REVERSE ACCUMULATION AND ACCURATE ROUNDING ERROR ESTIMATES FOR TAYLOR SERIES COEFFICIENTS

Bruce Christianson

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

Numerical Optimisation Centre Technical Report 239, July 1991
published Optimization Methods and Software (1992) 1, 81–94

Abstract

We begin by extending the technique of reverse accumulation so as to obtain gradients of univariate taylor series coefficients. This is done by re-interpreting the same formulae used to reverse accumulate gradients in the conventional (scalar) case. Thus a carefully written implementation of conventional reverse accumulation can be extended to the taylor series valued case by (further) overloading of the appropriate operators. Next, we show how to use this extended reverse accumulation technique so as to construct accurate (ie rigorous and sharp) error bounds for the numerical values of the taylor series coefficients of the target function, again by re-interpreting the corresponding conventional (scalar) formulae. This extension can also be implemented simply by re-engineering existing code. The two techniques (reverse accumulation of gradients and accurate error estimates) each require only a small multiple of the processing time required to compute the underlying taylor series coefficients. Space requirements are comparable to those for conventional (scalar) reverse accumulation, and can be similarly managed. We conclude with a discussion of possible implementation strategies and the implications for the re-use of code.

1. Reverse Accumulation. Suppose that we have a calculation which takes as input a number of independent variable values, and produces as output the values of one or more dependent variables. Such calculations are not in practice performed atomically, but are internally expressed as a number of elementary steps involving intermediate variables. (The precise sequence of such steps may depend upon the initial values of the independent variables.) The values of such intermediate variables are typically calculated by elementary operations of the unary form $x = f(u)$ where f is an operation such as $\sin, \log, \surd$ etc, or of the binary form $x = g(u, v)$ where g is an operation such as plus, times etc. Here x is an intermediate or a dependent variable, and u, v are intermediate or independent variables.

Note that we are assuming that each intermediate and dependent variable is assigned a value exactly once, and once assigned that value is not subsequently overwritten. The variables used here thus correspond, not to program variables in code to calculate the dependent variable values, but rather to nodes in the *computational graph* representing such a calculation. The computational graph can be constructed automatically from the evaluation code by overloading the arithmetic operators. For a fuller account see [3] and [18].

In what follows we also assume for simplicity of exposition that dependent variables are not themselves used as arguments to further operations.

Using the computational graph we can evaluate, for each dependent variable y, the gradient vector ∇y (which contains an element $\partial y/\partial u$ corresponding to each independent variable u.) This calculation is performed as follows.

With each variable x we associate an accumulator (or *adjoint variable*) $\bar{x}$, initially zero. To each atomic operation $x = f(u)$ we associate the reverse accumulation step $\bar{u} \mathrel{+}= \bar{x} * f'(u)$, where f' is the derivative of f, $*$ denotes (scalar) multiplication, and $l \mathrel{+}= r$ denotes the operation of incrementing l by r. Similarly, to the atomic operation $x = g(u, v)$ we associate the two reverse accumulation steps $\bar{u} \mathrel{+}= \bar{x} * g_1(u, v)$ and $\bar{v} \mathrel{+}= \bar{x} * g_2(u, v)$ where g_1 and g_2 denote the partial derivatives of g with respect to the first and second arguments respectively.

Gradients are evaluated as follows. Choose a dependent variable y, and set $\bar{y} = 1$ for that variable. Then carry out the reverse accumulation steps in the reverse order to the original sequence of atomic operations. After this is done, we have by the chain rule that $\bar{x} = \partial y/\partial x$ for each independent or intermediate variable x.

Alternatively, we may evaluate an arbitrary linear combination of the gradients corresponding to distinct dependent variables y_i by initializing the corresponding $\bar{y}_i$ to c_i so as to obtain $\sum_i c_i \nabla y_i = \mathbf{c} \cdot J\mathbf{y}$ where J denotes Jacobian.

It can be shown [10, §3.3][3, §6] that the computational cost (in terms of the number and time length of arithmetic operations) of obtaining ∇y (respectively $\mathbf{c} \cdot J\mathbf{y}$) in this way is a small multiple (about three times) that of calculating y (respectively $\mathbf{y}$) in the first place, regardless of the number of independent variables represented in the gradient vector.

This remarkable result is particularly significant in the case where the y_i share components of the calculation by making common use of intermediate variables, and hence have sub-additive total cost.

Even if the entire Jacobian is required, extraction one column at a time using reverse accumulation is attractive compared to forward accumulation provided that the number of dependent variables is small compared to the number of independent variables.

In particular all parallelism which could be exploited in the function computation can also be exploited in the reverse accumulation of the gradient.

In practice there is a performance penalty associated with the use of the computational graph, owing to the fact that the indirect referencing, operation interpretation, and cache usage pattern, are poorly supported by modern conventional hardware relative to the support modern architectures give to floating point arithmetic. Consequently when the computational graph is used to drive a function calculation, these overheads can increase the cpu time required by a factor of up to ten, thus leading to a total cpu time for gradient calculation of up to twenty times the underlying conventional calculation cost [3, §7][4] [21, §5.2].

