REVERSE ACCUMULATION OF FUNCTIONS CONTAINING GRADIENTS

Bruce Christianson

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

Numerical Optimisation Centre Technical Report 278, May 1993
presented Theory Institute Argonne National Laboratory Illinois

Abstract

We extend the technique of reverse accumulation so as to allow efficient extraction of gradients of scalar valued functions which are themselves constructed by composing operations which include taking derivatives of subsidiary functions. The technique described here relies upon augmenting the computational graph, and performs well when the highest order of derivative information required is at most fourth or fifth order. When higher order is required, an approach based upon interpolation of taylor series is likely to give better performance, and as a first step in this direction we introduce a transformation mapping reverse passes through an augmented graph onto taylor valued accumulations through a forward pass. The ideas are illustrated by application to a parameter free differentiable penalty function for constrained optimization problems.

1. Introduction. It is well known (see for example [2]) that reverse accumulation can be used to extract all components of the gradient vector ∇f of any scalar valued function f for about 3 times the floating point computational cost of a single evaluation of f, where the constant 3 is independent both of the form of f and of the number of parameters (independent variables). Similarly [op cit] if $\mathbf{b}$ is any constant column vector then reverse accumulation can evaluate the entire vector $(Hf)\mathbf{b}$ (ie an arbitrary linear combination of rows of the Hessian of f) for about 6 times the floating point computational cost of a single evaluation of f. This technique is further extended in [1].

Here we develop these techniques to derive a simple and elegant way of extracting gradients (and higher derivatives) of functions such as $y = \phi(\mathbf{u}, \nabla f(\mathbf{u}))$ which are constructed by composing operations which include taking gradients of subfunctions. We show how to obtain such information to the same level of accuracy as the function value for f, and at a small constant multiple of the computational cost.

The ability to form gradients of this kind has many applications in constrained optimization. As an example, we consider the following problem:

$$\text{Minimize } y = f(\mathbf{u}) \text{ subject to } \mathbf{k}(\mathbf{u}) = 0 \text{ where } dim(\mathbf{k}) < dim(\mathbf{u})$$

It is shown in [6] that this is equivalent to the augmented problem:

$$\text{Minimize } z = \phi(\mathbf{u}) = f(\mathbf{u} - \mathbf{n}(\mathbf{u})) - \lambda_*(\mathbf{u}) \cdot \mathbf{k}(\mathbf{u}) + \frac{1}{2}\mathbf{n}(\mathbf{u}) \cdot \mathbf{n}(\mathbf{u})$$

where

$$\mathbf{n} = \hat{N}\mathbf{k}, \hat{N} = N'(NN')^{-1}, N = J(\mathbf{k}) \text{ so that } N_{ij} = \frac{\partial k_i}{\partial u_j} \text{ and } \lambda_* = \mathbf{g}\hat{N}, \mathbf{g} = \nabla f$$

In contrast with traditional penalty functions, ϕ is differentiable and parameter free. In the past, the difficulty of obtaining accurate first and second derivatives for functions such as ϕ at reasonable cost has proved a disincentive to using such penalty formulations.

2. Unfolding Reverse Accumulation. We assume for ease of exposition that reverse accumulation is implemented in the style of [2], by operator overloading but without overloading assignment. Floating point program variables are redeclared as of type `vary` where

```
type vary = record (ref_node :  pointer to node)
type node = record (opcode :  integer; arg1 :  vary; arg2 :  vary;
        forward_value :  real; adjoint_value :  real)
```

Evaluation of the function f produces as a side effect a computational graph for f. The reverse accumulation sweep to evaluate ∇f begins by placing the value 1.0 in the adjoint value field of the end node. The reverse sweep then moves backwards through the graph incrementing the adjoint values by appropriate multiples of the operation derivatives, as required by the chain rule. For example, the adjoint accumulation step corresponding to the forward step $v = \sin u$ is $\bar{u} = \bar{u} + \bar{v}\cos u$, and the adjoint accumulation steps corresponding to the forward step $w = u * v$ are $\bar{u} = \bar{u} + \bar{w} * v$; $\bar{v} = \bar{v} + \bar{w} * u$. At the

end of the reverse sweep, the adjoint value in each node is the partial derivative of the function value in the end node with respect to the forward value in the given node. In particular, adjoint values in the nodes pointed at by the independent variables correspond to the components of ∇f.

