More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability†

David M. Gay

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
dmg@bell-labs.com

ABSTRACT

We describe computational experience with automatic differentiation of mathematical programming problems expressed in the modeling language AMPL. Nonlinear expressions are translated to loop-free code, which makes it easy to compute gradients and Jacobians by backward automatic differentiation. The nonlinear expressions may be interpreted or, to gain some evaluation speed at the cost of increased preparation time, converted to Fortran or C. We have extended the interpretive scheme to evaluate Hessian (of Lagrangian) times vector. Detecting partially separable structure (sums of terms, each depending, perhaps after a linear transformation, on only a few variables) is of independent interest, as some solvers exploit this structure. It can be detected automatically by suitable ‘‘tree walks’’. Exploiting this structure permits an AD computation of the entire Hessian matrix by accumulating Hessian times vector computations for each term, and can lead to a much faster computation of the Hessian than by computing the whole Hessian times each unit vector.

1. Introduction

Consider the minimization problem

(1a) minimize $f(x)$

(1b) subject to $c(x) = \mathbf{0}$,

where $f: \mathbb{R}^n \to \mathbb{R}$ and $c: \mathbb{R}^n \to \mathbb{R}^m$ are differentiable. If the constraint Jacobian matrix $c'(x^*)$ has full row-rank, the Lagrange multiplier rule gives a necessary condition for x^* to solve (1):

$$\nabla f(x^*) = \nabla c(x^*) \lambda \tag{2}$$

†This paper is written as a contribution to the proceedings of the Second International Workshop on Computational Differentiation, held in February, 1996 in Santa Fe, New Mexico.

for some $\lambda \in \mathbb{R}^m$ (the Lagrange multipliers), where $\nabla f(x) = f'(x)^{\mathrm{T}} \in \mathbb{R}^n$ and $\nabla c = c'(x)^{\mathrm{T}} \in \mathbb{R}^{n \times m}$ denote the gradient of f and transposed Jacobian matrix of c. Many algorithms for solving (1) are motivated by application of Newton's method to the necessary conditions (1b) and (2). A central ingredient in such algorithms is an approximation to the Hessian of the Lagrangian function,

$$H = \nabla^2 f(x) - \sum_{i=1}^{m} \lambda_i \nabla^2 c_i(x). \tag{3}$$

There is much current interest in algorithms that require H to be provided. Thus it is of interest to ask how efficiently we can compute (3) from a suitable symbolic representation of f and c. This paper discusses computing (3) by automatic differentiation (AD) and gives some computational experience with problems expressed symbolically in the AMPL modeling language [11, 12]. For simplicity, the remainder of this paper ignores constraints and just talks about computing $\nabla^2 f$. However, the implementation discussed below is designed to compute (3), which is also relevant when some of the constraints are changed to inequality constraints.

Use of AD in Hessian computations is not new; for example, Dixon [8] and Christianson [6] discuss using AD in this context. What is new in my work is automatic detection of partially separable structure and use of this structure to give faster Hessian computations.

The rest of the paper is organized as follows. Section 2 briefly reviews AMPL and AD computations of gradients of AMPL expressions. Section 3 discusses Hessian-vector computations. Section 4 describes partially separable structure and discusses automatically detecting it. A discussion of computing sparse Hessians appears in §5. Section 6 presents some computational experience, and the final section offers concluding remarks.

2. AMPL and AD for Gradients

AMPL is a language and modeling environment for expressing and working with optimization problems that involve finitely many variables, with objectives and constraints described by algebraic expressions. AMPL encourages separate descriptions of a ''model'' for a class of problems and the data for a particular problem — an instance of the model. The AMPL processor does not itself solve an instance, but rather writes a description of the problem to a ''`.nl`'' file and invokes a separate solver. The solver obtains loop-free expression graphs for the problem's nonlinear expressions by calling interface routines that read the `.nl` file. These routines walk expression graphs for the objective and constraints and set up data structures for derivative computations. The AMPL home page

`http://www.ampl.com/ampl/`

tells how to get source for the interface routines and gives pointers to other information related to AMPL.

The AMPL/solver interface routines provide ‘‘interpreted’’ evaluations of objectives and constraint bodies, as well as backward AD computations of their first derivatives. A brief overview of AMPL and a discussion of these first-derivative computations appear in [15]. In short, an operation is represented by a structure that provides storage for the partial derivatives of the operation and has pointers to its operands and to a function that carries out the operation. The AD ‘‘backward sweep’’ for computing ‘‘adjoints’’, i.e., the partial derivatives of an objective or constraint body with respect to the results of the operations used to compute it, is carried out by the C loop

```
do *d->a += *d->b * *d->c;
    while(d = d->next);
```

This is just a sequence of multiplications and additions of the form `a := a + b*c`.

‘‘Compiled’’ evaluations are also possible. To arrange for them, one runs a program called *nlc*, which turns a `.nl` file into C or Fortran for computing the objective function, constraint bodies, and their first derivatives. Because of the compilation and linking times they entail, compiled evaluations only save time for expressions that are to be evaluated a great many times. A brief discussion of *nlc* appears in [17].

The interpreted evaluations are fast enough to be useful in many situations. Some comparisons with hand-coded Fortran evaluations appear in [15], and more appear below in §6.

