partial E_O * k$
3	$\partial I_B/\partial q := \partial I_B/\partial q + \partial I_B/\partial E_E * (-V_{BE})$;	$\partial I_B/\partial V_{BE} := \partial I_B/\partial V_{BE} + \partial I_B/\partial E_E * (-q)$
4	$\partial I_B/\partial q := \partial I_B/\partial q + \partial I_B/\partial E_C * V_{CB}$;	$\partial I_B/\partial V_{CB} := \partial I_B/\partial V_{CB} + \partial I_B/\partial E_C * q$
5	$\partial I_B/\partial E_E := \partial I_B/\partial E_E + \partial I_B\ \partial X_E * (1/E_O)$;	$\partial I_B/\partial E_O := \partial I_B/\partial E_O + \partial I_B/\partial X_E * (-X_E/E_O)$
6	$\partial I_B/\partial E_C := \partial I_B/\partial E_C + \partial I_B/\partial X_C * (1/E_O)$;	$\partial I_B/\partial E_O := \partial I_B/\partial E_O + \partial I_B/\partial X_C * (-X_C/E_O)$
7	$\partial I_B/\partial X_E := \partial I_B/\partial X_E + \partial I_B/\partial Y_E * Y_E$	
8	$\partial I_B/\partial X_C := \partial I_B/\partial X_C + \partial I_B/\partial Y_C * Y_C$	
9	$\partial I_B/\partial Y_E := \partial I_B/\partial Y_E + \partial I_B/\partial W_E * 1$	
10	$\partial I_B/\partial Y_C := \partial I_B/\partial Y_C + \partial I_B/\partial W_C * 1$	
11	$\partial I_B/\partial \alpha_F := \partial I_B/\partial \alpha_F + \partial I_B/\partial A_F * (-1)$	
12	$\partial I_B/\partial \alpha_R := \partial I_B/\partial \alpha_R + \partial I_B/\partial A_R * (-1)$	
13	$\partial I_B/\partial I_{ES} := \partial I_B/\partial I_{ES} + \partial I_B/\partial U_E * W_E$;	$\partial I_B/\partial W_E := \partial I_B/\partial W_E + \partial I_B/\partial U_E * I_{ES}$
14	$\partial I_B/\partial I_{CS} := \partial I_B/\partial I_{CS} + \partial I_B/\partial U_C * W_C$;	$\partial I_B/\partial W_C := \partial I_B/\partial W_C + \partial I_B/\partial U_C * I_{CS}$
15		
16	$\partial I_B/\partial A_F := \partial I_B/\partial A_F + \partial I_B/\partial J_{1B} * (-U_E)$;	$\partial I_B/\partial U_E := \partial I_B/\partial U_E + \partial I_B/\partial J_{1B} * (-A_F)$
17	$\partial I_B/\partial A_R := \partial I_B/\partial A_R + \partial I_B/\partial J_{2B} * (-U_C)$;	$\partial I_B/\partial U_C := \partial I_B/\partial U_C + \partial I_B/\partial J_{2B} * (-A_R)$
18		
19	$\partial I_B/\partial J_{1B} := \partial I_B/\partial J_{1B} + \partial I_B/\partial I_B * 1$;	$\partial I_B/\partial J_{2B} := \partial I_B/\partial J_{2B} + \partial I_B/\partial I_B * 1$

$\partial I_B/\partial I_B := 1$; $\partial I_B/\partial w := 0$ $(w \neq I_B)$

Scheme 1.2. Computational scheme for the derivatives of I_B

computing both the function and all its partial derivatives are of the order of the "size" of the computational graph concerned, i.e., they are of the same order.

The purpose of this paper is to more rigorously define the problem, to carry out detailed analysis from the point of view of computational complexity, and to discuss related problems and possible applications in numerical mathematics.

2. Terminology and Notation

For a (directed) graph $G(V,A)$ with vertex set V and arc set A, we adopt a standard terminology and notation in graph theory [3,5,6]. Specifically, for an arc $a \in A$, $\partial^+ a$ $(\in V)$ is its initial vertex and $\partial^- a$ the terminal vertex; for a vertex $v \in V$, $\delta^+ v$ is the set of outgoing arcs from v, and $\delta^- v$ the set of incoming arcs; $d^+ v = |\delta^+ v|$ is the outdegree of vertex v, and $d^- v = |\delta^- v|$ its indegree; $\Gamma^+ v = \partial^- \delta^+ v$ is the set of vertices adjacent in the positive direction to vertex v, and $\Gamma^- v = \partial^+ \delta^- v$ the set of adjacent vertices in the negative direction:

$$\begin{aligned}
&\partial^{\pm} : A \to V \qquad \text{(naturally extended to } \partial^{\pm} : 2^A \to 2^V \text{)} , \\
&\delta^{\pm} : V \to 2^A , \\
&d^{\pm} : V \to \mathbb{Z}_+ , \\
&\Gamma^{\pm} : V \to 2^V .
\end{aligned} \tag{2.1}$$

The following relations are the most fundamental in graph theory, sometimes called the "First Theorem in Graph Theory" [5]:

$$\delta^+ u \cap \delta^+ v = \emptyset , \quad \delta^- u \cap \delta^- v = \emptyset \qquad \text{if} \quad u \neq v ; \tag{2.2.1}$$

$$\bigcup_{v \in V} \delta^+ v = \bigcup_{v \in V} \delta^- v = A ; \tag{2.2.2}$$

$$\sum_{v \in V} d^+ v = \sum_{v \in V} d^- v = |A| ; \tag{2.2.3}$$

$$\sum_{v \in V} (d^+ v + d^- v) = 2 \cdot |A| . \tag{2.2.4}$$

A <u>computational graph</u> $G(V,A)$ is an acyclic graph for which the following concepts are defined. (The name of computational graph is taken from [1]. The concept itself is a special case of the standard tool — sometimes, regarded as a generalization of "signal-flow graph" to the nonlinear case — now widely used in system analysis [8,9].)

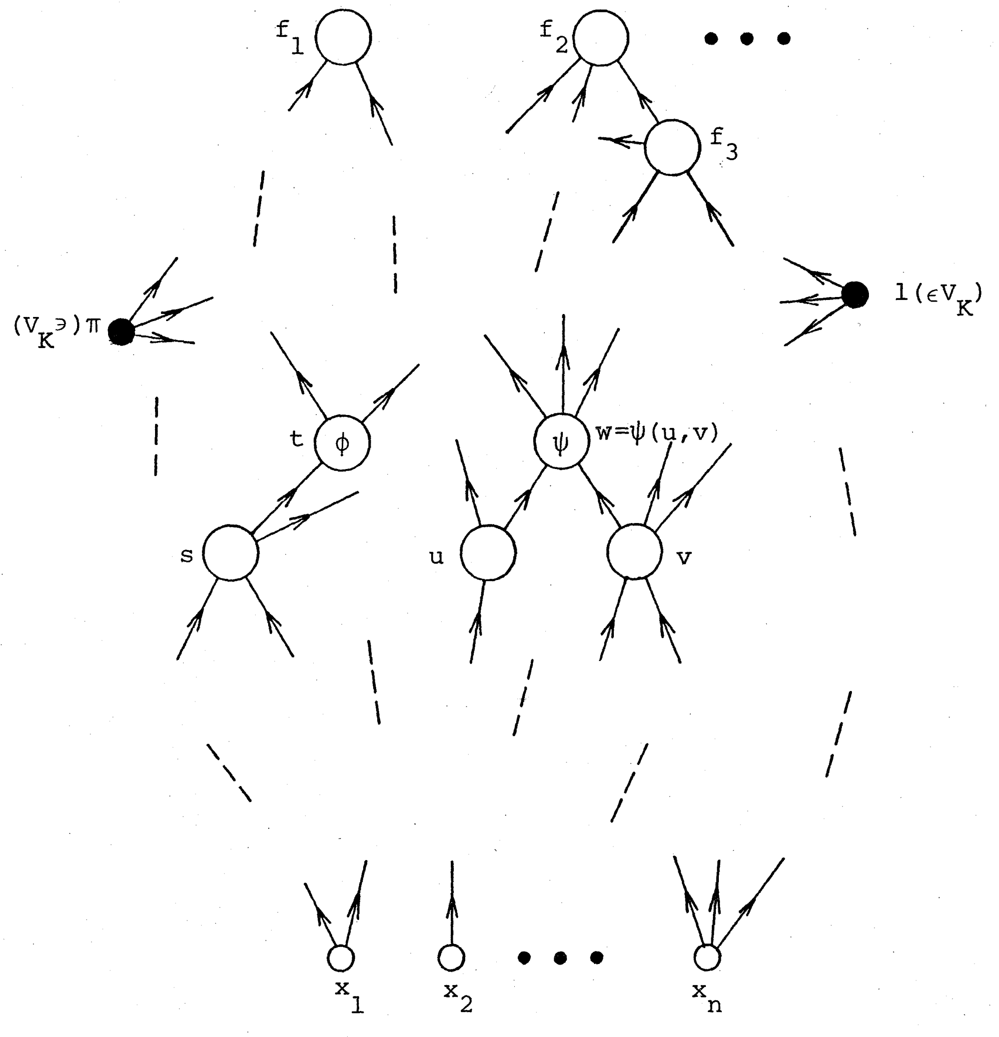


Fig. 2.1. A computational graph

The vertex set V is partitioned into three parts V_X, V_U and V_K, their elements being called, respectively, <u>input variables</u>, <u>intermediate variables</u> and <u>scalars</u> (or <u>constants</u>). (Thus, we shall not distinguish in nomenclature between "vertices" of a computational graph and the corresponding "variables".) Some of the intermediate variables are specified as (final) functions, for which the computational graph affords the manner of computing the values when the values are given to the input variables. (The values of the scalars are assumed to be defined a priori.) The input variables and scalars, and they only, have no incoming arcs:

$$\delta^- v = \emptyset \quad \text{iff} \quad v \in V_X \cup V_K \ . \tag{2.3}$$