Suppose now that we have some function $\phi(\nabla f(u))$ and we wish to evaluate $\nabla\phi$. This can be done by the very same code that we have just described, by making one crucial change. We redeclare the adjoint value field as

```
adjoint_value :  vary
```

The reverse accumulation step $a = a + b * c$ where a and b are now of type `vary` is implemented by overloading in such a way that $null + a$ returns a and $null * b$ returns a null pointer.

The effect of this redeclaration is that the reverse sweep now creates an additional segment of the computational graph, recording the calculation of the various operation derivatives and adjoint values. At the end of the reverse sweep,

`x.ref_node.adjoint_value.ref_node.forward_value` contains the floating point adjoint value (derivative component) corresponding to the independent variable x.

Part or all of the computational graph can be swept in this way, and similarly adjoint values, once calculated, can be used in subsequent constructions which can then themselves be reverse-swept. In this case, it is important to reset (to null pointers) the adjoint fields in the part of the graph to be re-swept before re-sweeping. This can be done as a side effect in the course of the previous reverse sweep. Note that this reset operation does not affect the node previously pointed at by the re-initialized field. The approach described here has been implemented by Kubota [9].

Some pseudocode showing how to extract a directional second derivative (such as a row of the Hessian) is now given to illustrate these ideas. Suppose that we wish to calculate, for a fixed direction vector $\mathbf{b}$, the components of $(Hf)\mathbf{b} = \nabla(\mathbf{b} \cdot \nabla f)$.

```
declare sp, ep, y, x[1..n], ybar :  pointer to node;
{assume that we have set initial values for x[i].fwd}
sp=mark_graph
y=f(x)
ep=mark_graph
ybar=vary(1.0)
y.adj=ybar
reverse_sweep(ep,sp)
{at this point x[i].adj.fwd=(∇f)_i}
ep=mark_graph
do for for each i from 1 to n
  x[i].adj.adj=vary(b[i])
  x[i].adj=null
end do
reverse_sweep(ep,sp)
{at this point ybar.adj.fwd=b · ∇f and x[i].adj.fwd=((Hf)b)_i }
```

In this pseudocode, `mark_graph` is a function which returns a pointer to the (current) end of graph position. It is assumed that this pointer points at a special graph node created to record the mark. The `reverse_sweep` routine is assumed to reset `node.adjoint_value` to `null` upon leaving each graph node which it visits. (Nodes corresponding to independent variables are incremented but not visited, and so must be reset explicitly.) The coercion function `vary` takes a floating point value and returns a pointer to a graph node which contains that floating point value in the forward field. Effectively this creates an additional independent variable.

The independent variable `ybar` is introduced purely in order to illustrate how a reference to an "old" adjoint value can be retained in spite of the re-initializing by `reverse_sweep`. In implementation, it may be desirable for `reverse_sweep` to accumulate adjoint values for independent variables, not in the variables themselves, but in a specially constructed sparse vector (of pointers to nodes) associated with the marked node pointed at by `sp`.

Since the Hessian is just the Jacobian of the gradient, the problem of calculating an entire Hessian is now reduced to that of evaluating a (sparse) Jacobian, as considered in [4].

3. Reverse^2=Taylor. We turn now to a consideration of strategies for evaluating the ideal penalty function introduced in §1 in such a way as to permit automatic extraction of the gradient and higher derivatives.

The Jacobian of a vector valued function whose component calculations share the values of intermediate variables can be extracted by following the strategy of [4] or [8]. This strategy, which relies upon manipulating sparse vectors of adjoint quantities, must be re-defined to manipulate sparse vectors of pointers to graph nodes containing the adjoint values. Once this has been done, the calculation of the Jacobian, including a record of the order of node elimination and of the relevent multiplications used to do this, will be automatically recorded in the computational graph, and thus be available for further automatic manipulation. The sparse increment operation, used to add a multiple of one sparse vector to another, will create a new node only for vector components which are present in both vectors.

