

## SHORT NOTE

# THE NUMERICAL EVALUATION OF ANALYTICAL DERIVATIVES

J. W. PONTON

Department of Chemical Engineering, University of Edinburgh, The King's Buildings, West Mains Road,  
Edinburgh, Scotland

(Received 10 December 1981; received for publication 13 March 1982)

**Abstract**—A simple computational scheme is presented for obtaining the numerical value of the analytical derivative of a function without the generation of computer code representing the differentiated expression. The original function is supplied as a reverse Polish string, and values of the function and partial derivatives with respect to all variables are obtained simultaneously. The compactness of the algorithm and data structure make the technique particularly suitable for use on personal or minicomputers.

### INTRODUCTION

Various packages for solving sets of equations or optimising have obtained analytical derivatives by manipulation of algebraic expressions (see, e.g. [1] and [2]). The usual procedure is to take a representation of the function in conventional "algebraic" or infix notation and process it to generate a set of Fortran statements, one for the evaluation of each partial derivative of the function. Symbol manipulation packages (e.g. [3]), have been available for some time to perform such operations.

The following are features of the above approach.

(1) Most symbol manipulation packages are large and cumbersome. This is partly a result of the difficulty of directly manipulating expressions in infix notation, or alternatively of having to convert from infix to a more flexible representation, and then regenerate the infix form to produce Fortran code (as in [2]).

(2) The algebraic form of the derivatives of an expression containing quotients or higher functions is commonly more complicated than the original expression.

(3) If partial derivatives are generated, then a statement is required to evaluate each one. A large number of functions can result when the set of equations is non-sparse.

We shall seek to avoid problem (1) by replacing infix notation completely by the more compact and easily manipulable Reverse Polish Notation (RPN).

Problem (2) is unavoidable, but its existence should make one consider carefully the reasons for wishing to use analytical derivatives in the first place. If the derivative is more complex, and hence takes longer to evaluate than the function, then numerical differentiation would be faster. The object of using analytical derivatives thus cannot be to speed function and derivative evaluation, it must be to obtain faster convergence by using a true tangent rather than a secant. Speed is indeed often a secondary consideration to the requirement to get to the solution at all.

The present method avoids problem (3) altogether by storing only the representation of the basic function and not of each of its derivatives. This incurs some com-

putation time overhead, but, as noted above, speed is often not the main objective.

### *Evaluation of RPN functions and their derivatives*

RPN is by now familiar to most, particularly to owners of Hewlett-Packard calculators. The expression, which is a string of operators and operands, is evaluated on a stack. Starting from the left of the expression string and moving right:

(a) an operand, i.e. a constant or the numerical value of a symbolic item, is "pushed" onto the stack;

(b) a unary operator implies that the operation specified be performed on the top element of the stack, the value of the result replacing that element;

(c) a binary operation is performed between the top two elements of the stack, the single result replacing the two top elements, thus causing the stack to be "popped", i.e. to become one element shorter.

Finally, the stack holds a single element, this being the value of the complete expression.

A selection of binary and unary operators are defined in Table 1, using the notation:  $T$  = top element of stack, and  $M$  = next element below the top of stack.

To evaluate the derivative of an R.P. expression we use a second "derivative" stack in parallel with the first or main stack. This will contain the values of the derivatives of corresponding elements of the main stack. On to it are pushed the derivatives of operands, constant or variable, at the same time as their values are pushed on to the main stack: the derivative of a constant is of course zero, and that of a variable (w.r.t. itself) is unity.

When operators are applied to the main stack, the corresponding derivative operators must be applied. These can readily be defined in terms of the usual rules for differentiation of simple functions, see Table 2. In general these operate on the top, or top two, elements of *both* main and derivative stacks. Simultaneous operations on both stacks are thus necessary, so that function and derivative are evaluated together; this is no disadvantage, as most applications require both values.

The simultaneous evaluation of a function of one variable and its derivative is illustrated in Table 3.

Simultaneous evaluation of partial derivatives

The operations and logic for differentiation one function w.r.t. several variables, i.e. the evaluation of a complete Jacobian row, are the same for each variable, only the derivatives pushed on to the derivative stack are

different. It is thus convenient to generate *n* partial derivative values simultaneously on *n* derivative stacks, together with a function stack. Table 4 illustrates a 3 variable example.

After evaluation, the top elements of each stack hold the partial derivative values. If evaluation continues with a further RPN function string, then the top two elements of each stack will hold, in order, the two Jacobian rows. Eventually, after *n* function strings, the stack will be *n* elements high, and will hold the complete Jacobian.

Table 1. RP arithmetic operators

| (T = top element of stack, M = next element) |                          |             |
|--|--------------------------|-------------|
| +  | $M + T \rightarrow M$    | T discarded |
| -  | $M - T \rightarrow M$    | "           |
| *  | $M * T \rightarrow M$    | "           |
| /  | $M / T \rightarrow M$    | "           |
| square                                       | $T^2 \rightarrow T$      | T replaced  |
| sqrt   | $\sqrt{T} \rightarrow T$ | "           |

Table 2. RP "Differentiated" operators

| (T', M' top elements of derivative stack<br>T, M top elements of main stack) |  |              |
|--|--|--------------|
| +'   | $M' + T' \rightarrow M'$                 | T' discarded |
| -'   | $M' - T' \rightarrow M'$                 | "            |
| *'   | $M' * T + T' * M \rightarrow M'$         | "            |
| /'   | $(M' * T - T' * M) / T^2 \rightarrow M'$ | "            |
| square'  | $2 * T * T' \rightarrow T'$              | T' replaced  |
| sqrt'  | $- 0.5 / \sqrt{T} * T' \rightarrow T'$   | "            |

Procedures and functions

Any system for solving extensive sets of equations, e.g. a "flowsheeting" system, must allow the user to supply procedures or functions for common operations such as physical property calculations. There will exist only one copy of the equations for these, but each "activation" will have its own variable or constant values.