We shall denote the set of arcs outgoing from nonscalars by

$$A_U = \bigcup_{v \in V_X \cup V_U} \delta^+ v \ , \tag{2.4}$$

distinguishing those arcs from the arcs outgoing from scalars:

$$A_K = \bigcup_{v \in V_K} \delta^+ v \ . \tag{2.5}$$

Sometimes, we denote the input variables by $x_1, \ldots, x_n$:

$$V_X = \{ x_1, \ldots, x_n \} \ , \tag{2.6.1}$$

and the functions by $f_1, \ldots, f_m$:

$$\{ f_1, \ldots, f_m \} \subseteq V_U \ . \tag{2.6.2}$$

We assume that a finite set of finitary operations Ψ is given as the set of basic operations. Ψ is assumed to contain at least the four arithmetic operations, addition $+$, subtraction $-$, multiplication $*$ and division $/$, all being binary operations, and the unary operation of changing the sign. (However, since the operation of changing the sign is not very essential in practical computation, we shall ignore it by merging it with the preceding or succeeding operations to get, e.g., $\psi(u,v) = -(u*v)$, $= -u/v$, $= -\sin(u)$, $= \log(-u)$, etc., in the following.) The closure of Ψ with respect to legal compositions will be denoted by Ψ^* . In particular, Ψ^* is considered to contain "constants" and the "identity". We assume that every partial derivative of any operation in Ψ belongs to Ψ^* . This is equivalent to assuming that Ψ is a set of operations such that Ψ^* is closed under partial differentiation.

To each intermediate variable v $(\in V_U)$ is attached a basic operation $\psi = \omega(v)$ $(\in \Psi)$, of which the number of operands is equal to $d^- v$, where it is prescribed "which operand" corresponds to the initial vertex $(\in V)$ of "which arc" of $\delta^- v$:

$$\begin{array}{l} \omega : V_U \to \Psi , \\ \psi = \omega(v) \text{ is } (d^- v)\text{-ary} . \end{array} \tag{2.7}$$

Since a computational graph is acyclic, a partial order is determined on $V = \{ u_1, \ldots, u_\ell \}$ as usual. Let $(u_1, u_2, \ldots, u_\ell)$ be a permutation of all the vertices such that, for no two u_i, u_j with $i < j$, u_j precedes u_i in that partial order, and let $(v_1, \ldots, v_k)$ be the sequence of intermediate variables obtained from $(u_1, \ldots, u_\ell)$ by suppressing the input variables and scalars. Then, we have the computational scheme:

$$\left\downarrow \begin{array}{l} v_1 := \psi_1(u_{11}, u_{12}, \ldots) \\ v_2 := \psi_2(u_{21}, u_{22}, \ldots) \\ \quad\vdots \\ \quad\vdots \\ v_k := \psi_k(u_{k1}, u_{k2}, \ldots) \end{array}\right.$$

Scheme 2.1. Computational scheme

where $\psi_i = \omega(v_i)$, and u_{ij} is a scalar $(\in V_K)$, an input variable $(\in V_X)$, or a v_h with $h < i$. There are in general several computational schemes corresponding to one and the same computational graph, but they differ from one another only in the order of arrangement of lines.

In the following, we shall abbreviate "addition(s)/subtraction(s)" to

"A", "multiplication(s)/division(s) by a scalar" to "S", "multiplication(s) of a nonscalar by a nonscalar" to "M", "division(s) by a nonscalar" to "D", and "other operation(s)" to "T". ("T" may be further classified into finer classes if we want. Note that we ignore the operation of changing the sign, merging it with a neighbouring operation, as has already been remarked.) The operation count

$$\nu(G) = n_A\cdot A + n_S\cdot S + n_M\cdot M + n_D\cdot D + n_T\cdot T \tag{2.8}$$

(read " n_A additions/subtractions, n_S scalar multiplications/divisions, n_M essential multiplications, n_D essential divisions and n_T other operations") for a computational graph is simply a quintuple (n_A, n_S, n_M, n_D, n_T) $(\in\mathbf{Z}_+^5)$ of nonnegative integers, each counting the number of respective operations to be performed in the computation specified by G (or the corresponding computational schemes). Let us introduce the mapping

$$\nu : V \to \mathbf{Z}_+^5 \tag{2.9}$$

such that

$$\begin{aligned} \nu(v) &= 1\cdot A \quad \text{if} \quad \omega(v) \quad \text{is an addition or subtraction,} \\ &= 1\cdot S \quad \text{if} \quad \omega(v) \quad \text{is a multiplication or division by a scalar,} \\ &= 1\cdot M \quad \text{if} \quad \omega(v) \quad \text{is a multiplication of two nonscalars,} \\ &= 1\cdot D \quad \text{if} \quad \omega(v) \quad \text{is a division by a nonscalar,} \\ &= 1\cdot T \quad \text{if} \quad \omega(v) \quad \text{is a nonarithmetic operation} \end{aligned}$$

for $v \in V_J$, and

$$\nu(v) = 0$$

for $v \in V_X \cup V_K$ (cf. also Table 2.1). Then we may write

$$\nu(G) = \sum_{v\in V} \nu(v) \ . \tag{2.10}$$

To each arc $a(\in A_U)$ whose initial vertex is not a scalar (i.e., $\partial^+ a \in V_X \cup V_U$) we attach an elementary partial derivative as follows. Take the terminal vertex $w = \partial^- a$, which is an intermediate veriable $(\in V_U)$ by virtue of (2.3). Then, there is a basic operation $\psi = \omega(w)$ (cf. (2.7)), and it is prescribed to which of the $d^{-}w$ operands of ψ the initial vertex $u = \partial^{+}a$ of arc a corresponds. Let us assume that the corresponding operand is the p-th. Thus, there is a step of computation

$$w = \psi(\overset{1 \cdot\cdot\cdot\cdot\cdot p \cdot\cdot\cdot\cdot\cdot}{\ldots , u, \ldots}) \qquad (2.11)$$

in the computational scheme. Therefore, after finishing this step of computation, we can determine the value $d(a)$ of the partial derivative of ψ with respect to its p-th argument for the current values of the arguments. (Note that, unless a has a parallel arc, we may write $d(a) = \frac{\partial w}{\partial u}$.) We shall call the values $d(a)$ attached to the arcs of A_U in the computational graph in this manner the elementary partial derivatives (see Fig. 2.2). An elementary partial derivative may be calculated in any manner in terms of the values of the intermediate variables involved in the computation of (2.11). It is worth noting that little additional computation is needed to compute the elementary partial derivatives after the computation of the functions is finished.

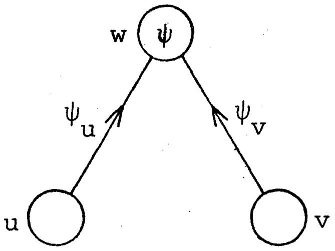


Fig. 2.2. Elementary partial derivatives

We shall denote by $\tilde{\nu}(w)$ the operation count for computing all the elementary partial derivatives $d(a)$ attached to the arcs $a \in \delta^{-}w \cap A_U$ from the values of w and u_i's when the computation of $w = \psi(u_1, u_2, \ldots)$ is over. The operation count for the elementary partial derivatives for all the arcs of A_U in the computational graph G is then defined as

$$\tilde{\nu}(G) = \sum_{w \in V_U} \tilde{\nu}(w) . \qquad (2.12)$$

In the following, we shall make use of another computation connected with the elementary partial derivatives attached to the arcs $a \in \delta^- w \cap A_U$, i.e.,

to compute $d(a) * y$'s for any given value of y .

We denote by $\tilde{\nu}^*(w)$ the operation count for this computation and define

$$\tilde{\nu}^*(G) = \sum_{w \in V_U} \tilde{\nu}^*(w) . \qquad (2.13)$$

Table 2.1 shows the operation counts $\tilde{\nu}(w)$ and $\tilde{\nu}^*(w)$ for typical vertices w , where the operation count for a division vertex indicated with an asterisk in the table is based on the folllowing algorithm (due to [2]):

$$\begin{aligned} &z := y/v , \\ &\frac{\partial w}{\partial u}*y \ (= (1/v)*y \ ) := z , \qquad (2.14) \\ &\frac{\partial w}{\partial u}*y \ (= -(w/v)*y \ ) := -w*z . \end{aligned}$$

As will be seen later, this algorithm corresponds to a tricky surgery (i.e., local modification) of the part of computational graph such as shown in Fig. 2.3.

