Revisiting Expression Representations for Nonlinear AMPL Models

David M. Gay

AMPL Optimization, Inc.

dmg@ampl.com

http://ampl.com

ABSTRACT

AMPL facilitates stating and solving nonlinear programming problems involving algebraically defined objectives and constraints. For solving such problems, the AMPL/solver interface library provides routines that compute objective functions, constraint residuals, and associated derivatives. Objectives and constraint bodies hitherto have been represented by ''executable'' expression graphs, in which each node points to its operands and to a function that computes the node's result. Nodes also store partial derivatives for use in computing gradients and Hessians by automatic differentiation. Storing these values makes the graphs nonreentrant. To enable several threads to evaluate the same expression at different points without having separate copies of the expression graphs, such details as variable values and partial derivatives must be stored in thread-specific arrays. We describe and compare some expression-graph representations for use in computing function, gradient, and Hessian values, and for extracting some auxiliary problem information. In particular, we describe some details of an updated AMPL/solver interface library that uses operation lists to represent expressions.

1. Introduction

The AMPL modeling language [7, 8] facilitates formulating, instantiating, solving, and examining solutions of mathematical programming problems, such as

$$\text{minimize} \quad f(x) \tag{1a}$$

$$\text{s.t.} \quad \ell \le c(x) \le u, \tag{1b}$$

with $x \in \mathbb{R}^n$ and $c\colon \mathbb{R}^n \to \mathbb{R}^m$, possibly with some components of x restricted to integer values. (If there are no constraints, then $m = 0$. When $m > 0$, the ith lower bound ℓ_i can be finite or $-\infty$, i.e., $\ell_i \in \{-\infty\} \cup \mathbb{R}$, and similarly $u_i \in \{+\infty\} \cup \mathbb{R}$, with $\ell_i = u_i \in \mathbb{R}$ if constraint i is an equality constraint.)

While AMPL has ''presolve'' facilities [6] for simplifying (1), and these simplifications sometimes find solutions, normally AMPL does not solve problems, but relies on separate *solvers* to find solutions. To use such a solver, AMPL writes a representation of the current problem instance to a ''.nl file'' and invokes the solver as a separate program. The solver typically uses facilities in a special library, the AMPL/solver interface library (ASL), to read the .nl file and acquire various problem details. When f or c is nonlinear, the solver calls ASL routines to compute function values $f(x)$ and $c(x)$ at a specified

vector $x \in \mathbb{R}^n$. Most nonlinear solvers also obtain gradient $\nabla f(x)$ and Jacobian $\nabla c(x)$ values from the ASL; some also use the ASL to obtain values of the Hessian of the Lagrangian function,

$$\nabla^2 L(x,y) = \nabla^2 f(x) + \sum_{i=1}^{m} y_i \nabla^2 c_i(x) \tag{2}$$

or Hessian-vector products $\nabla^2 L(x,y) \cdot v$ for specified vectors $v \in \mathbb{R}^n$. This paper explains how the ASL has computed these values and presents an alternative way to compute them. The goal of the present work is to revisit expression representations and evaluations in the ASL with an eye to separating expressions from data so multiple threads can make independent use of the same expressions.

For concreteness, let us consider a tiny problem with $n = 2$ and $m = 1$:

$$\text{minimize } f(x) = (x_1 - 3)^2 + (x_2 + 4)^2 \tag{3}$$

$$\text{s.t. } c(x) = x_1 + x_2 = 1.$$

An AMPL script for stating and solving this problem is

```
var x; var y;
minimize f: (x - 3)^2 + (y + 4)^2;
s.t. c: x + y == 1;
solve;
display x, y;
```

Putting this script into file `tiny.x` and invoking ‘‘`ampl tiny.x`’’, we get

```
MINOS 5.51: optimal solution found.
2 iterations, objective 2
Nonlin evals: obj = 6, grad = 5.
x = 4
y = -3
```

To process the ‘‘solve’’ command above, AMPL writes a .nl file containing

- problem statistics (number of variables, etc.)
- expression graphs for nonlinear parts of objectives and constraints
- linear parts of objectives and constraints
- starting guesses (if specified)
- suffixes, e.g., for a basis (if available)

2. Representations of Expression Graphs

There are various ways to represent expression graphs. The following four ways are roughly equivalent in size and evaluation time.

1. Polish postfix: operands are pushed onto a stack, and operators remove operands from the top of the stack, compute a result, and push the result onto the stack top. Thus

operators follow operands. For a while in the late twentieth century, Hewlett Packard sold hand calculators that worked this way. For example, on an HP 15C calculator, the computation of $3 \times 4 + 5 = 17$ proceeds as follows:

keystroke	*display*
3	3
Enter	3.00
4	4
×	12.00
5	5
+	17.00

Some interpreters for the Pascal programming language, e.g., [3], also used Polish postfix for their ‘‘compiled’’ program representations.

2. Polish prefix: operators are followed by expressions for their operands. The ASL has long worked this way [9]. For example, the expression graph for (3) could be represented visually by Figure 1.

