ADIFOR Working Note #6:

Structured Second- and Higher-Order Derivatives through Univariate Taylor Series*

MCS Preprint P296–0392

by

Christian Bischof, George Corliss, and Andreas Griewank

Abstract

Second- and higher-order derivatives are required by applications in scientific computation, especially for optimization algorithms. The two complementary concepts of interpolating partial derivatives from univariate Taylor series and preaccumulating of "local" derivatives form the mathematical foundations for accurate, efficient computation of second- and higher-order partial derivatives for large codes. We compute derivatives in a fashion that parallelizes well, exploits sparsity or other structure frequently found in Hessian matrices, can compute only selected elements of a Hessian matrix, and computes Hessian $\times$ vector products.

1 Introduction

Discussions of automatic differentiation for computing first-order partial derivatives and Taylor coefficients of arbitrary order have appeared in the literature regularly over the past 30 years [5,15, 16,23,24,30]. Juedes [20] includes a survey of 29 software packages for automatic differentiation. In this paper, we describe two concepts:

1. interpolation of partial derivatives from an ensemble of Taylor series of single independents, and
2. preaccumulation of "local" derivatives at the statement or scalar function level.

The complementary concepts of interpolating partial derivatives from univariate Taylor series and preaccumulating "local" derivatives form the mathematical basis on which to build a software tool capable of efficiently computing accurate second- and higher-order partial derivatives for large codes. This is a significant new capability: neither symbolic tools nor divided difference approximations can deliver that capability.

Our focus is on second-order partial derivatives as required by optimization software. Our approach for the efficient computation of second derivatives has the following characteristics:

Enhances Parallelism: Any parallelization or vectorization built into the original code is maintained. The additional code for computing first- and second-order derivatives increases the scope for efficient parallelization.

Utilizes Structure: The generated code takes advantage of the sparsity that is often present in Hessian matrices.

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38 and through NSF Cooperative Agreement No. CCR-8809615.

Computes Hessian $\times$ vector: The generated code can compute a Hessian $\times$ vector product directly in a manner that is much more efficient than first computing the Hessian and then multiplying.

Allows Slices: It is possible to compute selected elements of the Hessian without computing the entire matrix. This capability allows for run-time storage trade-offs when the Hessian is too large to compute the entire matrix at once, or if the application requires only some of the elements (the diagonal, for example).

Supports Vector Functions $f : \mathbf{R}^n \mapsto \mathbf{R}^m$: A separate Hessian is computed for each component with one invocation of the tool, subject only to limitations of memory size. This is necessary for many optimization algorithms and for efficient parallelization.

Provides a Growth Path: The framework for computing second-order derivatives generalizes to higher-order derivatives.

In Section 2, we outline the need for second- and higher-order derivatives in scientific computations, especially in optimization, and point out some of the reasons for the increased computational complexity of second derivatives. In Section 3, we describe automatic differentiation applied to the naive propagation of gradients and Hessians in the forward mode. In Sections 4 and 5, we present the two contributions of this paper: the interpolation of partial derivatives from univariate Taylor series and the preaccumulation of "local" derivatives, respectively. Section 6 shows how to access second derivatives, to exploit sparsity, to compute subsets of a Hessian, and to compute Hessian $\times$ vector products. In Section 7, we generalize the second-derivative results to the computation of higher-order mixed partial derivatives. Finally, we outline conclusions and directions for further research.

2 Need for High-Order Derivatives

Second- and higher-order derivatives are required by applications in scientific computation, especially for optimization algorithms.

2.1 Derivatives in Optimization

The primary motivation for adding the ability to compute second derivatives comes from optimization. Given $f : \mathbf{R}^n \mapsto \mathbf{R}$, unconstrained optimization algorithms minimize f locally by solving $\nabla f = 0$ using a Newton or a secant-type iterative method [12, 13]. The Newton iteration requires the Hessian $\nabla^2 f$. In nonlinearly constrained optimization, the curvature of the constraint surfaces is represented by the Hessians $\nabla^2 c_i$ of the active constraints $c_i(x) = 0$. Often, all these second derivatives are aggregated into the Hessian of the Lagrangian

$$\nabla^2 L = \nabla^2 f + \sum_i \lambda_i \nabla^2 c_i,$$

where the Lagrange multipliers λ_i are derived in some way from first-derivative information, that is, the gradients of the objective and the active constraints. In most large-scale optimization problems, the Hessians of the objective and constraints are sparse or otherwise structured.

Moreover, many optimization codes require at various stages of the calculation only limited second-derivative information, for example:

$\nabla^2 L\, v$	Hessian $\times$ vector products for iterative solvers
$v^T \nabla^2 L\, v$	Second directional curvature in line search
$\mathrm{Diag}(\nabla^2 L)$	Diagonals for preconditioning of equations
$\nabla^2 L\, Z$	One-sided projection onto tangent space
$Z^T \nabla^2\, L\, Z$	Two-sided projection onto tangent space

Here, v is an arbitrary vector, and Z is a rectangular matrix whose columns span the orthogonal complement of the active constraint gradients. Another interesting optimization technique that involves selected second-derivative information is the use of merit functions that are both smooth and exact [14]. That is, that obtain unconstrained minima exactly at the constraint minimizers. Williamson uses exact Hessian values in a nonlinear programming algorithm [31].

Secant methods for approximating second derivatives have been quite successful in the context of unconstrained optimization. For problems of this type, they are considered a useful alternative to the true Hessian, partly because they reduce the linear algebra effort per step from $\mathcal{O}(n^3/3)$ to $\mathcal{O}(n^2)$ in the dense case. However, this saving is lost in the nonlinearly constrained case, and the determination of a successful approximation to the Hessian of the Lagrangian is still an open research question. One primary reason is that approximating the Hessian in constrained optimization is a much more complicated problem than approximating the Hessian in unconstrained optimization, since the Hessian of the Lagrangian is not necessarily positive definite even at the solution.