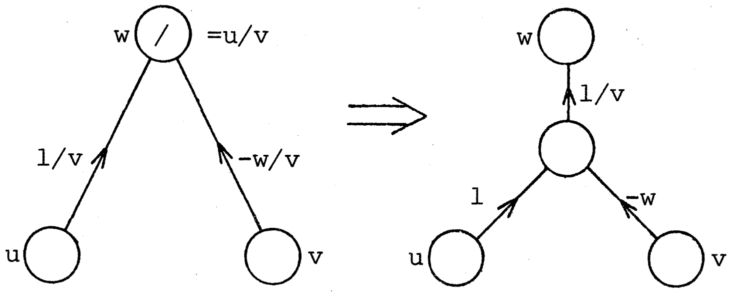


Fig. 2.3. Surgery for a division vertex

Table 2.1. Operation counts for typical vertices

($u, v \in V_X \cup V_U$; $w \in V_U$; $c \in V_K$; $\psi \in \Psi$; A = addition/subtraction,

S = scalar multiplication/division, M = essential multiplication,

D = essential division, T = other operations)

$w = \psi(u,v)$	ψ_u	ψ_v	$\nu(\psi)$	$\tilde{\nu}(\psi)$	$\tilde{\nu}^*(\psi)$
$w = u \pm v$	1	±1	1A	0	0
$w = u * v$	v	u	1M	0	2M
$w = -u * v$	-v	-u	1M	0	2M
$w = u * c$	c		1S	0	1S
$w = u/v$	1/v	-w/v	1D	2D	1M + 1D*
$w = -u/v$	-1/v	w/v	1D	2D	1M + 1D*
$w = c/u$	-w/u		1D	1D	1M + 1D
$w = \exp(u)$	w		1T	0	1M
$w = \log(u)$	1/u		1T	1D	1D
$w = \sin(u)$	cos(u)		1T	1T	1T + 1M
$w = \cos(u)$	-sin(u)		1T	1T	1T + 1M
exp w = ±log(±u) sin cos	the same as above				
$w = \sqrt{u}$	1/(2u)		1T	1D + 1S	1D + 1S
$w = \|u\|$	sgn(u)		0	0	0

$w = \max(u,v)$	if u > v
	then 1
	else 0
		if u > v
		then 0
		else 1
			1A	0	0

$w = \min(u,v)$	if u > v
	then 0
	else 1
		if u > v
		then 1
		else 0
			1A	0	0

3. Partial Derivatives and "Shortest Paths"

As is readily seen from the chain rule for the differentiation of a composite function, the partial derivative $\frac{\partial f_j}{\partial x_i}$ of the j-th function f_j with respect to the i-th input variable x_i can be expressed in terms of elementary partial derivatives as

$$\frac{\partial f_j}{\partial x_i} = \sum d(a_1) * d(a_2) * \dots * d(a_\ell) \ , \qquad (3.1)$$

where $(a_1, a_2, \dots, a_\ell)$ is the sequence of arcs on a directed path from x_i to f_j (i.e., $x_i = \partial^+ a_1$, $\partial^- a_1 = \partial^+ a_2$, ..., $\partial^- a_{\ell-1} = \partial^+ a_\ell$, $\partial^- a_\ell = f_j$) and the sum is taken over all the directed paths from x_i to f_j .

Here the analogy is evident between the formula (3.1) and the well-known formula for the shortest distance:

$$d(f_j, x_i) = \text{Min} \{ d(a_1) + d(a_2) + \dots + d(a_\ell) \} \ , \qquad (3.2)$$

where $d(f_j, x_i)$ is the distance along the shortest directed path from x_i to f_j in the graph $G(V_X \cup V_U, A_U)$ which has essentially the same topological structure as the computational graph we are considering but which has the "arc-distance" function $d : A_U \to \mathbb{R}$ defined on the arc set A_U instead of the elementary partial derivatives, and where $(a_1, a_2, \dots, a_\ell)$ as well as the range of taking the minimum in (3.2) is the same as in (3.1). The difference between (3.1) and (3.2) thus consists only in that the product in (3.1) is replaced by the sum in (3.2) and the sum in (3.1) by the minimum in (3.2).

The algorithmic theory for the shortest-path problem has long been studied in detail (see [4] for an earlier survey, which already contains enough information for our present purpose). For an acyclic graph the problem is rather easy to solve, the most primitive "potential method" or "label-correcting method" giving satisfactory results from the point of view of computational efficiency. The only choice is in which direction we should proceed the labelling, "from the entrance x_i to the exit f_j " or "from the exit to the entrance". The first method is to label each vertex v with the shortest distance $d(v,x_i)$ from the entrance x_i to v as its potential, where we spread the labels from the entrance to other vertices by scanning the graph in the breadth-first manner starting from the entrance. The second method is to adopt the distance $d(f_j,v)$ to the exit f_j from each vertex v for the label of v, and to scan the graph in the breadth-first manner starting from the exit, thus spreading the labels from the exit to other vertices. It is a commonsense in the algorithmic theory for the shortest-path problem that we can determine by means of one of the above two methods not only the shortest path from a vertex (entrance) to another (exit) but we can determine also either all the shortest paths from a single entrance to many exits simultaneously by means of the first method or all the shortest paths from many entrances to a single exit simultaneously by means of the second method, and furthermore that, for the multi-entrance multi-exit problem, there is no method definitely better than applying the first or the second method repeatedly to each of the entrances or of the exits, respectively. Hence, if there are less entrances than exits, we had better apply the first method as many times as there are entrances, whereas, if there are more entrances than exits, the repeated application of the second method to each of the exits is recommendable.

Let us consider what will result if we translate these facts for the shortest-path problem into our problem according to the analogy between the two problems. First, we note that the shortest distance from a vertex u to another v is translated into the partial derivative $\frac{\partial v}{\partial u}$ of v with respect to u. (If there is no directed path from u to v, the distance from u to v is taken to be infinite, and the partial derivative $\frac{\partial v}{\partial u}$ to be equal to zero.) Then, the method of determining the partial derivatives of all the intermediate variables v $(\in V_U)$ with respect to an input variable x_i —— which we shall call Method 1 —— can be described as follows (see also Fig. 3.1(a)), where $(v_1, v_2, \ldots, v_k)$ is the permutation of V_U in Scheme 2.1.

<Method 1> An array of size $|V_X \cup V_U|$ for $\frac{\partial v}{\partial x_i}$'s $(v \in V_X \cup V_U)$ is used for the working area.

1° Initialization :--

```
    for all $v \in V_X \cup V_U$ do $\frac{\partial v}{\partial x_i} := 0$ ;
    $\frac{\partial x_i}{\partial x_i} := 1$ .
```

2° Iteration :--

```
    for $h := 1$ until $k$ do
      for all $a \in \delta^- v_h \cap A_U$ do
        $u := \partial^+ a$ ; $\frac{\partial v_h}{\partial x_i} := \frac{\partial v_h}{\partial x_i} + d(a) * \frac{\partial u}{\partial x_i}$ .
```

On the other hand, the method of determining the partial derivatives of a function f_j with respect to all the nonscalar variables v $(\in V_X \cup V_U)$ —— which we shall call Method 2 —— is as follows (see also Fig. 3.1(b)), where $(v_1, v_2, \ldots, v_k)$ is the same as in Method 1.

<Method 2> An array of size $|V_X \cup V_U|$ for $\frac{\partial f_j}{\partial v}$'s $(v \in V_X \cup V_U)$ and a set Q of size $|V_X \cup V_U|$ are used for the working area.

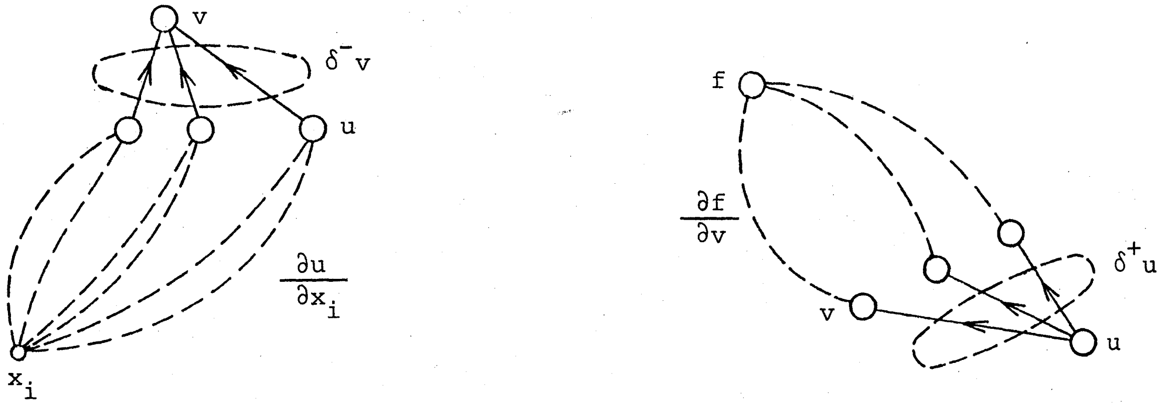
1° Initialization :--

```
    for all $v \in V_X \cup V_U$ do $\frac{\partial f_j}{\partial v} := 0$ ;
    $\frac{\partial f_j}{\partial f_j} := 1$ ;  $( Q := \{ f_j \} )$ .
```

```
2° Iteration :--
    for  h := k  until  1  step  -1  do
      ( if  $v_h \in Q$  then )
        ( $Q := Q - \{ v_h \}$ ;)
        for all  $a \in \delta^- v_h \cap A_U$  do
            $u := \partial^+ a$ ;  $\frac{\partial f_j}{\partial u} := \frac{\partial f_j}{\partial u} + d(a) * \frac{\partial f_j}{\partial v_h}$ ;
            ( $Q := Q \cup \{ u \}$ ) .
```

(The set Q may not be used, so that the parts enclosed in parentheses may be omitted from the above algorithm.)