3. Hessian Times Vector Computations

Many iterative methods for solving linear equations require only matrix-vector products. In particular, we can compute the search direction in Newton’s method for minimizing $f:\mathbb{R}^n \to \mathbb{R}$ by computing Hessian-vector products, $\nabla^2 f(x)\,v$ for suitable vectors v. We can also compute the entire Hessian by computing $\nabla^2 H(x)\,e^i$ for each of the n standard unit vectors e^i. Thus it is of interest to consider how to compute Hessian-vector products.

Computing $\nabla^2 f(x)\,v$ by backward AD is conceptually straightforward: we just apply backward AD to compute the gradient of $v^{\mathrm{T}}\nabla f(x)$, with v constant. I find it slightly easier to think about an alternative interpretation of this calculation that Christianson [6] has described: consider the scalar function $\psi(\tau) = f(x + \tau v)$; compute $\psi(0)$ and $\psi'(0)$ by forward AD, then apply backward AD to compute the derivative with respect to x of $\psi'(0)$. Both approaches result in the same calculation [6]. Figure 1 depicts the situation at a typical binary operation, $o(\ell, r)$, during the computation of ψ and ψ'. The dashed arrows represent derivatives with respect to τ; only they depend on the direction v. For each new v, it is necessary to make a ‘‘forward’’ pass to recompute the results represented by the dashed arrows. (If we reversed the dashed arrows, we would obtain a figure that shows the forward computation of $o(\ell, r)$ and the reverse computation of the adjoints $\partial f/\partial o$, $\partial f/\partial \ell$, and $\partial f/\partial r$. Specifically, the ‘‘+’’ node would accumulate the adjoint $\partial f/\partial o$, and the ‘‘$*$’’ nodes would represent multiplications whose results are added to the adjoints $\partial f/\partial \ell$ and $\partial f/\partial r$.)

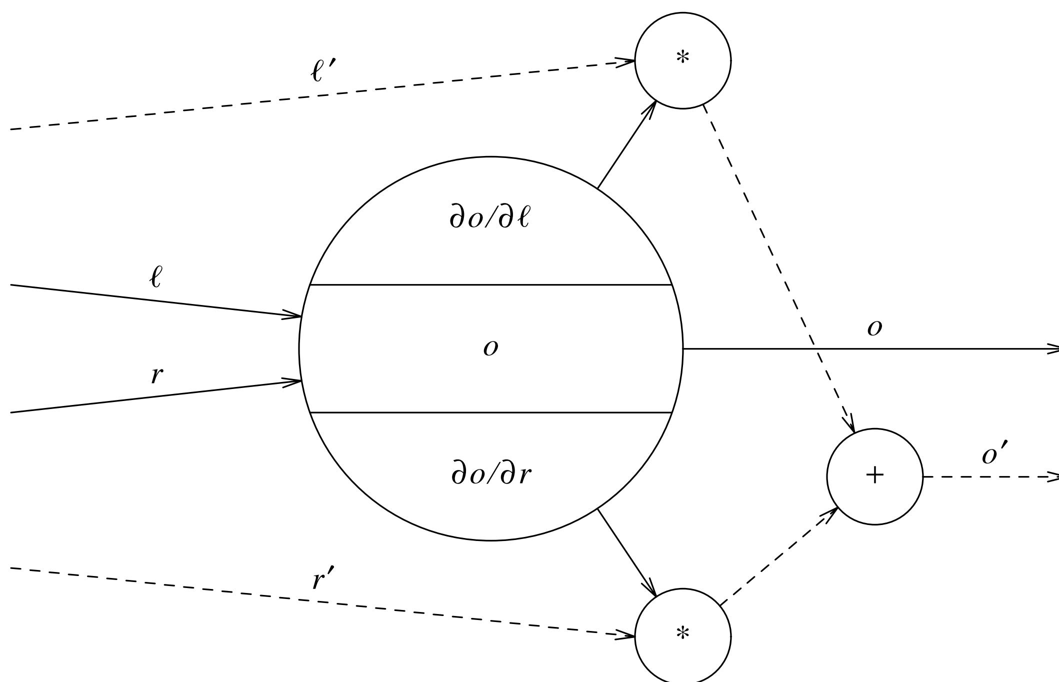

Figure 1. *A binary operation,* $o(\ell, r)$.

Figure 1 shows that six operations are associated with $o(\ell, r)$ in the computation of $\psi'(0)$: $o(\ell, r)$ itself, $\partial o/\partial \ell$, $\partial o/\partial r$, the two ''$*$'' nodes, and the ''+'' node. The partial derivatives of these operations (including the second partials of o) are involved in the backward AD computation of the derivative of $\psi'(0)$. Because a single operation o contributes several operations to this backward AD computation, and because there are many special cases in which the general computation in Figure 1 can be simplified, it seems reasonable to switch on the type of operation and do all the operations at once that are associated with $o(\ell, r)$ in the backward sweep. Figure 2 shows the C structure used in the computations reported below to represent a general binary operation $o(\ell, r)$. It contains fields for the first and second partials of o, including `dO` $= \partial o/\partial \tau$ in the computation of $\psi'(0)$, and for two adjoints: `aO` $= \partial \psi'/\partial o$ and `adO` $= \partial \psi'/\partial(\partial o/\partial \tau)$. The structure also contains forward and backward pointers to permit visiting operations in a suitable order, pointers `L` and `R` to the operands, an adjoint index used in setting up the data structures, and a pointer to the function that computes $o(\ell, r)$ and its partials: in the current implementation, o and its partials are computed at the same time. Since some algorithms compute function values at several points before they require any new gradient information, it might sometimes be worthwhile to compute the partials separately, but this detail remains for future research.

