Automatically Finding and Exploiting Partially Separable Structure in Nonlinear Programming Problems

David M. Gay

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Nonlinear programming problems often involve an objective and constraints that are partially separable — the sum of terms involving only a few variables (perhaps after a linear change of variables). This paper discusses finding and exploiting such structure in nonlinear programming problems expressed symbolically in the AMPL modeling language. For some computations, such as computing Hessians by backwards automatic differentiation, exploiting partial separability can give significant speedups.

Overview

To set the context for this paper, it is necessary to talk about various aspects of nonlinear programming problems and automatic differentiation. Accordingly, it is convenient to begin with brief overviews of Newton's method, nonlinear programming, and automatic differentiation. Since I report computational experience with problems expressed symbolically in the AMPL modeling language, a brief account of AMPL is also appropriate. In the initial overviews, I will omit most references.

Newton's Method for Nonlinear Equations

Newton's method is in some ways an ideal algorithm for solving systems of nonlinear equations. It is easily derived by a linearization argument, and it converges quickly when started close to a ''strong'' solution. As a simple example, Table 1 shows the sequence of residual errors for a square-root iteration; note how the residuals are approximately squared in successive iterations (''quadratic convergence'').

In more detail, if $f : \mathbb{R}^n \to \mathbb{R}^n$ is a differentiable mapping of real n-space to itself, then $f(x + y) \approx f(x) + f'(x)(y - x)$, where $f'(x)$ is the Jacobian matrix of f at x, so if $f'(x)$ is nonsingular and $y = x - f'(x)^{-1} f(x)$, then $f(y) \approx \mathbf{0}$, which gives Newton's method:

$$x^{k+1} = x^k - f'(x^k)^{-1} f(x^k). \tag{1}$$

To carry out a step of Newton's method, it is of course not necessary to explicitly form $f'(x^k)^{-1}$; rather it suffices to solve

iter.	resid.
0	1e+00
1	4e–01
2	8e–02
3	4e–03
4	1e–05
5	8–11

Table 1. *Sample residuals in Newton's method.*

$$f'(x^k)\,s = -f(x^k) \tag{2}$$

for s and then compute $x^{k+1} = x^k + s$. To solve (2), one can either work with $f'(x^k)$ explicitly or use an iterative method that just requires computing the matrix-vector product $f'(x^k)\,d$ for certain vectors d. See, for example, [19, 23, 30].

Unfortunately, Newton's method can only be guaranteed to work when when f' is sufficiently smooth (e.g., Lipschitz continuous) and the initial iterate, x^0, is close enough to a solution x^* at which $f'(x^*)$ is nonsingular. In practice it is usually necessary to use some safeguarded variant of Newton's method, such as the damped Newton method

$$x^{k+1} = x^k - \lambda^k f'(x^k)^{-1} f(x^k)$$

in which λ^k is a (positive) step length that should tend to 1 (for quadratic convergence), but that may have to be smaller initially. Trust-region methods are another popular way to encourage convergence of Newton's method; they maintain the radius of a region about the current iterate in which they consider the current approximation to f valid and restrict the steps they compute to this region. Various textbooks go into much more detail about safeguarding Newton's method than there is space for here; see, for example, [2, 7, 10, 18, 28].

One reason to modify Newton's method is to encourage convergence from poor starting guesses. Another is that computing the requisite derivatives is sometimes inconvenient or too costly. Indeed, much research on optimization algorithms has dealt with ways to approximate derivatives. Perhaps the most straightforward way is to use finite differences. For example,

$$\partial f_i/\partial x_j \approx [\, f_i(x + h_{ij} e^j) - f_i(x)] \,/\, h_{ij} \tag{3}$$

is a forward-difference approximation to component (i,j) of $f'(x)$, where e^j is the j-th standard unit vector and h_{ij} is a suitable step length, and

$$\partial f_i/\partial x_j \approx [\, f_i(x + h_{ij} e^j) - f_i(x - h_{ij} e^{j)}] \,/\, (2h_{ij}) \tag{4}$$

is a corresponding central-difference approximation; (4) takes more work but is often more accurate. Finite-difference approximations like (3) and (4) suffer from two sources of error: round-off error (when carried out in finite-precision arithmetic), and truncation error, which arises because of nonlinearity.

