AUTOMATIC HESSIANS BY REVERSE ACCUMULATION

Bruce Christianson

School of Information Sciences, University of Hertfordshire
Hatfield, Herts AL10 9AB, England, Europe

Numerical Optimisation Centre Technical Report 228, April 1990
published IMA Journal of Numerical Analysis (1992) 12, 135–150

Abstract

Let n be the number of independent variables of a function f, and let W, S respectively be the time and space bounds for the joint evaluation of $\{f, \nabla f\}$ using automatic differentiation with reverse accumulation. In this note, we examine an extension of the technique of reverse accumulation which allows the automatic extraction of the Hessian of f. The method allows the parallel evaluation of all rows of the Hessian matrix in about $2W$ time units and $3S$ space units on each of n processors, or sequential row-by-row evaluation in about $2nW$ time units and $3S$ space units on a single processor. The approach described here is intended for use with operator overloading (for example in Ada) and allows the conventional coding of the target function f.

1. Introduction and Summary. Automatic differentiation with reverse accumulation can be used to obtain the gradient vector of a function of many variables. The striking advantage of reverse accumulation is the economy of effort. If the computational cost of evaluating f is W_f then ∇f can be evaluated at an additional cost of about $3W_f$ [4, §3.3]. This cost does not explicitly depend upon the number of independent variables in f (which may be very large), but merely on the complexity of the computational graph of f.

Reverse accumulation is an alternative to gradient computation using forward accumulation [4, §3.2]. A naive approach to forward accumulation results in a gradient calculation with linear cost in the number of partial derivatives. However, a more sophisticated approach to storing and manipulating sparse data structures yields considerable improvement, in particular dramatically sublinear computational costs for the forward gradient calculation [1, Table 1 p 3][3].

As with forward accumulation, reverse accumulation allows all parallel processing permitted by the computational graph in the evaluation of f to be exploited in the evaluation of ∇f.

The disadvantages of reverse accumulation are the heavy run-time storage requirements (a naive approach to reverse accumulation gives a storage cost proportional to the number of nodes in the computational graph, ie proportional to the number of arithmetic operation executions required to calculate f), and the high degree of sophistication required from the implementor of the reverse accumulation package (overloaded arithmetic operators or pre-compiler generated code must be used to manipulate fairly sophisticated dynamically allocated data structures.) Dixon [2] suggests that these disadvantages will tend to become less compelling as the commercial cost of memory decreases and the computer-science expertise of numerical analysts (and of their tools) increases.

In this note, we examine an extension of the technique of reverse accumulation which allows the automatic extraction of the Hessian Hf of f.

Let n be the number of independent variables of the function f, and let W, S respectively be the time and space bounds for the joint evaluation of $\{f, \nabla f\}$ using reverse accumulation. Then the extended method of reverse accumulation described here allows the subsequent evaluation of an arbitrary directional second derivative of the form $(Hf)\mathbf{u}$ (where $\mathbf{u}$ is a constant vector) in about $2W$ time units and $3S$ space units. An arbitrary set of such directional derivatives may be evaluated serially or in parallel.

In particular, once the function value and gradient vector have been calculated, the method allows the parallel evaluation of all rows of the Hessian matrix in about $2W$ time units and $3S$ space units on each of n processors, or sequential row-by-row evaluation in about $2nW$ time units and $3S$ space units on a single processor.

We give an example for which the Hessian has an intrinsic computational cost of at least $\frac{1}{2}nW_f$, which shows that this bound is within a constant factor of being tight.

As with the gradient, all parallelism in the calculation of f permitted by the computational graph can also be exploited in the calculation of the Hessian.

The ability to calculate an accurate directional second derivative at a time cost of a few function evaluations is extremely useful in, for example, an optimization using a truncated Newton method which contains a conjugate gradient algorithm [1].

Methods similar to that discussed here have been developed by M. Iri *et al* [6][8] and independently by B.W. Stephens and J.D. Pryce [11] and are used by PADRE2 and DAPRE

respectively. Our approach is intended to be used with operator overloading (for example in Ada), and allows the direct compilation of target functions coded in conventional programming notation, whereas PADRE2 and DAPRE are pre-compilers. Techniques for obtaining automatic Hessians by reverse accumulation using operator overloading have also been developed by A. Griewank *et al* [5], and are implemented in C++ as the package ADOL-C. The details of graph representation and manipulation appear very different to those described here, but the underlying algorithms are quite similar.

This report is organized as follows. In the next section, we describe a method, based on operator overloading, for the automatic run-time generation of the computational graph corresponding to a function evaluation. In section 3 we show how this construction can be used to reverse-accumulate the gradient vector. So far the material is quite standard. Section 4 investigates the opportunities for parallelism in the reverse accumulation algorithm. Section 5 presents the promised extension to second derivatives. In section 6 we give the time and space calculations. In section 7 we compare these theoretical bounds with those observed for a serial implementation of the algorithm in Ada. In the final section, we review the situation and draw conclusions concerning the present relationship between numerical analysis and computer science.