Figure 3 shows fragments of the C code used in the computing reported below. They carry out operations in the backward AD sweep for computing the gradient of $\psi'(0)$. The first fragment (case `Hv_binaryLR`) shows calculations for a general binary operation $o(\ell, r)$. In many common operations, such as addition and multiplication, some partials always vanish and others have special values, such as 1 or -1. For

```
 struct
expr2 {
        efunc2 *op;
        int a;      /* adjoint index (in gradient comp.) */
        expr2 *fwd;
        expr2 *bak;
        uir  dO;    /* deriv of op w.r.t. t in x + t*v */
        real aO;    /* adjoint (in Hv comp.) of op */
        real adO;   /* adjoint (in Hv comp.) of dO */
        real dL;    /* deriv of op w.r.t. L */
        ei2 L, R;   /* left and right operands */
        real dR;    /* deriv of op w.r.t. R */
        real dL2;   /* second partial w.r.t. L, L */
        real dLR;   /* second partial w.r.t. L, R */
        real dR2;   /* second partial w.r.t. R, R */
        };
```

Figure 2. *C structure representing* $o(\ell, r)$.

example, the second fragment in Figure 3 shows code for the operation of adding a constant to the left operand.

```
case Hv_binaryLR:       /* general binary op */
        e1 = e->L.e;
        e2 = e->R.e;
        adO = e->adO;
        t1 = adO * e1->dO.r;
        t2 = adO * e2->dO.r;
        e1->aO  += e->aO*e->dL + t1*e->dL2 + t2*e->dLR;
        e2->aO  += e->aO*e->dR + t1*e->dLR + t2*e->dR2;
        e1->adO += adO * e->dL;
        e2->adO += adO * e->dR;
        break;
case Hv_plusL:          /* Left + constant */
        e1 = e->L.e;
        e1->aO += e->aO;
        e1->adO += e->adO;
        break;
```

Figure 3. *C fragments for backward AD of* $\psi'(0)$.

4. Detecting and Using Partial Separability

Many optimization problems involve objectives (and constraint bodies) that have the form

$$f(x) = \sum_{i=1}^{q} f_i(U_i x), \tag{4}$$

in which U_i is an $m_i \times n$ matrix with a small number m_i of rows. For instance, the protein-folding problem discussed later involves minimizing the energy of a configuration of atoms. When the locations of the atoms are expressed in Cartesian

coordinates, only the differences of the coordinates of pairs of atoms appear in the various objective terms, so a row of U_i contains all zeros except for one $+1$ and one -1. In this class of problems, $m_i = 3$, 6, or 9.

The gradient and Hessian of (4) exhibit useful structure:

$$\nabla f(x) = \sum_{i=1}^{q} U_i^{\mathrm{T}} \nabla f_i(U_i x) \tag{5}$$

and

$$\nabla^2 f(x) = \sum_{i=1}^{q} U_i^{\mathrm{T}} \nabla^2 f_i(U_i x) U_i . \tag{6}$$

Griewank and Toint [19, 20] originally pointed out the structure in $(4-6)$ and proposed using secant updates to approximate each $\nabla^2 f_i$ separately. Since the Hessians $\nabla^2 f_i$ in (6) are $m_i \times m_i$ matrices, secant updates often give a good approximation to $\nabla^2 f_i$ in only $m_i << n$ steps, resulting in an excellent overall Hessian approximation in $\max_i m_i$ steps.

Use of partially separable structure can make explicit Hessian computations by Hessian-vector products relatively fast — much faster than computing n products $\nabla^2 f(x) e^i$. For $m_i < n$, we can compute the contribution $U_i^{\mathrm{T}} \nabla^2 f_i(U_i x) U_i$ to $\nabla^2 f$ by first using m_i Hessian-vector products to compute $\nabla^2 f_i$, then forming

$$U_i^{\mathrm{T}} \nabla^2 f_i(U_i x) U_i = \sum_{jk} (\nabla^2 f_i)_{jk} (U_i^{\mathrm{T}} e^j)(e^{k\mathrm{T}} U_i) , \tag{7}$$

which is a sum of outer products.

Automatic detection of partially separable structure is appealing. It should make life easier for users than, say, having to state this structure explicitly in the input format SIF associated with LANCELOT [7], a solver that exploits partially separable structure.

The computations described below involve automatic detection of partially separable structure, starting with the expression graphs written by the AMPL processor. We find f_i and U_i by walking the graph for f. In doing so, we accumulate linear terms (rows of a U_i) as long as possible — until they appear in a nonlinear operation. Having found a linear term, we put it into a canonical form, sorting its coefficients and scaling them so the largest coefficient is 1, and enter this canonical form into a hash table, so we can find duplicate appearances of U_i in f_i. Similarly, once we have found a term f_i, we put U_i into a canonical form and hash it, so we can easily tell if a subsequent f_j has $U_j = U_i$, in which case we can merge f_j into f_i. The process involves walking the expressions for f (and for any constraints) once to determine the partially separable structure, and a second time to arrange for derivative and Hessian-vector computations.