Secant-update methods are much akin to finite-difference computations, except that they only update their approximation A^k to $f'(x^k)$ by a low-rank change each iteration (e.g., by a rank-1 change for Broyden's method). In general, secant updates satisfy the quasi-Newton equation,

$$A^{k+1}(x^k - x^{k-1}) = f(x^k) - f(x^{k-1}).$$

An alternative considered in more detail below is so-called automatic differentiation (AD), a mixture of analytic and numeric computation that uses partial derivatives of elementary operations and the chain rule to avoid truncation error, and is sometimes much more efficient than finite differences.

Nonlinear Programming

For a point x^* to minimize or maximize a smooth function $\phi:\mathbb{R}^n \to \mathbb{R}$, it is necessary that $\phi'(x^*) = 0$. Thus at first glance, an ideal iteration for minimizing ϕ would be to use Newton's method on $f(x) = \nabla\phi(x) \equiv \phi'(x)^{\mathrm{T}}$, the transpose of $\phi'(x)$. Of course, a complication here is that a point where $\nabla\phi(x^*) = \mathbf{0}$ could be a local maximizer or saddle point, rather than a local minimizer. To encourage convergence to a minimizer, people often use a descent method, one that requires $\phi(x^{k+1}) < \phi(x^k)$, or at least $\phi(x^{k+m}) < \phi(x^k)$ for bounded m. (The texts cited above give more details.)

For $f(x) = \nabla\phi(x)$, we have $f'(x) = \nabla^2\phi(x)$, the so-called Hessian matrix. Much research has gone into secant-update methods for approximating $\nabla^2\phi$. Below I describe an alternative: use of ''backwards'' automatic differentiation for computing $\nabla^2\phi$.

Now consider imposing m equality constraints $c(x) = \mathbf{0}$, where $c:\mathbb{R}^n \to \mathbb{R}^m$ is sufficiently smooth. For $x = x^*$ to minimize $\phi(x)$ subject to $c(x) = \mathbf{0}$, if the constraint Jacobian matrix $c'(x^*)$ has full row-rank, the Lagrange multiplier rule gives the necessary condition

$$\nabla\phi(x^*) = \nabla c(x^*)y,$$

where $\nabla c = c'^{\mathrm{T}}$ is the transpose of $c'(x) \in \mathbb{R}^{m\times n}$ and $y \in \mathbb{R}^m$ is a vector of Lagrange multipliers. Aside from safeguards to prevent convergence to saddle points and local maximizers, an ideal algorithm for this problem is Newton's method applied to $f(\tilde{x}) = \phi(x) - y^{\mathrm{T}}c(x)$, with $\tilde{x} = \begin{bmatrix} x \\ y \end{bmatrix}$. This requires computing (or being able to multiply by) both the Jacobian matrix of the constraints, $c'(x)$, and the Hessian of the Lagrangian function,

$$W(x) = \nabla^2\phi(x) - \sum_{i=1}^{m} y_i \nabla^2 c_i(x). \tag{5}$$

Further imposing p inequality constraints $d(x) \geq \mathbf{0}$, where $d:\mathbb{R}^n \to \mathbb{R}^p$ is smooth, leads to more complicated necessary conditions involving the signs of the Lagrange multipliers for d:

$$\nabla\phi(x^*) = \nabla c(x^*)y + \nabla d(x^*)z,$$

with $c(x^*) = \mathbf{0}$, $d(x^*) \geq \mathbf{0}$, $z \geq 0$, and $z_i d_i(x^*) = 0$ for $1 \leq i \leq p$ (''complementary slackness''), i.e., $z_i = 0$ if $d_i(x^*) > 0$. Active set methods guess which components of d will be 0, revising the guess if it proves wrong. The currently popular interior-point methods (which are closely related to the barrier methods) solve a sequence of problems in which the complementary slackness condition is replaced by $z_i d_i(x) = \mu$, $1 \leq i \leq p$, for a sequence of values of $\mu = \mu^k > 0$ with $\mu^k \to 0$. Either way, an ''ideal'' algorithm needs the Hessian of the Lagrangian function (5), which now becomes

$$W(x) = \nabla^2 \phi(x) - \sum_{i=1}^{m} y_i \nabla^2 c_i(x) - \sum_{i=1}^{m} z_i \nabla^2 d_i(x).$$

For simplicity, in what follows I will mainly discuss computing $\nabla^2 \phi(x)$, but the implementation discussed below computes $W(x)$.