The space requirements of the reverse accumulation algorithm can be extremely large compared to those of the underlying program to calculate the dependent variable values, since the entire computational graph must be explicitly represented. However, the fast RAM storage requirement can be held to a small multiple of that for the underlying program, with the remaining storage space being held in slow sequential access archive store, or re-calculated from program checkpoints. For further details of storage requirements refer to [10] [13] and [3].

2. Univariate Taylor Series Valued Variables. Frequently it would be useful to have not merely the numerical values of the dependent variables corresponding to particular values of the independent variables, but also information about the behaviour of the dependent variable values under perturbations of the independent variables. Complete information could be obtained by regarding each of the dependent variables as an analytic function of the independent variables, and calculating numerical values for the taylor coefficients of the corresponding multivariate taylor series.

This in turn requires calculating the gradient vector, Hessian matrix, and higher order derivative tensors at the point in question, but if the number of independent variables is large this labour is exhausting [22].

The use of sparse data structures may alleviate this overhead dramatically in the case of *separable* calculations, ie in the case where most of the intermediate variables depend individually (directly and indirectly) upon only a small number of the independent variables [8].

In many cases, however, it is possible to assume that each independent variable is itself a polynomial function (usually a linear function) of an underlying taylor variable t, and thus to be satisfied with expressing each dependent variable as a univariate taylor series in t. In particular, the required numerical coefficients of a multivariate taylor series expansion can usually be extracted fairly efficiently by interpolating a number of univariate taylor expansions. See [12, §3] for further details.

If for each independent variable u_i we have $u_i = a_i + b_i t$ then the terms in the univariate taylor series for a dependent variable y correspond to repeated directional derivatives of the form $d/dt = \sum_i b_i \, \partial/\partial u_i = \mathbf{b}\cdot\nabla$ evaluated at the point $t = 0$, ie for $u_i = a_i$. In effect we may regard t as an "infinitesimal" generator if we are interested in local behaviour of the dependent variables, or as a (possibly quite large) finite offset if we are concerned with global behaviour. In the latter case we generally require a relatively large number of terms in the taylor series (see for example [5]).

Provided the elementary operation are sufficiently smooth, each dependent or intermediate variable x can be regarded as a polynomial series $x = x^{(0)} + t.x^{(1)} + t^2.x^{(2)} + \cdots$, where the $x^{(k)}$ depend (potentially) upon the values for a_i and b_i, but not upon the value of t.

In general, if x is any (truncated) taylor series in t, we write $[x]_k$ to denote the coefficient of t^k in x, so that $x = \sum_k t^k [x]_k$.

We can consider dx/dt as well as the component partial derivatives $\partial x/\partial a_i = \partial x/\partial u_i$. Clearly $[dx/dt]_k = (k+1)[x]_{k+1}$, whence (by induction)

$$[x]_k = \frac{1}{k!}\left[\frac{d^k x}{dt^k}\right]_0$$

Each partial derivative $\partial x/\partial a_i$ is also a taylor series, and we could similarly consider the taylor series $\partial x/\partial b_i$. The interesting thing is that $[\partial x/\partial b_i]_k = [\partial x/\partial a_i]_{k-1}$ for $k > 0$.

This is a special case of the following more general result.

Theorem. Let u be a univariate taylor series in t, and let $x = f(u)$ for some smooth function f. Denote $[x]_k$ by $x^{(k)}$ and $[u]_k$ by $u^{(k)}$. Then for all $k \geq 0, p > 0$ we have

$$\frac{\partial x^{(k+p)}}{\partial u^{(p)}} = \frac{\partial x^{(k)}}{\partial u^{(0)}} \qquad\qquad \frac{\partial x^{(k)}}{\partial u^{(k+p)}} = 0$$

Proof: Since f is smooth, x can indeed be written as a taylor series $x = \sum_k t^k \cdot x^{(k)}$ in t. Expand $f(u)$ as a power series in u, substitute $u = \sum_k t^k \cdot u^{(k)}$ and collect terms in like powers of t. Each $x^{(k)}$ is a (multinomial) function of $u^{(j)}$ for $j \leq k$, but (obviously) $x^{(k)}$ does not itself depend on t, whereas x itself can be considered as a function of all the $u^{(k)}$ and of t.

Now consider the derivative of x with respect to $u^{(p)}$, say. The chain rule gives

$$\frac{\partial x}{\partial u^{(p)}} = f'(u) \cdot \frac{\partial u}{\partial u^{(p)}}$$

which since

$$\frac{\partial u}{\partial u^{(p)}} = t^p \qquad\qquad \frac{\partial x}{\partial u^{(0)}} = f'(u) \cdot \frac{\partial u}{\partial u^{(0)}} = f'(u)$$

gives

$$\frac{\partial x}{\partial u^{(p)}} = \frac{\partial x}{\partial u^{(0)}} \cdot t^p$$

This is true for a range of values of t about 0 hence, if we consider the two sides of this equation as taylor series, corresponding coefficients must be equal. In particular, since t and $u^{(p)}$ are mutually independent, considering the coefficient of $t^{(k+p)}$ gives

$$\frac{\partial x^{(k+p)}}{\partial u^{(p)}} = \left[\frac{\partial x}{\partial u^{(p)}}\right]_{k+p} = \left[\frac{\partial x}{\partial u^{(0)}} \cdot t^p\right]_{k+p} = \left[\frac{\partial x}{\partial u^{(0)}}\right]_k = \frac{\partial x^{(k)}}{\partial u^{(0)}}$$

as asserted by the theorem. A similar argument gives the second identity.

