AT&T
Bell Laboratories

Numerical Analysis Manuscript 91–05

Automatic Differentiation of Nonlinear AMPL Models

David M. Gay

August 22, 1991

Automatic Differentiation of Nonlinear AMPL Models

David M. Gay∗

Abstract. We describe favorable experience with automatic differentiation of mathematical programming problems expressed in AMPL, a modeling language for mathematical programming. Nonlinear expressions are translated to loop-free code, which makes analytically correct gradients and Jacobians particularly easy to compute — static storage allocation suffices. The nonlinear expressions may either be interpreted or, to gain some execution speed, converted to Fortran or C.

1. Introduction. Modeling languages for mathematical programming aim to expedite the task of giving an optimization problem to a solver. For nonlinear problems, this task generally includes providing derivatives to the solver. This note reports favorable experience with automatic differentiation to compute derivatives of problems expressed in the AMPL modeling language [8].

Our recent paper [8] provides much more discussion of modeling languages in general and of AMPL in particular than space would permit here. Although that paper deals mostly with linear problems, its concerns remain valid for nonlinear problems.

It is often convenient to work with a class of optimization problems parameterized by certain parameters and sets (which themselves are often used to compute other parameters and sets); we call such a class of problems a *model*. We obtain a particular problem — an instance of a model — by associating with the model data that define the parameters and sets which are used but not defined in the model.

2. Maximal *n*-gon example. It is helpful to see an example of a nonlinear problem. Consider the problem of finding a planar n-gon of maximal area inscribed in a circle of diameter 1. For odd n, the regular n-gon is known to solve this problem, but for even n, one can do better than the regular n-gon. The solution for even $n > 4$ is rigorously known only for $n = 6$: see [12].

Figure 1 shows an AMPL model for the maximal n-gon problem (derived from a GAMS [2] model of Francisco J. Prieto). The model illustrates several aspects of AMPL. It involves one free parameter, `n`, the number of sides in the n-gon. To get a particular problem (an instance of the model), we would specify a value for `n`; to get the maximum hexagon problem, for example,

∗AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974-0636; dmg@research.att.com .

```
# Maximum area for polygon of N sides.
# After a GAMS model by Francisco J. Prieto.

param n integer > 0;
set I := 1..n;

param pi := 4*atan(1);

var rho{i in I} <= 1, >= 0                    # polar radius
                := 4*i*(n + 1 - i)/(n+1)**2;

var theta{i in I} >= 0                        # polar angle
                  := pi*i/n;

inscribe{i in I, j in i+1 .. n}:
        rho[i]**2 + rho[j]**2
                  - 2*rho[i]*rho[j]*cos(theta[j] - theta[i])
        <= 1;

increasing{i in 2..n}:  theta[i] >= theta[i-1];

fix_theta: theta[n] = pi;
fix_rho:   rho[n] = 0;

maximize area:  .5*sum{i in 2..n}
                     rho[i]*rho[i-1]*sin(theta[i]-theta[i-1]);
```

FIG. 1. *AMPL model for maximum n-gon problem.*

we would specify

```
data;
param n := 6;
```

Comments start with # and extend to the end of the line. The statement

```
param I := 1..n;
```

defines `I` to be the set $\{1, 2, \cdots, n\}$. The statement

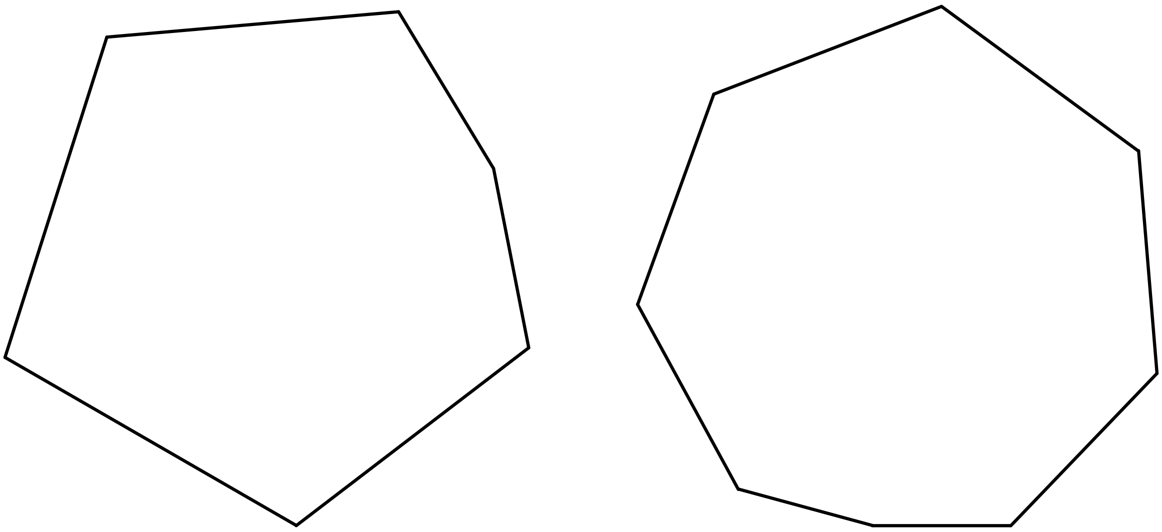
```
param pi := 4*atan(1);
```

defines `pi` to be the machine approximation to $\pi = 4 \cdot \tan^{-1}(1)$. The model expresses vertices of the *n*-gon in polar coordinates, say (ρ_i, θ_i), represented by the variables `rho` and `theta`, both of which are subscripted by elements of set `I`. Square brackets surround subscripts, so, e.g., `rho[i]` denotes ρ_i. The declaration (i.e., `var` statement) for `rho` specifies upper and lower bounds (1 and 0) and, in the `:=` clause, an initial guess; the indexing expression `{i in I}` in the declaration specifies that `i` is a dummy variable running over `I`. Indexing expressions always have a scope; in the declaration of `rho`, the scope of `i` extends over the whole statement, and `i` participates in specifying the initial guess for `rho[i]`. The statement that begins

```
inscribe{i in I, j in i+1 .. n}:
```