2. Automatic Construction of the Computational Graph. We assume that the calculation of the function $f(x_1, \ldots, x_n)$ can be broken into a sequence of simpler "atomic" operations thus:

for i from $n+1$ upto m do
    $x_i = f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$
enddo
$\{x_m = f(x_1, \ldots, x_n)\}$

where for each $i > n$, n_i is the arity of f_i and τ_i is a map from $\{1, \ldots, n_i\}$ into $\{1, \ldots, i-1\}$.

In practice, each f_i will generally be a unary or binary arithmetic operation (such as plus, times, sine, square root, etc.)

Where the evaluation of f is coded in conventional (in-line) form in a language which permits operator overloading, a list representing the sequence of atomic operations can be built as the function is evaluated. The operators from which the code for f is built are overloaded to append information about themselves and their arguments and results to the list dynamically as a side effect. Their arguments and return values (as well as all intermediate variables) are no longer REALs, but pointers to list items obtained from a heap.

Each list item contains a value identifying the operator which placed this item on the list (and returned a pointer to this item), a REAL value (which represents the result of performing the operation), a list of pointers to other items (which correspond to the arguments of the operator), and possibly some housekeeping information (such as pointers and counters.) A global environment pointer points to the (current) final list item.

Initially, items corresponding to the parameters $x_1 \ldots x_n$ are created and placed on the list. When the evaluation of f is complete, the final list item will contain the return value x_m for f.

Actually, the list of atomic operators built in this way is a linearization of the computational graph. The computational graph is the directed acyclic graph with nodes labelled from 1 to m and with an arc from node j to node i iff $j \in \mathcal{T}_i$ where $\mathcal{T}_i = Image\ \tau_i$. Note that we allow multiple arcs from j to i in the computational graph, corresponding to repeated arguments of f_i. Note also that (in accordance with convention) the arcs in the computational graph have the opposite sense to the pointers in the implementation.

By using a more highly concurrent data structure such as a *sibling trie* [10] rather than a list for housekeeping, overloading can be used to produce a representation of the computational graph which is more suited to parallel evaluation of the target function f.

Subgraphs of the computational graph (however the graph is represented) correspond to invocations of user-provided or library subprograms.

The code for f may also contain control statements. This means that the structure of the computational graph may depend on the parameter values. In particular, iterative loops are "unwrapped", which means that successive values of a single variable correspond to separate nodes in the computational graph. This contributes to the high run-time storage requirement mentioned above. However, since only the most recent value of a variable is ever required in subsequent construction of the graph, the working-set of the overloaded program for f requires space for only one item corresponding to each active stack or register variable. This is only a small multiple of the working-set storage required by the conventional program for f, and modern virtual-memory systems are capable of handling the "archived" items efficiently.

Since program variables are pointers, variable assignments are implemented as pointer assignments. No additional copying or duplication of the list items takes place (except in the case of pre-accumulation, which is discussed in section 4.)

Generally the structural dependence of the computational graph upon the parameter values causes no harm, but two points should be noted.

First, the use of control statements to define a piecewise-smooth function means that automatic differentiation will return the gradient of the smooth piece containing the current argument [4, §3.4]. As pointed out by Kedem [9], if the function is poorly programmed, this may not be the intended result in boundary cases.

Second, the evaluation of variables which are used only to determine the flow of control (and which are not functionally composed into f) means that not every node need be reachable from node m in the computational graph. This can also happen if the programmer is careless. There is an implication here for garbage-collecting the corresponding graph nodes. We return to this point in section 4.

If the function is programmed in a language which does not support operator overloading, the same effect can be achieved less elegantly by using a precompiler to convert operator invocations to function evaluations with correct dynamic side-effects (see for example [8].)

3. Reverse Accumulation of the Gradient. Define $\bar{x}_i = \partial x_m / \partial x_i$ for $1 \leq i \leq m$. (In evaluating $\bar{x}_i$ we de-link x_i from its dependence on the x_j with $j \in \mathcal{T}_i$ and regard x_i as

an independent variable, and $x_1 \ldots x_n$ as fixed.)

The vector $(\bar{x}_1, \ldots, \bar{x}_n)$ is just ∇f. Clearly $\bar{x}_m = 1$, and by the chain rule,

$$\bar{x}_j = \sum_{i>j} \bar{x}_i \frac{\partial f_i}{\partial x_j} \quad \text{where} \quad \frac{\partial f_i}{\partial x_j} = \sum_{\{p:\tau_i p = j\}} (D_p f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$$

Note that $\partial f_i / \partial x_j = 0$ unless $j \in \mathcal{T}_i$. Thus each arc leading out from j in the computational graph contributes exactly one term to $\bar{x}_j$. So we can compute ∇f as follows:

for i from 1 upto n do
    $\bar{x}_i = 0$
enddo
for i from $n+1$ upto m do
    $x_i = f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$
    $\bar{x}_i = 0$
enddo
$\{x_m = f(x_1, \ldots, x_n)\}$

$\bar{x}_m = 1$
for i from m downto $n+1$ do
    for p from 1 to n_i do
        $\bar{x}_{\tau_i p} = \bar{x}_{\tau_i p} + \bar{x}_i * (D_p f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$
    enddo
enddo
$\{(\bar{x}_1, \ldots, \bar{x}_n) = (\nabla f)(x_1, \ldots, x_n)\}$

To implement this, we alter the definition of the list items in the representation of the computational graph, so as to add a REAL field for the corresponding $\bar{x}_i$ and storage for a representation of $\nabla f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$. These partial derivatives of the atomic operations f_i may be represented explicitly as numerical values, or may already be implicit in the operation identifier and argument list (for example if $f_i(x, y) = x * y$ then $D_1 f_i(x, y) = y$.) The $\bar{x}_i$ are initialized to zero as items are generated and placed on the list.