However, for the particular form of the penalty funtion ϕ considered here, we do not need to develop the full Jacobian in this way. All we require are the vectors $\mathbf{n}$ and λ_*. It is well known that we can evaluate vectors of the form $\mathbf{y} = N'\mathbf{x}$ quite simply by a single reverse sweep of the graph which records the construction of $\mathbf{w} = \mathbf{k}(\mathbf{u})$ from $\mathbf{u}$. This reverse sweep starts by setting $\bar{\mathbf{w}} = \mathbf{x}$ and ends with $\bar{\mathbf{u}} = \mathbf{y}$.

The fact that reverse accumulation can also be used to evaluate vectors of the form $\mathbf{z} = N\mathbf{y}$ does not appear to be nearly so widely known. Here is some pseudocode illustrating how to do this.

```
declare sp, ep, u[1..n], y[1..n], w[1..m], x[1..m], :  pointer to node;
{assume that we have set initial values for u[i].fwd and x[i].fwd}
sp=mark_graph
w=k(u)
{at this point w[i].fwd=k_i(u)}
ep=mark_graph
```

```
do for each i from 1 to m
  x[i]=vary(x_i)
  w[i].adj=x[i]
end do
reverse_sweep(ep,sp)
{at this point u[i].adj.fwd=(N'x)_i}
sp=ep
ep=mark_graph
do for for each i from 1 to n
  y[i]=u[i].adj
  y[i].adj=y[i]
  u[i].adj=null
end do
reverse_sweep(ep,sp)
{at this point x[i].adj.fwd=(NN'x)_i }
```

What is happening here is this. We first build a graph $G(u \to w)$ which records the calculation of $\mathbf{w} = \mathbf{k}(\mathbf{u})$. Next we set $\bar{\mathbf{w}} = \mathbf{x}$ and reverse through $G(u \to w)$ to calculate $\mathbf{y} = \bar{\mathbf{u}} = N'\mathbf{x}$. As a side effect, this reverse sweep builds a new segment of the graph which we call $G(x \to y)$. Now suppose that we set $\bar{\mathbf{y}} = \mathbf{c}$ for a constant vector $\mathbf{c}$ and reverse through $G(x \to y)$ to obtain $\mathbf{z} = \bar{\mathbf{x}}$. This reverse pass builds $G(c \to z)$. We assert that $\mathbf{z} = N\mathbf{c}$, since

$$z_i = \sum_j c_j \frac{\partial y_j}{\partial x_i} = \sum_j c_j \frac{\partial \bar{u}_j}{\partial \bar{w}_i} = \sum_j c_j \frac{\partial w_i}{\partial u_j} = \sum_j c_j N_{ij} = (N\mathbf{c})_i$$

If the constant vector $\mathbf{c}$ is merely chosen to have numerical values which equal those of the vector $\mathbf{y} = N'\mathbf{x}$, then we would evaluate $\mathbf{z} = N\mathbf{c} = NN'\mathbf{x}$, as required, but the graph $G(c \to z)$ would not fully reflect the functional dependency of z on u and x, and so could not itself be subjected to further automatic differentiation. This functional dependence is however reflected in the pseudocode above by setting $\bar{\mathbf{y}} = \mathbf{y}$, ie setting `y[i]` to be its own adjoint. The reverse accumulation step $G(y \to z)$ accumulates the correct multiples of the derivatives, and the correct functional dependencies are recorded for subsequent differentiation passes.

