Appl. Math. Lett. Vol. 6, No. 3, pp. 47–50, 1993
Printed in Great Britain.

0893-9659/93 \$6.00 + 0.00

AUTOMATIC DIFFERENTIATION AND THE STEP COMPUTATION IN THE LIMITED MEMORY BFGS METHOD

JEAN CHARLES GILBERT
INRIA, Domaine de Voluceau, Rocquencourt
B.P. 105, 78153 Le Chesnay, Cedex, France

JORGE NOCEDAL*
Department of Electrical Engineering and Computer Science
Northwestern University, Evanston, IL 60208, U.S.A.

(Received and accepted January 1993)

Abstract—It is shown that the two-loop recursion for computing the search direction of a limited memory method for optimization can be derived by means of the reverse mode of automatic differentiation applied to an auxiliary function.

1. INTRODUCTION

The problem of finding efficient procedures for automatically calculating the gradient of a function has recently received much attention [1]. It is known that the *reverse mode* of automatic differentiation can provide the value of the gradient at a cost that is not much greater than that required to evaluate the function.

On the other hand, the limited memory BFGS method—an optimization method designed for very large unstructured problems—has recently become popular. This method attempts to mimic the very successful BFGS variable metric method, but without storing matrices. To do this, it saves several pairs of vectors $\{y_i, s_i\}$, $i = 1, \ldots, m$ that implicitly define the iteration matrix H. The search direction of the limited memory method is then computed by $d = -Hg$, where H can be viewed as an approximation of the inverse Hessian of the objective function at the current iterate and g is the gradient of this function. Since the matrix H is not formed but is only represented implicitly by the set of vectors $\{y_i, s_i\}$, a formula [2] is required to calculate the product Hg directly from the vectors $\{y_i, s_i\}$ and the gradient g. It turns out that this formula is not unique; one can devise various equivalent expressions, some of which are more economical than others [3]. For unconstrained optimization, the most efficient formula for computing d, given in [2], consists of a two-loop recursion involving the vectors $\{y_i, s_i\}$ and the gradient g. In this paper, we show that this two-loop recursion can be viewed as an application of the reverse mode of automatic differentiation. Thus, we find a connection between two apparently unrelated subjects: the step computation in a limited memory method and automatic differentiation.

2. ADJOINT CODE OF A PROGRAM COMPUTING A FUNCTION

Suppose that a function

$$f : x = (x_1, \ldots, x_n) \in \mathbb{R}^n \longrightarrow f(x) \in \mathbb{R}$$

is computed by a program executing the following sequence of instructions

$$\begin{array}{l} \textbf{for}\ \ k := 1\ \ \textbf{to}\ \ K\ \ \textbf{do}\ \ x_{\mu_k} := \varphi_k(\{x_{P_k}\}); \\ f := x_N. \end{array} \tag{1}$$

*This author was supported by the National Science Foundation Grant INT-9101901, and by the Department of Energy Grant DE-FG02-87ER25047.

For convenience, we have denoted by $x_1, \ldots, x_n$ the *independent variables*, with respect to which the gradient of f is desired, and by $x_{n+1}, \ldots, x_N$ $(N \geq n)$ all the other variables used in the program. There is, however, no meaningful ordering in this notation. Instruction k in (1) modifies the variable x_{μ_k}, $\mu_k \in \{1, \ldots, N\}$, by using an intermediate function φ_k depending on $\{x_{P_k}\}$, which is the collection of variables x_j with indices j in some subset $P_k \subset \{1, \ldots, N\}$. After assigning a value to the variable x, this program will provide the value of $f(x)$ in the variable x_N.