The other strong motivation for using the analytical Hessian is that it often has a great deal of structure that a secant approximation cannot take into account. For example, in many applications, the Hessian is very sparse, and by exploiting this sparsity, the computation the true Hessian is not expensive. Currently, this structuring is done by hand in some applications. Hence, it is important that a tool for computing second derivatives support sparse computations.

Software for solving problems in unconstrained optimization (Dennis and Schnabel [12], for example) often views analytic second-derivative information as optional, but desirable. If the user is able to supply code for computing $\nabla^2 f$, the algorithms often display the quadratic convergence rate of Newton's method, rather than the superlinear rate of the secant method. Writing code for the analytic second derivative is a tedious and error-prone process, even for problems of modest code size like those in the Hock and Schittkowski [19] or the MINPACK-2 [3] test suites. For applications such as multidisciplinary optimization [27,28] where the code defining the function f may be tens of thousands of lines long, hand-coding even ∇f is unthinkable. Software for constrained optimization has viewed analytic second-derivative information as so difficult to supply that only Williamson's code [31] even provides the opportunity for a knowledgeable user to supply it. The techniques outlined in this paper make the computation of accurate second-derivative information feasible. With this information, software for unconstrained optimization can routinely use second derivatives to attain quadratic convergence, and software for the constrained case can be redesigned without being restricted to Hessian-free algorithms.

Third-, fourth-, and higher-order partial derivatives are required by some algorithms for solving potentially degenerate nonlinear systems or for nested optimization problems. Berz [6] and Michelotti [22] discussed the problem of beam tracing in the Superconducting Super Collider. In that application, up to $m = 10$ derivatives in $n = 6$ variables are needed to describe the physical system. As scientists understand that high derivatives can be feasibly obtained when n is reasonably small, more applications requiring such high-order partial derivatives may be recognized, and algorithms may be developed that efficiently utilize higher derivatives.

2.2 Cost of Complete Jacobians and Hessians

Using the reverse mode of automatic differentiation, one can compute the gradient of a scalar function for no more than five times the arithmetic operations needed to evaluate the function itself [4]. However, this result does not extend to the Jacobians of vector functions. By applying the cheap gradient result to each component of the vector function, one can bound the total cost of evaluating the Jacobian by no more than five times the sum of the costs of evaluating all of the component functions separately. This total cost is often much larger than the cost of evaluating all components of the vector function simultaneously, where common subexpressions need to be calculated only once. For vector functions that are themselves gradients of f, such joint terms are typical. As an example, let us consider the sum of squares residual

$$f(x) = \tfrac{1}{2}\,||r||_2^2, \quad r = Ax - b,$$

where A is a matrix, and b is a vector of compatible size. The complexity of evaluating r and its norm f is given by the number of nonzero elements in A, which we assume to be significantly larger than the number n of independent variables. The gradient and Hessian of f are given by

$$\nabla f = A^T r, \quad \nabla^2 f = A^T A.$$

The entire vector ∇f can be evaluated for exactly twice the effort of calculating r. Evaluating just one component of ∇f separately costs half as much, since the intermediate vector r must be calculated first. Hence, the cheap gradient result applied to each component of ∇f implies that the Hessian can be calculated for no more than $5n$ times the effort of evaluating r. The factor 5 in that bound can be left off, but otherwise it provides a realistic estimate. Thus, we conclude that Hessians can indeed be up to $\mathcal{O}(n)$ times as expensive as the underlying scalar function and gradient. Moreover, the Hessians may have little or no sparsity, even if there is a lot of structure. For example, if A is the identity appended with one additional dense row, then the Hessian ∇f is dense.

2.3 Partitioning and Parallelism

Independent of what methodology one adopts for the evaluation of Jacobian and Hessian matrices, both run-time and memory requirements will usually grow at least linearly with the number of independent variables. Now suppose the simultaneous differentiation with respect to all n independent variables on one processor takes too long and/or requires too much memory. Then, one may partition the independent variables into a family of groups with no more than some $p < n$ elements and perform several derivative evaluation runs, either in parallel on several processors or successively in a serial mode. For first derivatives, this partitioning approach can be adopted quite easily, for example, by utilizing the interface of ADIFOR (Automatic Differentiation In FORtran) [7] or ADOL–C (Automatic Differentiation Of aLgorithms written in C) [18].

Each pair of independent variables that corresponds to a nonzero entry in the Hessian must occur together in at least one of the groups. In graph-theoretical terms, this requirement means covering the connectivity graph of the sparsity pattern by a set of subgraphs with no more than p nodes such that each edge occurs in at least one of the subgraphs. In the dense case, the graph is a clique with $n(n-1)/2$ edges. Since each p-element subgraph has at most $p(p-1)/2$ edges, one needs at least

$$\frac{n(n-1)}{p(p-1)} \approx \left(\frac{n}{p}\right)^2$$

differentiations with respect to groups of no more than p independent variables to calculate the Hessian in parallel or sequentially in pieces.

If p is even and divides n, then the following scheme comes within a factor of two of this lower bound. Partition the independent variables into $2n/p$ fixed subgroups of $p/2$ elements each. There are

$$\left(\frac{2n}{p} - 1\right)\frac{n}{p} \approx 2\left(\frac{n}{p}\right)^2$$

pairwise combinations of subgroups, each forming a group of p elements. Spreading the evaluation of a dense Hessian in time or across processors increases the (serial) complexity at most by a factor of two. Since no communication is necessary at all, linear speed up is achievable on any multiprocessor. Hence, roughly the same complexity growth by a factor of two must occur if the Hessian is simply treated as the Jacobian of the gradient, disregarding its symmetry.

It is not immediately clear how the simple scheme sketched above can be adopted to the sparse case. In this paper, we will propose a scheme that makes full use of sparsity and can be partitioned to any level without any communications overhead.