The present method lends itself readily to this. The equations associated with the function are written in RPN, and take their parameters, i.e. the appropriate values of variables and their derivatives, from the top of the stack where they have been pushed before calling the function. The evaluation of the function and its derivatives proceeds as described before, and the result or results, and derivatives, are returned on top of the stack.

Comments and applications

It might appear that the technique described above will be "inefficient" in that (a) it is interpretive, and (b) it involves redundant operations, such as multiplication by zero and one, and addition of zero.

While it is certainly true that an interpreter written in,

Table 3. Evaluation of (x+7)(3+x<sup>2</sup>)

| RPN form: x7 + 3x square + *   |               |              |
|--|---------------|--------------|
| Evaluate at x = 5, together with derivative w.r.t. x                                 |               |              |
| Operation  | Stacks        |              |
|  | main          | derivative   |
|  | S             | S'           |
| (1) Push 5 (= x) and 7 on S, and their derivatives, 1 and 0, on S'                   | 7<br>5        | 0<br>1       |
| (2) Apply + to S and +' to S'  | 12            | 1            |
| (3) Push 3 and 5 (= x) on S, and 0 and 1 on S'                                       | 5<br>3<br>12  | 1<br>0<br>1  |
| (4) Apply square (5 x 5) to S, and square' (2 x 5 x 1) to S', replacing top elements | 25<br>3<br>12 | 10<br>0<br>1 |
| (5) Apply + and +'   | 28<br>12      | 10<br>1      |
| (6) Apply * to S, and * (28 x 1 + 10 x 12) to S'                                     | 366           | 148          |
| The stacks now contain function and derivative.                                      |               |              |

Table 4. Evaluation of  $3x_1 + 2x_1x_2 + x_3^{0.5}$ 

| RPN form: $3x_1 * 2x_1 * x_2 * x_3 \text{ sqrt} + +$                  |              |   |             |                |
|---|--------------|---|-------------|----------------|
| at $\underline{x} = (2, 3, 4)$  |              |   |             |                |
| Operation   | Stacks       |   |             |                |
|   | main<br>s    | derivatives<br>s <sub>1</sub> s <sub>2</sub> s <sub>3</sub> |             |                |
| (1) Push 3,2 (= $x_1$ ) and derivatives                               | 2<br>3       | 1<br>0  | 0<br>0      | 0<br>0         |
| (2) Apply * and *'; note *' gives 0 except                            | 6            | 3   | 0           | 0              |
| (3) Push 2,2 (= $x_1$ )   | 2<br>2<br>6  | 1<br>0<br>3   | 0<br>0<br>0 | 0<br>0<br>0    |
| (4) Apply *, *'   | 4<br>6       | 2<br>3  | 0<br>0      | 0<br>0         |
| (5) Push 3 (= $x_2$ )   | 3<br>4<br>6  | 0<br>2<br>3   | 1<br>0<br>0 | 0<br>0<br>0    |
| (6) Apply *, *'; note non zero result for *' except on s <sub>3</sub> | 12<br>6      | 6<br>3  | 4<br>0      | 0<br>0         |
| (7) Push 4 (= $x_3$ )   | 4<br>12<br>6 | 0<br>6<br>3   | 0<br>4<br>0 | 1<br>0<br>0    |
| (8) Apply sqrt, sqrt'   | 2<br>12<br>6 | 0<br>6<br>3   | 0<br>4<br>0 | 0.25<br>0<br>0 |
| (9) Apply +, +'   | 14<br>6      | 6<br>3  | 4<br>0      | 0.25<br>0      |
| (10) Apply +, +' again  | 20           | 9   | 4           | 0.25           |

say, Fortran, will be slower than the execution of compiled code, it is worth noting that many language systems are nowadays interpretive: Basic of course, most Pascal implementations, and all languages on the popular UCSD operating system, including Fortran. If the interpreter for differentiation is written at the same level as the main language interpreter, then no additional inefficiency will be incurred.

The redundant operations can be trapped; if this too is done at language interpretation level it will be fast in comparison to the floating point arithmetic operations.

Against these, probably surmountable, disadvantages must be set the very considerable compactness, both in code and storage, of this method. A "desk calculator" differentiator, written in Basic, with the facility to handle expressions of up to 26 variables entered from the keyboard, requires only 65 lines of code. A run time stack depth of 6 has proved adequate for expressions of substantial complexity.

Finally the application for which the method was devised should be mentioned. This was the solution of chemical equilibrium problems involving simultaneous reactions, on a small computer. We wished to work with the equilibrium equations in the form:

$$K_R = \frac{\pi\{\text{products}\}}{\pi\{\text{reactants}\}}$$

rather than the more usual, and easily differentiable form:

$$\log K_R = \Sigma\{\text{products}\} - \Sigma\{\text{reactants}\}.$$

The properties of these equations are well suited to such a method:

- (1) The functions involved are complex and not convenient to differentiate.
- (2) Difficulties can arise in obtaining convergence with numerical differentiation.
- (3) Although the number of equations is not generally large, nor are they particularly sparse.

Another similar class of problem which might bear investigation is the calculation of nonideal vapour-liquid equilibria.

#### REFERENCES

1. R. Hernandez and R. W. H. Sargent, *12th Symp. of Computer Applications in Chemical Engineering*, Montreux, pp. 643-657 (1979).
2. T. G. Koup, E. H. Chimowitz, A. Blanz & L. F. Stutzman, *Comput. Chem. Engng* 5, 151 (1981).
3. H. Strubbe, *Comput. Phys. Commun.* 8, 1 (1974).