It has been shown [4] that the gradient $\nabla f(x)$ of f at a given point x can be evaluated by first executing the original program (1), storing some partial derivatives $\frac{\partial \varphi_k}{\partial x_j}$ for $j \in P_k$, and then executing the following *adjoint code*:

$$
\begin{array}{l}
\textbf{for} \;\; i := 1 \;\; \textbf{to} \;\; N-1 \;\; \textbf{do} \;\; \bar{x}_i := 0; \\
\bar{x}_N := 1; \\
\textbf{for} \;\; k := K \;\; \textbf{down to} \;\; 1 \;\; \textbf{do} \;\; \{ \\
\quad \bar{x}_i := \bar{x}_i + \frac{\partial \varphi_k}{\partial x_i} \bar{x}_{\mu_k}, \; \forall i \in P_k \backslash \{\mu_k\}; \\
\quad \bar{x}_{\mu_k} := \frac{\partial \varphi_k}{\partial x_{\mu_k}} \bar{x}_{\mu_k}; \} \\
\textbf{for} \;\; i := 1 \;\; \textbf{to} \;\; n \;\; \textbf{do} \;\; \frac{\partial f}{\partial x_i} := \bar{x}_i.
\end{array}
\tag{2}
$$

Here we used the notation $A \backslash B$ to denote the set of elements that belong to A but not to B. The variable $\bar{x}_i$ is called the *adjoint variable* associated to x_i. Its value is the current evaluation of the derivative of f with respect to x_i. This technique is also called the *reverse mode* of automatic differentiation; see [5–8] for an introduction to the subject. Its main advantage is that it computes $f(x)$ and its gradient $\nabla f(x)$ in a time $T(f, \nabla f)$ satisfying

$$
T(f, \nabla f) \leq C\, T(f), \tag{3}
$$

where C is some constant and $T(f)$ is the time to compute $f(x)$ by Algorithm (1). When the functions φ_k are restricted to be the ones available in FORTRAN, it is reasonable to bound C by 5, see [6].

3. APPLICATION TO THE L-BFGS METHOD

In a typical iteration of the limited memory BFGS method (L-BFGS), one is given a symmetric and positive definite matrix H_0 and m pairs of vectors $\{y_0, s_0\}, \ldots, \{y_{m-1}, s_{m-1}\}$, where m is some integer. Each of these pairs satisfies the condition $\rho_i \equiv (y_i^T s_i)^{-1} > 0$. The matrix H_0 is then updated m times using the BFGS formula, i.e., for $i = 0, \ldots, m-1$:

$$
H_{i+1} = V_i^T H_i V_i + \rho_i s_i s_i^T, \tag{4}
$$

where

$$
V_i = I - \rho_i y_i s_i^T; \tag{5}
$$

see [2]. The resulting matrix, H_m, is then used to compute the search direction

$$
d = -H_m g,
$$

where g is the current value of the gradient of the objective function.

To interpret this step computation in terms of automatic differentiation, we need to express $H_m g$ as the gradient of a real-valued function. A natural candidate for this is

$$
f_m(g) = \frac{1}{2} g^T H_m g,
$$

since, due to the symmetry of H_m, $\nabla_g f_m(g)$ is precisely $H_m g$. From (4), we find that $f_m(\cdot)$ can easily be expressed in terms of $f_{m-1}(\cdot)$:

$$
f_m(g) = f_{m-1}(V_{m-1}\, g) + \frac{\rho_{m-1}}{2} (s_{m-1}^T g)^2.
$$

Therefore, if we define

$$
q_m = g, \quad q_{i-1} = V_{i-1} q_i, \qquad \text{for } i = m, \ldots, 1, \tag{6}
$$

we find by induction that

$$f_m(g) = f_0(q_0) + \sum_{i=0}^{m-1} \frac{\rho_i}{2} (s_i^T q_{i+1})^2.$$

Using this formula, the computation of $f_m(g)$ can be performed by the following algorithm. We assume that the scalars ρ_i have been computed beforehand, and to save storage, we place all the q_i in the same vector q. Recall that the vectors q_i are updated by (6), where the matrices V_i are defined by (5).

$$\begin{array}{l} f := 0; \\ q := g; \\ \textbf{for } i := m-1 \textbf{ down to } 0 \textbf{ do } \{ \\ \quad \alpha_i := s_i^T q; \\ \quad f := f + \frac{\rho_i}{2} \alpha_i^2; \\ \quad q := q - \rho_i \alpha_i y_i; \} \\ f := f + \frac{1}{2} q^T H_0 q. \end{array} \tag{7}$$