Alternatively, we can also evaluate vectors of the form $\mathbf{z} = N\mathbf{y}$ with a single forward pass, using (in effect) a linear taylor series. We allow graph nodes to be chained together to form the terms of a taylor series. We form the taylor series v_i corresponding to the independent variables u_i, and set (initially) $\mathbf{v} = \mathbf{u} + \mathbf{y} \cdot t$ where t is the (nominal) taylor variable. We also declare overloaded operations to act on these taylor types. By making a forward pass through the existing graph for $\mathbf{k}$ using these overloaded operations, we augment the graph to compute the taylor values $\mathbf{k}(\mathbf{v}) = \mathbf{k}(\mathbf{u}) + N\mathbf{y} \cdot t$. In other words the first order taylor terms of the constraint values give (point at) the values for $N\mathbf{y}$ which we require. An alternative implementation approach is to place an entire taylor series inside a single graph node. We return to discuss this issue below.

We can thus evaluate $\mathbf{n}$ as follows. Use the techniques just described to evaluate $\mathbf{z} = NN'\mathbf{x}$ for arbitrary $\mathbf{x}$, and use an equation solver to find $\mathbf{x}_*$ such that $\mathbf{k} = NN'\mathbf{x}_*$.

Then $\mathbf{n} = N'\mathbf{x}_*$. Similarly λ_* is the solution of $\mathbf{g}N' = \lambda_* NN'$. We can evaluate λ_* by a simple modification to the procedure for $\mathbf{x}_*$, since if we set `y[i].adj=y[i]-g[i]` where $\mathbf{g}$ is the gradient of f (calculated by another reverse sweep) then we can evaluate $N(N'\lambda'_* - \mathbf{g}')$ directly, and feed this into the equation solver. Once the vectors $\mathbf{n}$ and λ_* have been obtained, it is a simple matter to compute the ideal penalty function ϕ. Note that both λ^* and $\mathbf{x}^*$ are of the same dimension as $\mathbf{k}$.

This computation of ϕ is now available in a form which is itself susceptible to automatic differentiation, and the extraction of gradients, directional or full Hessians and so forth. These can in turn be used by optimization software to find a local minimum point $\mathbf{u}_*$ of ϕ, which will correspond to the solution of the original constrained problem. Finally we can apply automatic differentiation to the components of $\mathbf{u}_*$ so as to perform an automatic error analysis or determine the sensitivity of the solution.

A Remark on Equation Solving. It is worth noting that we may use an iterative method of solving the linear equations for $\mathbf{n}$. Provided that we have a contractive iteration mapping near the fixed point, we can then use the methods of [5] to construct the adjoints efficiently as fixed points of an adjoint contraction. The contractive mapping which is dualised to construct the adjoints need not have been used to obtain the solution. The initial construction of the solution could be done by hook or by crook, using conventional floating point arithmetic, and followed by a single (graph constructing) iteration of a contractive mapping (such as a truncated newton step or an ABS conjugate gradient.)

A Remark on Differentiating Taylor Series. In previous work [3][7] there has been a tendency to place the entire taylor series inside a single graph node in order to minimize the amount of manipulative node handling overhead. This is possible because of the result [3] that if $y = f(x)$ where x and y are taylor series then $\partial y^{(p+k)}/\partial x^{(p)} = \partial y^{(k)}/\partial x^{(0)}$ provided f is made up of elementary functions. However, this result is no longer true if f includes operations such as taking derivatives. Although a left shift of a taylor series on the forward pass corresponds to right shift as the appropriate adjoint operation on the reverse pass, nevertheless a left shift does not correspond to differentiation with respect to a taylor variable, because of the scaling of the coefficient constants. The use of reverse accumulation to obtain gradients of functions containing differentiation operations with respect to taylor variables would then require the explicit representation of order r^2 coefficients for a taylor series of order r. Equivalently, there is no adjoint operation corresponding to a projection of a taylor series. The approach of this section therefore requires that the details of any taylor series evaluation are explicitly available in the computational graph.

4. Refolding the Graph. The repeated use of reverse accumulation on a problem of the form

$$f(\mathbf{z}, \nabla g(\mathbf{y}, \nabla h(\mathbf{x})))$$

where $\mathbf{y}, \mathbf{z}$ also depend partially on $\mathbf{x}$, will produce duplicate structures with the same form form as $G(h)$, the graph of h. The number of copies of $G(h)$ is exponential in the depth of gradient nesting. The question therefore arises, whether it might be more efficient to store the various coefficients in a single (enlarged) copy of the graph for h. We have shown in §3 (see also [2, §5],[3, §6]) that reversal through the reversed graph is equivalent to developing a first order taylor series in a single variable forwards through the original

graph.

