

USING EXPRESSION GRAPHS IN OPTIMIZATION ALGORITHMS

DAVID M. GAY*

Abstract. An expression graph, informally speaking, represents a function in a way that can be manipulated to reveal various kinds of information about the function, such as its value or partial derivatives at specified arguments and bounds thereon in specified regions. (Various representations are possible, and all are equivalent in complexity, in that one can be converted to another in time linear in the expression's size.) For mathematical programming problems, including the mixed-integer nonlinear programming problems that were the subject of the IMA workshop that led to this paper, there are various advantages to representing problems as collections of expression graphs. "Presolve" deductions can simplify the problem, e.g., by reducing the domains of some variables and proving that some inequality constraints are never or always active. To find global solutions, it is helpful sometimes to solve relaxed problems (e.g., allowing some "integer" variables to vary continuously or introducing convex or concave relaxations of some constraints or objectives), and to introduce "cuts" that exclude some relaxed variable values. There are various ways to compute bounds on an expression within a specified region or to compute relaxed expressions from expression graphs. This paper sketches some of them. As new information becomes available in the course of a branch-and-bound (or -cut) algorithm, some expression-graph manipulations and presolve deductions can be revisited and tightened, so keeping expression graphs around during the solution process can be helpful. Algebraic problem representations are a convenient source of expression graphs. One of my reasons for interest in the AMPL modeling language is that it delivers expression graphs to solvers.

Key words. Expression graphs, automatic differentiation, bound computation, constraint propagation, presolve.

AMS(MOS) subject classifications. Primary 68U01, 68N20, 68W30, 05C85.

1. Introduction. For numerically solving a problem, various problem representations are often possible. Many factors can influence one's choice of representation, including familiarity, computational costs, and interfacing needs. Representation possibilities include some broad, often overlapping categories that may be combined with uses of special-purpose libraries: general-purpose programming compiled languages (such as C, C++, Fortran, and sometimes Java), interpreted languages (such as awk, Java, or Python), and "little languages" specialized for dealing with particular problem domains (such as AMPL for mathematical programming or MATLAB for matrix computations — and much else). Common to most such representations is that they are turned into expression graphs behind the scenes: directed graphs where each node represents an operation, incoming edges represent operands to the operation, and outgoing edges represent uses of the result of the operation. This is illustrated in Figure 1, which shows an expression graph for computing the $f : \mathbb{R}^2 \rightarrow \mathbb{R}$

*Sandia National Laboratories, Albuquerque, NM 87185-1318. See [34].

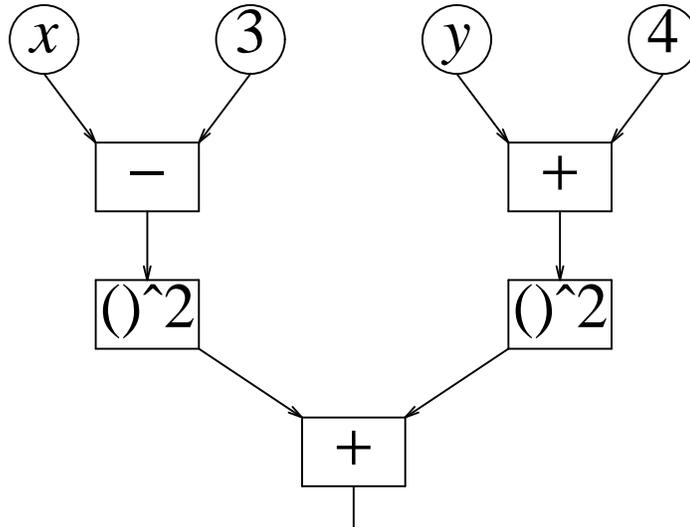


FIG. 1. *Expression graph for $f(x, y) = (x - 3)^2 + (y + 4)^2$*

for computing the function $f(x, y) = (x - 3)^2 + (y + 4)^2$, which involves operators for addition (+), subtraction (-) and squaring (()^2).

It can be convenient to have an explicit expression graph and to compute with it or manipulate it in various ways. For example, for smooth optimization problems, we can turn expression graphs for objective and constraint-body evaluations into reasonably efficient ways to compute both these functions and their gradients. When solving mixed-integer nonlinear programming (MINLP) problems, computing bounds and convex underestimates (or concave overestimates) can be useful and can be done with explicit expression graphs. Problem simplifications by “presolve” algorithms and (similarly) domain reductions in constraint programming are readily carried out on expression graphs.

This paper is concerned with computations related to solving a mathematical programming problem: given $D \subseteq \mathbb{R}^n$, $f : D \rightarrow \mathbb{R}$, $c : D \rightarrow \mathbb{R}^m$, and $\ell, u \in D \cup \{-\infty, \infty\}^n$ with $\ell_i \leq u_i \forall i$, find x^* such that $x = x^*$ solves

$$\begin{aligned}
 & \text{Minimize } f(x) \\
 (1.1) \quad & \text{subject to } \ell \leq c(x) \leq u \\
 & \text{and } x \in D.
 \end{aligned}$$

For MINLP problems, D restricts some components to be integers, e.g.,

$$(1.2) \quad D = \mathbb{R}^p \times \mathbb{Z}^q,$$

with $n = p + q$.