Automatic Differentiation

There are several ways to compute (exact or approximate) derivatives. Perhaps most familiar is symbolic computation: we apply the rules of calculus to find expressions for the desired derivatives, then evaluate these expression. Finite-difference approximations such as (3) and (4) have often been used in numerical computations, as they are easy to program and do not require detailed knowledge of the expressions being differentiated. Recently there has been growing interest in automatic differentiation. For example, some 31 papers on AD appear in the proceedings book [1], including a history [25] with many further references.

Consider computing first derivatives by AD. Most straightforward is the ''forward mode'', in which we compute the partials of each elementary operation with respect to the independent variables. For example, if operation $o = o(\ell, r)$ depends on operands ℓ and r, and we know the partials $\partial \ell / \partial x_i$ and $\partial r / \partial x_i$ of ℓ and r with respect to the independent variables x_i, $1 \leq i \leq n$, then we can use the chain rule to compute

$$\frac{\partial o}{\partial x_i} = \frac{\partial o}{\partial \ell} \cdot \frac{\partial \ell}{\partial x_i} + \frac{\partial o}{\partial r} \cdot \frac{\partial r}{\partial x_i}.$$

Forward AD is straightforward and is readily extended to recur higher derivatives, but it can turn an expression evaluation that requires $O(k)$ operations into one that takes $O(k^2)$ operations. (The same criticism applies to straightforward symbolic computation of derivatives, though fancier symbolic analysis can sometimes overcome this ''expression swell''.) Backward AD for computing $\nabla \phi(x)$ recurs the partials $\partial \phi / \partial o$ of ϕ with respect to each operation o involved in evaluating $\phi(x)$. To do this, it is necessary to visit the elementary operations first in ''forward'' order to compute $\phi(x)$, then in ''reverse'' order to recur the partials. At the end of this ''reverse sweep'', we have $\partial \phi / \partial x_i$, the components of $\nabla \phi(x)$. Backward AD has the advantage of computing both $\phi(x)$ and $\nabla \phi(x)$ in at most a small multiple of the time needed to compute $\phi(x)$ alone. It has the disadvantage of needing to save information for the reverse sweep. Thus it may involve considerably more storage than does forward AD.

It is conceptually straightforward to extend a backward AD computation of $\nabla\phi(x)$ to a Hessian-times-vector computation, i.e., $\nabla^2\phi(x)v$ for arbitrary (constant) vectors v; all we need do is apply backward AD to the computation of $v^T\nabla\phi(x)$. Bruce Christianson [5] has described another another way to arrange this computation: use forward AD to compute $\psi(0)$ and $\psi'(0)$, where $\psi:\mathbb{R}\to\mathbb{R}$ is the scalar function given by

$$\psi(\tau) = \phi(x + \tau v), \tag{6}$$

and apply backward AD to compute $\nabla^2\phi(x)v$, which is the gradient with respect to x of $\nabla\psi$. I have found Christianson's scheme easier to think about (than applying AD to $v^T\nabla\phi(x)$) and have used it in the programming behind the computations reported below.

We can compute $\nabla^2\phi(x)$ a column at a time by computing $\nabla^2\phi(x)e^i$ for each of the standard unit vectors e^i, $1 \le i \le n$. This is $O(n)$ times more expensive than a single evaluation of $\phi(x)$, which may be prohibitive. Fortunately, as discussed in more detail below, many problems exhibit partially separable structure, and exploiting this structure can make Hessian computations considerably cheaper.

AD with AMPL, a Modeling Language for Mathematical Programming

AMPL [11, 12] is a language and modeling environment for expressing linear and nonlinear programming problems in a notation close to what one writes on a blackboard. It encourages one to think separately about modeling a class of phenomena and providing data for a particular instance. Thus an AMPL model describes a class of constrained optimization problems, and separate data sets provide details (numeric or symbolic parameters and sets of tuples of such entities) needed to instantiate a particular problem. Models usually depend on some fundamental parameters and sets that will appear in a data set; most models specify computing other parameters and sets from fundamental or previously computed ones. After the AMPL processor has instantiated a particular instance, it makes the instance available to solvers (which are independent of the AMPL processor) by writing a ‘‘`.nl`’’ file describing the instance. In particular, the `.nl` file contains expression graphs for the problem's nonlinear expressions. Solvers invoke interface routines that read the `.nl` file and can arrange for derivative computations. The AMPL home page

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

gives pointers to source for the interface routines and to more information on AMPL in general.