defines a family of constraints that keeps each pair of points Euclidean distance at most 1 apart; ‘‘`inscribe`’’ is the (arbitrarily chosen) name of the constraint family, and once the problem has been solved, `inscribe[i,j]` denotes the dual variable (i.e., Lagrange multiplier) associated with constraint `inscribe[i,j].` Similarly, `increasing` is a family of constraints that force the polar angles to be monotone increasing. The constraints `fix_theta` and `fix_rho` serve to remove two unnecessary degrees of freedom by specifying the values of `theta[n]` and `rho[n]`; these constraints are not seen by the solver, since they are removed from the problem by AMPL's ''presolve'' phase (see [8]). Finally, the statement that begins ''`maximize area:`'' specifies the objective function, which involves a summation.

Fig. 2 depicts the solutions found by MINOS [18, 19, 20] version 5.3 for `n` = 6 and 8, and it lists the areas of the corresponding regular *n*-gons.



n = 6
area = 0.674981
regular area = 0.649519

n = 8
area = 0.726868
regular area = 0.707107

FIG. 2. *Solutions computed by MINOS 5.3.*

3. Translated problem representation. The AMPL translator communicates with solvers by writing and reading files. For instance, for use with MINOS, it writes the following files (when told to use stub ''`foo`'' in constructing file names):

Name	*Meaning*
`foo.mps`	MPS file (for linear constraints and objectives)
`foo.spc`	SPECS file (for specifying options)
`foo.row`	AMPL constraint names (for solution report)
`foo.col`	AMPL variable names (again for solution report)
`foo.nl`	nonlinear expressions (discussed below)
`foo.fix`	variables fixed by presolve
`foo.slc`	constraints eliminated by presolve
`foo.unv`	variables declared but not used
`foo.adj`	constant to be added to objective function

The MPS file encodes a description of linear constraints and objectives in a format that many linear programming solvers understand; see, e.g., [17, chapter 9].

The `.nl` file encodes nonlinear constraints and objectives in a Polish prefix form that is easy to read and manipulate. For example, for the maximal hexagon problem, the description of constraint `inscribe[1,2]`, including debugging comments that start with #, is

```
C0      #inscribe[1,2]
o54     #sumlist
3
o5      #^
v0      #rho[1]
n2
o5      #^
v1      #rho[2]
n2
o16     #-
o2      #*
o2      #*
o2      #*
n2
v0      #rho[1]
v1      #rho[2]
o46     #cos
o1      # -
v6      #theta[2]
v5      #theta[1]
```

Lines that start with `o` denote operations; such lines are followed by their operands. Lines that start with `v` are references to variables, and lines that start with `n` give numeric constants. Toward its end, the `.nl` file also tells which elements of the constraints' Jacobian matrix are nonzero and simultaneously provides coefficients for variables that appear linearly in the nonlinear constraints. A more thorough discussion of `.nl` files will have to be given in another paper.

There are at least two ways to use a `.nl` file. A solver can read it directly to construct expression graphs, and can walk the graphs to compute, e.g., function and derivative values. Alternatively, the `.nl` file can be translated to a convenient programming language, such as Fortran or C, then compiled and linked to a solver. Although this takes longer to set up, it generally leads to faster function and gradient evaluations. We report experience with both of these approaches below.

4. Automatic differentiation. Efficient nonlinear optimization routines (solvers) need ''good'' gradient approximations. Conventionally, people often find it easiest to approximate the requisite gradients by computing finite differences, but finite differences sometimes have drawbacks. Some problems are sufficiently nonlinear that straightforward finite difference approximations are too inaccurate. Some functions involve conditionals (e.g., if ... then ... else expressions) and thus have places where their gradients or even their function values are discontinuous; piecewise-linear terms, such as the absolute-value function, are perhaps the simplest examples, but much more elaborate examples also occur, such as functions whose evaluation uses adaptive quadrature or an adaptive differential equation solver. A solver that expects to deal with continuously differentiable functions may successfully cope with a problem that has discontinuities if the solution lies away from the discontinuities, but finite-difference approximations spoiled by discontinuities can lead it to fail. Finally, computation time sometimes matters, and the alternative gradient computations discussed below are sometimes much faster than finite differences.

Symbolic differentiation is a familiar technique, taught in calculus courses. Given an algebraic $f(x)$, one derives an expression for the gradient $\nabla f(x)$, which one subsequently evaluates to get numerical gradient approximations. On a computer, this yields an ''analytically correct'' approximation to $\nabla f(x)$, i.e., one that is marred, at most, by roundoff errors.

Symbolic differentiation can be useful, but it is not the only way of computing analytically correct gradient approximations. ''Automatic differentiation'' (henceforth AD) also does this job and is sometimes more convenient or more efficient than straightforward symbolic differentiation.