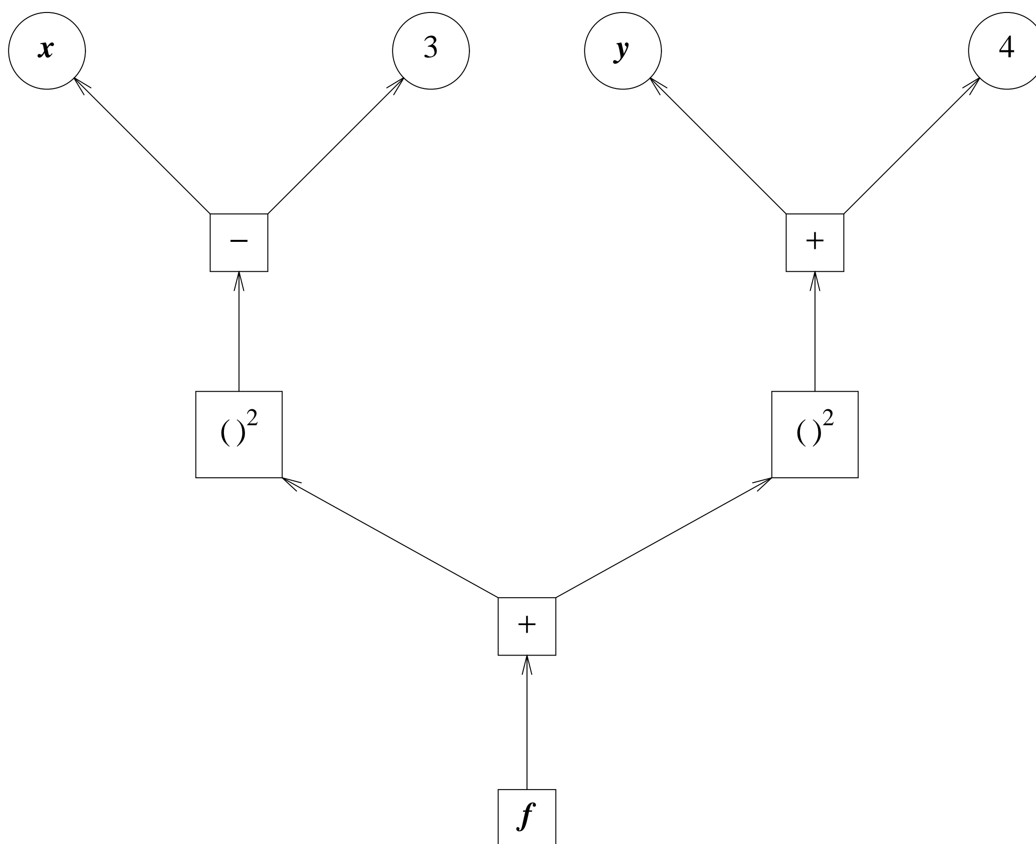


Figure 1. Expression graph for (3).

Inserting the lines

```
option nl_comments 1;
write gtiny;
```

before the ‘‘solve;’’ line in the ‘‘tiny.x’’ file shown above would cause AMPL to write file ‘‘tiny.nl’’ containing the following lines to represent (3):

```
O0 0#f
o0  # +
o5  #^
o0  # +
n-3
v0  #x
n2
o5  #^
o0  # +
n4
v1  #y
n2
```

The portion of each line starting with ‘‘#’’ is a comment that is only present when ‘‘option nl_comments 1’’ is in effect. For instance, the lines

```
o0  # +
n4
v1  #y
```

represent $y + 4$, and the lines

```
o5  #^
o0  # +
n4
v1  #y
n2
```

represent $(y + 4)^2$.

3. Executable expression graphs: the ASL has hitherto represented each operation by a data structure that includes pointers to operands and a pointer to a function that carries out the operation. For a binary operation, the data structure has the form

```
struct expr {
        real (*op)(struct expr*);
        int a;
        real dL;
        struct expr *L, *R;
        real dR;
        };
```

Figure 2. Old ASL binary expression for gradients only.

in the C programming language. The ‘‘left’’ and ‘‘right’’ operands are `L` and `R`, and the `dL` and `dR` fields are for the partial derivatives of the operation’s result with respect to the left and right operands. These fields make the code nonreentrant. (The ‘‘a’’ field is an ‘‘adjoint subscript’’ used when setting up derivative computations. Here and below, we assume ‘‘typedef double real;’’ has appeared.) For example, C source for the ‘‘op’’ function for division, prepared for function and gradient evaluations, is

```
real f_OPDIV(expr *e) {
        real L, R, rv;
        expr *e1 = e->L;
        L = (*e1->op)(e1);
        e1 = e->R;
        if (!(R = (*e1->op)(e1)))
                zero_div(L, "/"); /*no return*/
        rv = L / R;
        if (want_deriv)
                e->dR = -rv * (e->dL = 1. / R);
        return rv;
        }
```

Partial derivatives are not always needed. For instance, partials are not needed in computing the *test* part of an ‘‘if *test* then *texpr* else *fexpr*’’ expression. In the `f_OPDIV` source above, partials are only computed when `want_deriv` is true, in which case the line after ‘‘`if (want_deriv)`’’ computes both `dL` and `dR` values.

