U.S.-Mexico Workshop on Optimization and its Applications, 4–8 January 2016

Rethinking Expression Representations for Nonlinear AMPL Models

David M. Gay

AMPL Optimization, Inc.

dmg@ampl.com

http://www.ampl.com

Goal

Immediate goal: revisit expression and derivative evaluations in the AMPL/solver interface library (ASL) with an eye to separating expressions from data so multiple threads can use the same expressions.

Longer-term goal: prepare for adding recursive function declarations to AMPL.

Toy nonlinear example

```
ampl: var x; var y;
ampl: minimize f: (x - 3)^2 + (y + 4)^2;
ampl: s.t. c: x + y == 1;
ampl: solve;
MINOS 5.51: optimal solution found.
2 iterations, objective 2
Nonlin evals: obj = 6, grad = 5.
ampl: display x, y;
x = 4
y = -3
```

Operation of "`solve;`"

AMPL writes `.nl` file containing, e.g.,

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

Expression graph representations

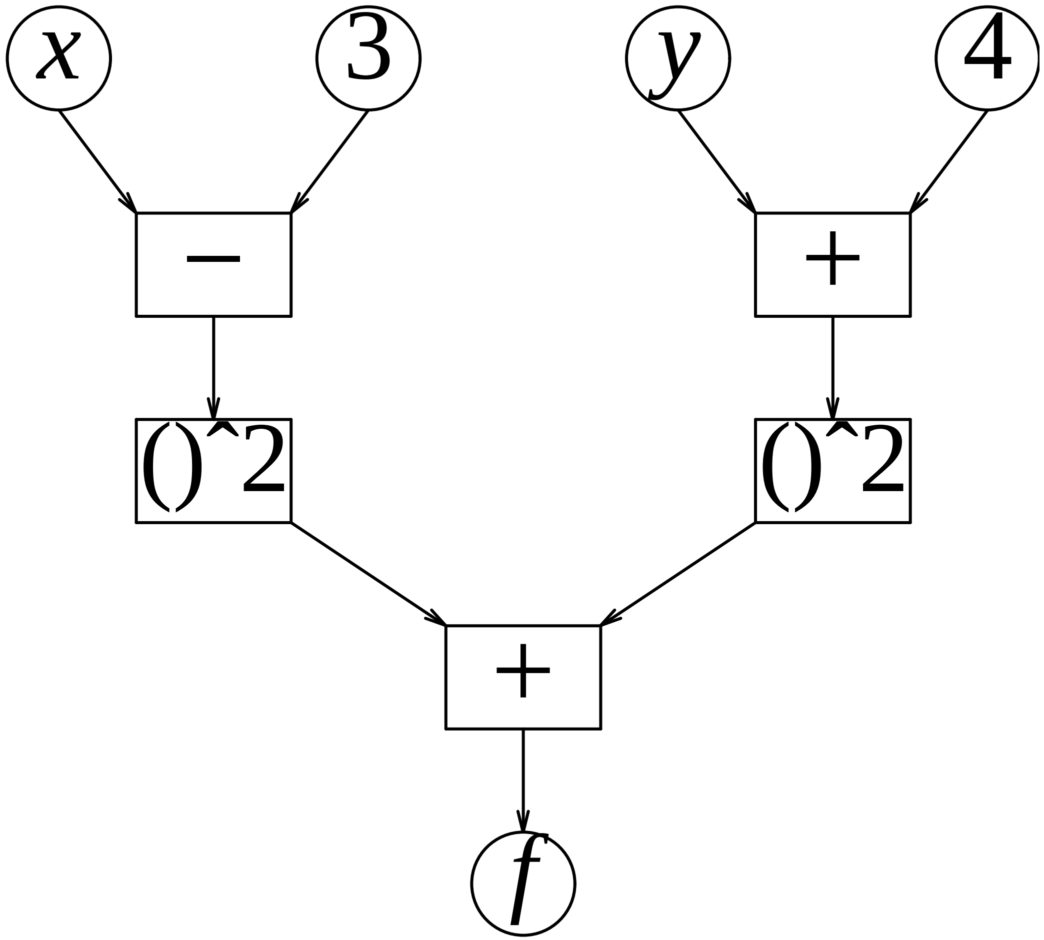
Several representations roughly equivalent in size and evaluation time:

- Polish postfix (as with HP calculators)
- Polish prefix (used in `.nl` files)
- executable expression graphs (current ASL)
- operation lists (considered here)

Linear time conversion from one form to another.