We evaluate the gradient by calling an accumulate routine at the end of the code evaluating f. This accumulate routine begins at node m by initializing $\bar{x}_m$, and then propagates the appropriate summations back up all arcs leading into node m. It then moves to node $m-1$ and repeats the propagation process, and so on for each node, working backwards through the list. The accumulate routine can perform node-by-node garbage collection as it moves down through the list.

The time and space requirements of this algorithm are discussed quantitatively in section 6.

4. Parallel Evaluation and Pre-accumulation of the Gradient. Two implementation optimizations of gradient accumulation are possible. First, we can recover any parallelism in the calculation of f by allowing the accumulation algorithm to be active at

more than one node at a time. The condition for correct evaluation is that accumulation from node j must not begin until accumulation has completed at all nodes i for which there is an arc from j to i.

To ensure this we could, for example, associate a "use count" with each list item, indicating the number of arcs leading out of the corresponding node. This count (initialized to zero) is incremented as the list is built (as each new item is placed on the list, the use count of each item pointed at by the argument list is increased by one.) The count is decremented by the accumulate algorithm (as the accumulation algorithm propagates a summation term to a list item representing an argument, the use count of that argument item is decreased by one.) As soon as the use count of a list item reaches zero, the accumulation algorithm is free to begin propagation (potentially in parallel) from that node.

As mentioned earlier, care must be taken in garbage collection if this parallel approach is used. Nodes which are not reached by the propagation process (ie which already have a use count of zero when the accumulation begins) must be collected by a separate mechanism.

The second optimization allows us to recover some space early at the expense of a little processing time. At any time we can "pre-accumulate" an arbitrary sub-section of the graph, in practice usually a section corresponding to an invocation of a user or library sub-program.

This is done by removing from the list all items placed there by the sub-program, and replacing them by a single item representing the sub-program, and containing (pointers to) its parameters, return value, and gradient vector. This item is calculated by allowing the accumulation routine to run from the item corresponding to the result of the sub-program back as far as (local copies of) the parameters to the sub-program.

In effect, we are treating the sub-program as an atomic operation, and calculating its gradient using reverse accumulation "on the fly". The benefit of this is that the space occupied in the computational graph by the sub-program is drastically reduced. This saving may be very dramatic if many iterated values of intermediate variables are involved in the calculation.

The cost of using pre-accumulation in this way is that when the item corresponding to the sub-program is itself (later) accumulated in the calculation of ∇f, up to an additional n_i multiplies are required in the inner loop of the accumulation (for p from 1 upto n_i), where n_i is the number of parameters to the sub-program. If pre-accumulation were not used, these multiplies would be avoided since the correct setting of the $\bar{x}_i$ corresponding to the sub-program return value (rather than the setting to 1 by the pre-accumulation) would ensure that the correct multiples were calculated in the first place.

However, provided the number of parameters n_i is small compared to the number of items initially generated by the subprogram (ie the number of items removed from the graph when pre-accumulating the sub-program), this optimization may be highly desirable, at least for some sub-programs. In practice, a variant (longer) format of list element to that used for atomic operations would be used to contain the result of a pre-accumulated sub-program.

Where a sub-program returns more than one output parameter value, the corresponding pre-accumulation must be performed once for each such output parameter, and the result of each such pre-accumulation is represented by a corresponding item in the graph. The

overhead of making multiple traversals of overlapping portions of the graph can make pre-accumulation unattractive in such a case. In particular, if the number of output parameters is greater than the number of input parameters to the subprogram, it is more efficient to use forward accumulation. When overlapping pre-accumulations are used, items generated by the sub-program must not be removed from the graph until they have been released by all the relevant pre-accumulations.

Pre-accumulations can also be nested in an arbitrary fashion. It is perhaps worth remarking that the immediate pre-accumulation of every (atomic) operation produces an algorithm equivalent to forward accumulation.