4. Operation list: a list, similar in spirit to computer machine instructions, of operators, input operands, and output result locations is sometimes useful. For example, Kearfott’s GlobSol solver [14] uses a list of quadruples of integers (called a ‘‘codelist’’). To convey the general idea, here is a list of instructions for computing (3), using a scratch array `w` initialized with `w[0]` = `x` and `w[1]` = `y`.

```
w[2] = w[0] - 3;    /* x - 3 */
w[2] = w[2] * w[2];
w[3] = w[1] + 4;    /* y + 4 */
w[3] = w[3] * w[3];
w[2] = w[2] + w[3];
```

The computation ends with $f(x)$ = `w[2]`.

The present work uses a list of tuples of varying lengths to represent operations and operands. For example, a unary operation is represented by three integers, a binary operation by four integers, and a sum of n terms by $n + 3$ integers. A ‘‘big switch’’, which a good compiler will turn into a jump table, determines the operation carried out:

```
real eval1(int *o, EvalWorkspace *ew) {
        real *w = ew->w;
 top:   switch(*o) {
          case nOPRET:
                return w[o[1]];
          case nOPUMINUS:
                w[o[1]] = -w[o[2]];
                o += 3;  goto top;
          case nOPPLUS:
                w[o[1]] = w[o[2]] + w[o[3]];
                o += 4;  goto top;
          case nOPMINUS:
                w[o[1]] = w[o[2]] - w[o[3]];
                o += 4;  goto top;
          case nOPMULT:
                w[o[1]] = w[o[2]] * w[o[3]];
                o += 4;  goto top;
        ...
```

The above `eval1(...)` routine has a separate work array, w, for computed results. The executable expression graph approach could also be modified to use a separate work array for computed results, but timing results shown below indicate it may be better to use operation lists.

3. Gradient Computations

It is convenient and efficient to use automatic differentiation (AD) to compute gradients, and possibly Hessians (2) or Hessian-vector products, related to (1). AD has been addressed by many papers and books; see, for example, the book [12] by Griewank and Walther, the publications cited therein, and those listed in the web site `http://www.autodiff.org`. The basis for AD is simply the chain rule. Suppose t is a variable whose role in (1) can be described as

$$\phi(t) = f(y_1(t), y_2(t), \ldots, y_k(t));$$

t might be a program variable x_i or an intermediate value involved in (1). The notion here is that t is directly involved in the computation of possibly intermediate values $y_1, \ldots, y_k$. The chain rule gives

$$\phi'(t) = \frac{\partial f}{\partial t} = \sum_{i=1}^{k} \frac{\partial f}{\partial y_i} \frac{\partial y_i}{\partial t}.$$

Once we know the *adjoint* $\frac{\partial f}{\partial y}$ of an intermediate variable y, we can add its contribution $\frac{\partial f}{\partial y} \frac{\partial y}{\partial t}$ to the adjoint of each variable t on which y depends directly. So-called backwards AD or reverse AD proceeds by computing adjoints in the reverse order of the

operations to compute $f(x)$. Since the ith component of the gradient $\nabla f(x)$ is the adjoint of x_i, reverse AD computes $\nabla f(x)$ in a number of operations proportional to the number needed to compute $f(x)$ itself, which makes reverse AD very appealing for computing gradients, at least for computations simple enough that relevant intermediate results can be stored.

As described in [9], the ASL has long used reverse AD to compute gradients by means of ''derivative propagation'' structures

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

in which a and b point to adjoints and c points to a previously computed partial derivative value. The reverse AD computation proceeds via

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

Would it be faster to implement reverse AD another way? To see how using integer subscripts rather than pointers would perform, we now consider three ways of computing an inner product, based on the following structures:

```
struct Rpair { double a, b; } *rp;
struct Aoff { double *a, *b; } *p;
struct Ioff { int a, b; } *q;
```

With Rpair, the values to be multiplied are in the structure itself. With Aoff, pointers to those values are in the structure. With Ioff, an auxiliary array, declared by

```
double *v;
```

contains those values. The basic operations for computing an inner product ''dot'' are

```
dot += rp->a * rp->b;      /* Rpair */
dot += *p->a * *p->b;      /* Aoff  */
dot += v[p->a] * v[p->b];  /* Ioff  */
```

In computing on a laptop computer with an Intel Celeron CPU, sequential memory accesses are faster than randomly permuted ones, as indicated in the following table of relative computation times for some inner products, with both 32- and 64-bit addressing:

	32-bit	64-bit
Rpair	1.0	1.0
Aoff sequential	1.0	1.0
Ioff sequential	1.0	1.0
Aoff permuted	1.6	1.8
Ioff permuted	1.6	1.7

Table 1. *Relative times for ‘‘dot’’ variants.*

On this machine, at least, there is no disadvantage to using integer subscripts. For separating data fixed by the problem instance from data that depend on current variable values, it is convenient to use integer subscripts into a thread-specific work array, and the above table suggests that doing so may not adversely affect single-thread computations, at least not greatly.