3 Forward-Mode Hessians

Automatic differentiation is a well-known technique for the accurate, efficient computation of derivatives [15,16,17,21,24]. It is neither symbolic nor based on potentially unstable finite-difference approximations. It propagates values according to the familiar rules of calculus. There are two fundamental modes for propagating derivative values: the forward mode that we use here, and the reverse mode (see [15]).

One could use the forward mode of automatic differentiation to compute the gradient and the dense Hessian of f by propagating the first- and second-derivative objects strictly in the forward mode [25]. We describe how this would be done to show that the combination of preaccumulation and interpolation yields much more efficient code.

Suppose that u and v are active variables (they depend on values of independent variables). The values of ∇u, ∇v, $\nabla^2 u$, and $\nabla^2 v$ have been computed along with the values for u and v. As an example of a typical operation, suppose that $f = u \cdot v$. Then by the chain rule, we have

$$\begin{aligned} f &= uv \\ \nabla f &= u \cdot \nabla v + \nabla u \cdot v \\ \nabla^2 f &= u \cdot \nabla^2 v + \nabla u \cdot (\nabla v)^T + \nabla v \cdot (\nabla u)^T + v \cdot \nabla^2 u. \end{aligned}$$

Table 1 gives the computational complexity for the $*$ operator.

Table 1. Computational complexity of the $*$ operator

Cost	+'s	*'s
Function	0	1
Gradient	n	$2n$
Hessian	$1.5n(n+1)$	$2n(n+1)$

The complexity of the other operators is similar, differing only in the constants. The storage complexity for the naive forward propagation of ∇f and $\nabla^2 f$ is proportional to $n^2/2$ times the storage required for computing f.

The time and storage complexity for the naive forward propagation contrasts sharply with the corresponding complexities for the univariate Taylor series whose complexities are a small multiple of (the number of nonzero elements of $\nabla^2 f$) $\times$ (the corresponding costs for f).

The alternative of reverse mode propagation of adjoint values [15] is attractive for computing gradients, but for the highly structured Hessians and higher-order derivatives, the forward mode is satisfactory.

4 Interpolating Derivatives from Taylor Series

The two central ideas of this paper are described in this section and the next. In this section, we compute second-order partial derivatives by interpolation from sets of three-term univariate Taylor series. The interpolation scheme for second-order partial derivatives is a special case of an interpolation scheme for arbitrarily high-order mixed partial derivatives, thus providing a natural growth path for any software tool based on this method. This treatment was inspired by a remark by Rall [26].

In the following section, we show how the preaccumulation of local derivatives complements the interpolation scheme, yielding an efficient method for computing second-order partial derivatives.

Let us consider the example that we have two independent variables x and y. We wish to compute the three distinct second-order partial derivatives of $f : \mathbf{R}^2 \mapsto \mathbf{R}$. The two-dimensional Taylor series tells us that

$$\begin{aligned} f(x_0 + h_x, y_0 + h_y) &:= f(h_x, h_y) \\ &= f(x_0, y_0) + (f_x, f_y) \begin{pmatrix} h_x \\ h_y \end{pmatrix} \\ &\quad + \tfrac{1}{2} \cdot (h_x, h_y) \begin{pmatrix} f_{xx} & f_{xy} \\ f_{xy} & f_{yy} \end{pmatrix} \begin{pmatrix} h_x \\ h_y \end{pmatrix} + O\left((h_x + h_y)^3\right), \end{aligned} \tag{1}$$

where all derivatives of f are evaluated at (x_0, y_0). For the special case $h_x = h_y = h$, we obtain

$$\begin{aligned} f(x_0 + h, y_0 + h) &:= f(h) \\ &= f(x_0, y_0) + (f_x + f_y) \cdot h + (f_{xx} + f_{yy} + 2f_{xy}) \cdot h^2/2 + O\left(h^3\right). \end{aligned} \tag{2}$$

Because of the uniqueness of the Taylor series, $f(h)$ is the same as the univariate Taylor expansion in the direction $u = x + y$:

$$f(h) = f(x_0, y_0) + hf_u + h^2 f_{uu}/2. \tag{3}$$

Equations (2) and (3) imply that

$$f_u = f_x + f_y \text{ and } f_{uu} = f_{xx} + 2f_{xy} + f_{yy}.$$

Hence

$$f_{xy} = 0.5(f_{uu} - (f_{xx} + f_{yy})), \tag{4}$$

and the three distinct second-order partial derivatives of f can be recovered from values computed as three univariate Taylor series.

In general, let $f : x \in \mathbf{R}^n \mapsto y \in \mathbf{R}$. We wish to compute the gradient ∇f and the Hessian $\nabla^2 f$. Assume that we know the locations of the p nonzero elements of $\nabla^2 f$. Let $l(i, j)$ map the index pairs (i, j) corresponding to nonzero Hessian elements into $1 \ldots p$. For each active variable w, we propagate the value $w := w(x + t \cdot u)|_{t=0}$ and the two p-vectors w' and w'' containing values for the first and second terms, respectively, of the Taylor series for $w(t)|_{t=0}$. Here, u represents the p directions corresponding to the nonzero elements of $\nabla^2 f$,

$$u = \left\{ \begin{array}{ll} e_i & \text{for } i = j \\ e_i + e_j & \text{for } i \neq j \end{array} \right\} \text{ for all } (i, j) \text{ s.t. } \frac{\partial^2 f}{\partial x_i \partial x_j} \neq 0,$$

and $'$ denotes differentiation with respect to t. The propagation of first and second derivatives parallelizes in the p-direction. That is, for both vectors w' and w'', there is no interaction between the p different elements. Hence, we omit indexing of the vectors w' and w'', except in an example where we use square braces $w'[i]$ to denote indexing across directions.