For simplicity, we find U_i even when $m_i \geq n_i$, where n_i is the number of variables involved in f_i, i.e., $n_i = | \{ j\colon 1 \leq j \leq n,\ U_i e^j \neq \mathbf{0} \} |$. In this case, rather than using (7) to compute the contribution to the overall Hessian, it is better to compute n_i Hessian-vector products

(8) $$(U_i^{\mathrm{T}} \nabla^2 f_i U_i) e^j, \quad 1 \le j \le n_i .$$

Sometimes there are many ways to express a function in partially separable form. The above process detects the ‘‘finest’’ partially separable structure one can find without applying the distributive law to nonlinear expressions. This does not necessarily lead to the most efficient computations. For example, as Andreas Griewank has pointed out [private communication, 1996], an expensive common expression might be involved in two or more of the f_i, and dealing separately with these f_i may take more work than handling them jointly. In AMPL models, ‘‘defined variables’’ give rise to common expressions. The implementation discussed below checks for expensive defined variables that appear in more than one f_i, and if doing so appears worthwhile, arranges to compute the Hessians of these defined variables before dealing with the f_i and to use the defined-variable Hessians in the computations for the f_i. How important this optimization is in practice remains to be seen. It does not play a role in the testing reported below.

5. Sparse Hessians

Problems involving a large number n of variables often have sparse Hessians, and some solvers that require explicit Hessians will need them in sparse form. By using a column-dispatch scheme much like the one described in [16], we can compute a Hessian column by column. This involves structures representing the current state of contributions from (7) and from (8); after processing a contribution to the current column, we dispatch its structure to the next column to which it will contribute.

It is helpful to determine the sparsity pattern of the Hessian in advance, in part because some solvers will need to know the sparsity pattern before they request any Hessian evaluations. Although this pattern is contained in the sparsity pattern of

(9) $$\sum_{i=1}^{q} U_i^{\mathrm{T}} e e^{\mathrm{T}} U_i ,$$

where e is a vector of ones, this may be an overestimate. For example, consider the AMPL model

```
var x{1..6};
var y{i in 0..2} = sum{j in 1..2} x[2*i+j];
minimize silly:  sum{i in 1..6} (x[i] - i)^2
                + if (x[1] > 1) then y[0]*y[1] else y[1]*y[2];
```

For this problem, $q = 7$, $U_i = e^{i\mathrm{T}}$ for $1 \le i \le 6$, and

$$U_7 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix},$$

so (9) would give a completely dense ‘‘sparsity’’ pattern, whereas the true pattern is

$$\begin{bmatrix} * & & * & * & & \\ & * & * & * & & \\ * & * & * & & * & * \\ * & * & & * & * & * \\ & & * & * & * & \\ & & * & * & & * \end{bmatrix}.$$

A straightforward way to compute the sparsity structure is to carry out a variation of the column-wise Hessian computation in which all partial derivatives that could be nonzero are treated as having the value 1. By summing contributions into a vector that is initially all zeros and noting the cells that become nonzero (a common manipulation in sparse-matrix computations), we can determine the ''true'' sparsity structure of the Hessian.

On small problems, it is faster to sum each $\nabla^2 f_i$ directly into a dense upper triangle, bypassing the overhead of column dispatching.

Griewank and Toint point out in [19] that sparsity implies partially separable structure. The implied representations are ones in which (9) would give the correct sparsity structure. Finding such representations involves applying the distributive law to nonlinear expressions, which could change the work of computing some functions and gradients from $O(k)$ to $O(k^2)$ operations.

6. Computational Results

Some comparisons of the ''interpreted'' evaluations summarized in §2 of functions and gradients with hand-coded Fortran evaluations appear in [15]. In the meantime, various new AD packages have become available, so it is of interest to see a few more comparative timings. I will present these new timings first, before considering Hessian computations, to give some scale and argue that the general approach described here is competitive, at least when sufficient memory is available. I carried out the present timings on an SGI Indy with one 100 MHz IP22 processor (MIPS R4010 floating-point chip and MIPS R4000 processor), with separate 8 kilobyte primary instruction and data caches and a unified secondary instruction/data cache of one megabyte, running IRIX 5.2. Compilations were by the SGI C and Fortran compilers; except for the C or Fortran from *nlc*, which was too large to optimize, compilations were with compiler flag `-O` (requesting optimization).

The first comparisons are with a small instance (22 atoms) of the protein-folding problem. The objective for this problem started out as a Fortran subroutine (the hand-coded Fortran in timings below, which evaluates an all-atom representation [22] of the empirical force field CHARMM [5]) that Teresa Head-Gordon and Frank Stillinger provided in connection with [21]; the AMPL model for the problem has various features, such as ''if'' expressions and heavy use of defined variables (in effect, named common subexpressions), that make it a good stress-test. Though partially separable, the problem has a completely dense Hessian matrix (parts of which, corresponding to atoms that are widely separated, will essentially be zero — but which parts depends on the

Evaluation method	*rel. time*	*kilo-bytes*
Hand-coded Fortran	1.0	860
Interpreted	2.6	856
Compiled code from *nlc*	2.0	964
Dense ADIFOR on *nlc* func	40.5	2064
Sparse ADIFOR on *nlc* func	35.7	2880
Dense ADIFOR on hand-coded func	10.0	1036
Sparse ADIFOR on hand-coded func	33.3	1380
ADOL-C record on *nlc* func	26.4	1772
ADOL-C replay on *nlc* func	12.3	1772
ADOL-C record on *f2c* func	17.6	1176
ADOL-C replay on *f2c* func	9.6	1176

Table 1. *Relative $f + \nabla f$ evaluation times for a 22-atom instance of the protein-folding problem.*

conformation of the atoms). For much more on the protein-folding problem and many pointers to the literature, see Neumaier's excellent survey [25].