4. Alternative Implementations of `derprop`

For computing gradients in a thread-safe way, i.e., with separate arrays for values that vary with the thread, we could use a computation analogous to the `derprop` routine shown above:

```
struct iderp { int a, b, c; } *d, *de;
for(d = ...; d < de; ++d)
        s[d->a] += s[d->b] * w[d->c];
```

However, this has the disadvantage of using a potentially large `s` array, many components of which must be initialized to zero. One possible alternative is to again use a ‘‘big switch’’, such as

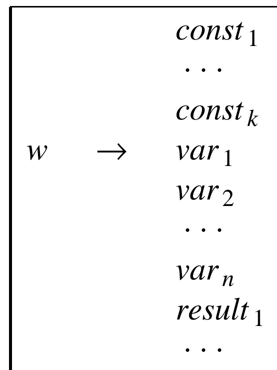
```
for(;;)
  switch(*u) {
    case ASL_derp_copy:      s[u[1]] = s[u[2]];
         u += 3;  break;
    case ASL_derp_add:       s[u[1]] += s[u[2]];
         u += 3;  break;
    case ASL_derp_copyneg:  s[u[1]] = -s[u[2]];
         u += 3;  break;
    case ASL_derp_addneg:   s[u[1]] -= s[u[2]];
         u += 3;  break;
    case ASL_derp_copymult: s[u[1]] = s[u[2]]*w[u[3]];
         u += 4;  break;
    case ASL_derp_addmult:  s[u[1]] += s[u[2]]*w[u[3]];
         u += 4;  break;
    ...
```

It is simpler to exploit the fact that, aside from ‘‘defined variables’’ (which amount to named common subexpressions and are discussed more in §5 below), each intermediate

value is used just once, which allows us to use a loop of the form

```
for(d = ...; d < de; ++d) {
        t = s[d->b] * w[d->c];
        if ((a = d->a) >= a0)
                s[a] = t;
        else
                s[a] += t;
        }
```

In this loop, components `s[`i`]` with i < `a0` correspond to decision variables x_i or to defined variables; such components must be initialized suitably, e.g., with linear coefficients in the case of decision variables. There is no need to initialize the other components of `s`, and the `s` array can be much shorter. The `w` array contains known constants, variable values, and computed results, including (where appropriate) partial derivatives, with a layout of the form



so that `w[0]` is the value of the first decision variable and negative subscripts are for constants.

5. Funneling Defined Variables

AMPL permits declaring a ‘‘variable’’ whose value is computed from an expression involving other variables. This is logically equivalent to introducing a new variable and an equality constraint that specifies the new variable’s value, except that some solvers may arrange for equality constraints only to be satisfied in the limit, whereas defined variables always have exactly their specified value (except for roundoff errors). Some problems that would otherwise involve equality constraints may be stated as unconstrained problems with the help of defined variables, making a wider range of solvers available to solve such problems.

When a defined variable is used in several constraints or objectives, and the expression for the defined variable is sufficiently complicated, it can be worthwhile to precompute the partials of the defined variable with respect to the variables on which it depends. This is sometimes called *funneling* the gradient computation. The current heuristic is to funnel μ adjoint operations to k variables when $\mu > 3k$, since with no funnel we would do at least 2μ adjoint-propagation operations (as the defined variable is shared by at least

two constraints or objectives); creating the funnel costs about $\mu + k$ adjoint operations, and applying it twice costs $2k$ more such operations. We only want to do this when it saves time, i.e., when $2\mu > \mu + 3k$.

The MINPACK [16] test problem ''Chebyquad'', as given in the following AMPL model, provides an example where funnels are worthwhile:

```
# chb50b.mod: MINPACK Chebyquad 50 as
# both objective and constraints
param n > 0 default 50;
var x {j in 1..n} := j/(n+1);
var Tj{j in 1..n} = 2*x[j] - 1;
var T{i in 0..n, j in 1..n} =
        if (i = 0) then 1
        else if (i = 1) then Tj[j]
        else 2 * Tj[j] * T[i-1,j] - T[i-2,j];
minimize ssq: sum{i in 1..n} ((1/n) * sum {j in 1..n} T[i,j]
                          - if (i mod 2 = 0) then 1/(1-i^2))^2;
s.t. eqn {i in 1..n}:
    (1/n) * sum{j in 1..n} T[i,j] =
          if (i mod 2 = 0) then 1/(1-i^2) else 0;
```

In this model, `T[`i`,`j`]` is the ith Cheybshev polynomial evaluated at variable `x[`j`]`, and `Tj` is a helper defined variable.