The Taylor series give us the first derivatives and the diagonal entries of the Hessian. The off-diagonal Hessian entries are recovered by interpolation as suggested by Equation (4):

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = 0.5\big(w''[l(i,j)] - (w''[l(i,i)] + w''[l(j,j)])\big) \tag{5}$$

In contrast to the naive forward propagation of Hessian matrices described in Section 3, the computation of Hessians as an ensemble of univariate Taylor series has the following advantages. The univariate Taylor series approach

- handles sparse Hessians by generating series only for nonzero entries,
- handles very large Hessians by generating elements in multiple sweeps,
- can generate arbitrary elements with little redundant computation,
- parallelizes and vectorizes,
- uses simple data structures – scalars and vectors, rather than symmetric matrices,
- is easier to understand when coding individual operators, and
- generalizes to higher derivatives (see Section 7).

In exchange, some computation is necessary to extract off-diagonal entries according to Equation (4), but that process is only done once at the end of the computation.

5 Preaccumulation of Derivatives

In the preceding section, we described an interpolation scheme for computing high-order derivatives from values obtained by propagating ensembles of univariate Taylor series. In this section, we discuss the second major contribution of this paper: the preaccumulation of derivatives. The preaccumulation of derivatives allows us to extend the results of the preceding section, where it was implicitly assumed that we were dealing with binary operations or unary elementary functions. Here, we cover complicated expressions in assignment statements, embedded function calls, or even basic blocks of code.

Let the variables y and z depend on a vector x of independent variables. The first and second derivatives ∇y, ∇z, $\nabla^2 y$, and $\nabla^2 z$ are available from earlier computations. If $w = f(y,z)$, the chain rule tells us that

$$\begin{aligned} \nabla w &= \frac{\partial w}{\partial y} \cdot \nabla y + \frac{\partial w}{\partial z} \cdot \nabla z, \text{ and} \\ \nabla^2 w &= \frac{\partial w}{\partial y} \cdot \nabla^2 y + \frac{\partial w}{\partial z} \cdot \nabla^2 z \\ &\quad + \frac{\partial^2 w}{\partial y^2} \cdot (\nabla y)^2 + 2\frac{\partial^2 w}{\partial y \partial z} \cdot \nabla y \cdot \nabla z + \frac{\partial^2 w}{\partial z^2} \cdot (\nabla z)^2. \end{aligned} \tag{6}$$

Hence, if we know the "local" derivatives $(\frac{\partial w}{\partial y}, \frac{\partial w}{\partial z})$ and $(\frac{\partial^2 w}{\partial y^2}, \frac{\partial^2 w}{\partial y \partial z}, \frac{\partial^2 w}{\partial z^2})$ of w with respect to z and y, we can easily compute ∇w and $\nabla^2 w$, the derivatives of w with respect to x. An example of Equation (6) is given in Equation (7) for the simple case $w = f(y,z) = y \cdot z$. Equation (6) for propagating Taylor series has the much simpler form given by Equation (8).

The idea is that the large "global" derivatives ∇w are propagated in the forward mode from one assignment statement to another, while the scalar "local" derivatives $(\frac{\partial w}{\partial y}, \frac{\partial w}{\partial z})$ are preaccumulated independently of the larger flow of control from one statement to the next. ADIFOR was the first

tool for automatic differentiation to use preaccumulation of local derivatives by applying the reverse mode at the statement level for the efficient computation of first derivatives [7,9]. The hierarchy of "local" and "global" derivatives extends to higher-order derivatives.

Consider the alternatives for computing first- and second-order derivatives of an active variable w that is given by an expression involving k active variables:

$$w = f(s_1, s_2, \ldots, s_k).$$

There are two alternatives for computing w': and w''

1. parse the expression for f, and propagate first- and second-order derivatives for each intermediate quantity; or

2. preaccumulate first- and second-order derivatives of f with respect to its local independent variables, and use the local derivatives to compute w' and w''.

All currently existing automatic differentiation software that can compute second-order derivatives uses the first alternative. We will see that these two alternatives lie at opposite ends of a spectrum. Each is optimal in operation counts for some expressions. We advocate a mixed strategy.

Next, we describe the two alternatives for computing w' and w'' in more detail.

5.1 Parse f and Propagate Derivatives for Each Intermediate

As is common in the automatic differentiation literature (see [15,24], for example), suppose that f has been parsed into a sequence of q unary and binary operations with operands either s_1, ..., s_k or else earlier intermediate results. Then in our case, the parsed code is annotated with code for generating the first- and second-order Taylor coefficients. As an example of a typical operation, suppose that $f = y \cdot z$:

$$\begin{aligned} f &= y \cdot z \\ f' &= y \cdot z' + y' \cdot z \\ f'' &= y \cdot z'' + 2y' \cdot z' + y'' \cdot z. \end{aligned} \tag{7}$$

(Comparison with the code in Section 3 for forward-mode Hessians shows why univariate Taylor series are preferred.) The important point to note about evaluating operators of this form is that there are $2q$ vector loops of length p.

The storage cost of propagating derivatives for each intermediate result is $2p$ times the storage requirements of f. The operation count for propagating derivatives for each intermediate result is about $5p$ times the operation count for evaluating f itself.

The special case in which f is a linear function of active variables $w = f(s_1, \ldots, s_k) = \sum a_i \cdot s_i$, where the a_i are constants, deserves special attention:

$$\begin{aligned} w &= \sum a_i \cdot s_i \\ w' &= \sum a_i \cdot {s_i}' \\ w'' &= \sum a_i \cdot {s_i}''. \end{aligned}$$

For assignment statements of this form, propagating derivatives by parsing into unary and binary operations is faster than the method of preaccumulation described in the following section.