5. Reverse Accumulation of the Hessian. To extend the reverse accumulation algorithm, we begin by noting that we can calculate $\mathbf{u}\cdot\nabla f$ for an arbitrary constant vector $\mathbf{u} = (u_1, \ldots, u_n)$ by re-writing the gradient accumulation algorithm in the (computationally inefficient) forward form as follows:

for i from 1 upto n do
    $w_i = u_i$
enddo

for i from $n+1$ upto m do
    $x_i = f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$
    $w_i = g_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i}, w_{\tau_i 1}, \ldots, w_{\tau_i n_i})$
enddo
$\{x_m = f(x_1, \ldots, x_n),\ w_m = \mathbf{u}\cdot\nabla f(x_1, \ldots, x_n)\}$

where

$$g_i(\mathbf{x}, \mathbf{w}) = \sum_{p=1}^{n_i} w_{\tau_i p}(D_p f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$$

Induction shows that at each stage,

$$w_i = u_1 \frac{\partial x_i}{\partial x_1} + \cdots + u_n \frac{\partial x_i}{\partial x_n}.$$

We regard this as an algorithm for calculating $w_m = \mathbf{u}\cdot\nabla f(x_1, \ldots, x_n)$, and apply reverse accumulation to this to evaluate the directional second derivative $\nabla(\mathbf{u}\cdot\nabla f) = \mathbf{u}\cdot(\nabla \wedge \nabla f) = (Hf)\mathbf{u}$. This requires us to evaluate the quantities $\partial w_m/\partial x_j$ and $\partial w_m/\partial w_j$.

First we show by induction that $\partial w_m/\partial w_j = \bar{x}_j$ for $1 \le j \le m$. Clearly $\partial w_m/\partial w_m = 1 = \bar{x}_m$, and if $\partial w_m/\partial w_i = \bar{x}_i$ for all $i > j$ then

$$\frac{\partial w_m}{\partial w_j} = \sum_{i>j} \frac{\partial w_m}{\partial w_i}\frac{\partial g_i}{\partial w_j} = \sum_{i>j} \bar{x}_i \frac{\partial f_i}{\partial x_j} = \bar{x}_j$$

since

$$\frac{\partial g_i}{\partial w_j} = \sum_{\{p:\tau_i p = j\}} (D_p f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i}) = \frac{\partial f_i}{\partial x_j}.$$

Next we define $\bar{w}_j = \partial w_m / \partial x_j$ for $1 \le j \le m$. Since none of the f_i or g_i has x_m as an argument, we have $\bar{w}_m = 0$, and the chain rule gives

$$\bar{w}_j = \sum_{i>j} \left\{ \bar{x}_i \frac{\partial g_i}{\partial x_j} + \bar{w}_i \frac{\partial f_i}{\partial x_j} \right\}$$

but

$$\frac{\partial g_i}{\partial x_j} = \sum_{k<i} w_k \frac{\partial^2 f_i}{\partial x_j \partial x_k}$$

$$\text{where} \quad \frac{\partial^2 f_i}{\partial x_j \partial x_k} = \sum_{\{p:\tau_i p = k\}} \sum_{\{q:\tau_i q = j\}} (D_p D_q f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$$

since $\partial^2 f_i / \partial x_j \partial x_k = 0$ unless $j, k \in \mathcal{T}_i$. Thus

$$\bar{w}_j = \sum_{i>j} \left\{ \bar{w}_i \frac{\partial f_i}{\partial x_j} + \bar{x}_i \sum_{k<i} w_k \frac{\partial^2 f_i}{\partial x_j \partial x_k} \right\}.$$

Using these formulae, we see that directional second derivative $(Hf)\mathbf{u}$ can be calculated using the following algorithm.

for i from 1 upto n do
    $\bar{x}_i = 0$
    $w_i = u_i$
    $\bar{w}_i = 0$
enddo

for i from $n+1$ upto m do
    $x_i = f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$
    $\bar{x}_i = 0$
    $w_i = 0$
    for p from 1 to n_i do
        $w_i = w_i + w_{\tau_i p} * (D_p f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$
    enddo
    $\bar{w}_i = 0$
enddo
$\{x_m = f(x_1, \ldots, x_n),\ w_m = (\mathbf{u} \cdot \nabla f)(x_1, \ldots, x_n)\}$

$\bar{x}_m = 1$
for i from m downto $n+1$ do
    for p from 1 to n_i do
        $\bar{x}_{\tau_i p} = \bar{x}_{\tau_i p} + \bar{x}_i * (D_p f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$
        $\bar{w}_{\tau_i p} = \bar{w}_{\tau_i p} + \bar{w}_i * (D_p f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$
        for q from 1 to n_i do
            $\bar{w}_{\tau_i p} = \bar{w}_{\tau_i p} + \bar{x}_i * w_{\tau_i q} * (D_p D_q f_i)(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$

enddo
enddo
enddo
$\{(\bar{x}_1, \ldots, \bar{x}_n) = (\nabla f)(x_1, \ldots, x_n),\ (\bar{w}_1 \ldots, \bar{w}_n) = \nabla(\mathbf{u}\cdot\nabla f)(x_1, \ldots, x_n)\}$

An alternative way of deriving this algorithm for calculating the directional second derivative $(Hf)\mathbf{u}$ is to "unfold" the reverse accumulation of the computational graph of x_m so as to obtain an explicit computational graph for

$$\bar{x} = u_1\bar{x}_1 + \cdots + u_n\bar{x}_n$$

and then apply reverse accumulation to this (augmented) graph in order to obtain $\nabla\bar{x} = (Hf)\mathbf{u}$. (This is the derivation given in [7] for their Algorithm C.)