AD has been reinvented many times: see Griewank's survey [13].

AD proceeds by computing numerical values for the partial derivatives of each operation with respect to its operands and combining these partial derivatives by the chain rule. Two kinds of AD are of interest: ''forward'' and ''backward''. Forward AD recurs partial derivatives with respect to input variables; backward AD recurs partial derivatives (sometimes called *adjoints*) of the final result with respect to the result of each operation. With forward AD, one can recur derivatives of arbitrarily high degree in one pass. Backward AD is more complicated, in that one must propagate derivative information from the final result back to the original independent variables, one derivative at a time (as discussed below in more detail). But for computing first derivatives, backward AD is sometimes faster than forward AD (or evaluation of straightforwardly computed symbolic derivatives). Indeed, as various authors point out, with backward AD we can compute a function and its gradient in at most a multiple of the time needed to compute just the function. As Griewank [13] shows, it is easy to find examples where computing analytically correct approximations of $f(x)$ and $\nabla f(x)$ takes $O(n)$ operations if we use backward AD and $O(n^2)$ operations if we use forward AD or straightforward symbolic differentiation.

Backward AD is particularly well suited to computing the first derivatives needed by many nonlinear optimization solvers; it automatically exploits sparsity and, in my experience, is convenient and reasonably efficient. Before relating this experience, I need to describe the mechanics of backward AD in more detail.

Suppose we wish to compute $f(x)$ and $\nabla f(x)$, where $f : \mathbb{R}^n \to \mathbb{R}$. In general, we can decompose the computation of f into a sequence of k function evaluations, henceforth called *operations*, which may be elementary arithmetic operations or general function evaluations. Thus we may consider the evaluation of f to proceed as follows. For notational convenience, let $v_i = x_i$, $1 \le i \le n$. Then for $i = n + 1, \cdots, n + k$, we compute

$$v_i := o_i(v_1, v_2, \cdots, v_{i-1}),$$

which yields $f(x) = v_{n+k}$. Typical operations o_i depend on just one or two of the previously computed v_j, but sometimes it is convenient to regard as an operation a function evaluation that involves several v_j. Indeed, when computing a Jacobian matrix for constraints, two or more of which share a common subexpression, it is sometimes efficient to treat the common subexpression as a single operation; following Speelpenning [21, p. 42], we call such an operation a *funnel*. As explained below, it is convenient to treat conditional common expressions as funnels. In general, use of funnels amounts to mixing forward and backward gradient accumulation.

Consider operation o_k and suppose for some $\ell > k$ and $m > k$ that operations o_ℓ and o_m depend on the result v_k of o_k. As far as o_k is concerned, we may regard f as having the form $f(o_\ell(o_k), o_m(o_k))$; then the chain rule implies

$$\frac{\partial f}{\partial o_k} = \frac{\partial f}{\partial o_\ell} \times \frac{\partial o_\ell}{\partial o_k} + \frac{\partial f}{\partial o_m} \times \frac{\partial o_m}{\partial o_k},$$

so once we know the adjoints $\frac{\partial f}{\partial o_\ell}$ and $\frac{\partial f}{\partial o_m}$ (and the partial derivatives $\frac{\partial o_\ell}{\partial o_k}$ and $\frac{\partial o_m}{\partial o_k}$), we can compute the adjoint $\frac{\partial f}{\partial o_k}$ of o_k. Similarly, if o_k in turn depends directly on v_j, then o_k contributes $\frac{\partial f}{\partial o_k} \times \frac{\partial o_k}{\partial o_j}$ to the adjoint of o_j. Thus to propagate adjoints (derivatives), it is convenient to visit the operations in reverse order, so that when we visit an operation, we have already computed its adjoint and can compute its contribution to the adjoints of the operations on which it directly depends. Propagation of adjoints thus reduces to a sequence of multiplications and additions or assignments.

Although it is convenient to visit the operations in the reverse order of their evaluation to propagate adjoints backwards, many other orders of visitation are possible. It is only necessary that an operation have received all contributions to its adjoint before it starts contributing to the adjoints of its predecessors. Thus there is much opportunity to schedule adjoint propagation in ways that are somehow efficient, e.g., that minimize memory bank conflicts or reduce cache misses. Unfortunately, the ‘‘best’’ scheduling strategy is machine-dependent. In the computing reported below, I simply propagate adjoints in straightforward reverse order.

5. Implementation. One can arrange for several adjoints to share the same storage; ADOL-C [14], for example, does this. In hopes of gaining some speed (in trade for a relatively modest amount of memory), in the computing reported below I use a separate memory cell for each adjoint when computing the gradient of a particular constraint (or the objective); I compute the constraint gradients one at a time, and use the same adjoint scratch vector for each gradient computation. My computing involves ‘‘interpreted’’ and compiled representations of the objective and constraints. To make the interpreted version fast, one must avoid burdening it with unnecessary tests. When propagating adjoints, for example, one could arrange that the first (and often only) contribution to an adjoint be a simple assignment to the adjoint’s cell, and that only the second and subsequent contributions be additions to the value in the cell. However, this would entail distinguishing between first and subsequent contributions, which would require a test or an indexed branch. Instead, I simply zero all the adjoint cells, then add all adjoint contributions to the cells. Specifically, the adjoint propagation loop is equivalent to the C code in Fig. 3.

```
struct derp {
                struct derp *next;
                double *a, *b, *c;
                };

void derprop(register struct derp *d)
{
        *d->b = 1;
        do *d->a += *d->b * *d->c;
                while(d = d->next);
        }
```

FIG. 3. *C for propagating adjoints (derivatives).*

When reading each operation, the subroutine for reading a nonlinear expressions (`.nl`) file simply builds one `derp` structure for each multiply/add pair needed to propagate adjoints. In this structure, `a` points to the target adjoint, `b` to the source adjoint, and `c` to the relevant partial derivative value. To propagate adjoints, I use the standard library routine `memset` to zero the adjoint cells, then invoke `derprop`.

