NAOIV-2017 — Numerical Analysis and Optimization
Sultan Qaboos University, Muscat, 2–5 January 2017

Revisiting Expression Representations for Nonlinear AMPL Models

David M. Gay

AMPL Optimization, Inc.

Albuquerque, New Mexico, U.S.A.

dmg@ampl.com

http://www.ampl.com

Background: AMPL, a language for mathematical programming, e.g.,

$$\text{minimize } f(x)$$
$$\text{s.t. } \ell \le c(x) \le u,$$

with $x \in \mathbb{R}^n$ and $c : \mathbb{R}^n \to \mathbb{R}^m$ given algebraically and some x_i discrete.

AMPL: Tiny 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)

AMPL/solver interface library (ASL) reads `.nl` file, provides details to solvers, e.g., function and gradient values.

Goal of current work

Immediate goal: revisit ASL expression and derivative evaluations 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 — for use in models and in callbacks for solvers.

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.

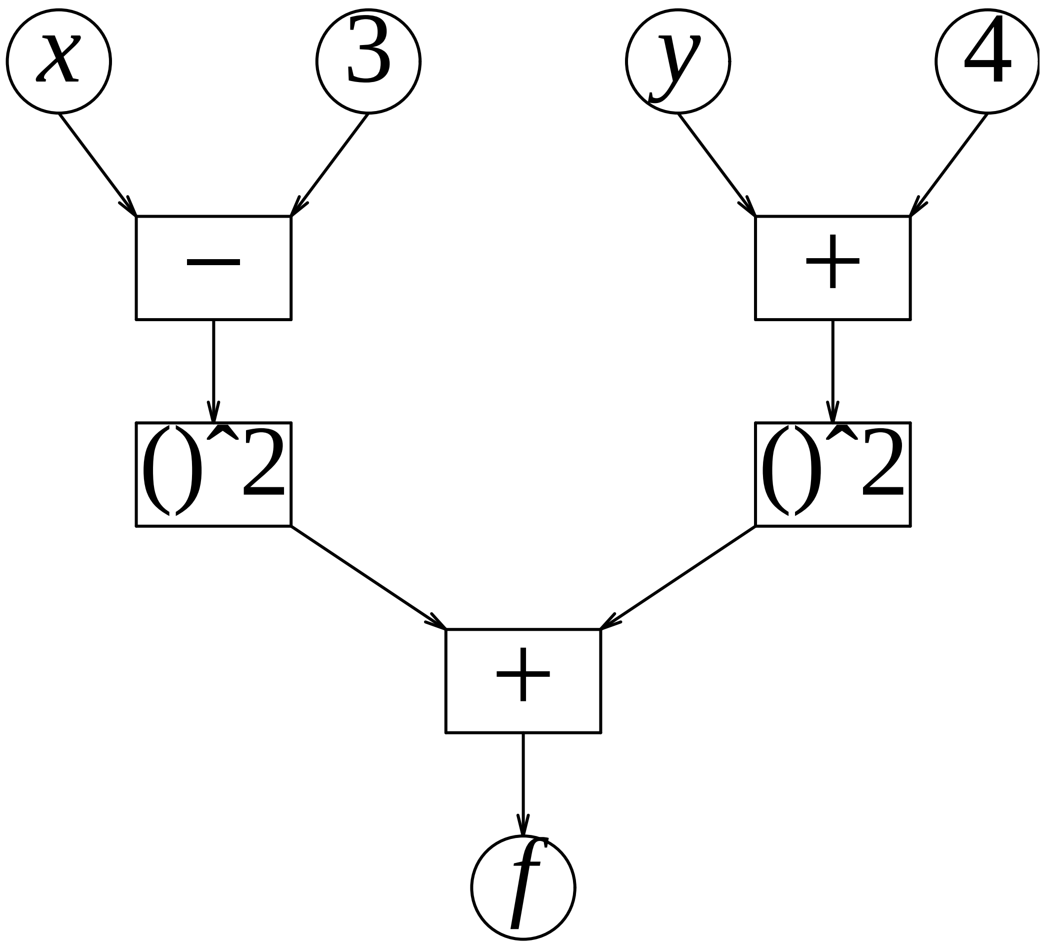
HP calculater-style Polish postfix

Example: compute $3 \times 4 + 5 = 17$:

keystroke	*display*
3	3
Enter	3
4	4
$\times$	12
5	5
$+$	17

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

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

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

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

Relative times for `derprop` alternatives

"Simple loop with if" divided by current ASL:

	32-bit no Hes.	64-bit no Hes.	32-bit Hes.	64-bit Hes.
Ex1 f, ∇f	0.50	0.42	0.76	0.75
Ex1 c, ∇c	0.95	0.94	0.83	0.79
Ex2 f, ∇f	0.44	0.43	0.79	0.60
Ex3 f, ∇f	0.40	0.32	0.26	0.22
Ex3 c, ∇c	0.50	0.40	0.19	0.18

Relative times for `derprop` alternatives

More relative times: "simple loop with if" with iterated defined defined variables, divided by current ASL (Jan. 2016):

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

Relative times for **derprop** alternatives

More relative times: "simple loop with if" with *derpcopy* for defined defined variables, divided by current ASL (July 2016):

	32-bit	64-bit
pfold3 f, ∇f	0.73	0.63
ch50 f, ∇f	0.61	0.54
ch50b f, ∇f	0.93	0.51
ch50b c, ∇c	0.87	0.48

ch50b.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;
```

Iterated defined-variable *derprop*

```
do {    i = *dv--;
        if ((t = s[i])) {
                ce = cx0 + i;
                if ((db = ce->dbf)) {
                        if (db == ce->db)
                                derpropa(db, i, s, w, t);
                        else
                                derprop(db, s, w, t);  }
                if ((lp = ce->lp)) {
                        lc = lp->lc;
                        lce = lc + lp->n;
                        do s[lc->varno] += t*lc->coef;
                          while(++lc < lce); }       }
        } while(dv > dvr);
```

Iterating defined variables caused sloth

The current ASL uses *derpcopy* to copy derivative propagations for all defined variables so a single *derprop()* call was needed. Looping as in the previous slide may sometimes take less memory, but takes more time. Now using *derpcopy* in the updated ASL.

Relative net memory use: new(Jan. 2016)/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

Relative net memory use: new(Jan. 2017)/old

	32-bit no Hes.	64-bit no Hes.	32-bit Hes.	64-bit Hes.
Ex1	0.92	0.62	0.87	0.86
Ex2	0.82	0.41	0.71	0.54
Ex3	0.71	0.45	0.60	0.41
pfold3	0.83	0.63	0.91	0.84
ch50	0.87	0.58	0.45	0.33
ch50b	0.92	0.55	0.49	0.36

Comparison of alternative `derprop` implementations

Of the `derprop` alternatives, "simple loop with if" plus *derpcopy* is often slightly faster than the others and often outperforms the current ASL implementation for function and gradient calculations.

One or (with setup for Hessians) two alternatives remain to be tested. Funneling all defined variables without *derpcopy* but with other data structures may be worthwhile. Machinery for Hessians offers another possibility.

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.

On to Hessians

Consider

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

for which

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

If we compute $\phi'(0)$ by forward AD, then reverse AD gives $\nabla^2 f(x)p$, i.e., a Hessian-vector product [Bruce Christianson, 1992].

Partially separable structure

Griewank & Toint (1982) point out that many objectives $f(x)$ have the form

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

in which each U_i has only a few rows. For such f,

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

and

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

Group partially separable structure

In LANCELOT, Conn, Gould & Toint exploit further structure:

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

in which $\theta_i()$ is a unary function. Then

$$\nabla f(x) = \sum_{i=1}^{q} \theta'(\cdots) U_i^{\mathrm{T}} \nabla f_i(U_i x)$$

$$\nabla^2 f(x) = \sum_{i=1}^{q} \{ \quad \theta'(\cdots) U_i^{\mathrm{T}} \nabla^2 f_i(U_i x) U_i$$

$$+ \quad \theta''(\cdots)(U_i^{\mathrm{T}} \nabla f_i(U_i x))(U_i^{\mathrm{T}} \nabla f_i(U_i x))^{\mathrm{T}} \}.$$

General group partially separable structure

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

and $\tilde{u}_i = \sum_{j=1}^{n_i} U_{ij}^{\mathrm{T}} \nabla f_{ij}(U_{ij}x)$, then

$$\nabla f(x) = \sum_{i=1}^{q} \theta'(\cdots)\tilde{u}_i$$

and

$$\begin{aligned}\nabla^2 f(x) = \sum_{i=1}^{q} \{ \quad & \theta'(\cdots) \sum_{j=1}^{n_i} U_{ij}^{\mathrm{T}} \nabla^2 f_{ij}(U_{ij}x) U_{ij} \\ + \quad & \theta''(\cdots)\tilde{u}_i \tilde{u}_i^{\mathrm{T}} \}.\end{aligned}$$

Finding group partially separable structure

Group partially separable structure can be found automatically by a suitable tree walk. *One can exploit it without knowing what it is.*

Example: excerpt from pfold.mod