In view of our knowledge concerning the shortest-path problem as stated in the above, we shall adopt Method 2 in the following, assuming there are usually not so many functions to compute as there are input variables.


v
$\delta^- v$
u
$\frac{\partial u}{\partial x_i}$
x_i
f
$\frac{\partial f}{\partial v}$
$\delta^+ u$
v
u

(a) (b)

Fig. 3.1. Fundamental recurrence relation used in (a) Method 1 and (b) Method 2

4. Complexity of the Computation of the Partial Derivatives of a Function

In this section we confine ourselves to the case of a single function. Then the algorithm for computing the function and all its partial derivatives is summarized as follows.

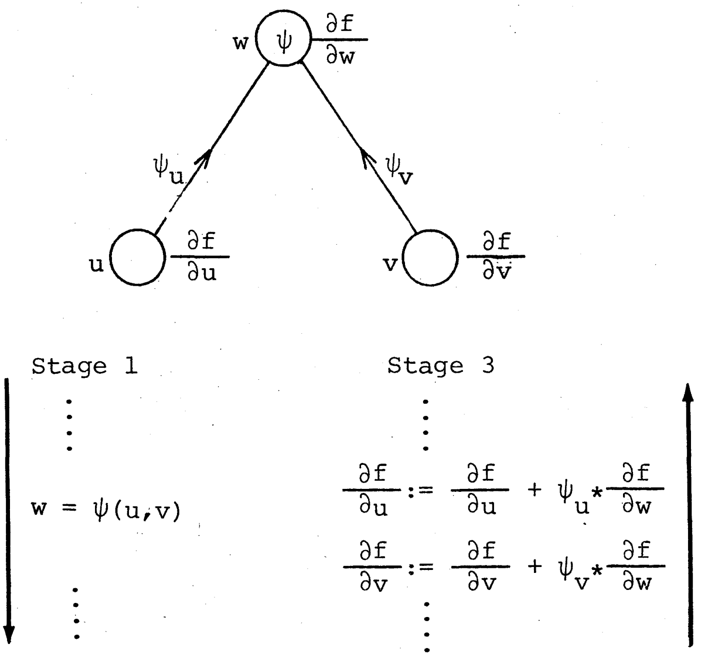
<Algorithm A>

Stage 1 :-- Compute the value of the function f as well as the values of the intermediate variables v_h according to the computational scheme, Scheme 2.1, of section 2.

Stage 2 :-- Compute the elementary partial derivatives $d(a)$'s to attach to the arcs $a \in A_U$ of the computational graph.

Stage 3 :-- Compute the partial derivatives $\frac{\partial f}{\partial v_h}$'s and $\frac{\partial f}{\partial x_i}$'s of the function with respect to the intermediate variables and the input variables by Method 2 of section 3.

The relation of Stage 1 to Stage 3 is illustrated in Fig. 4.1 figuratively.



w
ψ
$\frac{\partial f}{\partial w}$
ψ_u
ψ_v
u
$\frac{\partial f}{\partial u}$
v
$\frac{\partial f}{\partial v}$
Stage 1
$w = \psi(u,v)$
Stage 3
$\frac{\partial f}{\partial u} := \frac{\partial f}{\partial u} + \psi_u * \frac{\partial f}{\partial w}$
$\frac{\partial f}{\partial v} := \frac{\partial f}{\partial v} + \psi_v * \frac{\partial f}{\partial w}$

Fig. 4.1. Relation between Stage 1 and Stage 3 of Algorithm A.

As is seen from "iteration" step of Method 2, we may combine Stage 2 and Stage 3 into one stage to compute $d(a) * \frac{\partial f}{\partial v_h}$'s for $a \in \delta^- v_h \cap A_U$ with $\tilde{\nu}^*(v_h)$ operations, thus obtaining the following algorithm.

<Algorithm B>

Stage 1 :-- The same as in Algorithm A.

Stage 2 & 3 :-- Compute the partial derivatives $\frac{\partial f}{\partial v_h}$'s and $\frac{\partial f}{\partial x_i}$'s by Method 2, computing elementary partial derivatives as they become necessary.

The time complexity, i.e., the operation count of the algorithm is now obvious. For Stage 1 it is equal to $\nu(G)$ (cf. (2.10)), and for Stage 2 of Algorithm A it is $\tilde{\nu}(G)$ (cf. (2.12)). At Stage 3, one multiplication is required for each arc of A_U, so that the total number of multiplications needed is equal to $|A_U|$, whereas the number of additions needed is less than that of multiplications by one for each vertex of $V_X \cup V_U$, hence by $|V_X \cup V_U|$ in all, since the first addition for each such vertex in the expression of "Iteration" step of Method 2, being the addition to 0, is not to be counted in the operation count. Hence, the operation count for Stage 3 is

$$(|A_U| - |V_X \cup V_U|)\cdot A + |A_U|\cdot M . \tag{4.1}$$

Therefore, the total operation count for Algorithm A is

$$\nu(G) + \tilde{\nu}(G) + (|A_U| - |V_X| - |V_U|)\cdot A + |A_U|\cdot M . \tag{4.2}$$

For Algorithm B where Stages 2 and 3 are merged into the single stage, Stage 2 & 3, the operation count for Stage 2 & 3 is

$$\nu^*(G) + (|A_U| - |V_X \cup V_U|)\cdot A , \tag{4.3}$$

so that the total operation count is

$$\nu(G) + \tilde{\nu}^*(G) + (|A_U| - |V_X| - |V_U|)\cdot A . \tag{4.4}$$

In order to go into more details, we have to make some assumptions on the properties of basic operations. For the sake of simplicity, we first count every basic operation with an equal weight, i.e., we introduce a kind of "norm" of the operation count such as follows:

$$| n_A \cdot A + n_S \cdot S + n_M \cdot M + n_D \cdot D + n_T \cdot T | = n_A + n_S + n_M + n_D + n_T . \qquad (4.5)$$

Then, we have by definition

$$|\nu(w)| = 1 \quad \text{for every} \quad w \in V_U . \qquad (4.6)$$

Furthermore, it will be plausible, in view of Table 2.1, to assume that

$$|\tilde{\nu}(w)| \leqq d^- w \quad \text{and} \quad |\tilde{\nu}^*(w)| \leqq d^- w \quad \text{for every} \quad w \in V_U . \qquad (4.7)$$

(Obviously, these assumptions are practically too conservative.) It will also be a realistic assumption that every basic operation is at most binary and that there is no redundant vertex in the computational graph, i.e.,

$$d^- w \leqq 2 \quad \text{for every} \quad w \in V_U \quad \text{and} \quad d^+ w \geqq 1 \quad \text{for every} \quad w \in V - \{f\} . \qquad (4.8)$$

From (2.2.3) and (4.8), we have

$$|A| = \sum_{v \in V_X \cup V_K} d^- v + \sum_{v \in V_U} d^- v \leqq 2 \cdot |V_U| \qquad (4.9)$$

and

$$|A_U| = \sum_{v \in V_X \cup V_U} d^+ v = \sum_{v \in V} d^+ v - \sum_{v \in V_K} d^+ v \leqq |A| - |V_K| . \qquad (4.10)$$

From (4.6) and (2.10) we have

$$|\nu(G)| = |V_U| . \qquad (4.11)$$

From (4.7), (2.12), (2.13), (2.3) and (2.2.3) we have

$$|\tilde{\nu}(G)| \leqq \sum_{v \in V_U} d^- v = \sum_{v \in V} d^- v - \sum_{v \in V_X \cup V_K} d^- v = |A| , \qquad (4.12)$$

and, likewise,

$$|\tilde{\nu}^*(G)| \leqq |A| . \qquad (4.13)$$

Let us denote the operation count for the computation of the function f alone by L(f) and that for the simultaneous computation of f and all its derivatives by L(f, ∇f) . Then we have from (4.11)

$$|L(f)| = |\nu(G)| = |V_U| \ . \tag{4.14}$$

For Algorithm A, we have from (4.1), (4.10), (4.11) and (4.12)

$$\begin{aligned}|L(f,\ \nabla f)| &= |\nu(G)| + |\tilde{\nu}(G)| + (\ |A_U| - |V_X \cup V_U|\ ) + |A_U| \\ &\leqq |V_U| + |A| + 2\cdot|A_U| - |V_X| - |V_U| \\ &\leqq 3\cdot|A| - |V_X| - 2\cdot|V_K| \leqq 3\cdot|A| \ .\end{aligned} \tag{4.15}$$

Furthermore, taking account of (4.9), we have

$$|L(f,\ \nabla f)| \leqq 6\cdot|V_U| \tag{4.16}$$

for Algorithm A. Thus we have the relation

$$|L(f)| \leqq |L(f,\ \nabla f)| \leqq 6\cdot|L(f)| \tag{4.17}$$

for Algorithm A under our assumptions.