One of my reasons for interest in the AMPL [7, 8] modeling language for mathematical programming is that AMPL makes explicit expression graphs available to separate solvers. Mostly these graphs are only seen and manipulated by the AMPL/solver interface library [13], but one could also use them directly in the computations described below.

There are various ways to represent expression graphs. For example, AMPL uses a Polish prefix notation (see, e.g., [31]) for the nonlinear parts of problems conveyed to solvers via a “.nl” file. Kearfott [21] uses a representation via 4-tuples (operation, result, left, and right operands). Representations in XML have also been advocated ([9]). For all the specific representations I have seen, converting from one form to another takes time linear in the length (nodes + arcs) of the expression graph.

The rest of this paper is organized as follows. The next several sections discuss derivative computations (§2), bound computations (§3), pre-solve and constraint propagation (§4), convexity detection (§5), and outer approximations (§6). Concluding remarks appear in the final section (§7).

2. Derivative computations. When f and c in (1.1) are continuously differentiable in their continuous variables (i.e., the first p variables when (1.2) holds), use of their derivatives is important for some algorithms; when integrality is relaxed, partials with respect to nominally integer variables may also be useful (as pointed out by a referee). Similarly, when f and c are twice differentiable, some algorithms (variants of Newton’s method) can make good use of their first and second derivatives. In the early days of computing, the only known way to compute these derivatives without the truncation errors of finite differences was to compute them by the rules of calculus: deriving from, e.g., an expression for $f(x)$ expressions for the components of $\nabla f(x)$, then evaluating the derived formulae as needed. Hand computation of derivatives is an error-prone process, and many people independently discovered [18] a class of techniques called Automatic Differentiation (or Algorithmic Differentiation), called AD below. The idea is to modify a computation so it computes both function and desired partial derivatives as it proceeds — an easy thing to do with an expression graph. Forward AD is easiest to understand and implement: one simply applies the rules of calculus to recur desired partials for the result of an operation from the partials of the operands. When there is only one independent variable, it is easy and efficient to recur high-order derivatives with respect to that variable. For example, Berz et al. [3, 4] have done highly accurate simulations of particle beams using high-order Taylor series (i.e., by recurring high derivatives).

Suppose f is a function of $n > 1$ variables and that computing $f(x)$ involves L operations. Then the complexity of computing $f(x)$ and its gradient $\nabla f(x)$ by forward AD is $O(nL)$. It can be much faster to use “reverse AD” to compute $f(x)$ and $\nabla f(x)$. With this more complicated AD variant, one first computes $f(x)$, then revisits the operations in reverse order to compute the “adjoint” of each operation, i.e., the partial of f with respect to the result of the operation. By the end of this “reverse sweep”, the computed adjoints of the original variables are the partials of f with respect to these variables, i.e., $\nabla f(x)$. The reverse sweep just involves initializing variables representing the adjoints to zero and adding partials of individual operations times adjoints to the adjoint variables of the corresponding operands, which has the same complexity $O(L)$ as computing $f(x)$. For large n , reverse AD can be much faster than forward AD or finite differences.

The AMPL/solver interface library (ASL) makes arrangements for reverse AD sweeps while reading expression graphs from a “.nl” file and converting them to internal expression graphs. This amounts to a preprocessing step before any numerical computing is done, and is one of many useful kinds of expression-graph walks. Many ways of handling implementation details are possible, but the style I find convenient is to represent each operation (node in the expression graph) by a C “struct” that has a pointer to a function that carries out the operation, pointers to the operands, and auxiliary data that depend on the intended use of the graph. For example, the “expr” structure used in the ASL for binary operations has the fields shown in Figure 2 when only function and gradient computations are allowed [12], and has the more elaborate form shown in Figure 3 when Hessian computations are also allowed [14]. The intent here is not to give a full explanation of these structures, but just to illustrate how representations can vary, depending on their intended uses. In reality, some other type names appear in the ASL, and some fields appear in a different order. Figures 2 and 3 both assume typedefs of the form

```
typedef struct expr expr;
typedef double efunc(expr*);
```

so that an “efunc” is a double-valued function that takes one argument, a pointer to an “expr” structure. Use of such a function is illustrated in Figure 4, which shows the ASL’s “op” function for multiplication. This is a particularly simple binary operation in that the left partial is the right operand and vice versa. Moreover the second partials are constants (0 or 1) and need not be computed. In other cases, such as division and the “atan2” function, when Hessian computations are allowed, the function also computes and stores some second partial derivatives.