However, this alternative derivation leads to just the same algorithm, since

$$\frac{\partial\bar{x}}{\partial\bar{x}_i} = w_i \quad \text{and} \quad \frac{\partial\bar{x}}{\partial x_i} = \bar{w}_i$$

To see the first equality, note that

$$w_i = (u_1\frac{\partial x_i}{\partial x_1} + \cdots + u_n\frac{\partial x_i}{\partial x_n})$$

so it suffices to show for $1 \le j \le n$ that

$$\frac{\partial\bar{x}_j}{\partial\bar{x}_i} = \frac{\partial x_i}{\partial x_j}$$

which is (almost) obvious. For the second equality, note that $w_m = \bar{x}$ identically as functions of $(x_1, \ldots, x_n)$ so

$$\frac{\partial\bar{x}}{\partial x_i} = \frac{\partial w_m}{\partial x_i} = \bar{w}_i \quad \text{by definition.}$$

In fact, not only the values but the calculations performed to obtain them are the same. For example, going backwards through the (reversed) part of the augmented graph for the $\bar{x}_i$ calculating $\partial\bar{x}/\partial\bar{x}_i$ is the same as going forwards through the original graph for the x_i calculating w_i. Finally, the two passes through the augmented graph can be re-combined so as to reproduce exactly the algorithm already given above.

To implement this algorithm for the directional second derivative we add further fields to the list items: two REALs to contain the values for $\bar{w}_i$ and w_i, and space to represent the second derivative values for the atomic operation f_i. For the majority of binary and unary arithmetic and special functions, these values are trivially computed from the operation result and gradient. The inner loops can then be completely unfolded, and the multiplication by $\bar{x}_i$ distributed over the $q-$summation.

We also modify the code for the overloaded operators and for the accumulate routine so that they propagate the appropriate summations.

If $\mathbf{u} = \mathbf{e}_i$, the i-th unit vector, then the algorithm returns the i-th row of the Hessian. The entire Hessian can be calculated by using n passes of the algorithm, one for each row. This can be done in parallel on n processors, or serially by n passes on a single processor.

In the case where a single processor is used, some time savings can be easily made. The computational graph constructed on the first pass can be re-used for subsequent passes, as can the calculated values of the $\bar{x}_i$, and useful multiples of the atomic operation derivative values.

In the parallel case, these values can be pre-calculated and broadcast, along with the graph structure, to each of the processors. Thereafter the computation of the Hessian proceeds in parallel with no interlocking. Alternatively, the n rows of the Hessian could instead be calculated completely independently. In this case absolutely no interlocking of the processors is required. However the communication cost is likely to be (considerably) lower than the cost of re-computing the function and gradient.

As with the gradient, all parallelism in the calculation of f permitted by the computational graph can also be exploited in the calculation of the Hessian. If we are re-using the computational graph, then additional information must be placed in the nodes as the graph is built, recording the trace through the graph of each individual processor.

6. Performance of the Reverse Accumulation Algorithm. In this section we analyze the incremental computational cost of calculating a single directional second derivative, as a multiple of the cost of calculating the function and gradient.

We consider the total amount of time which the algorithm spends processing the list element corresponding to f_i, and show that for each elementary atomic operation the w/x-ratio is at most two, ie that the time spent calculating w_i and $\bar{w}_i$ is at most twice the time spent calculating x_i and $\bar{x}_i$.

We assume that each atomic operation is either a univariate special function, or a bivariate arithmetic operation. The arithmetic operations are plus, minus and times. We assume that x/y is overloaded as $x * r(y)$ where $r(y) = 1/y$. (It turns out that this is not particularly inefficient.)

We also make the assumptions that multiplies t take a lot longer than adds a, that special functions s and their first derivatives s' take longer than multiplies, and that divides d take at least one and a half times as long as multiplies.

We assume that the second derivatives s'' of special functions can be evaluated in less time than it takes to evaluate the special function and its first derivative together.

$$a << t \quad 3t < 2d \quad t < s \quad t < s' \quad s'' < s + s'$$

We ignore the cost of cache misses and of pointer manipulation, including garbage collection, and interprocessor communication costs. (In section 7 we shall argue that this assumption need not be as simplistic as it may seem.)

Let us now consider each atomic operation in turn. For plus and minus, the first partial derivatives are unity, and the second partial derivatives are zero. These operations therefore require no multiplies. One add is required to calculate x_i, two to accumulate $\bar{x}_i$, one to calculate w_i and two to accumulate $\bar{w}_i$. The w/x-ratio for plus and for minus is therefore one.

For times, the first partial derivatives are available as parameters and the two non-zero second partial derivatives are unity. Times therefore requires one multiply to calculate x_i, $2t+2a$ to accumulate $\bar{x}_i$, $2t+a$ to calculate w_i and $4t+4a$ to accumulate $\bar{w}_i$. The w/x-ratio for times is therefore

$$\frac{6t+5a}{3t+2a} < 2 + \frac{a}{3t} \approx 2 \quad \text{since } a << t.$$

Reciprocal requires a divide to calculate the operation value, and an additional multiply to calculate each of the two derivatives. The remaining operations come from propagating the accumulations. Reciprocal therefore requires d to calculate x_i, $2t+a$ to accumulate $\bar{x}_i$, t to calculate w_i and $4t+3a$ to accumulate $\bar{w}_i$. The w/x-ratio for reciprocal is therefore

$$\frac{5t+3a}{d+2t+a} < 2.$$

However, if more than one second directional derivative is being calculated, a multiply can be saved from the cost of accumulating $\bar{w}_i$ subsequently by storing $\bar{x}_i * D^2 r(x_j)$ instead of $D^2 r(x_j)$.