The following table shows evaluation times for the above model with some ASL variants, some without funneling, relative to evaluation time for ''old ASL'', which is the ASL that has long been available; here and below, unless otherwise noted, timing is on the same machine used for Table 1. The ''bad ASL with funnels'' variant was my initial attempt to improve funneling by using data structures that I thought would take less memory. The ''new ASL'' variant uses an operation list, but the same funneling approach as the old ASL.

	f, ∇f		c, ∇c	
ASL variant	32-bit	64-bit	32-bit	64-bit
old ASL with funnels	1.00	1.00	1.00	1.00
old ASL without funnels	1.68	1.14	1.62	1.15
bad ASL with funnels	1.01	0.67	6.47	3.68
new ASL with funnels	0.40	0.28	0.39	0.28
new ASL without funnels	0.45	0.29	0.44	0.29

Table 2. *Relative evaluation times for* `ch50b` *with ASL variants.*

Using funnels sometimes gives faster gradient computations. It remains to be seen whether other ways of handling funnels would sometimes give still faster gradient computations.

6. Detecting and Extracting Quadratic Forms

Some solvers provide special treatment for quadratic objectives and possibly quadratic constraints. With the old ASL, determining whether an objective or constraint is quadratic and, if so, extracting its Hessian matrix, is done with `qpcheck()` functions that require calling a special `qp_read()` routine to read the `.nl` file. To compute the values of nonlinear, nonquadratic objectives or constraints, it is then necessary to call a `qp_opify()` routine, and to compute Hessians or Hessian-vector products, it is necessary to completely re-read the `.nl` file. With the new ASL, the `qpcheck()` routines operate directly on operation lists, carrying out an ‘‘evaluation’’ that computes expression information rather than numeric values, so `qp_opify()` is not needed and Hessians or Hessian-vector products can be computed without re-reading the .nl file. Similar comments apply to the `indicator_constrs()` routine that helps process indicator constraints: the old ASL required calling `qp_read`, and the new ASL does not.

7. Hessians and Hessian-vector Products

For computing Hessians, an approach described by Bruce Christianson in [4] works well. Let $p \in \mathbb{R}^n$ be nonzero and consider

$$\phi(\tau) = f(x + \tau p).$$

Then

$$\phi'(\tau) = \nabla f(x + \tau p)^{\mathrm{T}} p. \tag{4}$$

If we compute $\phi'(0)$ by forward AD, then we can use reverse AD to compute $\nabla^2 f(x) p$, i.e., a Hessian-vector product. Solvers that use a (possibly preconditioned) nonlinear conjugate-gradient algorithm can use Hessian-vector products directly. For solvers that use explicit (possibly sparse) Hessian matrices, we can use Hessian-vector products to assemble the Hessian matrix.

Griewank and Toint [11] point out that many objectives $f(x)$ have the form

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

in which each matrix $U_i \in \mathbb{R}^{m_i \times n}$ has only a few rows (i.e., $m_i << n$). For such a function f,

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

and

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

In LANCELOT [5], Conn, Gould and Toint exploit ‘‘group partially separable’’ structure:

$$f(x) \;=\; \sum_{i=1}^{q} \theta_i \Big(\sum_{j=1}^{k_i} f_{ij}(U_{ij} x) \Big)$$

in which $\theta_i()$ is a unary function. For such f, if $\psi_i(x) = \sum_{j=1}^{k_i} f_{ij}(U_{ij} x)$, then

$$\nabla f(x) \;=\; \sum_{i=1}^{q} \theta_i'(\psi_i(x)) \sum_{j=1}^{k_i} U_{ij}^{\mathrm{T}} \nabla f_{ij}(U_{ij} x)$$

and

$$\nabla^2 f(x) \;=\; \sum_{i=1}^{q} \Big\{ \quad \theta_i'(\psi_i(x)) \sum_{j=1}^{k_i} U_{ij}^{\mathrm{T}} \nabla^2 f_{ij}(U_{ij} x)\, U_{ij} + \theta_i''(\psi_i(x)) \sum_{j=1}^{k_i} (U_{ij}^{\mathrm{T}} \nabla f_{ij}(U_{ij} x))(U_{ij}^{\mathrm{T}} \nabla f_{ij}(U_{ij} x))^{\mathrm{T}} \;\Big\}.$$

By using a suitable expression-graph walk [10], we can find the structure (5) automatically. This graph walk is fairly elaborate, so the new ASL initially keeps the same expression graph representation as the old ASL. After the structure (5) is found, the new ASL does another graph walk to produce the operation lists used for evaluations. With both ASL versions, the net effect is that AMPL users can exploit the structure (5) without even being aware of it.