The compiled version of function and gradient evaluation largely mimics the interpreted version. In particular, it also zeros all the adjoint cells, then adds contributions to them. Of course, the compiled version would be faster if it avoided the initial adjoint zeroing and assigned rather than added the first contribution to each adjoint; it would then be reasonable for it to use the same cell for several adjoints. How much faster this would make the compiled version remains to be seen.

To save function evaluation time, I try to minimize conditional branches per arithmetic operation. Therefore I use one function invocation per elementary operation. For example, the C code for `f_OPMULT`, which implements multiplication, appears in Fig. 4.

The layout in Fig. 4 of the `expr` structure permits omitting the end of it for unary operations. The functions that implement operations all have the same calling sequence as

```
struct expr {
                double (*op)(struct expr *);
                int a;          /* subscript of adjoint */
                double dL;      /* partial w.r.t. L */
                struct expr *L, *R;     /* operands */
                double dR;      /* partial w.r.t. R */
                };

double f_OPMULT(register struct expr *e)
{
        register struct expr *e1, *e2;
        e1 = e->L;
        e2 = e->R;
        return (e->dR = e1->op(e1)) * (e->dL = e2->op(e2));
        }
```

FIG. 4. *C for multiplication.*

`f_OPMULT`: the single argument points to a structure that contains relevant information and storage for partial derivatives. Notice that the `return` line evaluates the left and right operands, stores their values in the relevant partial derivative cells, and computes the operation's value. (With some compilers, writing `f_OPMULT` this way — combining what could be written as three statements into one — may save some register loads and stores; of course, for a suitably smart optimizing compiler, this is not an issue. It appears to have little or no effect on the computing reported below.)

Some operations, such as the `min` and `max` functions and AMPL's general `if` ... `then` ... `else` ... expressions, involve conditionals. The interpreted implementations of such operations adjust pointers to insert the relevant chain of `derp` structures into the chain of `derp` structures traversed in Fig. 3. The result is the usual one for AD: away from breakpoints (i.e., points in whose every neighborhood more than one conditional outcome is possible), the computed derivative approximations are analytically correct; at breakpoints, they are analytically correct directional derivatives.

There is a complication when a conditional expression appears in a common subexpression shared by two or more constraints. To facilitate zeroing the adjoint cells, it is convenient to make them consecutive, which in general entails making copies of `derp` structures and using different adjoint cells for the common expression in different constraints. To adjust pointers efficiently for a conditional expression's adjoint propagation, however, it is desirable to have just one copy of the expression's `derp` structures and adjoint cells. A convenient way to get by with just one copy of the `derp` structures affected by a conditional common expression is to treat the expression as a funnel (i.e., as a single composite operation). This is how I handle conditional common expressions in the computing reported below.

On the one hand, if a common expression is used by several other common expressions, it may save time to treat the first common expression as a funnel — computing its gradient with respect to its inputs before propagating gradient information for the other common expressions. On the other hand, blindly following this strategy can turn an $O(n)$ computation into an $O(n^2)$ computation. If gradients were to be computed sufficiently often, it would pay to choose the funneling strategy by dynamic programming, but in general I do not know how to designate funnels in a way that minimizes overall execution time. In the computing reported below, I keep track of the inputs for each common expression, so long as there are not too many of them (no more than 5), and if two uses (in different constraints) of the expression would benefit from treating it as a funnel, then I designate it as such. Adopting this strategy led to substantial speedups for the *Chebyquad* problem discussed below.

6. Test results. Below I report timings on two machines. Bowell is a VAX 8550 with 128 megabytes of physical memory and 64 kilobytes of general cache, running a 10th Edition UNIX® system. Pyxis is an SGI 4D/240S with 128 megabytes of physical memory, two levels of cache, 25MHz MIPS R2000A/R3000 central processing units, and MIPS R2010A/R3010 floating-point units, running IRIX System V Release 3.3.2. Each primary cache (one for data, one for instructions) on Pyxis has 16384 slots, each covering four bytes, but 64 bytes are read (filling 16 slots) when a read request cannot be satisfied from the cache. (The computing reported below uses only one CPU.) Compilation for Bowell was with `-g` (debugging mode) by a combination of the native Fortran compiler, *f77* [6], and *lcc* [10], an experimental ANSI C compiler. Compilation on Pyxis was by *f2c* [5] and the native C compiler, *cc*, with `-O` (optimization enabled).