Table 1 compares several ways of computing the protein-folding objective f and its gradient ∇f. The times in Table 1 are relative to that for the original hand-coded Fortran, which gives the fastest evaluations. The table also shows the maximum working-set size (in kilobytes) for each test program. Since larger values may imply more cache misses, this size may affect some results.

The interpreted evaluations in Table 1 are those of the AMPL/solver interface [17]; the compiled evaluations are for C and Fortran produced by the *nlc* program [17].

The Fortran preprocessor ADIFOR [3, 2] offers both ''dense'' and ''sparse'' modes, each of which is sometimes preferable. Table 1 shows ADIFOR evaluations for the Fortran objective produced by *nlc* and for the original hand-coded objective. The Fortran from *nlc* is loop-free, whereas the hand-coded Fortran has various loops, which permit the ADIFOR evaluations to operate with less overhead. I expected ADIFOR gradient evaluations to be about as fast as those from *nlc*, but then I learned that ADIFOR does not make direct provision for computing gradients. Rather, it computes JS, where J is a Jacobian matrix and S is a specified matrix; to compute $\nabla f(x)$, one can use $S = I$ (the identity matrix), but this leads to considerable overhead when n is large. Much of this overhead could be avoided if one could specify S when ADIFOR processed the Fortran, but currently one can only provide S as a run-time argument.

With more programming work, it should be possible to make the ADIFOR evaluations faster. Specifically, Bischof, Bouaricha, Khademi, and Moré [4] have described a way to use partially separable structure to speed up ADIFOR evaluations. On the minimal-surface problem considered below, for example, this idea is easy to apply, and it leads to ADIFOR evaluations that are only about a factor of 3 slower than hand-coded Fortran evaluations.

The last four lines in Table 1 are for gradient evaluations by the very general C++ package ADOL-C [18], working with C from variants of the protein-folding objective produced by *nlc* and and by the Fortran-to-C converter *f2c* [10], applied to the original hand-coded Fortran. In its derivative computations, ADOL-C uses some internal ''tape'' arrays recorded during during its evaluation of the function $f(x)$. ADOL-C can also replay the tape to evaluate f at a new x, though the result is only correct if no conditional outcomes change, i.e., if all relevant ''if'' tests give the same truth value as when the tape was recorded, and all ''min'' and ''max'' expressions give results from the same arguments. Evaluations that replay the tape are faster than those that record it, as the contrast between the ''ADOL-C record'' and ''ADOL-C replay'' lines in Table 1 shows. (The protein-folding problem involves some ''if'' expressions, so some rerecording of the tape would probably be necessary in the course of optimizing the protein-folding objective with ADOL-C tape evaluations.) The contrast in times between the interpreted and ADOL-C tape evaluations suggests there may be room to speed up the latter. (The ''ADOL-C record'' lines involve calls on ADOL-C's lower level `vec_jac` routines, rather than the higher-level `gradient` routine; with the latter, the first ''ADOL-C record'' line would have shown a ratio of 30.)

The protein-folding objective is rich in trigonometric functions, square roots, and exponentiations, which are expensive enough to mask some overhead. As an example with less expensive expressions and a sparse Hessian, we now consider the minimal-surface problem of MINPACK-2 [1, §4.2]. Figure 4 shows an AMPL model for this problem. Figure 5 shows some timings, relative to hand-coded Fortran function and gradient evaluation times, for some square meshes ($M = N = 2^k + 1$, giving $n = 4^k$ variables for several integers k). The solid line depicts interpreted evaluations with data structures that only provide for computing f and ∇f. The dashed line shows corresponding evaluations with data structures that store second partial derivatives for use in Hessian evaluations. As these structures are considerably larger, they cause cache misses for smaller n values; the runs for the dashed line allocated about 2.8 times as much memory as did the corresponding runs for the solid line. The runs with the largest n values involved some paging.

The dotted line in Figure 5 shows relative times to compute the Hessian, $\nabla^2 f$, from the partial derivatives stored when f was evaluated. The Hessian has at most 4 nonzeros per column. Evaluating it costs less than 3 interpreted function and gradient evaluations (of the more expensive variety).

Figure 6 shows the time it takes, relative to an interpreted function and gradient evaluation, to read the `.nl` files for the minimal-surface problems and create the data structures that permit function, gradient, and (for the dashed line) Hessian evaluations. The times for the smallest n are small enough that they are probably not very accurate. The times in the dashed line include expression walks to find the partially separable structure.