For Algorithm B, we have from (4.3), (4.10), (4.11) and (4.13)

$$\begin{aligned}|L(f,\ \nabla f)| &= |\nu(G)| + |\tilde{\nu}^*(G)| + |A_U| - |V_X \cup V_U| \\ &\leqq |V_U| + |A| + |A_U| - |V_X| - |V_U| \\ &\leqq 2\cdot|A| - |V_X| - |V_K| \leqq 2\cdot|A| \ ,\end{aligned} \tag{4.18}$$

and, taking account of (4.9),

$$|L(f,\ \nabla f)| \leqq 4\cdot|V_U| \ . \tag{4.19}$$

Thus we have the relation

$$|L(f)| \leqq |L(f,\ \nabla f)| \leqq 4\cdot|L(f)| \tag{4.20}$$

for Algorithm B.

(4.17) or (4.20) can be read:

> "If f is computed with $|L(f)|$ operations, then there is an algorithm for simultaneously computing f and all its partial derivatives with at most $6\cdot|L(f)|$ or $4\cdot|L(f)|$ operations."

The space complexity for the simultaneous computation of a function and all its derivatives is easy to estimate. We need $|V|$ places for storing the values of the input and the intermediate variables (and the constants) at Stage 1, and, if we use Algorithm A, $|A_U|$ places for the values of the elementary partial derivatives at Stage 2, and $|V_X \cup V_U|$ places for the values of the partial derivatives $\frac{\partial f}{\partial v_h}$ and $\frac{\partial f}{\partial x_i}$ at Stage 3. If we use Algorithm B, we do not need $|A_U|$ places at Stage 2. In any case, we need at most $2 \cdot |V| + |A|$ ($\leqq 4 \cdot |V|$ under the assumption (4.8)), or $3 \cdot |V|$, extra places for computing the partial derivatives in addition to the function itself. (Note that we do not consider here the space for representing the topological structure of the computational graph or the structure of the computational scheme. It is common both to the function and to the derivatives.) Again, the space complexity is at most three or four times as large as the number of vertices of the computational graph.

In the case where we take only the four arithmetic operations for the basic operations, it is possible to make more detailed analysis of complexity. In fact, let

$$\nu(G) = n_A \cdot A + n_S \cdot S + n_M \cdot M + n_D \cdot D . \qquad (4.21)$$

Then it is seen from Table 2.1 that

$$\begin{aligned} \tilde{\nu}^*(G) &= n_S \cdot S + 2 \cdot n_M \cdot M + (n_D \cdot M + n_D \cdot D) \\ &= n_S \cdot S + (2 \cdot n_M + n_D) \cdot M + n_D \cdot D . \end{aligned} \qquad (4.22)$$

In order to evaluate the number of additions appearing in the second term of (4.3), let us rewrite $|A_U|$ and $|V_X \cup V_U|$ in terms of n_A, n_S, n_M and n_D . Since

$$|A_U| = \sum_{v \in V_U} |\delta^- v \cap A_U| \qquad (4.23)$$

and

$$\begin{aligned} |\delta^- w \cap A_U| &\leqq 2 && \text{if } \omega(w) \text{ is an addition/subtraction (the inequality} \\ & && \text{takes place when an operand is a scalar),} \\ &= 1 && \text{if } \omega(w) \text{ is a scalar multiplication/division,} \\ &= 2 && \text{if } \omega(w) \text{ is an essential multiplication,} \\ &\leqq 2 && \text{if } \omega(w) \text{ is an essential division (the inequality} \\ & && \text{takes place when the numerator is a scalar),} \end{aligned}$$

we have

$$|A_U| \leqq 2\cdot n_A + n_S + 2\cdot n_M + 2\cdot n_D \; ; \tag{4.24}$$

it is more trivially seen that

$$|V_X \cup V_U| \geqq |V_U| = n_A + n_S + n_M + n_D \; . \tag{4.25}$$

Combining (4.24) and (4.25), we have

$$|A_U| - |V_X \cup V_U| = n_A + n_M + n_D \; . \tag{4.26}$$

Summing (4.21), (4.22) and (4.26) all together, we have the operation count by Algorithm B in the case of the four arithmetic operations as follows:

$$L\,(f) = n_A\cdot A + n_S\cdot S + n_M\cdot M + n_D\cdot D \; , \tag{4.27}$$

$$L(f,\, \nabla f) \leqq (\; 2\cdot n_A + n_M + n_D \;)\cdot A + 2\cdot n_S\cdot S + (\; 3\cdot n_M + n_D \;)\cdot M + 2\cdot n_D\cdot D \; . \tag{4.28}$$

This result for the special case refines and sharpens the upper bound for $L(f,\, \nabla f)$ in [2] where M and D are not distinguished from each other so that $(\; 3\cdot n_M + n_D \;)\cdot M + 2\cdot n_D\cdot D$ here is counted as $3\cdot(\; n_M + n_D \;)\cdot M/D$ there. Moreover, it is easy to see from (4.24), (4.25), (4.26) that the right-hand side of (4.28) can further be reduced by the sum of the following numbers:

the number of input variables, i.e., $|V_X|$,

the number of additions/subtractions one of whose operands is a scalar

and

the number of divisions of a scalar by a nonscalar.

5. Estimation of Rounding Error

As a by-product of the algorithm for simultaneously computing a function and all its partial derivatives whose complexity we analyzed in the preceding section, we can get a rather good estimate of the rounding error incurred in the value of the function by the computation, with no substantial extra labour.

We shall denote by ε the so-called "machine epsilon", which is the maximum relative error occurring in the floating-point representation of a real number on a fixed-length word of the computer memory. For a typical familiar computer used with a typical compiler or interpreter, ε is ordinarily 16^{-5} (single-precision hexadecimal), 16^{-13} (double-precision hexadecimal), 2^{-24} (single-precision binary), 2^{-56} (double-precision binary), or one half of them.

For the generation and propagation of rounding errors, we adopt the following model [11]. Let Δw (to be called the rounding error in w) be the difference of the value of the variable w obtained by the actual computation from the value which would be obtained if we made the infinite-precision computation. When we compute the value of an intermediate variable w at a certain step of the computational scheme $w = \psi(u, v, \ldots)$, the rounding error Δw is assumed to be determined by

$$\Delta w = \delta w + \sum_{\substack{a \in \delta^- w \\ (u = \partial^+ a)}} d(a) * \Delta u . \tag{5.1}$$

(Usually, (5.1) is written in the form

$$\Delta w = \delta w + \sum_i \psi_{u_i} * \Delta u_i , \tag{5.1'}$$

which is valid unless the same intermediate variable u_i occurs as arguments of ψ more than once.) Here, the second term on the right-hand side of (5.1) is the term representing the "propagation" of rounding error, whereas the first term δw represents the "generation" of rounding error at this step of computation. It is not very easy to decide how large δw is, since it depends on the machine (and even on the particular compiler) we use and on the kind of operation ψ . But, when we work with a "good" machine and a "good" compiler, it might not be so unrealistic to assume that

$$|\delta w| \leq \varepsilon \cdot |w| . \tag{5.2}$$

Then, simple elementary calculation will yield

$$\Delta f = \sum_{v \in V_U} \frac{\partial f}{\partial v} * \delta v + \sum_{x \in V_X} \frac{\partial f}{\partial x} * \Delta x \ , \tag{5.3}$$

from which follows the inequality

$$|\Delta f| \leqq \varepsilon \cdot [\sum_{v \in V_U} |\frac{\partial f}{\partial v}| * |v| \ ] + \sum_{x \in V_X} |\frac{\partial f}{\partial x}| * |\Delta x| \ . \tag{5.4}$$

Since we have already got the values of the intermediate variables $v \in V_U$ and of the partial derivatives $\frac{\partial f}{\partial v}$ of the function with respect to them through the computation at Stages 1 to 3 (see section 4), it is straightforward to compute the sum of products of the first term on the right-hand side of (5.4) and to multiply it by ε , which stage of computation we shall call Stage 4.

Stage 4 :-- Compute the sum of products $|\frac{\partial f}{\partial v}| * |v|$ over all the intermediate variables $v \in V_U$.

The operation count for this additional stage is obviously

$$(\ |V_U| - 1 \ ) \cdot A + |V_U| \cdot M \ , \tag{5.5}$$

and we need no special working space for this stage.

Whether the second term on the right-hand side of (5.4) is meaningful or not will depend on the circumstances in which we are laid. Sometimes we may neglect that term setting $\Delta x = 0$ for all $x \in V_X$, or sometimes we should take account of the effect of the errors already contained in the input variables on the final function value. Whichever is the case, we can readily do anything we want to since we have enough information on $\frac{\partial f}{\partial x}$'s .

6. Practical Implications

It may be said that there have been two means in use when derivatives, in addition to a function or functions, are necessary in practical numerical computation. One is to differentiate the expression (or the computational scheme) of the function as a "formula" either by hand or by some program which is capable of formula manipulation. By this means, we ordinarily get apparently redundant formulas (see the example in section 1) and, what is worse, we have to write down as many expressions as there are input variables. If we perform differentiation by hand, we are apt to make mistakes in manipulating formulas and in coding them. Even if we have access to a formula-manipulating program, it is not without difficulty that we combine the result from such a program with an ordinary program written in FORTRAN, say.

This is probably one of the reasons why the other means, i.e., "numerical differentiation", has often been used in practice. However, as is well known, the simplest formula for numerical differentiation reduces the number of significant digits, roughly speaking, to half. Furthermore, if there are n input variables, we have to compute the values of the function for at least n different sets of values of the input variables.

Thus, whichever means we may use, cumbersome formula manipulation or inaccurate numerical differentiation, the amount of necessary computation will increase with the number of input variables.