First we consider a small linearly constrained nonlinear programming problem, the *Weapon* example described in [1, chapter 2] and[15, pp. 423–425]. In AMPL, the objective is written

```
var x{target, weapon} >= 0;
minimize zot:
   sum{t in target} (u[t] * prod{w in weapon} q[t,w]^x[t,w] - u[t]);
```

Weapon involves 5 targets and 20 weapons, so there are $5 \times 20 = 100$ variables; both `u` and `q` are parameters, i.e., constants. At first glance, all 100 variables appear to be nonlinear, but for 35 (t,w) pairs the parameters `q[t,w]` are 1, so the exponentiation `q[t,w]^x[t,w]` $=$ $\mathtt{q[t,w]}^{\mathtt{x[t,w]}}$ is 1, independent of `x[t,w]`. Because the AMPL translator records these variables as linear variables, whereas they are treated nonlinearly in the version of *Weapon* distributed with MINOS, MINOS winds up computing a different sequence of iterates for the AMPL version of the problem than for the version in the MINOS distribution. Nonetheless, it is interesting to see the times taken to solve the two versions.

Table 1 reports times, ratios and counts for solving *Weapon* three ways. The lines labeled Fortran are for the version of *Weapon* that comes with MINOS, in which a Fortran subroutine computes the objective function. The other lines are for the AMPL version of *Weapon*. Those labeled m53 use the interpreted evaluation described above, and those labeled nlc use a C version of the objective function derived from the nonlinear expressions (`.nl`) file that the AMPL translator writes. The program, *nlc*, that derives C or Fortran from a `.nl` file will be described elsewhere. (Note that the *relative* columns in Table 1 give ratios of CPU seconds to Fortran CPU seconds, computed before the CPU seconds were rounded to the values in the *CPU seconds* columns.)

TABLE 1
Average times, ratios, and counts for Weapon.

	CPU seconds		relative		counts	
	`FUNOBJ`	solve	`FUNOBJ`	solve	iters	NF
Bowell (VAX 8850):						
Fortran	0.9	3.9	1.00	1.00	195	393
m53	2.3	4.9	2.51	1.25	171	323
nlc	1.6	4.2	1.74	1.07	174	335
Pyxis (SGI/MIPS):						
Fortran	0.32	1.39	1.00	1.00	195	393
m53	0.56	1.42	1.76	1.02	171	323
nlc	0.46	1.39	1.45	1.06	174	334

Table 1 suggests that interpreted evaluations are cheap enough to make them a reasonable choice for some applications. Table 1 also shows that compiled (*nlc*) evaluations can be somewhat faster than interpreted ones. The times in Table 1 are averages of 10 repetitions of solving the problem. Timing results vary from run to run on both machines, partly because of the granularity of the timer and the short execution time. The `FUNOBJ` columns are for the total time to evaluate the objective function and its gradient NF times, and the solve columns are for the total time to solve the problem, excluding the time to read the problem in and write the solution out. For the m53 rows, reading the `.nl` file took less than 2% of the solve time and less than 17% of the time to read the linear constraint information.

Much of the time to evaluate *Weapon*'s objective is spent in the library routine that computes the power function x^y. For problems that only involve elementary arithmetic operations, the ratio of interpreted to compiled evaluation times (m53 to nlc) is larger than it is for *Weapon*. For example, Table 2 shows timings for solving another of Prieto's maximal n-gon models that only involves arithmetic operations. The `FUNCON` column is for evaluating the constraints and their gradients; there are 25 nonlinear constraints and 48 variables in this problem, which corresponds to $n = 50$ — Prieto's second model reduces the problem size by assuming symmetry.

TABLE 2

Average times and ratios for Prieto's second model, n = 50.

	CPU seconds			relative		
	`FUNCON`	`FUNOBJ`	solve	`FUNCON`	`FUNOBJ`	solve
Bowell:						
m53	1.34	0.68	11.1	3.8	3.6	1.16
nlc	0.35	0.19	9.5	1.0	1.0	1.00
Pyxis:						
m53	0.36	0.24	3.32	2.0	2.4	1.12
nlc	0.18	0.10	2.96	1.0	1.0	1.00

In the nlc rows of Table 2, objective and constraint evaluations are by C derived from the `.nl` file; times for corresponding Fortran would be similar.