An example with a rich structure (5) is an empirical energy function for protein folding. In connection with [13], Teresa Head-Gordon provided us with Fortran for the CHARM empirical energy function, and we converted it to an AMPL model, `pfold.mod`, of which the following is an excerpt:

```
# CHARM empirical energy function, derived
# from Fortran supplied by Teresa Head-Gordon.
set D3 circular := 1..3;
set Atoms; var x{i in Atoms, j in D3};

set Bonds;
param ib{Bonds} integer;
param jb{Bonds} integer;
param fcb{Bonds}; param b0{Bonds};

var bond_energy = sum{i in Bonds} fcb[i] *
 (sqrt(sum{j in D3} (x[ib[i],j] - x[jb[i],j]})^2) - b0[i])^2;
# ...
minimize energy: bond_energy + angle_energy + torsion_energy
             + improper_energy + pair14_energy + pair_energy;
```

The decision variables are the Cartesian coordinates of a collection of atoms. The overall empirical energy is the sum of six energy terms, each a sum of nonlinear expressions involving differences of the coordinates of the atoms involved. Thus each $U_{ij} x$ is a

vector of coordinate differences.

8. Implementations of Hessian-vector Products

When arranging to compute Hessians and Hessian-vector products, the old ASL used a more detailed version of the `expr` structure shown in Figure 1:

```
struct expr2 {
   real (*op)(expr2 *);
   int a;         /* adjoint index, then operator class */
   expr2 *fwd, *bak;
   real dO;       /* deriv of op w.r.t. t in x + t*p */
   real aO;       /* adjoint (in Hv computation) of op */
   real adO;      /* adjoint (in Hv computation) of dO */
   real dL;       /* deriv of op w.r.t. left operand */
   expr2 *L, *R; /* left and right operands */
   real dR;       /* deriv of op w.r.t. right operand */
   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 3. Old ASL binary expression for gradients and Hessians.

The `fwd` and `bak` pointers enable computing (4), i.e., $\phi'(\tau)$, and applying reverse AD to this computation. In the old ASL, computing (4) used code of the form

```
void hv_fwd(expr *e) {
for(; e; e = e->fwd) {
    e->aO = e->adO = 0;
    switch(e->a) {
        ...
        case Hv_binaryLR:
            e->dO = e->L->dO*e->dL + e->R->dO*e->dR;
            break;
        case Hv_minusR:
            e->dO = -e->R->dO;
            break;
        ...
        }}}
```

Figure 4. Old ASL forward computation of ϕ'.

This code uses partial derivatives stored in the `expr2` structure shown in Figure 3, making the code nonreentrant. The new ASL stores thread-dependent quantities in a separate workspace, some of which is composed of structures of the form

```
struct Eresult {
  real O;   /* op value */
  real dO;  /* deriv of op w.r.t. t in x + t*p */
  real aO;  /* adjoint (in Hv computation) of O */
  real adO; /* adjoint (in Hv computation) of dO */
  real dL;  /* deriv of op w.r.t. left operand L */
  real dL2; /* second partial w.r.t. L,L (R,R for OPDIV01) */
  real dR;  /* deriv of op w.r.t. right operand R */
  real dLR; /* second partial w.r.t. L,R */
  real dR2; /* second partial w.r.t. R,R */
  };
```

In the new ASL, computing (4) uses code of the form

```
    void hv_fwd(int *o, real *w, ...) { ...
    for(;;) {
        switch(*o) { ...
            case nOPDIV2:
                r = (Eresult*)(w + o[2]);
                L = (Eresult*)(w + o[3]);
                R = (Eresult*)(w + o[4]);
                r->dO = L->dO*r->dL + R->dO*r->dR;
                o += 5;
                break;
            ... }
            r->aO = r->adO = 0.;
            }}
```

Figure 5. New ASL forward computation of ϕ'.

The assignment ``e = e->fwd'' in Figure 4, to move forward to the next relevant operation, is replaced in Figure 5 by the assignment ``o += 5'' (i.e., ``o = o + 5''). The assignment ``r->aO = r->adO = 0.'' after the switch statement in Figure 5 is an initialization in preparation for the following reverse AD on ϕ'.

In the old ASL, applying reverse AD to (4) used code of the form

```
void hv_back(expr *e) { ...
for(; e; e = e->bak) {
    switch(e->a) { ...
        case Hv_binaryLR:
            e1 = e->L;
            e2 = e->R;
            adO = e->adO;
            t1 = adO * e1->dO;
            t2 = adO * e2->dO;
            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;
     ... }}}
```

Figure 6. Old ASL reverse AD of ϕ'.

In the new ASL, the corresponding code has the form

```
void hv_back(int *o, real *w) { ...
for(;;) {
    switch(o[0]) { ...
        case nOPPOW2: case nOP_atan22:
            r = (Eresult*)(w + o[2]);
            L = (Eresult*)(w + o[3]);
            R = (Eresult*)(w + o[4]);
            L->adO += r->adO * r->dL;
            R->adO += r->adO * r->dR;
            t1 = r->adO * L->dO;  t2 = r->adO * R->dO;
            L->aO  += r->aO*r->dL + t1*r->dL2 + t2*r->dLR;
            R->aO  += r->aO*r->dR + t1*r->dLR + t2*r->dR2;
            break; ...}
     o -= o[1];
     }}
```

Figure 7. New ASL reverse AD of ϕ'.

The assignment ‘‘`o -= o[1]`’’ in Figure 7 corresponds to ‘‘`e = e->bak`’’ in Figure 6.

9. Comparative Timings

The tables below present some timings of the new ASL relative to the old on some test problems summarized in Table 3. Problems *bearing*, *clnlbeam*, *gasoil*, and *henon80* were provided by Hans Mittelmann [15]; problem *denhex* is from the AMPL model