Divide is a reciprocal and a times, that is, $d+t$ to calculate x_i, $4t+3a$ to accumulate $\bar{x}_i$, $3t+a$ to calculate w_i and $8t+7a$ to accumulate $\bar{w}_i$. The w/x-ratio for divide is therefore

$$\frac{11t+8a}{d+5t+3a} < 2.$$

Again, if more than one second directional derivative is being sought, a multiply can be salvaged from the numerator.

We note in passing that if divide is calculated and accumulated directly as a single operation (instead of being overloaded as a reciprocal and a times) then we save $t+a$ from the denominator and $2t$ from the numerator, although we still require the storage cost for five REAL values.

Special functions are similar to reciprocals. A function which requires s to calculate x_i and $s'+t+a$ to accumulate $\bar{x}_i$ requires t to calculate w_i and $s''+3t+2a$ to accumulate $\bar{w}_i$. The w/x-ratio for this operation is therefore

$$\frac{s''+4t+2a}{s+s'+t+a} < 2 \qquad \text{since } s'' < s+s', t < s, t < s'.$$

Under our assumptions, therefore, and assuming no pre-accumulation, the incremental cost of evaluating $(Hf)\mathbf{u}$ is less than twice the cost of evaluating $\{f, \nabla f\}$ by reverse accumulation. In particular, all n rows of the Hessian can be evaluated by evaluating the n directional derivatives $(Hf)\mathbf{e}_i$.

We now give an example to show that this approach is within a constant factor of being optimal. Let f be the function of n variables defined by

$$f(\mathbf{x}) = (\mathbf{x} \cdot \mathbf{x})^2$$

$$\nabla f = 4(\mathbf{x} \cdot \mathbf{x})\mathbf{x}$$

$$Hf = 4(\mathbf{x} \cdot \mathbf{x})I + 8(\mathbf{x} \wedge \mathbf{x}).$$

The computational cost of f is $(n+1)t + (n-1)a$, and ∇f by reverse accumulation requires an additional $2(n+1)t + 4na$. The additional computational cost of evaluating Hf (by whatever means) is at least $\frac{1}{2}n(n+1)t$.

As far as storage space is concerned, for the gradient accumulation each list item requires at least two REALs and one or two pointers. (If x_i and $\bar{x}_i$ share storage [4, §4.1], then space is required to store derivative values calculated during the forward sweep.) Efficient Hessian accumulation requires at most another four REALs to store the derivatives of the unary operators, and the new accumulated values $w_i, \bar{w}_i$. (In fact, as mentioned earlier, we would store $\bar{x}_i * D^2 f_i(x_j)$ rather than the unscaled second derivative.)

The storage requirement for the reverse accumulation of the Hessian together with the function value and gradient is therefore less than three times that required for the evaluation of the gradient alone.

Again, it should be emphasized that this storage requirement is proportional, not to the storage requirement for the conventional program for evaluating f alone, but rather to the (potentially vast) number of arithmetic operations required by such an evaluation. Nevertheless, as remarked in section 2, the working set (fast RAM) storage requirement for the reverse accumulation algorithm is proportional to the working set storage requirement for the conventional program. (The crucial point is that the set of active items at each point on the reverse pass is the same as it was at the corresponding point on the forward pass.) The bulk of the graph can therefore be stored in (slow, sequential access) archive store.

Pre-accumulation of a sub-graph can still be used to salvage storage space as the algorithm proceeds, but the time penalty is now considerably more severe because the second derivatives must also be pre-accumulated, and this requires n_i passes forward and backward through the portion of the graph corresponding to the sub-program. There is also a storage penalty associated with pre-accumulation of the Hessian, since all the (non-zero) second derivatives of the sub-program must be stored.

In the full case (where all n_i^2 second derivatives are non-zero) pre-accumulation will produce a worthwhile storage saving only if the number of items m_i corresponding to the sub-program is large relative to n_i^2.

However, if we do have m_i large relative to n_i^2, then there may actually be a positive time benefit to using pre-accumulation of the Hessian, provided that the number k of second directional derivatives required is large compared with n_i.

To see this, argue as follows. Let W_i'' be the computational cost of one complete pass (forward and backward) through the sub-program graph accumulating a row of second derivatives. The cost of pre-accumulating all the second derivatives for the sub-program is thus $n_i W_i''$. Subsequent accumulation of a pre-accumulated sub-program with n_i parameters will now require up to an additional n_i^2 multiplies in the innermost loop of the reverse accumulation. Consequently the cost of subsequently accumulating k directional second derivatives if pre-accumulation has been used is $n_i W_i'' + k n_i^2 t$ (where t is the cost of a multiply), whereas the cost without pre-accumulation is $k W_i''$. The use of pre-accumulation will therefore result in a time saving provided that W_i'' is large relative to $n_i^2 t$ and that k is large relative to n_i (for example it suffices that $W_i'' > 2n_i^2 t$ and $k > 2n_i$).