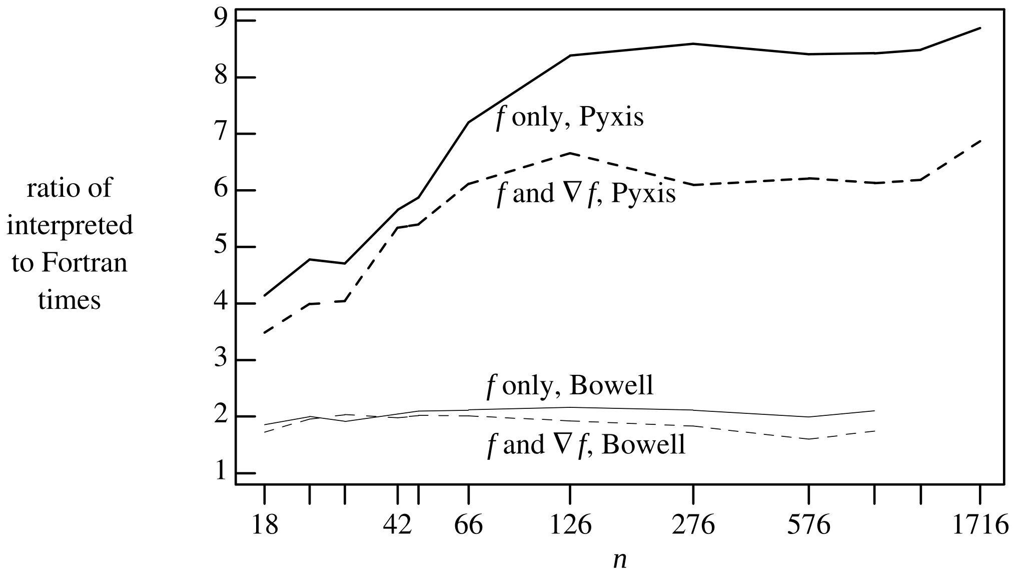


FIG. 5. *Protein-folding evaluation time ratios.*

For larger problems, the ratio of interpreted to hand-coded evaluation times can be substantially larger than Tables 1 and 2 would suggest, particularly when the interpreted evaluation suffers many cache misses. We first illustrate this with a set of protein-folding problems involving alanine polypeptides. (See [3] for a description of the general protein-folding objective function. Teresa Head-Gordon kindly provided hand-coded Fortran for this general objective and data for the particular alanine polypeptide problems; I modified the Fortran slightly to make joint computation of function and gradient more efficient.) The ‘‘real’’ problems involve 66, 126, 276, 576, 876, 1176, and 1716 variables (3 spatial coordinates per atom); to illustrate caching effects, I created similar smaller problems involving 18, 24, 30, 42, and 48 variables by arbitrarily discarding terms from the smallest ‘‘real’’ problem. Figure 5 shows ratios of interpreted to hand-coded Fortran evaluation times. Figure 6 shows that the interpreted evaluations involved substantial memory. The dashed ‘‘amax’’ line shows how many adjoint cells were used. These problems involve many named common subexpressions, called ‘‘defined variables’’ in AMPL (a recent extension). Because of funnels, there are fewer adjoint cells than `derp` structures, the number of which is shown by the solid ‘‘derps’’ line. The dotted line shows the total bytes of problem-dependent storage used. (As of this writing, *nlc* does not yet handle defined variables, so the protein-folding and *Chebyquad* test results below do not mention *nlc*.)

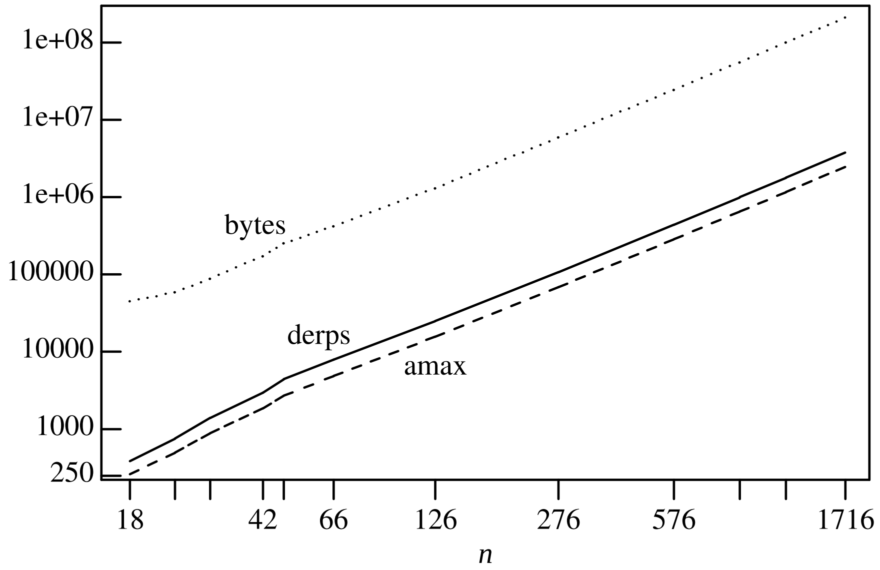


FIG. 6. *Protein folding (Pyxis) memory usage.*

In theory, backward automatic differentiation lets one compute function and gradient values in time proportional to that for computing just the function. Figure 7 illustrates that the proportionality constant is reasonably small, and for comparison, it also shows the ratio of function and gradient evaluation time to function only evaluation for the hand-coded Fortran.