```
# denhex.mod: dense Hessian
# expressed without structure
param n integer > 0 default 500;
set I = 1 .. n;
var x{I};
minimize q:   sum{i in I} x[i]^2
            + sum{i in I, j in I} i*j*x[i]*x[j];
```

problem *denhop* is from the AMPL model

```
# denhop.mod: dense, structured Hessian
param n integer > 0 default 5000;
set I = 1 .. n;
var x{I};
minimize q:    sum{i in I} x[i]^2
            + (sum{i in I} i*x[i])^2;
```

problem *pfold3* is the `pfold.mod` mentioned in §7, together with data connected with [13]; problem *chemeq* is from `chemeq.mod` in [1]; problem *ch50b* corresponds to the `ch50b.mod` shown above, and *ch50* is `ch50b.mod` with just the least-squares objective (no constraints).

name	*n*	*m*	*comments*
			n = number of variables
			m = number of constraints
bearing	16000	0	sparse quadratic objective
ch50	50	0	many defined variables
ch50b	50	50	many defined variables
chemeq	38	12	nonlinear objective, linear constraints
clnlbeam	59999	40000	all nonlinear
denex	500	0	dense quadratic, all explicit
denop	5000	0	quadratic, dense due to outer-product
gasoil	32001	31998	quadratic objective, nonlinear constraints
henon80	21601	161	linear objective, 80 quadratic constraints
			81 more nonlinear constraints
pfold3	66	0	many defined variables

Table 3. Test problems.

Problems *denex* and *denop* have dense Hessians and are for timing Hessian-vector products. The other problems have sparse Hessians, and the timings are for computing them explicitly.

Table 4 shows relative times: new ASL time divided by old ASL time, for computing function and gradient values (‘‘f, ∇f’’ indicates function and gradient values for the objective, and ‘‘c, ∇c’’ indicates corresponding values for the constraints). The ‘‘no Hes.’’ results only involve computing and storing first partial derivatives, whereas the

‘‘Hes.’’ results include time for computing (but not using) second partial derivatives. With both old and new ASL, possibly useful partial derivatives are computed during function evaluations and are stored for possible later use in computing gradients and Hessians or Hessian-vector products.

		32-bit no Hes.	64-bit no Hes.	32-bit Hes.	64-bit Hes.
bearing	$f, \nabla f$	0.57	0.41	0.41	0.42
ch50	$f, \nabla f$	0.62	0.52	0.43	0.28
ch50b	$f, \nabla f$	0.92	0.49	0.40	0.28
ch50b	$c, \nabla c$	0.11	0.08	0.40	0.28
chemeq	$f, \nabla f$	0.68	0.67	0.77	0.85
clnlbeam	$f, \nabla f$	0.63	0.55	0.74	0.68
clnlbeam	$c, \nabla c$	0.23	0.23	0.77	0.64
denex	$f, \nabla f$	0.56	0.43	0.15	0.21
denop	$f, \nabla f$	0.53	0.48	0.64	0.54
gasoil	$f, \nabla f$	0.56	0.39	0.28	0.27
gasoil	$c, \nabla c$	0.48	0.36	0.82	0.68
henon80	$c, \nabla c$	0.11	0.12	0.58	0.45
pfold3	$f, \nabla f$	0.74	0.64	0.73	0.74

Table 4. Relative function and gradient times: new ASL divided by old, computed on an Intel Celeron CPU with 2048 KB of cache.

Cache size and various CPU details may affect the results in Table 4. The Intel Celeron CPU used for Table 4 has 2048 KB of cache. Tables 5 and 6 are similar to Table 4, and were computed using the same binaries (compiled by gcc with –O2), but for different CPUs. The results in Table 5 were computed on an Intel Core 2 Quad CPU with 4096 KB of cache, and those in Table 6 were computed on an Intel Core i7-4700MQ CPU with 6144 KB of cache.

Despite the differences among the CPUs considered here, the general trend is that the new ASL, with its use of operation lists, often runs faster for function and gradient evaluations than does the old ASL. Moreover, the new ASL often takes less memory to represent nonlinear expressions when just one thread is used, as indicated by Table 7 below, which shows ratios of net memory use for the new ASL relative to the old.