The foregoing analysis assumes that the sub-program has a full matrix of second partial derivatives. If the Hessian of the sub-program is sparse, then pre-accumulation may give

a performance benefit even under weaker conditions.

An alternative form of pre-accumulation can be exploited to save storage space (but not processing time) in the case where $k = 1$, provided the direction $\mathbf{u}$ is known in advance of the pre-accumulation.

In this case, we can unwrap the innermost loop in the algorithm, as described in section 5. If the multiplication by $\bar{x}_i$ is distributed over the q-summation, the q-summation can then itself be performed when the sub-graph corresponding to the subprogram is initially built, since all the w_i are known at this time.

When the pre-accumulation is performed we need only calculate and store the appropriate linear combination of rows of the sub-program Hessian. We end up storing n_i components, which represents a space saving provided m_i is large relative to n_i (rather than n_i^2 as before.)

For the subsequent reverse sweep of accumulation the time overhead of accumulating a directional second derivative is $n_i t$. This is tolerable provided W_i'' is large relative to $n_i t$, but cannot represent a time saving.

In view of the fact that archive store is relatively plentiful in modern computer systems, it would appear that the principal advantage of pre-accumulating Hessians occurs where this enables a time saving to be made, as described above.

7. Implementation in Ada. The serial form of the extended method of automatic differentiation has been implemented in Ada by the author. The total amount of Ada code involved is of the order of 600 lines. The data types do not depend upon the number of independent variables or computational steps. The implementation is similar to that for the sparse forward accumulation (also implemented in Ada at the Hatfield Numerical Optimization Centre [1][3]), but the details are simpler.

As an example, we give the measured cpu times in milliseconds for the Helmholtz energy function

$$f(x_1, \ldots, x_n) = RT \sum_{i=1}^{n} x_i \log \frac{x_i}{1 - b^T x} - \frac{x^T A x}{\sqrt{8 b^T x}} \log \frac{1 + (1 + \sqrt{2}) b^T x}{1 + (1 - \sqrt{2}) b^T x}$$

with $n = 100$. The computation was carried out on a Sun 3/160 using VADS 5.5 Ada. The resulting graph has $10,206$ items corresponding to independent variables and constants, and an additional $20,818$ items corresponding to elementary operations.

The following table shows the cpu times when software library functions are used to perform floating point operations. The first column gives the actual cpu time in milliseconds, the second column gives the same figures normalized relative to the first row.

3186	100	calculate f using conventional real arithmetic
4148	130	build graph of f using overloaded operations
198	6	dismantle graph
3614	113	re-use graph to calculate f
9360	294	accumulate gradient ∇f
20138	632	accumulate directional second derivative $(Hf)\mathbf{u}$

As an alternative to dismantling and re-building the computational graph when the parameter values are changed, we can re-calculate new x_i values using the old graph, provided that the structure of the graph is not changed by the change in parameter values (see section 2). This is the figure given as "re-use graph" above.

By entering into the graph items corresponding to relational operators we can check that the shape has not in fact changed, and signal an exception if it has.

As these figures illustrate, when software floating point operations are used, the overheads due to overloading, graph traversal, indirect address (pointer) manipulation and floating-point assignment of intermediate results are not high, and the ratios between the various cpu times give good agreement with the values predicted by the theoretical analysis in section 6.

When hardware support for floating point operations is used, a different picture emerges. Here are the corresponding cpu time figures when use of the MC68881 floating point co-processor is enabled on the Sun.

258	100	calculate f using conventional real arithmetic
1298	503	build graph of f using overloaded operations
194	75	dismantle graph
706	274	re-use graph to calculate f
974	378	accumulate gradient ∇f
1892	733	accumulate directional second derivative $(Hf)\mathbf{u}$

The ratios between the measured cpu times are dependent upon the target hardware, operating system, compiler and mathematical library. For comparison, here are the corresponding cpu time measurements for native ADA on a VAX/VMS (with built in hardware support for floating point).

41	100	calculate f using conventional real arithmetic
403	983	build graph of f using overloaded operations
85	287	dismantle graph
331	807	re-use graph to calculate f
303	739	accumulate gradient ∇f
663	1617	accumulate directional second derivative $(Hf)\mathbf{u}$

These figures show a dramatic ratio of one to ten between the times for the conventional real arithmetic calculation of f and that using overloaded operators. The time spent performing floating point arithmetic (as distinct from assignments) is a small fraction of the total cpu time, and the cpu time for performing an accumulation is almost directly proportional to the number of times the graph is traversed. The given ratios are relatively insensitive to the value of n over the range 20 to 150.

In all the figures quoted above, the directional second derivative was calculated by re-traversing the graph (forward and backward) following the accumulation of the gradient. In fact this calculation can be done for a single second directional derivative without requiring these additional sweeps. Nevertheless, the graph manipulation still represents a significant overhead, and we are investigating possible hard and soft support for automatic differentiation which would reduce this overhead in the same way as specialized support for floating point arithmetic reduces the cpu time required for these operations.