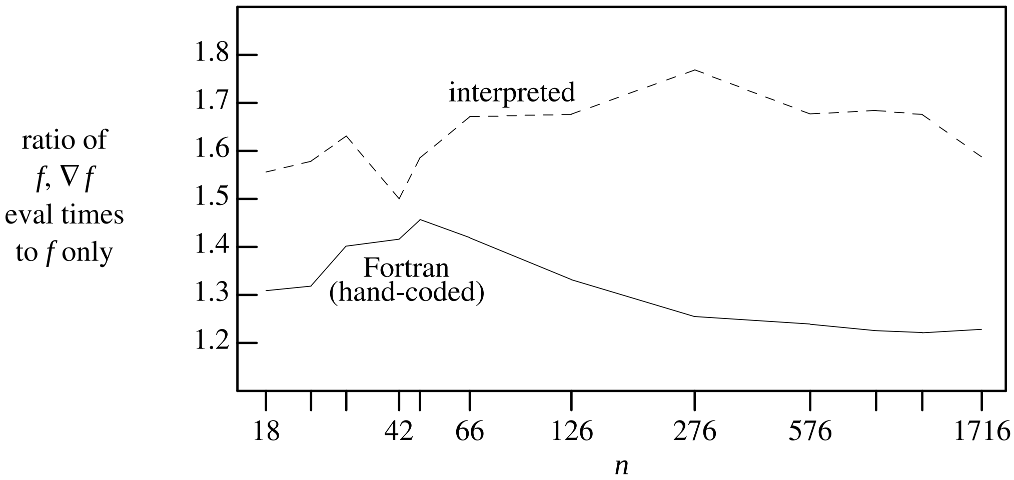


FIG. 7. *Protein folding (Pyxis): ratios of f and ∇f to f evaluation time.*

The protein-folding problems heavily involve trigonometric functions and square roots. Ratios of interpreted to hand-coded Fortran evaluation times are naturally larger for large calculations that only use elementary arithmetic. Figure 8 shows an AMPL model (courtesy of Bob Fourer) for the *Chebyquad* [7, 16] nonlinear least-squares test problem. This simple model uses defined variables `T[i,j]` to recur the value $T_i(x_j)$ of the ith Chebyshev polynomial at variable x_j = `x[j]`.

```
# ======================================
# CHEBYQUAD FUNCTION (argonne minpack #7)
# ======================================

param n > 0;

var x {j in 1..n} := j/(n+1);

var T {i in 0..n, j in 1..n} =
            if (i = 0) then 1
             else if (i = 1) then 2*x[j] - 1
              else 2 * (2*x[j]-1) * T[i-1,j] - T[i-2,j];

eqn {i in 1..n}:   (1/n) * sum {j in 1..n} T[i,j]
                 = if (i mod 2 = 0) then 1/(1-iˆ2) else 0;
```

FIG. 8. *AMPL model for Chebyquad.*

A nonlinear least-squares solver can treat the equality constraints in Figure 8 as defining the residual vector, the sum of the squares of whose components is to be minimized. For example, we have arranged for using DN2G (the variant of NL2SOL [4] that appears in the third edition of the PORT subroutine library [9]) to solve nonlinear least-squares problems expressed in AMPL. We turn Figure 8 into a general minimization problem by replacing the `eqn` constraints with an objective function:

```
minimize zot: sum{i in 1..n}
 ((1/n) * sum {j in 1..n} T[i,j] - if (i mod 2 = 0) then 1/(1-iˆ2)) ˆ 2;
```

As another illustration of using solvers with AMPL, we use SUMSOL [11] (known as `DMNG` in PORT 3) to solve unconstrained optimization problems expressed in AMPL. Figures 9 and 10 show ratios of interpreted to hand-coded Fortran evaluation times for *Chebyquad*, with *n* ranging from 5 to 50 in steps of 5. The hand-coded Fortran is that of Moré, Garbow, and Hillstrom [16].

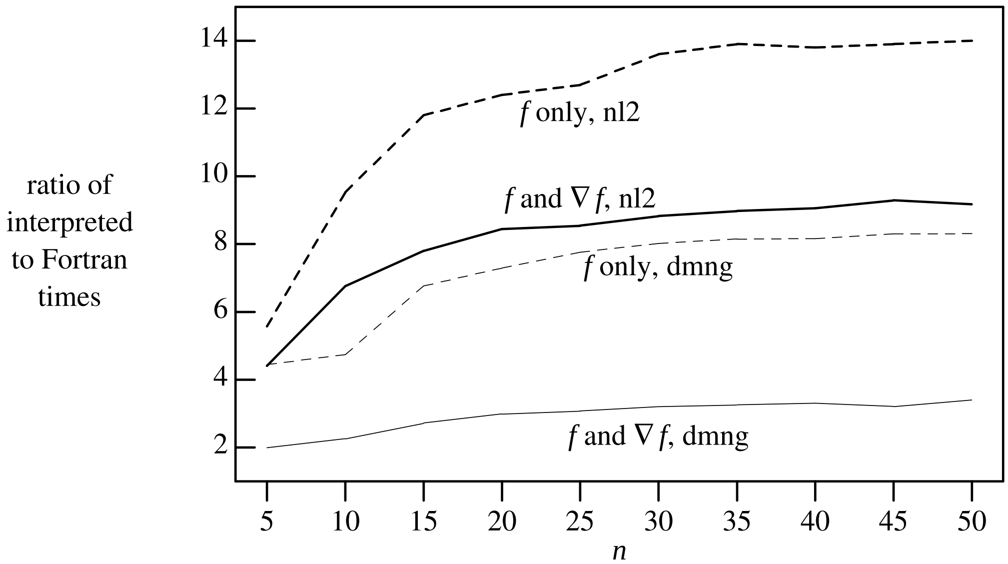


FIG. 9. *Chebyquad on Bowell.*
nl2 = nonlinear least-squares
dmng = general unconstrained minimization