The figures in Table 7 include memory for the operation lists as well as for one thread-specific work array. Each additional thread just needs its own work array, so substantial memory savings are often possible on large problems when multiple threads are used. Allocation of large memory blocks may cause small problems like *chemeq* to take more memory for one thread, but even small problems generally take less additional memory for each additional thread. For example, with *chemeq*, each additional thread requires less than a quarter of the memory needed by the old ASL.

		32-bit no Hes.	64-bit no Hes.	32-bit Hes.	64-bit Hes.
bearing	$f, \nabla f$	0.35	0.29	0.40	0.36
ch50	$f, \nabla f$	0.75	0.73	0.69	0.36
ch50b	$f, \nabla f$	0.89	0.78	0.47	0.29
ch50b	$c, \nabla c$	0.12	0.11	0.48	0.30
chemeq	$f, \nabla f$	0.85	0.84	0.86	0.89
clnlbeam	$f, \nabla f$	0.57	0.46	0.42	0.42
clnlbeam	$c, \nabla c$	0.23	0.20	0.71	0.65
denex	$f, \nabla f$	0.32	0.25	0.30	0.25
denop	$f, \nabla f$	0.58	0.61	0.73	0.74
gasoil	$f, \nabla f$	0.72	0.54	0.31	0.29
gasoil	$c, \nabla c$	0.37	0.32	0.63	0.57
henon80	$c, \nabla c$	0.13	0.10	0.44	0.33
pfold3	$f, \nabla f$	0.78	0.79	0.84	1.07

Table 5. Relative function and gradient times: new ASL divided by old, computed on an Intel Core 2 Quad CPU with 4096 KB of cache.

		32-bit no Hes.	64-bit no Hes.	32-bit Hes.	64-bit Hes.
bearing	$f, \nabla f$	0.44	0.29	0.39	0.29
ch50	$f, \nabla f$	0.58	0.51	0.48	0.19
ch50b	$f, \nabla f$	0.78	0.58	0.38	0.17
ch50b	$c, \nabla c$	0.12	0.10	0.41	0.18
chemeq	$f, \nabla f$	0.63	0.64	0.81	0.89
clnlbeam	$f, \nabla f$	0.55	0.45	0.55	0.41
clnlbeam	$c, \nabla c$	0.25	0.21	0.62	0.53
denex	$f, \nabla f$	0.49	0.32	0.15	0.14
denop	$f, \nabla f$	0.51	0.48	0.66	0.64
gasoil	$f, \nabla f$	0.49	0.43	0.22	0.23
gasoil	$c, \nabla c$	0.37	0.27	0.76	0.62
henon80	$c, \nabla c$	0.12	0.11	0.50	0.36
pfold3	$f, \nabla f$	0.63	0.54	0.71	0.75

Table 6. Relative function and gradient times: new ASL divided by old, computed on an Intel Core i7-4700MQ CPU with 6144 KB of cache.

	32-bit no Hes.	64-bit no Hes.	32-bit Hes.	64-bit Hes.
bearing	0.65	0.42	0.68	0.60
ch50	0.75	0.51	0.45	0.33
ch50b	0.85	0.57	0.49	0.36
chemeq	6.65	3.70	3.04	2.49
clnlbeam	0.78	0.58	0.52	0.48
denex	0.68	0.41	0.78	0.58
denop	0.85	0.59	0.90	0.86
gasoil	0.89	0.70	0.90	0.86
henon80	0.69	0.42	0.70	0.61
pfold3	0.89	0.62	0.88	0.82

Table 7. Relative memory use (one thread): new ASL divided by old.

10. Concluding Remarks

For large problems, using lists of operations rather than executable expression graphs often leads to faster computations and less memory use, particularly when multiple threads are deployed that do independent function (and gradient, etc.) evaluations. Source for the new ASL, using operation lists, appears in [2].

11. REFERENCES

[1] `http://ampl.com/netlib/ampl/models/nlmodels`.

[2] `http://ampl.com/netlib/ampl/solvers2.tgz`.

[3] *Turbo Pascal Reference Manual,* Borland International, 1983.

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

[5] 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.

[6] ROBERT FOURER AND DAVID M. GAY, ‘‘Experience with a Primal Presolve Algorithm,’’ pp. 135–154 in *Large Scale Optimization: State of the Art*, ed. W. W. Hager, D. W. Hearn, and P. M. Pardalos, Kluwer Academic Publishers (1994).

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

[8] ROBERT FOURER, DAVID M. GAY, AND BRIAN W. KERNIGHAN, *AMPL: A Modeling Language for Mathematical Programming,* Duxbury Press / Brooks/Cole Publishing Company, 2003. second edition, ISBN: 0-534-38809-4.

[9] 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).

[10] D. 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).

[11] 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).

[12] ANDREAS GRIEWANK AND ANDREA WALTHER, *Evaluating Derivatives,* SIAM, 2008.

[13] 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.

[14] R. BAKER KEARFOTT, ''GlobSol User Guide,'' *Optimization Methods and Software* **24** #4-5 (2009), pp. 687–708.

[15] HANS MITTELMANN, private communication (2017).

[16] J. J. MORÉ, B. S. GARBOW, AND K. E. HILLSTROM, ''User Guide for MINPACK-1,'' ANL-80-74 (1980), Argonne National Laboratory, Argonne, IL.