Qed

3. Forward Calculation of Taylor Series Coefficients. We can evaluate the taylor coefficients for $x = f(u)$ and $x = g(u, v)$ numerically given the corresponding values for u and v.

One obvious (but inefficient) way to do this for $x = f(u)$ is by noting that

$$\begin{aligned}
x = f(u) &= f(u^{(0)} + t.u^{(1)} + t^2.u^{(2)} + \cdots) \\
&= f(u^{(0)}) + f'(u^{(0)}).(t.u^{(1)} + t^2.u^{(2)} + \cdots) \\
&+\frac{1}{2!}f''(x^{(0)}).(t.u^{(1)} + t^2.u^{(2)} + \cdots)^2 + \cdots \\
&= f(u^{(0)}) + f'(u^{(0)}).ts + \frac{1}{2!}f''(u^{(0)}).t^2s^2 + \cdots
\end{aligned}$$

where $s = u^{(1)} + tu^{(2)} + t^2u^{(3)} + \ldots$ and so we can evaluate

$$[s^k]_p = [s.s^{k-1}]_p = \sum_{q=0}^{p}[s]_{p-q}[s^{k-1}]_q = \sum_{q=0}^{p}[u]_{p-q+1}[s^{k-1}]_q$$

$$x^{(p)} = \sum_{k=0}^{p}\frac{1}{k!}f^{(k)}(u^{(0)})[s^k]_{p-k}$$

where $[s^0]_0 = 1, [s^0]_p = 0$ for $p \geq 1$ whence

$$x^{(0)} = f(u^{(0)}) \qquad x^{(p)} = \sum_{k=1}^{p}\frac{1}{k!}f^{(k)}(u^{(0)})[s^k]_{p-k} \text{ for } p \geq 1$$

Here for each k we require order n^2 operations (multiplies and adds) to evaluate the first n taylor coefficients for s^k given the corresponding values for s^{k-1}, and hence we require order n^3 operations to evaluate the first n taylor coefficients for x.

This operation count can be reduced to order $n^2 \log n$ by using fast fourier multiplication to convolve the polynomial coefficients, and even further to $(n \log n)^{3/2}$ by using the strategy of [2, Algorithm 2.2].

The (quite large) increase in the order constant means that a large value for n is required before the use of fast convolution algorithms gives a significant performance improvement. Nevertheless, where the use of fast convolution methods has potential to improve performance, this will be noted in what follows.

4. Alternative Calculation Strategies. Alternatively, we can derive the taylor coefficients for $x = f(u)$ as follows. Clearly $x^{(0)} = f(u^{(0)})$. Recall that $x^{(k)} = (1/k) \cdot [dx/dt]_{k-1}$ whence for $k \geq 1$ we have

$$\begin{aligned}
x^{(k)} &= \frac{1}{k}\left[\frac{dx}{dt}\right]_{k-1} = \frac{1}{k}\left[f'(u)\cdot\frac{du}{dt}\right]_{k-1} = \frac{1}{k}\sum_{l=0}^{k-1}[f'(u)]_{k-1-l}\left[\frac{du}{dt}\right]_l \\
&= \frac{1}{k}\sum_{l=0}^{k-1}(l+1)u^{(l+1)}[f'(u)]_{k-(l+1)} = \frac{1}{k}\sum_{l=1}^{k}lu^{(l)}.[f'(u)]_{k-l}
\end{aligned}$$

Note that if we require the first n terms of the taylor series for $f(u)$ then we require only the first $n-1$ terms of the taylor series for $f'(u)$, which in turn require only the first $n-2$ terms of $f''(u)$ and (eventually) only the constant (zero order) term for $f^{(n)}(u)$. Thus once again the calculation requires order n^3 operations.

In the two variable case where $x = g(u, v)$ we can show in a similar fashion that

$$x^{(k)} = \frac{1}{k} \sum_{l=1}^{k} l \left(u^{(l)}.[g_1(u, v)]_{k-l} + v^{(l)}.[g_2(u, v)]_{k-l} \right)$$

again requiring n^3 operations (or $n^2 \log n$ if fast fourier multiplication is used [2, §7].)

However, in the case of the vast majority of common mathematical operations it is well known that the first n terms of the taylor series for x can be calculated in ascending order at a computational cost proportional to n^2 rather than n^3 (respectively $n \log n$ rather than $(n \log n)^{3/2}$ using fast fourier methods), by using the (already computed) lower order series terms for x, $f'(u)$ and u itself.