5.2 Preaccumulate First- and Second-Order Derivatives

If $w = f(s_1, \ldots, s_k)$, let ∇f and $\nabla^2 f$ denote the "local" gradient and Hessian, respectively, of f with respect to $s_1, \ldots, s_k$. If we extend Equation (6) to complicated right-hand sides and to second derivatives, we get

$$\begin{aligned} w &= f(s_1, \ldots, s_k) \\ w' &= \sum_{i=1}^{k} (\nabla f)_i \cdot s_i{}' \\ &= \nabla f^T \cdot s' \\ w'' &= \sum_{i=1}^{k} \left[(\nabla f)_i \cdot s_i{}'' + s_i{}' \cdot \sum_{j=1}^{k} \left[(\nabla^2 f)_{i,j} \cdot s_j{}' \right] \right] \\ &= \nabla f^T \cdot s'' + s'^T \cdot \nabla^2 f \cdot s'. \end{aligned} \tag{8}$$

Equation (8) represents derivatives in each of the p directions, which may be computed in parallel.

The important point to note in Equation (8) is that there are only two vector loops of length p, independent of the numer of variables or operations on the right-hand side of the assignment statement. The local k-element gradient ∇f and the local k^2-element Hessian $\nabla^2 f$ can be computed in any manner. We may apply preaccumulation again to less complicated subfunctions, or we may use the forward mode, the reverse mode, a combination of the two, or analytic formulas, if they are easy to derive.

The storage cost of preaccumulating local derivatives is about $2p$ times the storage allocated by f, plus $k + k^2$, where k is the largest number of active variables appearing on the right-hand side of any assignment statement. Usually, $k << p$.

The operation count for preaccumulating local derivatives is about $(3k + k^2)p$, plus the cost of computing the local derivatives. In practice, many elements of the local Hessian $\nabla^2 f$ are zero, and computations can be omitted.

The special case in which f is a composition of nonlinear functions of a small number of active variables (eg. $w = \exp(s_1^{s_2})$) deserves special attention. For this example, the method based on parsing given in the previous section requires 4 vector loops of length p, while the preaccumulation method requires only 2 vector loops of length p. Further, the bodies of the loops, as well as the computation of the local derivatives, are relatively simple.

A successful tool for second-order partial derivatives satisfying the requirements given in Section 1 can be built using either the method of propagating derivatives for each intermediate result or the method of preaccumulating local derivatives. The examples given here show that neither method, by itself, is optimal. Hence, we prefer a mixed strategy. For example, suppose a code computes conductivity as an ugly nonlinear function of an average of values from 7 nearest neighbors in a three-dimensional grid such as

$$w = \exp(a(s_{down} + s_{up} + s_{east} + s_{west} + s_{south} + s_{north} + s_{center})).$$

The method based on parsing to unary and binary operations generates 6 intermediate results and 14 vector loops of length p. The method of preaccumulation generates only 2 vector loops, but the local Hessian has 49 nonzero entries which must be accumulated and then accessed inside the loop bodies. On the other hand, rewriting the expression as

$$\begin{aligned} w_1 &= a(s_{down} + s_{up} + s_{east} + s_{west} + s_{south} + s_{north} + s_{center}) \\ w &= \exp(w_1) \end{aligned}$$

yields generated code with only 4 vector loops. The local "Hessian" for the second assignment is a scalar instead of a 7×7 matrix. The advantage is much more pronounced if exp is replaced by a more complicated nonlinear function.

The characterization of expressions for which preaccumulation is best and those for which parsing to elementary operations is best is a question that is under continuing study.

6 Structured Evaluations

In this section, we describe how users of our approach are able to access second derivatives, to exploit sparsity, to compute subsets of a Hessian, and to compute Hessian $\times$ vector products.

6.1 Seeding to Exploit Sparsity

Let us assume that we have independent variables x_i, $i = 1, \ldots, n$. If we are interested in computing all nonzero entries in the Hessian, then we set $x_i{}'[l] = x_j{}'[l] = 1$ for $l = l(i,j)$, the index function discussed in Section 4. The other first- and all second-directional derivatives $x_i{}''[l]$ are set to zero, since clearly

$$\frac{\partial^2 x_i}{\partial x_j \partial x_k} = 0 \text{ for all } i, j, k.$$

For example, suppose that $n = 5$ and that $\nabla^2 f$ has the sparsity structure

$$\nabla^2 f = \begin{array}{ccccc} \times & & & & \\ \times & \times & & & \\ \times & 0 & \times & & \\ \times & \times & 0 & \times & \\ \times & 0 & 0 & \times & \times \end{array}$$

Then, the first-derivative Taylor series are initialized as follows. The order does not matter. We illustrate using column-major order for the nonzero elements of the lower triangle of the Hessian matrix.

Direction Index l	Index of Nonzero Hessian Element	$x_1{}'$	$x_2{}'$	$x_3{}'$	$x_4{}'$	$x_5{}'$
1	(1,1)	1	0	0	0	0
2	(2,1)	1	1	0	0	0
3	(3,1)	1	0	1	0	0
4	(4,1)	1	0	0	1	0
5	(5,1)	1	0	0	0	1
6	(2,2)	0	1	0	0	0
7	(4,2)	0	1	0	1	0
8	(3,3)	0	0	1	0	0
9	(4,4)	0	0	0	1	0
10	(5,4)	0	0	0	1	1
11	(5,5)	0	0	0	0	1

The number of nonzeros in the Taylor coefficients for x_i is exactly the number of nonzeros in column i of the lower triangle or in row i of the upper triangle of the Hessian matrix. In this example, the number of distinct nonzero elements in the Hessian matrix is $p = 11$. All second-order derivatives are initially set to 0. That is, $x_i{}''[i] = 0$.

On return from the subroutine that has been generated for computing first- and second-order derivatives, f and the p-vectors f' and f'' contain the first three terms of the univariate Taylor series in the p directions corresponding to the nonzero entries of the Hessian. The elements of the Hessian themselves are reconstructed from Equation (5).