It turns out (using similar arguments) that nested reverse traversals amount to maintaining precisely the completely heterogenous terms of a multivariate taylor series (ie no variable appearing in power two or higher.)

For example if $\mathbf{p} = \nabla_x\ h(\mathbf{x}), \mathbf{q} = \nabla_x\ g(\mathbf{y}, \mathbf{p})$ then we can evaluate $\mathbf{q}$ as follows: build the graph $G(h)$, reverse though $G(h)$ to obtain the values $\bar{\mathbf{x}} = \mathbf{p}$, copy these into the base of the graph for g, build $G(g)$, reverse through $G(g)$ to obtain $\bar{\mathbf{p}} = \nabla_p\ g$, set $\mathbf{x}_1 = \mathbf{x} + \bar{\mathbf{p}}.t$ where t is the (first) taylor variable, then make a second pass forwards and backwards through $G(h)$ computing the linear taylor terms in t. The first order terms in t for $\bar{\mathbf{x}}_1$ give the value for $\mathbf{q}$. These in turn are built into the base of the graph for f, and the reverse pass through $G(f)$ requires a second pass forward and back through $G(g)$ in a direction $\bar{\mathbf{q}}$ corresponding to the second taylor variable s. This in turn requires a further pass forward and back through $G(h)$ evaluating the coefficients of the terms of order s and st. The next level of nesting would require passes for terms r, rt, rs, rst and so on (hence the exponential growth with nesting level.)

We have already considered representing a reversal through a previously built graph segment as an explicit computational step (corresponding to a graph node). This could be extended so as to define operations representing the addition of another taylor variable to the (forward or reverse portion of the) graph. Combining this with the interpolated taylor series approach [1] holds out the prospect of some time and space savings if the total order of differentiation is higher than about fourth or fifth order, and this is identified as as a promising avenue for future research.

This paper was presented at the Theory Institute on Combinatorial Challenges in Automatic Differentiation, held at Argonne National Laboratories, Illinois, 24-27 May 1993.

References.

[1] Christian Bischof *et al*, 1992, Structured Second- and Higher-Order Derivatives through Univariate Taylor Series, Optimization Methods and Software, *to appear*

[2] Bruce Christianson, 1992, Automatic Hessians by Reverse Accumulation, IMA Journal of Numerical Analysis **12**, 135–150

[3] Bruce Christianson, 1992, Reverse Accumulation and Accurate Rounding Error Estimates for Taylor Series Coefficients, Optimization Methods and Software **1** 81–94

[4] Bruce Christianson *and* Laurence Dixon, 1992, Reverse Accumulation of Jacobians and Optimal Control, Technical Report, Numerical Optimisation Centre, University of Hertfordshire, England, Europe

[5] Bruce Christianson, 1992, Reverse Accumulation and Attractive Fixed Points, Technical Report, Numerical Optimisation Centre, University of Hertfordshire, England, Europe

[6] Bruce Christianson, 1993, A Geometric Approach to Fletcher's Ideal Penalty Function, Technical Report, Numerical Optimisation Centre, University of Hertfordshire, England, Europe

[7] Andreas Griewank *et al*, 1991, ADOL-C: A Package for Automatic Differentiation of Algorithms written in C/C++, ACM Transactions on Mathematical Software *to appear*

[8] Andreas Griewank *et al*, 1993, Some Bounds on the Complexity of Gradients, Jacobians, and Hessians, *in* Complexity in Numerical Optimization, *ed* P.M. Pardalos, World Scientific

[9] Koichi Kubota, 1989, An Implementation of Fast Automatic Differentiation with C++, Abstracts of the 1989 Spring Meeting of the Operations Research Society of Japan, 175–176 *(in Japanese)*