The backward AD computations of gradients in the AMPL/solver interface routines are described in [15]. Briefly, partials of operations are computed and stored when the operations are executed, and in ‘‘interpreted’’ evaluations, the AD backward sweep is simply a sequence of multiplications and additions of the form `a := a + b*c`, which are carried out by the C loop

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

The ‘‘a’’ variables are ‘‘adjoints,’’ partials of the form $\partial\phi/\partial o$; they are initialized to zero (except for the adjoints corresponding to the independent variables x: linear terms are computed separately, and the adjoints for $\partial\phi/\partial x_i$ are initialized to the coefficients in the linear terms).

The interpreted evaluations themselves take the form

```
value = e->op(e);
```

in which `e` points to a structure representing an operation. The `op` field of this structure points to a function that carries out the operation and stores its partial derivatives.

‘‘Compiled’’ evaluations are also possible. A program called *nlc* turns the `.nl` file into a C or Fortran routine for evaluating the objective function, constraints, and their first derivatives. *Nlc* arranges to avoid some operations that arise in the interpreted evaluation, which can lead to faster gradient computations. However, because of the extra compilation and linking that compiled evaluations involve, they only save overall time if the function and gradient are evaluated a great many times. Compiled evaluations are discussed briefly in [16].

For backwards computation of Hessian-vector products, some elementary operations in the evaluation of ϕ engender several multiplications and additions in $\nabla^2\phi(x)v$, so it seems reasonable to switch on the operation type and carry out several operation-specific adjoint computations at once. There is a forward sweep to compute $\psi'(0)$ for (6), and a backward sweep, both with switching on operation type. More details appear in [17].

Partially Separable Structure

Many nonlinear programming problems involve objectives (and constraint bodies) of the form

$$\phi(x) = \sum_{i=1}^{q} \phi_i(U_i x), \tag{7}$$

where $U_i \in \mathbb{R}^{m_i \times n}$ is a matrix with a small number m_i of rows. For example, some computational results for the protein-folding problem appear below; when this problem is 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, and $m_i = 3$, 6, or 9.

There is interesting structure in the gradient and Hessian of (7):

$$\nabla\phi(x) = \sum_{i=1}^{q} U_i^{\mathrm{T}} \nabla\phi_i(U_i x) \tag{8}$$

and

$$(9) \qquad \nabla^2 \phi(x) \;=\; \sum_{i=1}^{q} U_i^{\mathrm{T}} \nabla^2 \phi_i(U_i x)\, U_i .$$

Griewank and Toint [21, 22] originally pointed out the possibility for exploiting the structure in $(7-9)$ and proposed using secant updates to approximate each $\nabla^2 \phi_i$ separately.

In general, a secant-update scheme for approximating an $n \times n$ Hessian matrix only gains information about one direction per update (even though most symmetric secant updates involve rank-2 changes), so even for secant updates to approximate the Hessian of a quadratic function (i.e., a function whose Hessian is constant), it can take n updates before a secant approximation to the Hessian is ''good'' in all directions. (Fortunately, it is not always necessary for them to be good in all directions. Indeed, when used in quasi-Newton methods, secant updates can lead to superlinear convergence, even though the Hessian approximations are ''bad'' in some directions. But that is another story.) Since the Hessians $\nabla^2 \phi_i$ in (9) are only $m_i \times m_i$ matrices, secant approximations to them can give good approximations in only $m_i << n$ steps. Thus after $\max_i m_i$ updates, secant approximations to $\nabla^2 \phi_i$ may give a very good overall Hessian approximation.

Use of partially separable structure can speed the Hessian computation described above. Rather than computing n matrix vector products $\nabla^2 \phi(x) e^i$, we separately compute each $\nabla^2 \phi_i$ with m_i Hessian-vector products, and accumulate $U_i^{\mathrm{T}} \nabla^2 \phi_i(U_i x) U_i$. The computational results reported below show that this can give a substantially faster Hessian evaluation than n Hessian-vector products $\nabla^2 \phi(x) e^i$.