In contrast, the method we have developed in this paper is more practical than formula manipulation, gives more accurate results than numerical differentiation, and is more efficient than either of them. Moreover, it is easy to implement in a FORTRAN compiler or a preprocessor, which we are trying to do including also the extension to the case discussed in section 8 [7].

Analysis of rounding errors in numerical computation has remained to be a theoretical subject in spite of its practical importance, probably because it is difficult to get a reasonable estimate for the bound for rounding errors in parallel with computation of functions. ("Interval algebra" [12] is terribly laborious and often gives uselessly too big error bounds.) It is now possible, as we showed in section 5.

7. Remarks on the Case of Several Functions and on Higher Derivatives

It is possible immediately to extend the algorithm developed in section 4 and section 5 to the case where there are more than one function.

Stage 1 and Stage 2 are to be performed as they are described in Algorithm A (at the beginning of section 4). Then, Stage 3 (as well as Stage 4 if we want it) is to be performed for each of the functions separately with Method 2 (of section 3). It might be possible to devise an algorithm to simultaneously compute the partial derivatives of all the functions more efficiently than to compute them for individual functions separately, but it seems that we have not yet known such an algorithm which is applicable to the general case (see [4] for analogous problems for the shortest paths). Even if we compute them for each function separately, we have only to make computation of Stages 1 and 2 once for all. The extra operations needed for computing for another function are at most $|A_U|$ multiplications and $|A_U| - |V_X \cup V_U|$ additions at Stage 3.

If there are more functions than input variables, Method 1 (of section 3) might better be used than Method 2 at Stage 3. However, if we want to find the estimate of rounding errors (Stage 4), Method 2 will still be recommendable.

We have so far considered the first derivatives, or the gradient, of a function. It is in principle possible to apply the same technique to higher derivatives. For example, for the second derivatives, or the Hessian, of f, we may proceed as follows.

We can write down all the computations we have done at Stages 1 to 3 for a function and its gradient in the form of a computational scheme or a computational graph. The computational graph representing the computations for the function and the gradient by Algorithm A (of section 4) will consist of three parts, like that shown in Fig. 7.1, which we shall call the "extended computational graph".

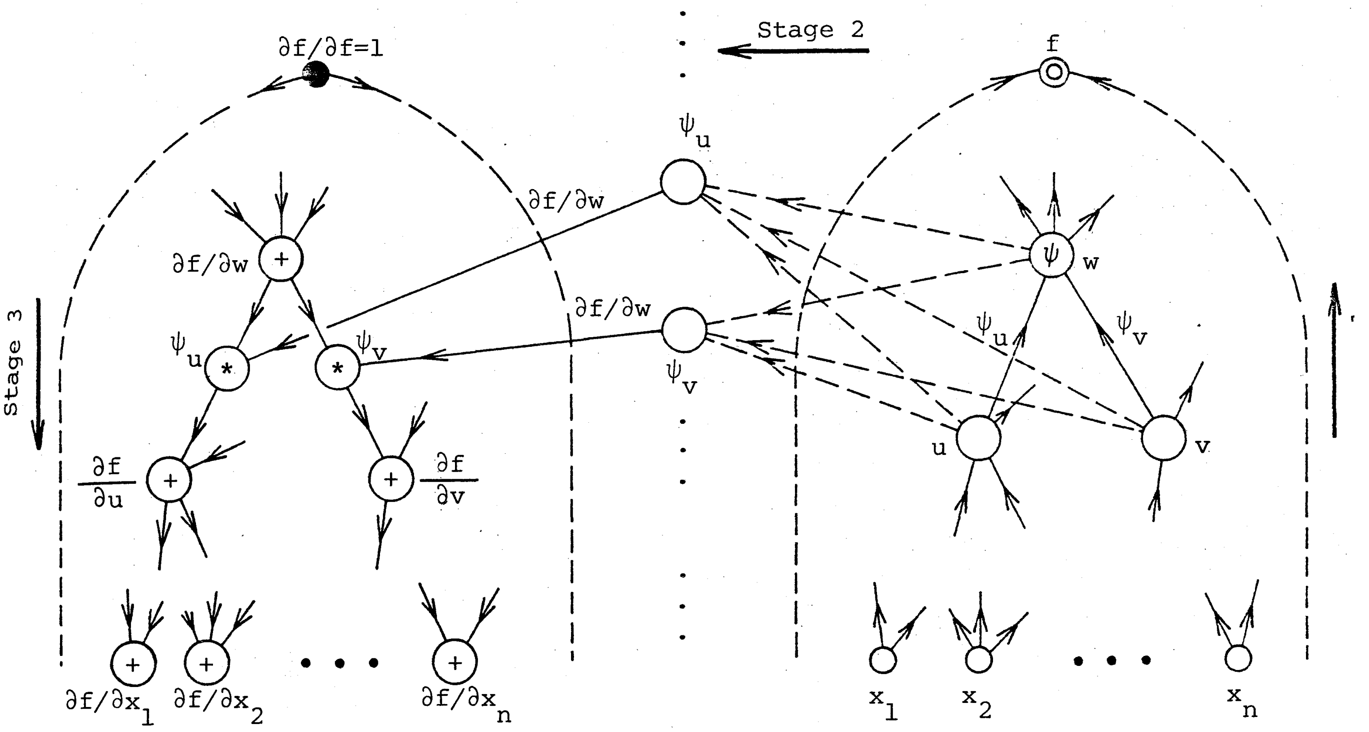


Fig. 7.1. Extended computational graph for "gradient"

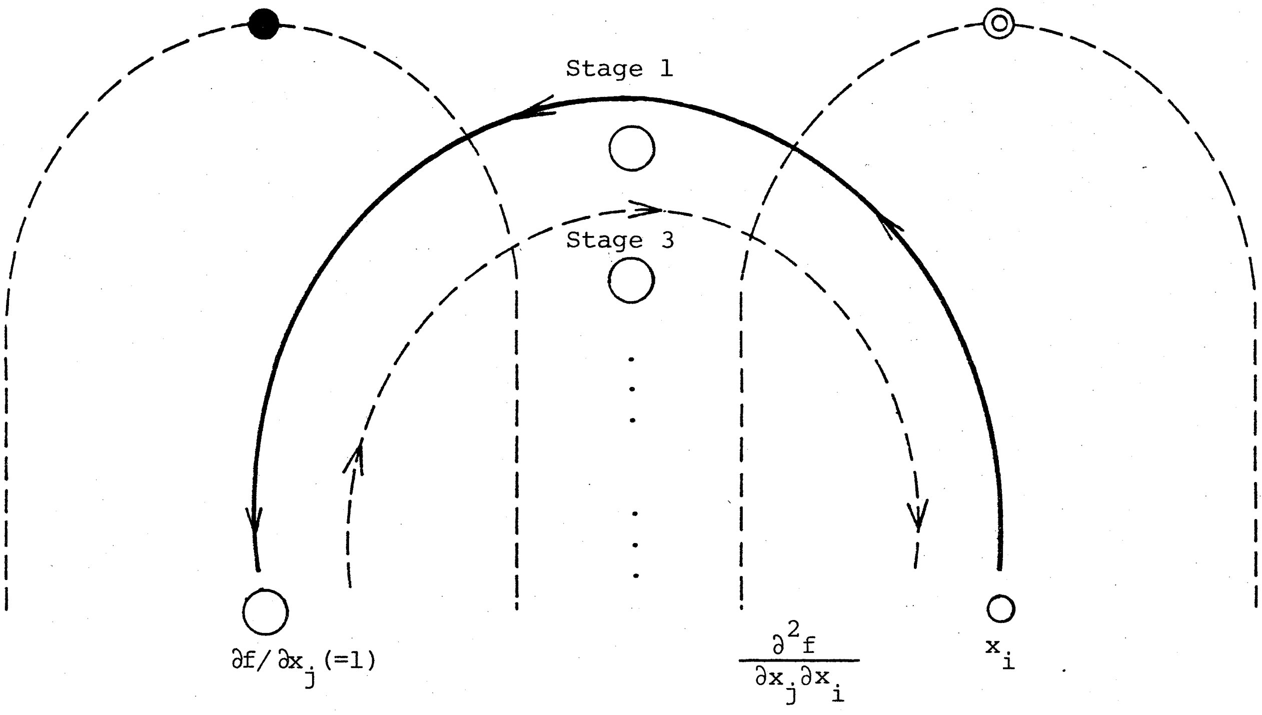


Fig. 7.2. Computation of the Hessian on the extended computational graph

The right half of the extended computational graph is the same as the original computational graph for the function, which represents the computation at Stage 1. The left half has almost the same topology as the right half but with the orientation of the arcs reversed and with one additional multiplication vertex in the middle of each arc. All the vertices corresponding to the intermediate variables, except that corresponding to f , in the left half, are addition vertices, and they are the vertices representing the (now intermediate) "variables" $\frac{\partial f}{\partial v}$'s . In between the right half and the left half, there are as many additional vertices as the arcs of A_U in the original computational graph, representing the "variables" d(a)'s (i.e., "elementary partial derivatives"). They are connected to the additional multiplication vertices in the left half to represent, together with the left half, the computation at Stage 3. The broken-line arcs connecting the vertices of the right half to the vertices for the elementary partial derivatives represent the computation of the latter from the input and intermediate variables of the right half at Stage 2. Note that the connection represented by the broken-line arcs is "local", i.e., it is confined within the variables which appear in one and the same step of computation at Stage 1.