The product $H_m g$ will now be computed by the adjoint code of (7). Let $\bar{f}$, $\bar{q}$, $\bar{\alpha}_0, \ldots, \bar{\alpha}_{m-1}$ be the adjoint variables corresponding to f, q, $\alpha_0, \ldots, \alpha_{m-1}$. The adjoint code is obtained by writing the adjoint instructions corresponding to the instructions in (7), in the reverse order of execution. After the initialization of the adjoint variables,

$$\bar{f} := 1; \ \bar{q} := 0; \ \bar{\alpha}_0 := 0; \ldots; \bar{\alpha}_{m-1} := 0,$$

the code continues as follows:

$$\begin{array}{l} \bar{q} := H_0 q; \\ \textbf{for } i := 0 \textbf{ to } m-1 \textbf{ do } \{ \\ \quad \bar{\alpha}_i := \bar{\alpha}_i - \rho_i y_i^T \bar{q}; \\ \quad \bar{\alpha}_i := \bar{\alpha}_i + \rho_i \alpha_i \bar{f}; \\ \quad \bar{q} := \bar{q} + \bar{\alpha}_i s_i; \\ \quad \bar{\alpha}_i := 0; \} \\ \bar{g} := \bar{q}; \\ \bar{q} := 0; \\ \bar{f} := 0. \end{array} \tag{8}$$

We now combine (7) and (8), omitting those instructions needed only for the evaluation of f (since we are only interested in $H_m g$). To save space in the adjoint portion of the code, we store all values of $\bar{\alpha}_i$ in the same location β and the values of $\bar{q}$ in the location of q. After deleting all unnecessary instructions, we obtain

$$\begin{array}{l} q := g; \\ \textbf{for } i := m-1 \textbf{ down to } 0 \textbf{ do } \{ \\ \quad \alpha_i := s_i^T q; \\ \quad q := q - \rho_i \alpha_i y_i; \} \\ q := H_0 q; \\ \textbf{for } i := 0 \textbf{ to } m-1 \textbf{ do } \{ \\ \quad \beta := \rho_i(\alpha_i - y_i^T q); \\ \quad q := q + \beta s_i; \}. \end{array} \tag{9}$$

The value of $H_m g$ is placed in the vector q. Algorithm (9) is identical to the two-loop formula used for the computation of the search direction in the L-BFGS method [2].

This derivation shows why Algorithm (9) is efficient: it is based on the compact Algorithm (7) computing f_m, and on the reverse mode of automatic differentiation, which is known to be very efficient.

REFERENCES

1. A. Griewank and G. Corliss, Eds., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Proceedings in Applied Mathematics 53, SIAM, (1991).
2. J. Nocedal, Updating quasi-Newton matrices with limited storage, *Mathematics of Computation* **35**, 773–782 (1980).
3. R.H. Byrd, J. Nocedal and R. Schnabel, Representations of quasi-Newton matrices and their use in limited memory methods, Tech. Rep. NAM-04, EECS Department, Northwestern University, (1992).
4. B. Speelpenning, Compiling fast partial derivatives of functions given by algorithms, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A., (1980).
5. L.B. Rall, *Automatic Differentiation, Techniques and Applications*, Lecture Notes in Computer Science 120, Springer-Verlag, Berlin, (1981).
6. A. Griewank, On automatic differentiation, In *Mathematical Programming: Recent Developments and Applications*, (Edited by M. Iri and K. Tanabe), pp. 83–108, Kluwer Academic Publishers, (1989).
7. D. Juedes, A taxonomy of automatic differentiation tools, In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, (Edited by A. Griewank and G.F. Corliss), SIAM, Philadelphia, (1991).
8. J.Ch. Gilbert, G. Le Vey and J. Masse, La différentiation automatique de fonctions représentées par des programmes, INRIA Research Report 1557, (1991).