Minimization algorithms involve varying amounts of overhead in addition to the time needed for function and derivatives evaluations. Table 2 shows a range of overall

```
# Minimal surface problem, after MINPACK 2

param M integer > 0;     # discretization:
param N integer > 0;     # regular M x N grid

var v{0..M, 0..N};

param bdx{0..N, 0..1};  # boundary data, x axis
param bdy{0..M, 0..1};  # boundary data, y axis

param hx := 1 / N;
param hy := 1 / M;

var lower = 0.5*hx*hy
        *sum{i in 0 .. M-1, j in 0 .. N-1}
                 sqrt(1 + ((v[i+1,j] - v[i,j])/hx)^2
                        + ((v[i,j+1] - v[i,j])/hy)^2);

var upper = 0.5*hx*hy
        *sum{i in 1..M, j in 1..N}
                 sqrt(1 + ((v[i-1,j] - v[i,j])/hx)^2
                        + ((v[i,j-1] - v[i,j])/hy)^2);

minimize surface: lower + upper;

# Constraints to fix the boundary values:
s.t. xbdry{i in 0..N,   j in 0..1}: v[j*M,i] == bdx[i,j];
s.t. ybdry{i in 1..M-1, j in 0..1}: v[i,N*j] == bdy[i,j];
```

Figure 4. *AMPL model for minimal-surface problem.*

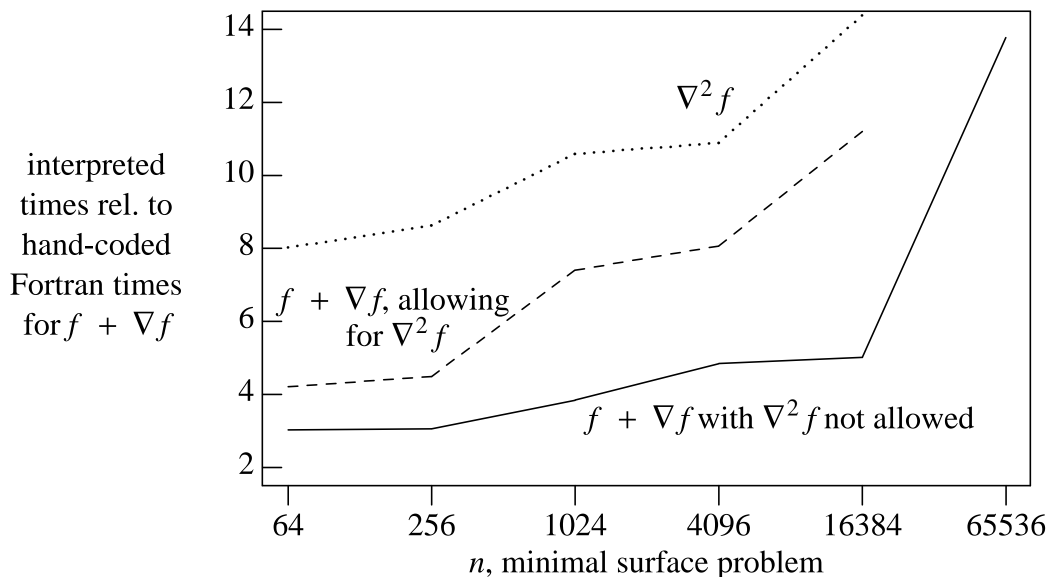

Figure 5. *Evaluation-time ratios, minimal-surface problems with $M = N$.*

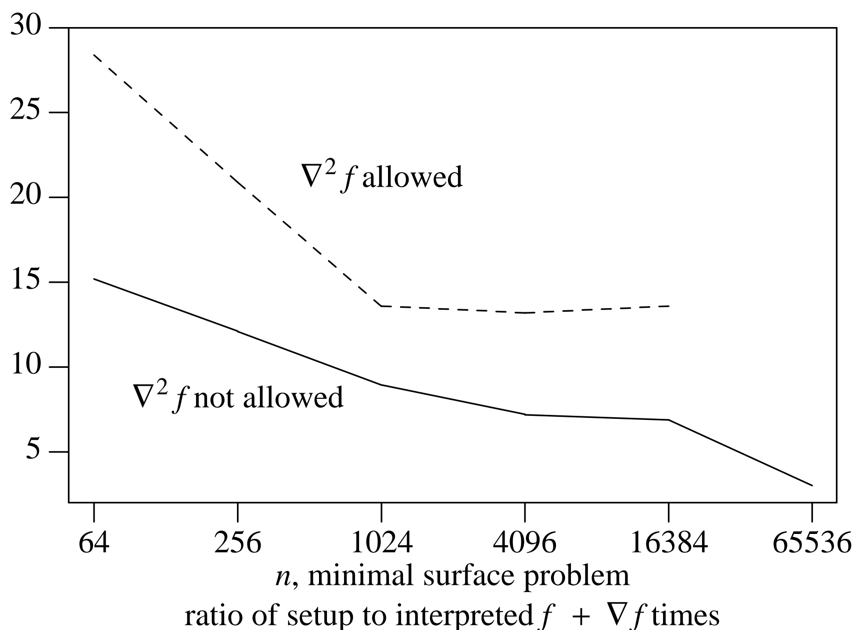

Figure 6. *Relative setup times, minimal-surface problems.*

solution times for some solvers to solve the 22-atom protein-folding problem discussed above, and for three instances of the minimal-surface problem. Solvers *mnh* and *mnhp* are variants of HUMSOL [14], which factors a dense Hessian matrix to carry out Newton's method and is thus impractical when n is large. The HUMSOL variants differ in that *mnhp* exploits partially separable structure in computing the Hessian matrix, whereas *mnh* simply computes n Hessian-vector products $\nabla^2 f(x) e^i$. Solvers *ve08* and *v8* are based on Toint's VE08 [26], which is designed to exploit partially separable structure; it uses a secant update to approximate each element Hessian and applies a truncated preconditioned conjugate gradient algorithm to compute an approximate Newton step. The variants differ in that *ve08* treats the problem as having just one element, thus reducing VE08 to a simple secant-update (quasi-Newton) method, whereas *v8* exploits the structure. The final three solvers, *tn*, *htn-fd* and *htn-hv*, are variants of Nash's truncated-Newton code TN [23, 24], which approximates Hessian-times-vector products by finite differences of gradients as it runs a preconditioned linear conjugate-gradient algorithm to compute an approximate Newton step. Variant *tn* runs without the overhead of data structures to compute Hessian-vector products, while the other two variants incur this overhead; *htn-fd* still uses finite-differences of gradients to approximate Hessian-vector products, and *htn-hv* uses backward AD to compute these products.