6.2 Computing Slices

By "slices" we mean a structured subset of the entire Hessian matrix. For example, one might need the diagonal elements, a single row, a set of rows (or columns), or a square submatrix. The use of univariate Taylor series makes it easy to compute only selected elements of the Hessian matrix. Equation (4) shows that in order to extract the element $(\nabla^2 f)_{i,j}$, one need only to propagate univariate Taylor series in the three directions x_i, x_j, and $u = x_i + x_j$.

6.3 Computing Hessian $\times$ Vector

The components of the vector $\nabla^2 f \times v$ are not Taylor coefficients themselves. However, they can be readily propagated by using univariate Taylor series. The key observation is that since

$$f(x + t\,v) = f(x) + t\,(\nabla f)^T v + \tfrac{1}{2} t^2 v^T \nabla^2 f v,$$

seeding (without loss of generality) the first component of $x_i{}' = v_i, i = 1, \ldots n$, yields

$$\begin{aligned} f' &= (\nabla f)^T v, \text{ and} \\ f'' &= v^T \nabla^2 f v. \end{aligned}$$

This is a generalization of the seeding that we have done so far, where $v = e_i$ or $v = e_i + e_j$. In particular, for $z = v + e_j$, we obtain

$$\begin{aligned} f' &= (\nabla f)^T w + \frac{\partial f}{\partial x_j}, \text{ and} \\ f'' &= v^T \nabla^2 f v + \frac{\partial^2 f}{\partial {x_j}^2} + 2 e_j^T \nabla^2 f v. \end{aligned} \tag{9}$$

Equation (9) contains the information that we are looking for, namely, $e_j^T \nabla^2 f v$, the jth component of the Hessian-vector product $\nabla^2 f v$. Hence, after computing a Taylor series each for $u = v, u = e_j, u = v + e_j$ and indexing the coefficients in that order, we can recover $e_j^T \nabla^2 f v$ as

$$e_j^T \nabla^2 f v = 0.5\,(f''[n + j + 1] - (f''[j + 1] - f''[1]))\,. \tag{10}$$

The subscript indicates the direction with respect to which the Taylor series was initialized. Hence, for n variables, we need to compute Taylor series for

$$v, e_1, \ldots, e_n, v + e_1, \ldots, v + e_n.$$

For maximum efficiency, two Taylor series evaluations can be saved. Let us assume w.l.o.g. that

$$e_1 = v - \sum_{i=2}^{n} \alpha_i e_i.$$

Then

$$e_1^T \nabla^2 f v = v \nabla^2 f v - \sum_{i=2}^{n} \alpha_i e_i^T \nabla^2 f v.$$

Exploiting this fact, the computation of a Hessian-vector product requires the evaluation of $2n - 1$ Taylor series, a slight advantage in complexity over divided differences. This complexity compares very favorably with the $n(n + 1)/2$ Taylor series required to compute the full, dense Hessian.

A generalization of this scheme allows us to compute the product of the Hessian with a rectangular matrix Z.

7 Higher-Order Derivatives

Our primary interest in this paper has been in second-order derivatives, but the techniques of interpolation from univariate Taylor series and of preaccumulation of local derivative values can be generalized to higher-order derivatives. In this section, we survey the propagation of Taylor series of a single variable which forms the basic building block we then apply to the computation of high-order mixed partial derivatives.

7.1 Univariate Taylor Series

We briefly survey the propagation of Taylor series of a single independent variable $w = f(t)$. A discussion of univariate Taylor series demonstrates that in this restricted context, the computation of derivatives of arbitrarily high order is well understood (see [23] or [24], for example). The interpolation scheme described below shows how arbitrary partial derivatives of a function of many independent variables can be efficiently computed from the much simpler univariate Taylor series case.

With a few exceptions [6,29], the automatic evaluation of higher derivatives has been restricted to cases with one independent variable. The k-th derivatives are usually scaled by $1/k!$ to store Taylor coefficients. Scaling by $1/k!$ reduces operation counts and reduces the risks of overflow. If the client algorithm actually wants derivatives, the Taylor coefficients can be rescaled at the end of the computation. The forward propagation of truncated Taylor series with m terms requires $O(m)$ arithmetic operations and memory accesses for each addition or subtraction operation and $(m+1)(m+2)/2$ arithmetic operations for each multiplication, division, or special function evaluation [23, 24]. We may use the factor of $(m+1)(m+2)/2$ as a measure of the cost ratio between the forward propagation of m-term series and as a means for evaluating the underlying function.

The use of Fourier transforms and other fast convolution methods yields the product of two polynomials with an asymptotic complexity $O(m \log m)$. One could determine for each value of m the best scheme for each computing environment, exploiting the obvious vectorizability. In this paper, we will continue to use $O(m^2)$, rather than the theoretical $O(m \log m)$ as the complexity of multiplication, and consequently as the estimate for the cost ratio. The use of the larger asymptotic bound is justified because we are primarily concerned with relatively low-order derivatives (Hessians are second order) and because the generation of Taylor series one term at a time for ODEs is not known to yield to Fourier transform techniques.

Figure 1 illustrates the forward propagation of Taylor series of a single independent variable, where we store the Taylor coefficients $U_i := u^{(i)}(t)/i!$. We give only the code corresponding to the simple assignment statement $w = -y/(z * z)$ using a parsing of the expression into a sequence of elementary unary and binary operations.

$$
\begin{aligned}
S_0 &= -Y_0 \\
T_0 &= Z_0 * Z_0 \\
W_0 &= S_0/T_0 \\
S_i &= -Y_i \quad \text{for } i = 1, \ldots, m \\
T_i &= \sum_{j=0}^{i} Z_j * Z_{i-j} \quad \text{for } i = 1, \ldots, m \\
W_i &= \left(S_i - \sum_{j=1}^{i} W_{i-j} * T_j \right) / T_0 \quad \text{for } i = 1, \ldots, m
\end{aligned}
$$

Figure 1. Forward propagation of univariate Taylor coefficients

The appropriate recurrence relations for each operation or elementary function are given by Moore [23] or by Rall [24]. Multiplication of two polynomials of degree m (or, equivalently, the convolution of their coefficients) is the central workhorse of all higher-derivative calculations [11], including the multivariate case and the reverse mode. Hence, it is important for the performance of automatic differentiation that the core routine for multiplying two polynomials be implemented with maximal efficiency for each m, much as dense linear algebra computations rely on efficient implementations of the BLAS [2,1].