For example, consider the case $x = \sin u$. Set $w = \sin' u = \cos u$. Then as before

$$x^{(k)} = \frac{1}{k} \sum_{l=1}^{k} l u^{(l)}.[f'(u)]_{k-l} = \frac{1}{k} \sum_{l=1}^{k} l u^{(l)}.w^{k-l}$$

and similarly since $\cos' u = -\sin u$ we have

$$w^{(k)} = \frac{1}{k} \sum_{l=1}^{k} l u^{(l)}.[f''(u)]_{k-l} = -\frac{1}{k} \sum_{l=1}^{k} l u^{(l)}.x^{k-l}$$

Noting that each $w^{(k)}$ depends only upon $x^{(p)}$ for $p < k$ and conversely, we see that the two sets of taylor coefficients for x and w can be calculated together for n terms at a total cost of order n^2.

As an example of a two variable operation consider $x = g(u, v) = u/v$. Then $u = vx$ so

$$u^{(k)} = \sum_{l=0}^{k} v^{(l)}.x^{(k-l)} = v^{(0)} x^{(k)} + \sum_{l=1}^{k} v^{(l)}.x^{(k-l)}$$

whence

$$x^{(0)} = \frac{u^{(0)}}{v^{(0)}} \qquad x^{(k)} = \frac{u^{(k)} - \sum_{l=1}^{k} v^{(l)}.x^{(k-l)}}{v^{(0)}} \text{ for } k \geq 1$$

Again, note that the total computational cost is order n^2.

For details of use of fast fourier methods to reduce the operation count to order $n \log n$, see [2, §5].

These taylor series may be evaluated by further overloading the operators which build the computational graph, or by a subsequent re-traversal of the computational graph in the forward direction for specific choices of the coefficients $u_i^{(1)}$ in the independent variables $u_i = u_i^{(0)} + u_i^{(1)} t$. For further details see [19, pp 155–157], [11, C++ code], and [5], [6].

5. Reverse Accumulating Gradients of Taylor Series Coefficients.

The first objective of this note is to point out that the technique of reverse accumulation can be applied to calculate gradients of univariate taylor series.

Let us clarify what we mean by this. If y is a dependent variable with taylor series coefficients $y^{(k)}$ then we can consider for specific k the gradient vector $\nabla y^{(k)}$ which contains an element $[\partial y/\partial u]_k = [\partial y/\partial u^{(0)}]_k = \partial y^{(k)}/\partial u^{(0)}$ corresponding to each independent variable u. (Recall that each independent variable u is itself a taylor series, usually linear, in t.)

Our assertion is that we can use reverse accumulation to calculate simultaneously all the gradient vectors $\nabla y^{(k)}$ for the given y. It makes no difference whether we regard this as taking a gradient of each term of the taylor series for y or as evaluating a taylor series for the gradient vector ∇y, since $\nabla \left(y^{(k)}\right) = [\nabla y]_k$.

In fact, our purpose here is to show that the very same formulae used in the scalar case can be re-interpreted as applying to taylor series values. For let x, u also be univariate taylor series in t with $x = f(u)$. Let $\bar{x}$ be the taylor series defined by

$$\bar{x} = \frac{\partial y}{\partial x} = \frac{\partial y}{\partial x^{(0)}} \quad \text{ie } \bar{x}^{(k)} = \frac{\partial y^{(k)}}{\partial x^{(0)}} = \frac{\partial y^{(k+p)}}{\partial x^{(p)}} \text{ for } k, p \geq 0$$

The partial derivative $\partial y^{(p)}/\partial u^{(0)}$ is, by the chain rule, made up of a sum of contributions over all intermediate variables x such that x depends directly upon u. Considering the effect that a small change in $u^{(0)}$ would have upon $y^{(p)}$ through x we see that the contribution to $\partial y^{(p)}/\partial u^{(0)}$ corresponding to $x = f(u)$ is

$$\sum_{k \leq p} \frac{\partial y^{(p)}}{\partial x^{(k)}} \cdot \frac{\partial x^{(k)}}{\partial u^{(0)}} = \sum_{k \leq p} \frac{\partial y^{(p-k)}}{\partial x^{(0)}} \cdot \frac{\partial x^{(k)}}{\partial u^{(0)}} = \sum_{k \leq p} [\bar{x}]_{p-k} [f'(u)]_k = [\bar{x} \cdot f'(u)]_p$$

where $\bar{x} \cdot f'(u)$ denotes the polynomial product of $\bar{u}$ and $f'(u)$ considered as taylor series. Thus to the forward step $x = f(u)$, where now x and u are taylor series, we associate the reverse accumulation step $\bar{u} \mathrel{+}= \bar{x} * f'(u)$, where f' is the derivative of f, $\bar{x}, \bar{u}, f'(u)$ are all taylor series, and $*$ now denotes taylor series product (ie coefficient convolution.)

A similar argument applies to the two-variable case. With the atomic operation $x = g(u, v)$ we associate the two reverse accumulations $\bar{u} \mathrel{+}= \bar{x} * g_1(u, v)$ and $\bar{v} \mathrel{+}= \bar{x} * g_2(u, v)$ where g_1 and g_2 denote the partial derivatives of g with respect to the first and second arguments respectively.