Expression graph example

Graph for $f = (x - 3)^2 + (y + 4)^2$:



Polish prefix in `.nl` file

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

ASL first-order expression-graph node

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

Example "op" function

```
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, "/");
        rv = L / R;
        if (want_deriv)
                e->dR = -rv * (e->dL = 1. / R);
        return rv;
        }
```

Operation list

List of instructions, e.g.,

```
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];
```

Operation list via `switch()`

```
real eval1(int *o, EvalWorkspace *ew) {
        real *w = ew->w;
top:    switch(*o) {
          case nOPRET:
                return w[o[1]];
          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;
        ...
```

Chain rule: basis for automatic differentiation (AD)

Suppose for scalar x that

$$\phi(x) = f(y_1(x), y_2(x), ..., y_k(x)).$$

The chain rule gives

$$\frac{\partial \phi}{\partial x} = \frac{\partial \phi}{\partial f} \sum_{i=1}^{k} \frac{\partial f}{\partial y_i} \frac{\partial y_i}{\partial x} = \sum_{i=1}^{k} \frac{\partial \phi}{\partial y_i} \frac{\partial y_i}{\partial x}.$$

In general, once we know the *adjoint* $\frac{\partial \phi}{\partial y}$ of an intermediate variable y, we can add its contribution $\frac{\partial \phi}{\partial y} \frac{\partial y}{\partial x}$ to the adjoint $\frac{\partial \phi}{\partial x}$ of each variable x on which y directly depends.

Current Reverse AD in ASL

Reverse AD: visiting operations in reverse order, we compute the contributions of each intermediate variable to the adjoints of its immediate prerequisites. Then the adjoints of the original variables are the gradient $\nabla\phi$. In ASL, this is currently done by

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

Possible data types for `derprop`

For performance, is it OK to use integer subscripts rather than pointers? Consider three inner-product alternatives:

```
struct Rpair { double a, b; } *rp;
==> dot += rp->a * rp->b;

struct Aoff { real *a, *b; } *p;
==> dot += *p->a * *p->b;

struct Ioff { int a, b; } *q;
real *v;
==> dot += v[q->a] * v[q->b];
```

Timing of data types for `derprop`

	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

Conclusion: integer subscripts are OK.

Alternative implementations of `derprop`

Simple loop:

```
struct derp { int a, b, c; } *d, *de;

for(d = ...; d < de; ++d)
        s[d->a] += s[d->b] * w[d->c];
```

Disadvantages:

- must initialize much of `s` array to zeros
- big `s` array.

Alternative implementations of derprop

Switch variant:

```
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;
```

Alternative implementations of `derprop`

Currently prefer simple loop with if:

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

No need to initialize `s` array to zeros; can use much smaller `s` array; smaller `u` array.

Organization of `w` array

$$
\begin{array}{ll}
 & const_1 \\
 & \dots \\
 & const_m \\
w \rightarrow & var_1 \\
 & var_2 \\
 & \dots \\
 & var_n \\
 & result_1 \\
 & \dots
\end{array}
$$

Relative times for `derprop` alternatives

Relative times: "simple loop with if" divided by current ASL:

	32-bit	64-bit
Ex1 f, ∇f	0.52	0.42
Ex1 c, ∇c	0.98	0.96
Ex2 f, ∇f	0.43	0.43
Ex3 f, ∇f	0.42	0.31
Ex3 c, ∇c	0.52	0.39

Relative times for `derprop` alternatives

More relative times: "simple loop with if" divided by current ASL:

	32-bit	64-bit
pfold3 $f, \nabla f$	0.80	0.65
ch50 $f, \nabla f$	0.62	0.69
ch50b $f, \nabla f$	1.01	0.67
ch50b $c, \nabla c$	6.47	3.68

ch50b.mod

```
# MINPACK Chebyquad 50 as both objective and connstraints
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;
```

Why the sloth with some defined variables?

AD can be viewed as a product of matrices [Griewank?]. Applying the associative law can lead to different numbers of operations. The draft revised ASL is recurring shared defined variables differently than the current ASL. This may change...

Relative net memory use: new/old

	32-bit	64-bit
Ex1	0.93	0.62
Ex2	0.82	0.41
Ex3	0.71	0.45
pfold3	0.83	0.63
ch50	0.87	0.57
ch50b	1.08	0.63

Comparison of alternative `derprop` implementations

Of the `derprop` alternatives, "simple loop with if" is often slightly faster than the others and sometimes outperforms the current ASL implementation.

Still to come: adjustments to "funneling" gradient contributions by defined variables used in several constraints and objectives; Hessian computations with separate workspace so multiple threads can use the same problem representation but different workspaces.

Adjusting qpcheck() routines

The existing ASL `qpcheck()` routines require special preparation — invoking `qp_read()` rather than `fg_read()` and calling `qp_opify()` before doing nonlinear evaluations. With the operations-list approach, we can dispense with `qp_read()` and `qp_opify()`.

The modified `qpcheck()` routines carry out an "evaluation" that computes expression information rather than numeric values.

Conclusion

After more testing, hope to replace ASL evaluations with a form that is more convenient for parallel executions and is somewhat faster on many problems.

Style of expression walks in updated `qpcheck()` routines may be grist for setting up gradient and Hessian computations in multi-level problems.