In the next section, we show how the relatively simple recurrence relations for univariate Taylor coefficients as illustrated in Figure 1 can be used to compute mixed partial derivatives of any order for multivariate functions of interest in such applications as optimization.

7.2 General Interpolation Scheme

Equation (5) is a special case of a general-purpose, efficient scheme for computing high-order partial derivatives from values obtained by propagating univariate Taylor series, which we describe here. The general scheme may be applied in the future to extend the capabilities of the source transformation tool to support higher derivatives.

A polynomial P of degree m in n variables is uniquely characterized by its values at the grid-points

$$\mathbf{i} = (i_1, i_1, \ldots, i_n) \quad \text{with} \quad 0 \le i_j \quad \text{and} \quad \sum_{j=1}^{n} i_j \le m.$$

Given the values $P(\mathbf{i})$ at these grid points, the value of P at any other point $x \in \mathbf{R}^n$ can be obtained by the simple Lagrangian interpolation formula:

$$P(x) = \sum_{0 \le \mathbf{i}, e^T \mathbf{i} \le m} L_i(x)\, P(\mathbf{i}) / L_i(\mathbf{i}), \tag{11}$$

where e denotes the vector of 1's and

$$L_i(x) = \prod_{j > e^T \mathbf{i}}^{j \le m} (e^T x - j) \cdot \prod_{j=0}^{j < i_1} (x_1 - j) \cdots \prod_{j=0}^{j < i_j} (x_j - j) \cdots \prod_{j=0}^{j < i_n} (x_{\hat{n}} - j)$$

is the unique polynomial (up to scaling) of degree m that vanishes at all gridpoints except $\mathbf{i}$. In practice, this polynomial should be converted into a more efficient representation.

Now, we sketch how the grid values $P(\mathbf{i})$ of a Taylor polynomial

$$P(x) = f(x) + \mathcal{O}\left(\|x\|^{m+1}\right)$$

can be obtained by using a bundle of univariate Taylor expansions.

A polynomial of degree m in n variables contains

$$b(n+m-1, m) := \binom{n+m-1}{m}$$

terms of the highest order (i.e., m). The corresponding variable exponents form multi-indexes $\mathbf{k}$ with $e^T \mathbf{k} = m$. These n-vectors may be interpreted as grid points in the domain of f (see Figure 2), and we can calculate the univariate Taylor polynomials

$$P_k(t) = f(t \cdot \mathbf{k}) + \mathcal{O}(t^{m+1})$$

by automatically differentiating the program that defines f at $t = 0$. The functions P_k represent restrictions of P to lines running through the origin. For each $\mathbf{i}$,

$$P(t \cdot \mathbf{k}) = P_k(t) \text{ with } t \in \mathbf{R}.$$

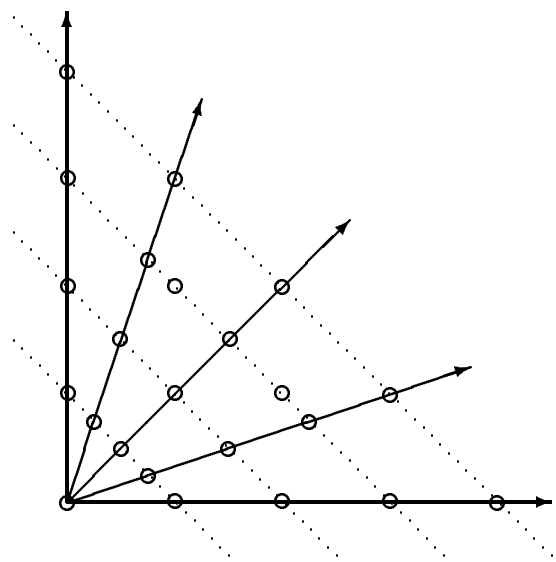

Figure 2: Interpolation grid with lines and hyperplanes for $m = 4$ and $n = 2$

The lines $t \cdot \mathbf{k}$ intersect each of the m hyperplanes

$$H_j := \{x \in \mathbf{R}^n : e^T x = j\} \text{ for } j = 1, 2, \ldots, m$$

in $b(n + m - 1, m)$ points. Some of these intersection points correspond to the cubic grid points $\mathbf{i} \subset \mathbf{R}^n$ considered above, but many of them do not. To obtain the value of P at a general grid point $\mathbf{i}$, we can interpolate P on the hyperplane H_j with $j = e^T \mathbf{i}$. On this $(n-1)$-dimensional subspace, the intersections with the lines $t \cdot \mathbf{k}$ form a regular grid of $b(n - 1 + m, m)$ points at which the values of P are known. The number $b(n - 1 + m, m)$ corresponds exactly to the degrees of freedom of an m-th degree polynomial with $n - 1$ variables, the restriction of P to H_j, in this case. With respect to suitable internal coordinates, the intersection points form a cubic grid. The Lagrangian formula (Equation (11)) can be applied to obtain the values $P(\mathbf{i})$ for all grid points $\mathbf{i}$ with $e^T \mathbf{i} \leq m$. Subsequently, we may use the values $P(\mathbf{i})$ to interpolate $P(x)$ for arbitrary $x \in \mathbf{R}^n$. This is an argument for feasibility; it may be possible to replace this two-stage interpolation procedure with a more efficient scheme. Further possible savings do not affect the complexity.