Recognizing more structure may be worthwhile. For example, LANCELOT [6] is a solver that exploits the structure in

$$(10) \qquad \phi(x) \;=\; \sum_{i=1}^{q} \theta_i \Big(\sum_{j=1}^{r_i} \phi_{ij}(U_{ij} x) \Big),$$

where $\theta_i : \mathbb{R} \to \mathbb{R}$ is a unary operator. This ϕ has

$$\nabla^2 \phi(x) \;=\; \sum_{i=1}^{q} \theta_i' \Big(\sum_{j=1}^{r_i} \phi_{ij}(U_{ij} x) \Big) \sum_{j=1}^{r_i} U_{ij}^{\mathrm{T}} \nabla^2 \phi_{ij}(U_{ij} x)\, U_{ij}$$

$$+ \sum_{i=1}^{q} \theta_i'' \Big(\sum_{j=1}^{r_i} \phi_{ij}(U_{ij} x) \Big) \Omega_i \Omega_i^{\mathrm{T}}$$

with

$$\Omega_i \;=\; \sum_{j=1}^{r_i} U_{ij}^{\mathrm{T}} \nabla \phi_{ij}(U_{ij} x).$$

For simplicity, the rest of this paper focuses on (7) rather than (10).

Automatic detection of partially separable structure seems worthwhile, as it should encourage exploitation of this structure. Users are likely to find automatic detection

much more convenient than, say, explicitly stating this structure by expressing the problem in the input format SIF associated with LANCELOT [6]. Given the expression graphs written by the AMPL processor, it is conceptually straightforward to find partially separable structure (including the structure in (10)): we find ϕ_i and U_i by walking the graph for ϕ. In so doing, we accumulate linear terms (rows of a U_i) as long as possible — until they appear in a nonlinear operation. Once we have 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 it into a hash table, so we can find duplicate appearances of U_i in ϕ_i. Similarly, once we have found a term ϕ_i, we put U_i into a canonical form and hash it, so we can easily tell if a subsequent ϕ_j has $U_j = U_i$, in which case we can merge ϕ_j into ϕ_i. The computations described below involve walking the expressions for ϕ (and for any constraints) once to determine the partially separable structure, and a second time to arrange for derivative and (where appropriate) Hessian times vector computations.

Computational Comparisons

For simplicity, the results reported here are for a single objective, the empirical energy function for a protein-folding problem. The function started life as a Fortran subroutine that Teresa Head-Gordon provided in connection with [24], and it is interesting to compare the function and gradient evaluation times for this hand-coded Fortran with those for interpreted evaluations of an AMPL version of the energy function. For much more on the protein-folding problem and many pointers to the literature, see Neumaier's excellent survey [29]. The results reported in this section also appear in [17], along with some other results.

My AMPL model for the protein-folding problem makes heavy use of ''defined variables'', which amount to named common expressions. For example, the van-der-Waals term for a pair of atoms involves $\sigma^2\rho^{-12} - 2\sigma\rho^{-6}$, where ρ is the distance between the atoms and σ is a scale factor. The AMPL model gives the names ''`rinv`'' and ''`r6`'' to ρ^{-1} and $\sigma\rho^{-6}$:

```
var rinv{i in Pairs} = 1. / sqrt(
        sum{j in D3} (x[inb[i],j] - x[jnb[i],j])^2 );

var r6{i in Pairs} =
        ((sigma[inb[i]] + sigma[jnb[i]]) * rinv[i])^6;
```

and later uses both of these defined variables in declaring another one, `pair_energy`, the overall contribution from pairwise interactions:

```
var pair_energy =
        sum{i in Pairs} (
                  332.1667*q[inb[i]]*q[jnb[i]]*rinv[i]
                + sqrt(eps[inb[i]]*eps[jnb[i]])
                        *(r6[i]^2 - 2*r6[i]) );
```

This, in turn, appears in the overall objective:

```
minimize energy: bond_energy + angle_energy
                 + torsion_energy + improper_energy
                 + pair14_energy + pair_energy;
```

Some of the other terms involve if-then-else expressions and other forms that make this example a good stress-test for derivative computations. (The set D3 is that of the three spatial dimensions, and for convenience in debugging and providing data, the double subscripting of various parameters follows that in the hand-coded Fortran.)