Thus we can reverse accumulate $[\nabla y]_k = \nabla y^{(k)}$ as follows. Set $\bar{y} = 1$ (ie set $\bar{y}^{(0)} = 1$ and $\bar{y}^{(p)} = 0$ for $p > 1$) and then carry out the reverse accumulation steps in taylor form, in the reverse order to the original sequence of atomic operations. After this is done we have that $\bar{x} = \partial y/\partial x$, ie $\bar{x}^{(k)} = \partial y^{(k)}/\partial x^{(0)}$, for each independent or intermediate variable x.

These gradient components may correspond to mixed derivatives which are of interest in themselves to a calculation. For example, the efficient numerical approximation of the bifurcation points and cusp singularities of dynamical systems requires (interpolations of) gradients of exactly this form [12, §2,3].

Alternatively, gradients of taylor coefficients may be useful in estimating the sensitivity of the taylor series coefficient values to changes in the initial values of the independent variables. One particular application of this is the estimation of the effects of rounding error, an issue to which we return below (§7).

Note that, since "constant" parameters may be regarded as independent variables with fixed (unchanging) values, this method may also be used to estimate sensitivities to such parameters.

6. Performance Analysis. The method described in the previous section requires that we obtain a taylor series for the first derivatives of each elementary operation involved in the function calculation, but only for the first derivatives. As we have seen, frequently these first derivative series are available anyway as a by-product of the taylor series calculation for the underlying function, or can be made available be re-arranging that calculation.

Although $[f'(u)]_k \neq (k+1)[f(u)]_{k+1}$ or anything as simple as that, nearly all interesting functions are solutions to some ODE of order (at most) two, from which recurrence relations for the derivative coefficients can be derived.

Once the taylor series corresponding to these derivatives are available, the reverse accumulation requires of order n^2 multiplies and adds per intermediate variable to produce the whole of ∇y to n terms (reducing to order $n \log n$ if fast fourier methods are used to perform the convolutions.)

Assuming that the derivative taylor series can be made available in a time comparable to that required to calculate the taylor series corresponding to the underlying function value (a safe assumption), the time requirement for calculating the whole of the matrix ∇y is then a small multiple of the time required to evaluate the taylor series for y.

The space requirement is the product of the space requirement for conventional (scalar) reverse accumulation with a number of the form $1 + \alpha n$, where typically $\alpha << 1$. Similar remarks apply to those made above in §1.

The calculations given in [3, §6] to evaluate a linear combination of rows of the Hessian of a function f are a special case of the construction described here. To see this, argue as follows. Observe first that if $y = f(\mathbf{u})$ where $\mathbf{u} = \mathbf{a} + \mathbf{b}t$ then $\mathbf{b} \cdot Hf = \nabla(\mathbf{b} \cdot \nabla f) = \nabla(dy/dt) = \nabla y^{(1)}$.

In the forward pass of the code to calculate the Hessian given in [3, §4] each x_i corresponds to $x_i^{(0)}$ in our notation, and w_i to $x_i^{(1)}$. The values

$$D_p f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i}) \quad \text{and} \quad \sum_{q=1}^{n_i} w_{\tau_i q}.D_q D_p f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$$

(which as remarked in [3] could be calculated on the forward pass) correspond to the zero-th and first order terms of the taylor series for $D_p f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$.

In the reverse pass, we see that each $\bar{x}_i$ corresponds to $\bar{x}_i^{(0)}$ in our notation, and also by induction each $\bar{w}_i$ corresponds to $\bar{x}_i^{(1)}$ since if $\bar{w}_i = \bar{x}_i^{(1)}$ then the term added to $\bar{w}_{\tau_i p}$in [3] is in fact

$$\bar{w}_i.D_p f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i}) + \bar{x}_i \sum_q w_{\tau_i q}.D_q D_p f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})$$

$$= \bar{x}_i^{(1)}[D_p f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})]_0 + \bar{x}_i^{(0)}[D_p f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})]_1$$
$$= [\bar{x}_i.D_p f_i(x_{\tau_i 1}, \ldots, x_{\tau_i n_i})]_1$$

whence $\bar{w}_{\tau_i p} = \bar{x}^{(1)}_{\tau_i p}$ in agreement with the algorithm given here.

This approach can be used to obtain the entire Hessian row by row, as suggested in [3], at a total cost which is linear in the number of independent variables. However, for separable problems where the entire Hessian is required, better performance can be obtained by using sparse data structures and conventional (forward) accumulation techniques [8],[9] as mentioned above (§2).

Where the entire third or higher order derivative tensor is required, the method outlined here is even less attractive. However where only part of the tensor is required, ie the number of required (linear combinations of) components of the tensor is small relative to the number of independent variables, good performance may be obtained using this approach.

Reverse accumulation of gradients for taylor series coefficients is not itself a new idea [11]. However the analysis presented here allows reverse accumulation code written for the scalar case using operator overloading to be extended with relatively little modification so as to provide reverse accumulation of taylor series coefficient gradients. We return to this point below (§9).