Regarding this extended computational graph as the given computational graph, we may apply Algorithm B (of section 4) starting from each $\frac{\partial f}{\partial x_i}$ -vertex at the bottom of the left half, to get $\frac{\partial^2 f}{\partial x_i \partial x_j}$'s at the bottom of the right half (see Fig. 7.2).

Since the extended computational graph is approximately three times as large as the original, we can compute all the second derivatives $\frac{\partial^2 f}{\partial x_i \partial x_j}$ for each fixed i in the time which is of the same order as we compute the function itself. If there are n input variables, we can find all the n^2 second derivatives in time O(n * F) , where F is the time complexity of the computation of the function.

In Fig. 7.1, the elementary partial derivatives for the arcs of the extended computational graph are also shown, except for the broken-line arcs representing the computation of elementary partial derivatives. The structure of the part consisting of the broken-line arcs is most chracteristic to the extended computational graph. It is illustrated in Fig. 7.3 for each of the typical basic operations.

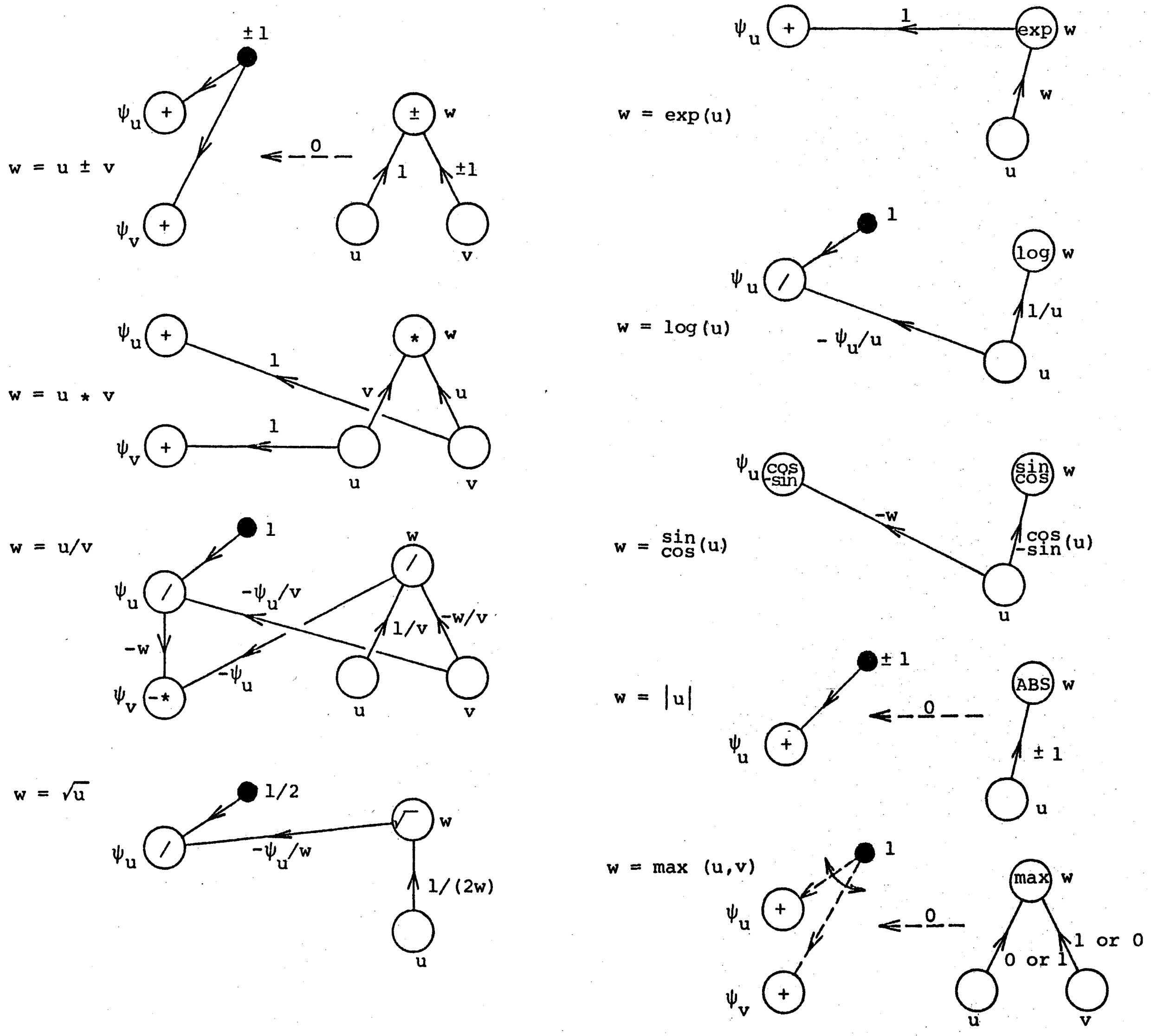


Fig. 7.3. Part of the extended computational graph for the elementary partial derivatives of typical basic operations, with the elementary (second) derivatives attached to arcs

As is seen from Fig. 7.3, this part is as simple in structure as, sometimes even simpler than, the other part.

8. Remarks on Non-straight-line Computations

Throughout the preceding sections, we have taken it for granted that a function, or functions, is defined in terms of a computational scheme or a computational graph, i.e. a so-called "straight-line program". However, the functions we have to deal with in practical situations are often defined in terms of a program containing conditional branches such as IF- and DO-statements (say, in FORTRAN). Even to such a case, we can apply the technique developed in this paper. In fact, we may imagine of (possibly infinitely) many computational graphs for all the possible combinations of conditions, and suppose that one of them is realized for each particular set of input data. It will also be a realistic assumption that, for "almost all" values of input data, any sufficiently small perturbation of the values of input data does not change the corresponding computational graph. Then, so long as the computational graph remains the same, we can apply our technique to find the partial derivatives. Thus, the problem is to determine the computational graph which is realized under the given condition. This is not possible a priori, but can be done on the run time. We may log the "history" of computation we have actually done as we proceed the computation, to complete the computational scheme realized under that particular condition. To that computational scheme we may apply the procedures of Stages 2 and 3 (of section 4) to get the partial derivatives. By means of this approach we can substantially extend the area of application of our technique.

If we apply the same technique to exceptional cases where the computational graph may change as the values of input data are subject to some perturbation however small it may be, we shall obtain a directional derivative, provided there is a perturbation which does not change the computational graph.

In the case where every perturbation of the input data changes the computational graph, no general strategy seems to work well.

9. Application to Systems of Ordinary Differential Equations

The technique we have studied in this paper may conceptually be applied to the numerical problems containing ordinary differential equations if we consider infinite graphs. To be specific, let us choose two examples from among the problems often encountered in mathematical treatments of systems.

9.1. Identification of system parameters. Suppose we are given the mathematical model of a real-world system in the form of a system of ordinary differential equations in y^κ ($\kappa = 1,\ldots, n$)

$$\frac{dy^\kappa}{dt} = f^\kappa(y,\ t;\alpha) \qquad (\kappa = 1,\ldots, n) \tag{9.1}$$

containing a certain number of unknown parameters α^i ($i = 1,\ldots, m$) as well as a set of observational data of the real system in the form of the functions $\tilde{y}^\kappa(t)$ ($\kappa = 1,\ldots, n$) of t on a time interval $[a,b]$, and we are required to determine the values of the parameters α^i so as to minimize a given objective functional of the form

$$I = \int_a^b F(y(t);\ t)\ dt\ , \tag{9.2}$$

where $y^\kappa(t)$'s are the solution of the initial-value problem of (9.1) with the initial conditions

$$y^\kappa(a) = \tilde{y}^\kappa(a) \qquad (\kappa = 1,\ldots, n)\ . \tag{9.3}$$

(For example, we take $F(y;\ t) = w(t)\cdot\|y(t) - \tilde{y}(t)\|^2$ for the least-square criterion with a weight function $w(t)$, where $\|\cdot\|$ is a certain norm.)

To determine the values of α^i's , the standard method will be to start from some appropriately chosen initial guess of α^i's , to numerically solve the system of ordinary differential euqations (9.1) under the initial conditions (9.3), to compute the value of I as well as the values of its derivatives $\frac{\partial I}{\partial \alpha^i}$'s with respect to α^i's , to "improve" the values of α^i's using the values of I and $\frac{\partial I}{\partial \alpha^i}$'s , and to repeat the same until convergence is attained.

Here we face the problem of simultaneously computing I and $\frac{\partial I}{\partial \alpha^i}$'s. In order to find the continuum counterpart of our technique, we resort to a kind of heuristics, i.e., the discretization of the problem followed by the re-continuization. Thus, we first replace the differential equations (9.1) and the integral (9.2) by any discrete approximation scheme, say, by the Euler formula

$$y^K(t_{\ell+1}) = y^K(t_\ell) + h\cdot f^K(y(t_\ell),\ t_\ell;\alpha) \qquad (\ \ell = 0,\ 1,\dots,\ N-1\ ) \qquad (9.4)$$

and by the rectangle formula

$$I = h\ \cdot\ \sum_{\ell=1}^{N} F(y(t_\ell);\ t_\ell)\ , \qquad (9.5)$$

respectively, where

$$t_{\ell+1} = t_\ell + h \qquad (\ \ell = 0,\ 1,\dots,\ N-1\ )\ ,\quad h = (b-a)/N\ . \qquad (9.6)$$

Unless we are given explicit expressions of f^K's as functions of t , y^K's and α^i's , we have to regard f^K's themselves as $(n+m+1)$-ary basic operations with y^K's , α^i's and t as operands. We may regard $\tilde{y}^K(a)$'s in the initial conditions as scalars in this context.