For more simplicity and speed of testing, the results below are for a very small instance: 22 atoms, with 66 variables in Cartesian coordinates. I carried out 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. Some caveats are in order. Cache details can strongly affect relative timings, as does the mix of algebraic and transcendental operations (such as trigonometric functions and exponentials) appearing in the expressions. The protein-folding objective is rich in transcendental operations, which mask some overhead.

Table 2 compares some ways of computing the protein-folding objective ϕ and its gradient $\nabla\phi$. It shows times relative to that for the original hand-coded Fortran, which gives the fastest evaluations. It also shows the maximum working-set size (in kilobytes) for each test program. This size may affect some results, as larger values may imply more cache misses. The interpreted evaluations are those of the AMPL/solver interface [16]; the compiled evaluations are for Fortran produced by the *nlc* program [16]. ADIFOR [4, 3] is an ambitious Fortran 77 preprocessor, the result of a collaboration among people at Argonne National Laboratory and Rice University. It has ‘‘dense’’ and ‘‘sparse’’ modes, each of which is sometimes preferable. Table 2 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 may permit the ADIFOR evaluations to operate with less overhead. Finally, the ADOL-C evaluations show the very general C++ package ADOL-C [20] working on two variants of C for the objective: C from *nlc* and C from the Fortran-to-C converter *f2c* [9], applied to the original hand-coded Fortran.

In part, at least, the ADIFOR evaluations are slower than those with the AMPL/solver interface because ADIFOR mainly uses the forward AD mode; it does use backward AD to deal with a single Fortran statement. The ADOL-C evaluations involve recording a ‘‘tape’’ and replaying it to carry out a backward AD gradient evaluation. When conditional outcomes do not change, ADOL-C can also compute $f(x)$ at a new x by replaying the tape. As Table 2 suggests, the tape handling involves significant overhead, but replaying the tape to compute $f(x)$ is faster than recording the tape.

Evaluation method	*rel. time*	*kilo-bytes*
Hand-coded Fortran	1.0	860
Interpreted	2.6	856
Compiled	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 2. *Relative* $\phi + \nabla\phi$ *evaluation times for a 22-atom instance of the protein-folding problem.*

On the 22-atom protein-folding problem, the automatic extraction of partially separable structure sketched above finds

```
6 defined variables split into 242
483 or more initial elements
609 distinct linear terms
600 duplicate linear terms in different elements
192 duplicate linear terms in the same element
242 adjusted elements.
```

How worthwhile is finding this structure? The answer depends on the use to which it is put. Table 3 shows the times (Indy seconds) for several solvers to solve the 22-atom protein-folding problem. `Mng`, `Mnh` and `mnhp` use the current PORT library [13] versions of SUMSOL (`mng`) and HUMSOL [14]; these versions are available from *netlib* [8] as ‘‘`dmngb dmnhb from port`’’. SUMSOL uses the BFGS secant-update to approximate the Hessian, whereas HUMSOL carries out Newton’s method, both with trust-region regularizations. The HUMSOL variants differ in that `mnh` computes the Hessian by n Hessian-times-vector products, ignoring the partially separable structure, whereas `mnhp` exploits this structure, summing the element Hessians to compute the overall Hessian. In this instance, finding the partially separable structure speeds the overall HUMSOL computation by over an order of magnitude, making it faster than SUMSOL. (In contrast to the solvers discussed next, neither SUMSOL nor HUMSOL is suitable for high dimensional problems, since they use dense-matrix techniques. On a problem with n variables, SUMSOL’s overhead is $O(n^2)$ per iteration, and HUMSOL’s is $O(n^3)$.)

Finding the partially separable structure in this example takes 0.22 seconds, the time of about 8.3 function and gradient evaluations. This is about the same time it takes to

read the problem in and build data structures for just function and gradient evaluations. In other words, finding the partially separable structure roughly doubles the setup time, at least in this instance. (The graph walks and hashing to find partially separable structure are essentially linear-time algorithms. Many of the algorithms within the AMPL processor also have this character; computing the original expression graphs and writing them out took AMPL about 0.7 seconds.)