An alternative approach to the problem would be to regard each computational step as a calculation of the n taylor terms of the result (as dependent variables) from the corresponding taylor terms of the parameters (as independent variables) by "sub-atomic" operations, and use overloading at this (lower) level to construct the computational graph for the whole calculation. The gradients can then be computed in one reverse pass, by starting from $y^{(n)}$ and afterwards using the fact that

$$\frac{dy^{(p)}}{dx^{(q)}} = \frac{dy^{(n)}}{dx^{(n+q-p)}}$$

This approach has not been followed here for three reasons.

Firstly, it involves a larger interpretive overhead in traversing the computational graph. The trend in automatic differentiation is towards each node in the graph representing more computation (ie towards "larger" elementary operations).

Secondly, the use of exact arithmetic to reduce roundoff error in the taylor steps (see §8 below) would be hampered by this approach.

Thirdly, the approach followed here appears to lay a better foundation for our next task, the derivation of accurate error bounds for taylor series coefficient values.

7. Accurate Error Bounds for Scalar Valued Calculations. It is frequently of interest to have accurate bounds on the extent to which the calculated numerical value of a dependent variable may have been affected by rounding errors during the course of the calculation.

Interval arithmetic usually gives bounds which are too loose (frequently by several orders of magnitude), since it takes no account of those rounding errors in calculating intermediate values which in fact are guaranteed to cancel one another out.

For example, suppose that $x = f(u), y = x + 1/x$, and suppose that the calculated value for x is known to be within δ of 2, where δ represents the rounding error in the calculation of $f(u)$ from u.

Then (assuming just for the sake of argument that δ is small relative to 1 and large relative to the subsequent rounding errors) $1/x$ is within $\frac{1}{4}\delta$ of $\frac{1}{2}$, and so (naively) y is within $\frac{5}{4}\delta$ of $2\frac{1}{2}$. In fact, however, we can see that y must be within $\frac{3}{4}\delta$ of $2\frac{1}{2}$. This corresponds to the fact that $dy/dx = 1 - 1/x^2 = \frac{3}{4}$.

For convenience we suppose in what follows that the intermediate and dependent variables are indexed in some common fashion consistent with the dependencies in the computational graph, so that each dependent variable y can be identified with x_m for some index m.

In the general case, we associate with each atomic step of the calculation an upper bound δ_i on the rounding error introduced at that step into the value of the corresponding intermediate variable x_i. Recall that following a reverse accumulation starting from a dependent variable $y = x_m$ we have $\bar{x}_i = \partial y/\partial x_i$. Consequently an approximate worst-case upper bound on the rounding error for the dependent variable y is given by

$$\epsilon = \sum_i |\bar{x}_i| \; \delta_i$$

where the summation is over all computational steps upon which y (functionally) depends (ie all steps in which rounding error affecting y may have been introduced).

If desired, the summation for ϵ may be extended to contain terms corresponding to initial uncertainties in the independent variables, as well as the terms corresponding to rounding errors introduced during the course of the computation, thus producing an estimate of the total uncertainty in y.

Usually for each given hardware and software environment we can specify a small constant η such that δ_i is bounded by the greater of $\eta \cdot |x_i|$ and *minreal* for all steps of all possible calculations.

However, as Iri shows in [17], it is possible to give a more exact estimate of the rounding error, as follows.

Supplement the (forward) calculation of the scalar quantities $x_i = f(u_i)$ by corresponding interval calculations for interval quantities $X_i = F(U_i)$. Clearly $x_i \in X_i$.

Replace each reverse accumulation step $\bar{u} \mathrel{+}= \bar{x} * f'(u)$ by the corresponding interval-valued operation $\bar{U} \mathrel{+}= \bar{X} * F'(U)$ (where $F'(U)$ denotes the interval-valued derivative and $*$ denotes interval multiplication).

Finally set $\Delta_i = [-\delta_i, \delta_i]$ where now δ_i is a (tight) upper bound for the worst-case rounding error that could occur in the step $X_i = F(U_i)$, and define

$$E = \sum_i \bar{X}_i * \Delta_i$$

Then this gives an error bound for the rounding error of $y = x_m$ which is accurate in the sense of being both rigorous and sharp.

Rigorous means that the correct value for y and the calculated value can differ by at most half the width of E (ie the true value for y must lie in the interval $y + E$, where y is the scalar calculated quantity.) Note that usually the width of the interval E will be much less than the width of the interval Y, sometimes by several orders of magnitude.

Sharp means that this bound is asymptotically tight as the rounding errors diminish. More precisely, the ratio between the width of E and the actual worst-case rounding error tends to unity as the accuracy of machine arithmetic improves (ie as $\eta \to 0$). In practice, of course, the bound tends to be pessimistic, since the worst-case scenario, although possible, rarely occurs.

These results can be proved by considering the effects upon y of a perturbation, representing the effects of rounding, applied to each variable in turn, in the order of their calculation. The use of the mean value theorem in conjunction with interval arithmetic allows us to disregard the effect of second and higher order derivatives (contrast with [21]).