8. Discussion and Conclusion. We have described a way of extending the reverse accumulation method for automatic differentiation so as to provide efficient calculation of the Hessian. This method has been implemented in Ada using operator overloading, and the observed performance agrees well with the theoretical prediction, although graph manipulation overheads are high when floating point hardware is used. Nevertheless, the method is sufficiently fast and accurate to be usefully applied to real problems. Only minimal alterations to the target function code and optimization code are required to allow the automatic Hessians to be incorporated. In particular, no restrictions are placed upon the use of Ada data or control structures in target function code.

The Ada language is a natural implementation choice for a system such as that described here, in view of the excellent support which it offers for floating point arithmetic. However, two remarks should be made.

First, Ada does not allow the assignment operator to be overloaded. This requires the programmer to insert explicit calls to type conversion routines, after reads and before prints, and possibly elsewhere. Nor does Ada provide a guarantee to call a (user defined) "deconstructor" routine when a variable instance is de-allocated. This limits the ability of the automatic differentiation package to detect certain types of user error, and also limits the opportunity to pass hint information about which variables are (about to become) active to the memory management system. The use of a language such as C++ allows reverse accumulation to be implemented much more transparently.

Second, the implementation of parallel versions of reverse accumulation techniques will clearly assume increasing importance. The extent to which the multitasking approach used by Ada is adapted to the parallel processing requirements of function evaluation with reverse accumulation is unclear. Implementing graph nodes as objects in a language such as C++ may lead to significantly improved performance.

In contrast with forward accumulation, the method of reverse accumulation associates the gradient and Hessian components with the target function parameter list rather than directly with the returned result. This can have unfortunate consequences.

For example, consider a finite element approach to solving a differential equation. Typically this involves calculating the values and gradients of a large number of component functions, each of which involves only a small number of the independent variables. Moreover, the number of elementary functions involved in calculating a single component is generally very small compared with the total number of independent variables. In such a case, unless prior use is made of the (explicitly known) sparse form of the gradient and Hessian, a great deal of time is wasted initializing and subsequently examining components of items corresponding to independent variable parameters which are not (in fact) involved in the particular component computation.

The data structures and accumulation techniques needed in order to present gradient and Hessian components to the user in (explicitly) sparse form are similar to those appropriate for sparse forward Hessian accumulation, and are almost identical with those required by the pre-accumulation techniques discussed above. Much work remains to be done in this area.

We conclude with two remarks of a more general nature. First, the method of automatic differentiation described here underlines the benefits to numerical analysts of using what were until comparatively recently regarded as computer-science concepts: operator overloading, dynamic data management (with garbage collection), and pointer-oriented (rather than array-oriented) concurrent data structures.

Second, there is a clear imperative to computer scientists. Even quite small computer systems should be designed in such a way that they are capable of efficiently supporting structures which embody these concepts. At present this is not the case. In particular, high-performance support should be provided for the pre-fetching of a stream of operands to cache from a potentially vast backing store (say an optical disk) by moving down dynamic chains of pointers according to fixed access rules. This would allow the cache-miss rate and pointer manipulation overhead for automatic differentiation by reverse accumulation to be reduced to practically nothing.

References.

[1] M.C. Bartholemew-Biggs *et al*, 1989, Three Papers on Automatic Differentiation, Technical Report 223, Numerical Optimization Centre, Hatfield Polytechnic

[2] L.C.W. Dixon, 1987, Automatic Differentiation and Parallel Processing in Optimization, Technical Report 180, Numerical Optimization Centre, Hatfield Polytechnic

[3] L.C.W. Dixon *et al*, 1990, Automatic Differentiation of Large Sparse Systems, Journal of Economic Dynamics and Control (North Holland) **14** 299–311

[4] A. Griewank, 1989, On Automatic Differentiation, *in* Mathematical Programming 88, Kluwer Academic Publishers, Japan

[5] A. Griewank *et al*, 1990, User's Guide for ADOL-C: Version 1.0, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, USA.

[6] M. Iri, 1984, Simultaneous Computation of Functions, Partial Derivatives and Estimates of Rounding Errors - Complexity and Practicality, Japan Journal of Applied Mathematics **1**(2) 223–252

[7] M. Iri and K. Kubota, 1987, Methods of Fast Automatic Differentiation and Applications, Technical Report METR 87-6, Department of Mathematical Engineering and Instrumentation Physics, University of Tokyo

[8] M. Iri *et al*, 1988, Automatic Computation of Partial Derivatives and Rounding Error Estimates with Application to Large-scale Systems of Non-linear Equations, Journal of Computational and Applied Mathematics (North Holland) **24** 365-392

[9] G. Kedem, 1980, Automatic Differentiation of Computer Programs, ACM Transactions on Mathematical Software **6**(2) 150–165

[10] G.D Parker, 1989, A Concurrent Search Structure, Journal of Parallel and Distributed Computing **7** 256–278

[11] B.R. Stephens and J.D. Pryce, 1990, The DAPRE/UNIX Preprocessor Users' Guide Version 1.2, Applied and Computational Mathematics Group, Royal Military College of Science, Shrivenham, England