```
# 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;
```

Binary operator for Hessians in current ASL

```
struct expr2 {
   efunc2 *op;
   int a;          /* adjoint index, then operator class */
   expr2 *fwd, *bak;
   real  d0;       /* deriv of op w.r.t. t in x + t*p */
   real a0;        /* adjoint (in Hv computation) of op */
   real ad0;       /* adjoint (in Hv computation) of d0 */
   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 */
   };
```

Current forward computation of ϕ'

```
void hv_fwd(expr *e) {
...
for(; e; e = e->fwd) {
    e->a0 = e->ad0 = 0;
    switch(e->a) {
        ...
        case Hv_binaryLR:
            e->d0 = e->L->d0*e->dL + e->R->d0*e->dR;
            break;
        case Hv_minusR:
            e->d0 = -e->R->d0;
            break;
        ...
        }}}
```

New forward computation of ϕ'

```
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.;
        }}
```

Current reverse AD on ϕ'

```
void hv_back(expr *e) { ...
for(; e; e = e->bak) {
    switch(e->a) { ...
        case Hv_binaryLR:
            e1 = e->L;
            e2 = e->R;
            ad0 = e->ad0;
            t1 = ad0 * e1->d0;
            t2 = ad0 * e2->d0;
            e1->a0  += e->a0*e->dL + t1*e->dL2 + t2*e->dLR;
            e2->a0  += e->a0*e->dR + t1*e->dLR + t2*e->dR2;
            e1->ad0 += ad0 * e->dL;
            e2->ad0 += ad0 * e->dR;
            break;
     ... }}}
```

New reverse AD on ϕ'

```
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->ad0 += r->ad0 * r->dL;
            R->ad0 += r->ad0 * r->dR;
            t1 = r->ad0 * L->d0;  t2 = r->ad0 * R->d0;
            L->a0  += r->a0*r->dL + t1*r->dL2 + t2*r->dLR;
            R->a0  += r->a0*r->dR + t1*r->dLR + t2*r->dR2;
            break; ...}
    o -= o[1];
    }}
```

Relative times for Hessians: new(Jan. 2017)/old

	32-bit	64-bit
Ex1	1.00	0.96
Ex2*	0.58	0.55
Ex3	0.56	0.50
pfold3	1.07	1.13
ch50	0.65	0.55
ch50b	0.59	0.48
chemeq	0.94	0.88

* Hessian-vector product

Discussion

Hessians now working and on large problems generally take less memory; often but not always faster. Significant memory savings when using multiple threads.

Plan soon to replace ASL evaluations with the updated ones — after testing another *derpcopy* alternative.

Need to update "Hooking Your Solver to AMPL" sometime.

More discussion

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

$$\phi(\sigma,\tau) = f(x + \sigma p + \tau q),$$

reverse AD on $\frac{\partial^2 \phi}{\partial\sigma\partial\tau}$ may be useful on bilevel problems.

More on AMPL and AD therewith

The AMPL web site

`http://ampl.com`

has more on AMPL, including pointers to papers on AD with AMPL and on the AMPL/solver interface library (ASL).

For more on AD in general, see

`http://www.autodiff.org`