Solver	seconds
`mng`	8.1
`mnh`	13.9
`mnhp`	4.5
`ve08`	8.3
`v8`	15.4
`tn`	5.0
`htn-fd`	6.6
`htn-hv`	5.3

Table 3. *Indy solve times.*

`Ve08` and `v8` are based on Phillipe Toint's partially-separable solver VE08AD, which is available from *netlib* as ''`ve08 from opt`''. It uses a secant update to approximate each element Hessian and applies a truncated preconditioned conjugate gradient algorithm to compute an approximate Newton step. `Ve08` and `v8` differ in that `ve08` ignores partially separable structure, whereas `v8` uses it. Table 3 shows that in this particular case the extra overhead needed to deal with 242 elements costs more time than it saves. Table 3 does not show that `v8` solves the problem in considerably fewer iterations and function evaluations than `ve08`: 57 quasi-Newton iterations and 68 overall function evaluations for `v8` versus 277 iterations and 287 function evaluations for `ve08`.

Solvers `tn`, `htn-fd` and `htn-hv` use Stephen Nash's truncated-Newton code TN [26, 27], which is available from *netlib* as ''`tn from opt`''. It 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. As Table 3 shows, TN can be very efficient. The three variants of TN differ in that `tn` only does the work needed for interpreted gradient evaluations, whereas the `htn` variants include the overhead during function evaluations of saving second partials for use in Hessian-times-vector computations. `Htn-fd` still computes finite differences of gradients, whereas `htn-hv` replaces them with an analytic Hessian-times-vector computation. It is slightly disappointing that the latter does not save time in this example. `Htn-hv` does happen to save some Hessian-times-vector products (327 versus 359 for `htn-fd`), but takes three more iterations (39 versus 36). Another variant is possible: we could compute the element Hessians and use them in computing Hessian-times-vector products for TN. This remains for future research.

Conclusions

There is much current interest in using explicit Hessians in algorithms for constrained optimization. Exploiting partially separable structure can lead to considerable time savings in their computation. By walking the expression graphs representing the objective and constraints, it is possible to detect partially separable structure at the cost of only a few function-and-gradient evaluation times. Such automatic detection should prove much more convenient to many users than schemes that require explicitly specifying this structure.

The preliminary computational results reported above are encouraging, but need to be augmented by more testing. The interpreted evaluations provided by the AMPL/solver interface are somewhat memory-intensive, requiring about 64 bytes per binary operation (with both operands differentiable) and 48 bytes per unary operation on a 32-bit machine. Thus, while interpreted AMPL evaluations of expressions and their first and perhaps second derivatives are relatively fast and convenient for some applications, they are impractical for others.

REFERENCES

[1] *Automatic Differentiation of Algorithms: Theory, Implementation, and Application,* SIAM, 1991. Edited by A. Griewank and G. Corliss

[2] Dimitri P. Bertsekas, *Nonlinear Programming,* Athena Scientific, P.O.Box 391, Belmont, MA, 02178-9998, U.S.A., 1995.

[3] 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`.

[4] 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`.

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

[6] 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

[7] J. E. Dennis, Jr. and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations,* Prentice-Hall, Englewood Cliffs, NJ, 1983.

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

[9] 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.

[10] R. Fletcher, *Practical Methods of Optimization*, Wiley, 1987. Second edition.

[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] 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`.

[17] David M. Gay, ‘‘More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability,’’ in *Computational Differentiation: Applications, Techniques, and Tools*, ed. George F. Corliss, SIAM (1996). `http://www.ampl.com/ampl/REFS/ad96.ps.gz` is the original *troff* version.

[18] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, 1981.

[19] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD, 1989. Second edition.

[20] 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`.

[21] 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).

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

[23] L. Hageman and D. Young, *Applied Iterative Methods*, Academic Press, 1981.

[24] 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.

[25] Masao Iri, ‘‘History of Automatic Differentiation and Rounding Error Estimation,’’ pp. 3–16 in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, ed. A. Griewank and G. F. Corliss, SIAM (1991).

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

[27] 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.

[28] Stephen G. Nash and Ariela Sofer, *Linear and Nonlinear Programming*, McGraw-Hill, 1996.

[29] 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`.

[30] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing, 1996. ISBN: 0-534-94776-X