It was shown above that we must propagate $b(n+m-1, m)$ univariate series for general n and m. Consequently, the complexity ratio for the interpolation approach versus the conventional approach is

$$\frac{m * n}{n + m} \leq \min\{n, m\}$$

in terms of storage and

$$q(n, m) = \frac{1}{2}(m + 2)(m + 1) \cdot \frac{(n + m - 1)(n + m - 2) \ldots n}{(2n + m)(2n + m - 1) \cdots (2n + 1)} \leq \frac{3}{2}$$

in terms of arithmetic operations per convolution. If there is not enough storage to compute all the univariate expansions in one pass, they can be calculated in groups over several forward sweeps.

The interpolation method promises great savings in terms of arithmetic operations because $q(n, m)$ is monotonically decreasing in m and equals roughly $m^2/2^{m+1}$ when n is significantly larger than m. Thus, we have an exponential complexity reduction in terms of m.

8 Conclusions and Future Research Directions

ADIFOR (Automatic Differentiation In FORtran) [7,8,9] is a source translation tool implemented by using the data abstractions and program analysis capabilities of the ParaScope Parallel Programming Environment [10]. ADIFOR accepts arbitrary Fortran 77 code defining the computation of a function and writes portable Fortran 77 code for the computation of its first derivatives. Work is in progress to extend ADIFOR to provide second- and higher-order derivatives as described in this paper.

Acknowledgments

We gratefully acknowledge the contributions of John Dennis, Alan Carle, and Karen Williamson of Rice University to the ADIFOR project and to our understanding of the use of second derivatives in optimization algorithms.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, Penn., 1992. To appear.

[2] Edward Anderson, Zhaojun Bai, Christian Bischof, James Demmel, Jack Dongarra, Jeremy DuCroz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In Joanne Martin, editor, *SUPERCOMPUTING '90*, pages 2–10, New York, 1990. ACM Press.

[3] Brett Averick, Richard G. Carter, and Jorge J. Moré. The MINPACK–2 test problem collection (preliminary version). Technical Memorandum ANL/MCS–TM–150, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., May 1991.

[4] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[5] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova. Programs for automatic differentiation for the machine BESM. Technical Report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR, 1959. (In Russian).

[6] Martin Berz. Forward algorithms for high orders and many variables with application to beam physics. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 147–156. SIAM, Philadelphia, Penn., 1991.

[7] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. Generating derivative codes from Fortran programs. *Scientific Computing*, to appear. ADIFOR Working Note # 1. Also appeared as Preprint MCS–P263–0991, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1991, and as Technical Report 91185, Center for Research in Parallel Computation, Rice University, Houston, Tex., 1991.

[8] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Memorandum ANL/MCS–TM–159, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., January 1992. ADIFOR Working Note # 3.

[9] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Memorandum ANL/MCS–TM–158, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., October 1991. ADIFOR Working Note # 2.

[10] D. Callahan, K. Cooper, Robert T. Hood, Ken Kennedy, and Linda M. Torczon. ParaScope: A parallel programming environment. *International Journal of Supercomputer Applications*, 2(4), December 1988.

[11] Y. F. Chang. The conduction-diffusion theory of semiconductor junctions. *Journal of Applied Physics*, 38(2):534–544, 1967.

[12] John Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, N.J., 1983.

[13] John E. Dennis and Robert B. Schnabel. A view of unconstrained optimization. In G. L. Nemhauser, editor, *Handbooks in Operations Research and Mathematical Software*, volume 1, pages 1–72. Elsevier, 1989.

[14] G. Di Pillo and L. Grippo. An exact penalty method with global convergence properties for nonlinear programming problems. *Math. Programming*, 36:1–18, 1986.

[15] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[16] Andreas Griewank. The chain rule revisited in scientific computing. *SIAM News*, 24, May & July 1991. No. 3, p. 20 & No. 4, p. 8.

[17] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, Penn., 1991.

[18] Andreas Griewank, David Juedes, Jay Srinivasan, and Charles Tyner. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, to appear. Also appeared as Preprint MCS–P180–1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1990.

[19] Willi Hock and Klaus Schittkowski. *Test Examples for Nonlinear Programming Codes*, volume 187 of *Lecture Notes in Economics and Mathematical Systems.* Springer-Verlag, Berlin, 1981.

[20] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–329. SIAM, Philadelphia, Penn., 1991.

[21] Harriet Kagiwada, Robert Kalaba, Nima Rasakhoo, and Spingarn Karl. *Numerical Derivatives and Nonlinear Analysis*, volume 31 of *Mathematical Concepts and Methods in Science and Engineering.* Plenum Press, Inc., New York, 1985.

[22] Leo Michelotti. MXYZPTLK: A C++ hacker's implementation of automatic differentiation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 218–227. SIAM, Philadelphia, Penn., 1991.

[23] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1966.

[24] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1981.

[25] Louis B. Rall. Differentiation in Pascal-SC: Type GRADIENT. *ACM Trans. Math. Software*, 10(2):161–184, June 1984.

[26] Louis B. Rall. Point and interval differentiation arithmetics. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 17–24. SIAM, Philadelphia, Penn., 1991.

[27] Greg R. Shubin. Obtaining "cheap" optimization gradients from computational aerodynamics codes. Applied Mathematics and Statistics Technical Report AMS–TR–164, Boeing Computer Services, June 1991.

[28] Greg R. Shubin and P. D. Frank. A comparison of two closely-related approaches to aerodynamic design optimization. In G. S. Dulikravich, editor, *Proceedings of the Third International Conference on Inverse Design Concepts and Optimization in Engineering Sciences, Washington, D.C., Oct. 23–25, 1991*, 1991.

[29] Leigh Tesfatsion. Automatic evaluation of higher-order partial derivatives for nonlocal sensitivity analysis. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 157–165. SIAM, Philadelphia, Penn., 1991.

[30] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.

[31] Karen A. Williamson. *A Robust Trust Region Algorithm for Nonlinear Programming.* PhD thesis, Rice University, Department of Mathematical Sciences, 1990. Technical Report TR90-22.