In practice, for a given level η of machine precision, it may be necessary or desirable to apply this accurate error bounding technique recursively in order to tighten the bounds on certain intermediate variable values. This can be done in a similar fashion to the "preaccumulation" technique used for derivatives [3], provided care is taken to distinguish between the (tight) bound on the error in the tightened intermediate variable (which will include the summation terms corresponding to uncertainties in the parameters to the subgraph being tightened) and the (tight) bound on the rounding error introduced in the calculation of the intermediate variable being tightened (which will exclude such terms.)

One obvious class of candidates for this "premature tightening" is singularities, for example points in the computational graph where division is attempted by an interval containing zero. In this case, premature tightening may remove the singular value from the relevant interval.

Another case where premature tightening may be useful is at intermediate variables for which roundoff errors in the (forward) calculation of the values of the corresponding elementary function or of its derivative have a large effect on the worst-case error bound.

One strategy for identifying such variables is by checking the size of $\delta_i \bar{x}_i$ relative to ηy. Assuming that $\delta_i \approx \eta |x_i|$, the variables sought are those for which $|\bar{x}_i x_i| >> |y|$. If any improvement is to result from premature tightening, then at least one of the corresponding intervals $X_i, \bar{X}_i$ must span several (binary) orders of magnitude (or include zero).

If this is the case for $X_i = F_i(X_j)$ then it may be worth tightening X_j. It may also be worth tightening X_j when $F'(X_j)$ spans several binary orders of magnitude, if this has the prospect of tightening the appropriate $\bar{X}$'s.

8. Accurate Error Bounds for Taylor Series Coefficients.

The second objective of this note is to show that the results of the previous section can be extended to the case where the variable values correspond to taylor series. Indeed, once again the very same formulae can be re-interpreted so as to apply to taylor series valued variables.

The effect on $y^{(p)}$ corresponding to roundoff error in the calculation step corresponding to the intermediate variable x is approximately

$$\sum_{k=0}^{p} \frac{\partial y^{(p)}}{\partial x^{(k)}} \cdot \delta^{(k)} = \sum_{k=0}^{p} \frac{\partial y^{(p-k)}}{\partial x^{(0)}} \cdot \delta^{(k)} = \sum_{k=0}^{p} \bar{x}^{(p-k)} \cdot \delta^{(k)} = [\bar{x} \cdot \delta]_p$$

where $\delta^{(k)}$ is the roundoff error in the calculation of $x^{(k)}$ introduced in that calculation step.

Suppose that we have associated with each atomic step of the calculation an upper bound $\delta_i^{(k)}$ on the rounding error introduced at that step into the value of the corresponding intermediate variable coefficient $x_i^{(k)}$.

Define the taylor series ϵ by

$$\epsilon = \sum_i |\bar{x}_i| \, \delta_i$$

where the modulus of the taylor series for $\bar{x}_i$ is taken component by component.

Then for a dependent variable $y = x_m$ an approximate worst-case upper bound on the rounding error for $y^{(p)}$ will be given by $\epsilon^{(p)}$.

Once again, augmenting the scalar taylor series operations by interval valued operations in the same way as above allows us to produce error bounds on the taylor coefficients for y which are accurate in the sense of being both rigorous and sharp.

Specifically, supplement the (forward) calculation of the taylor series quantities $x_i = f(u_i)$ by corresponding interval valued taylor series calculations $X_i = F(U_i)$ which associate an interval $X_i^{(p)}$ with each term $x_i^{(p)}$.

Replace each reverse accumulation step $\bar{u} \mathrel{+}= \bar{x} * f'(u)$ by the corresponding interval-valued operation $\bar{U} \mathrel{+}= \bar{X} * F'(U)$ (where $F'(U)$ denotes the interval-valued taylor series for the derivative and $*$ denotes multiplication of interval valued taylor series, ie convolution of interval valued coefficients).

Finally set $\Delta_i^{(k)} = [-\delta_i^{(k)}, \delta_i^{(k)}]$ where $\delta_i^{(k)}$ is a (tight) upper bound for the worst-case rounding error that could occur in the k-th term of the step $X_i = F(U_i)$, ie the worst rounding error in $x^{(k)}$ that could occur for any point $u \in U_i$ in the step $x = f(u)$, and define

$$E = \sum_i \bar{X}_i * \Delta_i$$

Then $E^{(k)}$ gives an error bound for the rounding error of $y^{(k)}$ which is accurate in the sense of being both rigorous and sharp.

Furthermore, provided the values $\delta^{(k)}$ are made available in the computational graph, this error estimate calculation may readily be combined with the reverse accumulation process for the calculation of the gradient of the respective dependent variable, at an additional cost of order n^2 per step (order $n \log n$ if fast convolution is used.)

It remains to determine tight bounds $\delta^{(k)}$ for the roundoff errors introduced at each step into the taylor series calculations for $x^{(k)}$ with $k > 0$.

For some operations this is straightforward. We suggest two (non-exclusive) alternative ways by which this might be done in general. First, by using the methods of Corliss [5].