The *mnh* and *mnhp* times for small n show significant savings from using the partially-separable structure in computing the Hessian; for larger n, extra cache misses overwhelm this savings. The protein-folding results show there is nontrivial overhead in VE08: ignoring the structure of the problem caused it to take less time but substantially more iterations and function evaluations (277 iterations rather than 57 for *v8*, and 287

Problem	Protein folding	minimal surface		
Solver/*n*	66	81	256	1521
`mnh`	13.9	2.3	33.8	
`mnhp`	4.5	1.1	34.1	
`ve08`	8.3	0.8	10.8	
`v8`	15.4	0.6	2.8	28.2
`tn`	5.0	0.4	1.6	21.4
`htn-fd`	6.6	0.4	2.0	29.5
`htn-hv`	5.3	0.4	1.6	27.8

Table 2. *Indy solve times (seconds).*

overall function evaluations rather than 68). For the other problems, exploiting the structure was worthwhile for VE08; indeed, for $n = 1521$ *v8* ran quickly, but *ve08* took so much time that I did not complete the run.

Although TN is a very efficient solver, I was hoping it would run still faster when modified to use analytic Hessian-vector products. A comparison of the results for *htn-fd* and *htn-hv* suggests the disappointing conclusion that with my current implementation, it is better to use finite differences in this context.

Source for the basic solvers underlying the variants in Table 2 is available from *netlib* [9]: *mnh* and *mnhp* use ''`dmnhb from port`'', the current PORT Library [13] variant of HUMSOL, which enforces simple-bound constraints; both VE08 and TN are available from *netlib*'s `opt` directory as ''`ve08 from opt`'' and ''`tn from opt`''.

7. Concluding Remarks

One can detect partially separable structure automatically by walking expression graphs. Detecting this structure facilitates explicit Hessian computations, and should help make various algorithms more convenient for people to use.

The style of backward AD for Hessian-vector products described in this paper is somewhat memory-intensive, requiring about 64 bytes to represent a binary operation (with both operands differentiable) on a 32-bit machine, and about 48 bytes for a unary operation. Thus the approach described here is not universally applicable, but it is convenient and relatively fast for some applications. I hope soon to add to [17] an explanation of using routines that implement this approach.

For simplicity, I have been evaluating partial derivatives while carrying out the associated operations. In the ideal case where Newton's method is converging nicely and one computes ∇f and $\nabla^2 f$ as often as f, this costs no extra time, but in many calculations there is probably an opportunity to save time by postponing computation of partials until they are needed. How much time this might save remains a topic for further research. Would it, for example, make analytic Hessian-vector products preferable to finite differences of gradients in the solver TN discussed above? Would computing (by

backward AD) and storing the $\nabla^2 f_i$, then, for several vectors v, computing $\sum_{i=1}^{q} U_i^{\mathrm{T}}((\nabla^2 f_i)(U_i v))$ be faster than using backward AD to compute $\nabla^2 f_i(U_i v)$ directly?

Another topic for future research is seeing how worthwhile exploiting ‘‘group partially separable structure’’ might be. This is the structure in functions of the form

$$f(x) = \sum_{i=1}^{q} \theta_i (\sum_{j=1}^{r_i} f_{ij}(U_{ij} x)),$$

where $\theta_i : \mathbb{R} \rightarrow \mathbb{R}$ is a unary operator. LANCELOT [7], for example, is prepared to handle such structure. Automatically recognizing this structure is a relatively straightforward extension to the work reported in this paper.

Acknowledgements

I thank Bruce Christianson, Bob Fourer, Tanner Gay, Andreas Griewank, Brian Kernighan and Margaret Wright for helpful comments on the manuscript.

REFERENCES

[1] Brett M. Averick, Richard G. Carter, Jorge J. Moré, and Guo-Liang Xue, ‘‘The Minpack-2 Test Problem Collection,’’ Preprint MCS-P153-0692 (1992), Math. & Computer Science Div., Argonne National Lab. `ftp://-info.mcs.anl.gov/pub/MINPACK-2/tprobs/P153.ps.Z`.

[2] Christian Bischof, Alan Carle, Paul Hovland, Peyvand Khademi, and Andrew Mauer, ‘‘ADIFOR 2.0 User’s Guide,’’ ANL/MCS-TM-192 (1995), Mathematics and Computer Science Div., Argonne National Lab. `ftp://-info.mcs.anl.gov/pub/tech_reports/reports/TM192.ps`.

[3] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer, ‘‘The ADIFOR 2.0 System for the Automatic Differentiation of Fortran 77 Programs,’’ Argonne Preprint ANL-MCS-P481-1194 (1994), Mathematics and Computer Science Div., Argonne National Lab. `ftp://info.mcs.anl.gov/pub/-tech_reports/reports/P481.ps.Z`.