It is interesting to see that the algorithm of section 4, when applied to the computational graph thus constructed, makes a discrete approximation to the continuous problem of finding $G_K(t)$'s , for $t \in [a,b]$, satisfying the system of differential equations

$$\frac{dG_\kappa}{dt} = - \sum_{\lambda=1}^{n} G_\lambda \cdot \frac{\partial f^\lambda}{\partial y^\kappa} - \frac{\partial F}{\partial y^\kappa} \qquad (\kappa = 1,\ldots, n) \tag{9.7}$$

and the initial (or, more precisely, the terminal) conditions

$$G_\kappa(b) = 0 \qquad (\kappa = 1,\ldots, n) , \tag{9.8}$$

and then calculating the integrals

$$\frac{\partial I}{\partial \alpha^i} = \int_a^b \sum_{\kappa=1} G_\kappa \cdot \frac{\partial f^\kappa}{\partial \alpha^i} dt \qquad (i = 1,\ldots, m) . \tag{9.9}$$

The complexity of numerically solving (9.7), (9.8) and computing (9.9) by discretizing the interval $[a,b]$ into N meshes is

$O(n^2 \cdot N)$ for the solution of (9.7) and (9.8)

and

$O(m \cdot n \cdot N)$ for the integration of (9.9).

This approach to the computation of $\frac{\partial I}{\partial \alpha^i}$'s should be compared with the standard approach which first solves the m perturbation equations of (9.1):

$$\frac{d\eta_i^\kappa}{dt} = \sum_{\lambda=1}^{n} \frac{\partial f^\kappa}{\partial y^\lambda} \cdot \eta_i^\lambda + \frac{\partial f^\kappa}{\partial \alpha^i} \qquad (\kappa = 1,\ldots, n) \tag{9.10}$$

with the initial conditions

$$\eta_i^\kappa(a) = 0 \qquad (\kappa = 1,\ldots, n) \tag{9.11}$$

for $i = 1,\ldots, m$, and then calculates the m integrals

$$\frac{\partial I}{\partial \alpha^i} = \int_a^b \sum_{\kappa=1}^{n} \frac{\partial F}{\partial y^\kappa} \cdot \eta_i^\kappa dt \qquad (i = 1,\ldots, m) . \tag{9.12}$$

With the same discretization as the above, the complexity of the latter approach will be

$O(m \cdot n^2 \cdot N)$ for the solution of (9.10) and (9.11)

and

$O(m \cdot n \cdot N)$ for the integration of (9.12) .

The essential difference between the two approaches consists in that we have to solve m systems of ordinary differential equations by the latter approach whereas we have only to solve a single system by the former.

It may be of some interest to note that $G_\kappa(t)$ can be interpreted as the Lagrange multipliers for the problem of minimizing (9.2) under the constraints (9.1) and (9.3) in the traditional framework of theory of ordinary differential equations.

9.2. Solving the two-point boundary-value problem by the shooting method with primitive descent. The two-point boundary-value problem we shall consider is constituted from the system of ordinary differential equations

$$\frac{dy^\kappa}{dt} = f^\kappa(y, t) \qquad (\kappa = 1,\dots, n) \tag{9.13}$$

for $t \in (a,b)$ and the boundary conditions

$$y^\kappa(a) = \alpha^\kappa \qquad \text{for } \kappa \in K_1 \ (\subset \{1,\dots, n\}), \tag{9.14}$$

$$y^\kappa(b) = \beta^\kappa \qquad \text{for } \kappa \in K_2 \ (\subset \{1,\dots, n\}), \tag{9.15}$$

where K_1 and K_2 are subsets of $\{1,\dots, n\}$ such that

$$|K_1| + |K_2| = n . \tag{9.16}$$

The shooting method for solving this two-point boundary-value problem converts the problem into the initial-value problem with the same system of differential equations (9.13) and the initial conditions (9.14) and

$$y^\kappa(a) = \alpha^\kappa \quad \text{for} \quad \kappa \in \{1,\dots, n\} - K_1 \tag{9.17}$$

with $m = n - |K_1|$ ($= |K_2|$) unknown parameters α^κ ($\kappa \in \{1,\dots, n\} - K_1$), and requires to determine the values of the m unknown parameters in such a way that

$$I = F(y(b)) \equiv \sum_{\kappa \in K_2} | y^\kappa(b) - \beta^\kappa|^2 \tag{9.18}$$

becomes minimum (desirably, zero), where $y^\kappa(b)$'s are the values at $t = b$ of the solution of the initial-value problem (9.13), (9.14), (9.17).

Here again, we need $\frac{\partial I}{\partial \alpha^\kappa}$'s for $\kappa \in \{1,\dots, n\} - K_1$. Entirely in the same manner as we did for the problem in section 9.1, we are led to the "single" system of ordinary differential equations

$$\frac{dG_\kappa}{dt} = - \sum_{\lambda=1}^{n} G_\lambda \cdot \frac{\partial f^\lambda}{\partial y^\kappa} \qquad (\kappa = 1,\ldots, n) \tag{9.19}$$

for $t \in (a,b)$ with the initial (or, more precisely, the terminal) conditions

$$G_\kappa(b) = \frac{\partial F}{\partial y^\kappa(b)} \qquad \text{for } \kappa \in K_2 , \tag{9.20}$$

$$G_\kappa(b) = 0 \qquad \text{for } \kappa \in \{ 1,\ldots, n \} - K_2 , \tag{9.21}$$

from the solution of which we can readily get

$$\frac{\partial I}{\partial \alpha^\kappa} = G_\kappa(a) \qquad \text{for } \kappa \in \{ 1,\ldots, n \} - K_1 . \tag{9.22}$$

The complexity of the solution of (9.19), (9.20), (9.21) is $O(n^2 \cdot N)$ if we discretize the interval $[a,b]$ into N meshes. This is in contrast with the complexity $O(m \cdot n^2 \cdot N) + O(m^2)$ of the standard approach where we first solve the m system of perturbation equations

$$\frac{d\eta_\lambda^\kappa}{dt} = \sum_{\mu=1}^{n} \frac{\partial f^\kappa}{\partial y^\mu} \cdot \eta_\lambda^\mu \qquad (\kappa = 1,\ldots, n) \tag{9.23}$$

with the initial conditions

$$\begin{aligned} \eta_\lambda^\kappa(a) &= 0 \quad \text{for } \kappa \neq \lambda , \\ \eta_\lambda^\kappa(a) &= 1 \quad \text{for } \kappa = \lambda \end{aligned} \tag{9.24}$$

for $\lambda \in \{ 1,\ldots, n \} - K_1$ (i.e., for each unknown parameter α^λ), and then calculate

$$\frac{\partial I}{\partial \alpha^\lambda} = \sum_{\kappa \in K_2} \frac{\partial F}{\partial y^\kappa(b)} \cdot \eta_\lambda^\kappa(b) \qquad (\lambda \in \{ 1,\ldots, n \} - K_1) . \tag{9.25}$$

Here again, $G_\kappa(t)$ may be interpreted as the Lagrange multipliers.

Acknowledgements

Personal conversation with Professor Volker Strassen which the author could have when he visited Japan, as well as subsequent personal communication with him, greatly stimulated the author's ideas in this paper, for which the author is deeply grateful to him. The author thanks also Dr. K. Murota of the University of Tsukuba for his valuable comments, Mr. N. Iwata, one of the author's graduate students, for his collaboration in implementing the ideas as a computer program, and Mr. T. Araki, a younger colleague of the author's, for reading the manuscript and suggesting improvements.

References

[1] F. L. Bauer: Computational graphs and rounding errors. *SIAM Journal on Numerical Analysis*, Vol.11 (1974), pp.87-96.

[2] W. Baur and V. Strassen: The complexity of partial derivatives (extended version, January 1982). Unpublished note.

[3] C. Berge: *Graphes et Hypergraphes*. Dunod, Paris, 1970.

[4] S. E. Dreyfus: An appraisal of some shortest-path algorithms. *Operations Research*, Vol.17 (1969), pp.395-421.

[5] F. Harary: *Graph Theory*. Addison-Wesley, Reading 1969.

[6] M. Iri: *Network Flow, Transportation and Scheduling —— Theory and Algorithms*. Academic Press, New York, 1969.

[7] M. Iri and N. Iwata: Automatic computation of partial derivatives. (in preparation)

[8] M. Iri, J. Tsunekawa and K. Murota: Graph-theoretic approach to large-scale systems of equations —— Structural solvability and block-triangularization (in Japanese). *Transactions of Information Processing Society of Japan*, Vol.23 (1982), pp.88-95. (English translation available from the authors)

[9] K. Murota: Structural Solvability and Controllability of Systems. Doctor's dissertation at the Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, April 1983.

[10] M. Jerrum: private note (August 1982). Personally communicated by V. Strassen to the author, December 1983.

[11] J. H. Wilkinson: Error analysis of floating-point computation. Numerische Mathematik, Bd.2 (1960), pp.319-340.

[12] R. E. Moore: Interval Analysis. Prentice-Hall, Inc., New Jersey, 1966.