Second, by using *exact arithmetic* as described in (for example) [20] and [1, Chapter Three], extended to intervals. Since the taylor series formulae involve mainly convolutions, similar techniques to those used for scalar products should work well in minimizing the accumulated error during the forward step. Exact arithmetic could also be used to advantage during the reverse accumulation calculation and interval error estimate summation.

The numerical stability of alternative forms for evaluating the taylor coefficients of intermediate variables and the corresponding first derivatives is a live issue, particularly if fast convolution methods are used.

Premature tightening techniques can be applied to the roundoff error estimation for taylor series, just as in the scalar valued case, although the criteria must now be applied on a maximum component basis.

9. Re-using Code. We have shown that reverse accumulation and accurate error bounds can be extended to taylor series valued calculations by re-interpreting the same formulae used in the scalar valued case. This has implications for the re-use of code implementing such techniques.

Where code to implement reverse accumulation is written in a language which permits operator overloading, such as Ada, or better yet in an object-oriented language such as C++, then it becomes possible to combine the reverse accumulation code with similar code written to perform interval arithmetic, exact arithmetic, and taylor series arithmetic, in such a way as to produce code which will reverse accumulate gradients of taylor series coefficients.

Similarly the very same code used to calculate accurate roundoff error bounds for function values in the scalar case can be used to produce corresponding bounds for taylor series coefficients, by combining that code with other, existing, well understood code, provided that sufficient care has been taken in the specification of the different pieces of code.

One possible development strategy would be as follows. Begin with a "plain vanilla" scalar reverse accumulation package and modify (if necessary) so as to store elementary function values as pairs $(f(x), f'(x))$.

Re-declare the relevant variables as being of truncated-taylor type, and combine with (pre-existing or independently developed) code for manipulating such types. Re-building the resulting package to support more efficient manipulation of the taylor coefficients (for example, by incorporating fast convolution, exact arithmetic, and sparse data structures in the taylor manipulation code) is straightforward, whether done immediately or at a later stage.

Take an interval processing package, including accurate scalar product if desired, and combine with the taylor manipulation code. Re-overload the latter to match the former where necessary. (In particular, coefficient convolution can be interpreted as a scalar product.) Test the resulting interval-valued taylor manipulation package and then lay to one side.

To the original (vanilla) package, add approximate error estimates. Re-declare the relevant variables in the resulting package to be of interval type (or of (scalar, enclosing interval) composite type, as appropriate.) Combine directly with the same interval processing package used earlier, and test the resulting error estimates for accuracy.

When satisfied, combine this package with the interval taylor package prepared earlier, by re-overloading appropriate variables as of type "taylor series of (previous type)".

The fact that the very same code can be re-used in these ways not only shortens development time, but also heightens the confidence of the implementor in the correctness of the final artifact.

References.

[1] M.C. Bartholemew-Biggs, 1990, An Introduction to Numerical Computation using Ada, Technical Report 235, Numerical Optimisation Centre, Hatfield Polytechnic

[2] R.P. Brent and H.T. Kung, 1978, Fast Algorithms for Manipulating Formal Power Series, JACM **25**(4) 581–595

[3] D.B. Christianson, 1991, Automatic Hessians by Reverse Accumulation, IMAJNA (to appear)

[4] D.B. Christianson and S. Davis, 1991, Why is Automatic Differentiation so Slow and What can be Done About it, *in preparation*

[5] G.F. Corliss and L.B. Rall, 1991, Computing the Range of Derivatives, IMACS Annals on Computing and Applied Mathematics, *to appear*

[6] G.F. Corliss, 1991, Overloading Point and Interval Taylor Operators, *in* Automatic Differentiation of Algorithms, SIAM, Philadelphia.

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

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

[9] L.C.W. Dixon, 1991, Use of Automatic Differentiation for Calculating Hessians and Newton Steps, *in* Automatic Differentiation of Algorithms, SIAM, Philadelphia.

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

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

[12] A. Griewank, 1991, Automatic Evaluation of First and Higher-Derivative Vectors, International Series of Numerical Analysis Vol 97, Birkhauser Verlag, Basel

[13] A. Griewank, 1991, Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation, Optimization Methods and Software, *to appear*

[14] 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

[15] 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

[16] 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

[17] M. Iri *et al*, 1988, Estimates of Rounding Error with Fast Automatic Differentiation and Interval Analysis, Technical Report RMI 88-12, Department of Mathematical Engineering and Instrumentation Physics, University of Tokyo

[18] M. Iri, 1991, History of Automatic Differentiation and Rounding Error Estimation, *in* Automatic Differentiation of Algorithms, SIAM, Philadelphia.

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

[20] U. Kulish and W. Miranker, 1981, Computer Arithmetic in Theory and Practice, Academic Press.

[21] S. Linnainmaa, 1976, Taylor Expansion of the Accumulated Rounding Error, BIT **16** 146–160

[22] R.D. Neidinger, 1991, An Efficient Method for the Numerical Evaluation of Partial Derivatives of Arbitrary Order, ACM Transactions on Mathematical Software, *to appear*