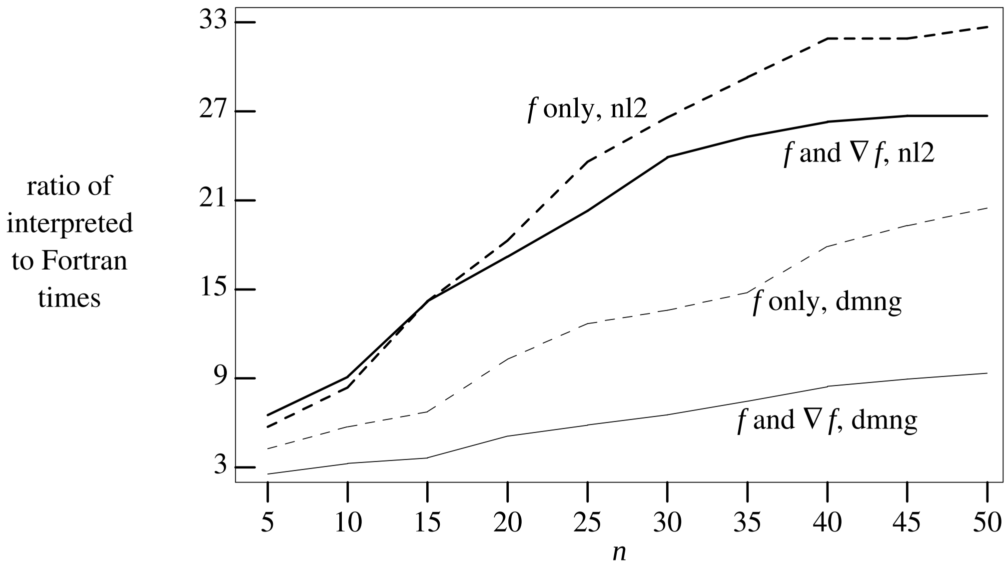


FIG. 10. *Chebyquad on Pyxis.*
nl2 = nonlinear least-squares
dmng = general unconstrained minimization

7. Conclusions. Automatic differentiation can be convenient to use in solvers for finite dimensional nonlinear optimization problems. Many such problems have a primitive recursive objective and constraints, i.e., in many such problems the objective and constraints can be evaluated by a sequence of instructions with no backward branches. If sufficient memory is available, one can use AD and the ''interpreted'' evaluation scheme sketched in §5 to compute the objective, constraints, and their analytically correct gradients, in time that may be competitive with corresponding hand-coded evaluations. Whether this yields sufficiently fast evaluations depends on the problem, machine, and one's purpose. At any rate, AD is convenient to use for exploratory modeling in conjunction with modeling languages like AMPL.

Acknowledgements. I thank Tanner Gay, Linda Kaufman, Brian Kernighan, Norm Schryer, and the referees for helpful comments on the manuscript, and Margaret Wright for information on the maximal *n*-gon example.

REFERENCES

[1] J. BRACKEN AND G. P. MCCORMICK, *Selected Applications of Nonlinear Programming,* Wiley, 1968.

[2] A. BROOK, D. KENDRICK, AND A. MEERAUS, *GAMS: A User's Guide,* Scientific Press, Redwood City, CA, 1988.

[3] 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.

[4] J. E. DENNIS, JR., D. M. GAY, AND R. E. WELSCH, ''Algorithm 573. NL2SOL—An Adaptive Nonlinear Least-Squares Algorithm,'' *ACM Trans. Math. Software* **7** (1981), pp. 369–383.

[5] 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.

[6] S. I. FELDMAN AND P. J. WEINBERGER, ''A Portable Fortran 77 Compiler,'' in *Unix Programmer's Manual, Volume II*, Holt, Rinehart and Winston (1983).

[7] R. FLETCHER, ''Function Minimization Without Evaluating Derivatives—A Review,'' *Comput. J.* **8** (1965), pp. 33–41.

[8] R. FOURER, D. M. GAY, AND B. W. KERNIGHAN, ''A Modeling Language for Mathematical Programming,'' *Management Science* **36** #5 (1990), pp. 519–554.

[9] 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.

[10] C. W. FRASER AND D. R. HANSON, ''A Retargetable Compiler for ANSI C,'' Research Report CS-TR-303-91 (Feb. 1991), Dept. of Computer Science, Princeton Univ..

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

[12] R. L. GRAHAM, ''The Largest Small Hexagon,'' *J. Combinatorial Theory (A)* **18** (1975), pp. 165–170.

[13] A. GRIEWANK, ''On Automatic Differentiation,'' pp. 83–107 in *Mathematical Programming*, ed. M. Iri and K. Tanabe, Kluwer Academic Publishers (1989).

[14] A. GRIEWANK, D. JUEDES, AND J. SRINIVASAN, ''ADOL-C, a Package for the Automatic Differentiation of Algorithms Written in C/C++,'' manuscript (Dec. 1990), Mathematics and Computer Science Div., Argonne National Laboratory.

[15] D. M. HIMMELBLAU, *Applied Nonlinear Programming,* McGraw-Hill, 1972.

[16] J. J. MORÉ, B. S. GARBOW, AND K. E. HILLSTROM, ''Algorithm 566. FORTRAN Subroutines for Testing Unconstrained Optimization Software,'' *ACM Trans. Math. Software* **7** #1 (1981), pp. 136–140.

[17] B. A. MURTAGH, in *Advanced Linear Programming: Computation and Practice*, McGraw-Hill, New York (1981).

[18] B. A. MURTAGH AND M. A. SAUNDERS, ''Large-Scale Linearly Constrained Optimization,'' *Math. Programming* **14** (1978), pp. 41–72.

[19] B. A. MURTAGH AND M. A. SAUNDERS, ''A Projected Lagrangian Algorithm and its Implementation for Sparse Nonlinear Constraints,'' *Math. Programming Study* **16** (1982), pp. 84–117.

[20] B. A. MURTAGH AND M. A. SAUNDERS, ''MINOS 5.1 User's Guide,'' Technical Report SOL 83-20R (1987), Systems Optimization Laboratory, Stanford University, Stanford, CA.

[21] B. SPEELPENNING, ''Compiling Fast Partial Derivatives of Functions Given by Algorithms,'' UIUCDCS-R-80-1002, University of Illinois, 1980.