[4] Christian H. Bischof, Ali Bouaricha, Peyvand M. Khademi, and Jorge J. Moré, ‘‘Computing Gradients in Large-Scale Optimization Using Automatic Differentiation,’’ Report ANL/MCS-P488-0195 (1995), Argonne National Laboratory. `ftp://info.mcs.anl.gov/pub/tech_reports/-reports/P488.ps.Z`.

[5] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, ‘‘CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations,’’ *J. Computational Chemistry* **4** #2 (1983), pp. 187–217.

[6] B. Christianson, ‘‘Automatic Hessians by Reverse Accumulation,’’ *IMA J. Numer. Anal.* **12** (1992), pp. 135–150.

[7] A. R. Conn, N. I. M. Gould, and Ph. L. Toint, *LANCELOT, a Fortran Package for Large-Scale Nonlinear Optimization (Release A),* Springer-Verlag, 1992. Springer Series in Computational Mathematics 17

[8] Laurence C. W. Dixon, ‘‘Use of Automatic Differentiation for Calculating Hessians and Newton Steps,’’ pp. 114–125 in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, ed. A. Griewank and G. F. Corliss, SIAM (1991).

[9] J. J. Dongarra and E. Grosse, ‘‘Distribution of Mathematical Software by Electronic Mail,’’ *Communications of the ACM* **30** #5 (May 1987), pp. 403–407.

[10] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, ‘‘A Fortran-to-C Converter,’’ Computing Science Technical Report No. 149 (1990), AT&T Bell Laboratories, Murray Hill, NJ.

[11] R. Fourer, D. M. Gay, and B. W. Kernighan, ‘‘A Modeling Language for Mathematical Programming,’’ *Management Science* **36** #5 (1990), pp. 519–554.

[12] Robert Fourer, David M. Gay, and Brian W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming,* The Scientific Press, 1993. ISBN: 0-89426-232-7

[13] P. A. Fox, A. D. Hall, and N. L. Schryer, ‘‘The PORT Mathematical Subroutine Library,’’ *ACM Trans. Math. Software* **4** (June 1978), pp. 104–126.

[14] D. M. Gay, ‘‘ALGORITHM 611—Subroutines for Unconstrained Minimization Using a Model/Trust-Region Approach,’’ *ACM Trans. Math. Software* **9** (1983), pp. 503–524.

[15] David M. Gay, ‘‘Automatic Differentiation of Nonlinear AMPL Models,’’ pp. 61–73 in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, ed. A. Griewank and G. F. Corliss, SIAM (1991).

[16] D. M. Gay, ‘‘Massive Memory Buys Little Speed for Complete, In-Core Sparse Cholesky Factorizations on Some Scalar Computers,’’ *Linear Algebra and Its Applications* **152** (1991), pp. 291–314.

[17] David M. Gay, ‘‘Hooking Your Solver to AMPL,’’ Numerical Analysis Manuscript 93–10 (1993), AT&T Bell Laboratories. `ftp://netlib.bell-labs.com/netlib/att/cs/doc/93/4-10.ps.Z`.

[18] A. Griewank, D. Juedes, and J. Utke, ‘‘ADOL-C, A Package for the Automatic Differentiation of Algorithms Written in C/C++,’’ *ACM Trans. Math Software* (to appear). `ftp://info.mcs.anl.gov/pub/tech_reports/reports/-TM162.ps`.

[19] A. Griewank and Ph. L. Toint, ‘‘On the Unconstrained Optimization of Partially Separable Functions,’’ pp. 301–312 in *Nonlinear Optimization 1981*, ed. M. J. D. Powell, Academic Press (1982).

[20] A. Griewank and Ph. L. Toint, ''Partitioned Variable Metric Updates for Large Structured Optimization Problems,'' *Numer. Math.* **39** (1982), pp. 119–137.

[21] Teresa Head-Gordon, Frank H. Stillinger, David M. Gay, and Margaret H. Wright, ''Poly(L-alanine) as a Universal Reference Material for Understanding Protein Energies and Structures,'' *Proc. Natl. Acad. Sci. USA* **89** (1992), pp. 11513–11517.

[22] Frank A. Momany, Valentine J. Klimkowski, and Lothar Schäfer, ''On the Use of Conformationally Dependent Geometry Trends from *Ab Initio* Dipeptide Studies to Refine Potentials for the Empirical Force Field CHARMM,'' *J. Comp. Chem.* **11** (1990), pp. 654–652.

[23] S. G. Nash, ''Newton-type Minimization via the Lanczos Method,'' *SIAM J. Num. Anal.* **21** (1984), pp. 770–788.

[24] S. G. Nash, ''User's Guide for TN/TNBC: Fortran Routines for Nonlinear Optimization,'' Report 397 (1984), Mathematical Sciences Dept., The Johns Hopkins Univ., Baltimore, MD.

[25] Arnold Neumaier, ''Molecular Modeling of Proteins: A Feasibility Study of Mathematical Prediction of Protein Structure,'' manuscript (Jan. 1996). `http://solon.cma.univie.ac.at/~neum/ms/protein.ps.gz`.

[26] Ph. L. Toint, ''User's Guide to the Routine VE08 for Solving Partially Separable Bounded Optimization Problems,'' Technical Report 83/1 (1983), FUNDP, Namur, Belgium.