Once a function evaluation has stored the partials of each operation, the “reverse sweep” for gradient computations by reverse AD takes on a

```

struct expr {
    efunc *op;      /* function for this operation */
    int a;          /* adjoint index */
    real dL, dR;   /* left and right partials */
    expr *L, *R;   /* left and right operands */
};
    
```

FIG. 2. ASL structure for binary operations with only f and ∇f available.

```

struct
expr {
    efunc *op;      /* function for this operation */
    int a;          /* adjoint index (for gradient comp.) */
    expr *fwd, *bak; /* used in forward and reverse sweeps */
    double d0;      /* deriv of op w.r.t. t in x + t*p */
    double a0;      /* adjoint (in Hv computation) of op */
    double ad0;     /* adjoint (in Hv computation) of d0 */
    double dL;      /* deriv of op w.r.t. left operand */
    expr *L, *R;    /* left and right operands */
    double dR;      /* deriv of op w.r.t. right operand */
    double dL2;     /* second partial w.r.t. L, L */
    double dLR;     /* second partial w.r.t. L, R */
    double dR2;     /* second partial w.r.t. R, R */
};
    
```

FIG. 3. ASL structure for binary operations with f , ∇f , and $\nabla^2 f$ available.

very simple form in the ASL:

$$(2.1) \quad \begin{aligned} & \text{do } *d \rightarrow a.\text{rp} += *d \rightarrow b.\text{rp} * *d \rightarrow c.\text{rp}; \\ & \quad \text{while}(d = d \rightarrow \text{next}); \end{aligned}$$

in which $d \rightarrow a.\text{rp}$ points to an adjoint being updated, $d \rightarrow b.\text{rp}$ points to an adjoint of the current operation and $d \rightarrow c.\text{rp}$ points to a partial derivative of this operation. Thus for each of its operands, an operation contributes the product of its adjoint and a partial derivative to the adjoint of the operand.

Hessian or Hessian-vector computations are sometimes useful. Given a vector $v \in \mathbb{R}^n$ and a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ represented by an expression graph of L nodes, we can compute $\nabla^2 f(x)v$ in $O(L)$ operations by what amounts to a mixture of forward and reverse AD, either applying reverse AD to the result of computing $\phi'(0)$ with $\phi(\tau) \equiv f(x + \tau v)$ (computing $\phi(0)$ and $\phi'(0)$ by forward AD), or by applying forward AD to $v^T \nabla f(x)$, with $\nabla f(x)$ computed by reverse AD. Both descriptions lead to the same numerical operations (but have different overheads in Sacado context discussed below). The ASL offers Hessian-vector computations done this way, since some nonlinear optimization solvers use Hessian-vector products in iterative “matrix-free” methods, such as conjugate gradients, for computing

```

double
f_OPMULT(expr *e A_AS�)
{
    expr *eL = e->L.e;
    expr *eR = e->R.e;
    return  (e->dR = (*eL->op)(eL))
           * (e->dL = (*eR->op)(eR));
}

```

FIG. 4. ASL function for multiplication.

search directions.

Many mathematical programming problems (1.1) involve “partially separable” functions. A function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is partially separable if it has the form

$$f(x) = \sum_i f_i(A_i x),$$

in which the A_i are matrices having only a few rows (varying with i) and n columns, so that f_i is a function of only a few variables. A nice feature of this structure is that f ’s Hessian $\nabla^2 f(x)$ has the form

$$\nabla^2 f(x) = \sum_i A_i^T \nabla^2 f_i(x) A_i,$$

i.e., $\nabla^2 f(x)$ is a sum of outer products involving the little matrices A_i and the Hessians $\nabla^2 f_i(x)$ of the f_i . Knowing this structure, we can compute each $\nabla^2 f_i(x)$ separately with a few Hessian-vector products, then assemble the full $\nabla^2 f(x)$ — e.g., if it is to be used by a solver that wants to see explicit Hessian matrices.

Many mathematical programming problems involve functions having a more elaborate structure called partially-separable structure:

$$(2.2) \quad f(x) = \sum_i \theta_i \left(\sum_{j=1}^{r_i} f_{ij}(A_{ij} x) \right),$$

in which $\theta_i : \mathbb{R} \rightarrow \mathbb{R}$ is smooth and each A_{ij} matrix has only a small number of rows. The full $\nabla^2 f(x)$ is readily assembled from the pieces of this representation (and their partials). By doing suitable expression-graph walks, the ASL finds partially-separable structure (2.2) automatically and arranges for it to be exploited when explicit Hessians are desired. More graph walks determine the sparsity of the desired Hessians — usually the Hessian of the Lagrangian function. See [14] for more details. (For use in a parallel computing context, I have recently been convinced to add a way to express the Lagrangian function as a sum of pieces and to arrange for

efficient computation of the Hessian of each piece, with the sparsity of each piece made available in a preprocessing step.)

The expression-graph walks that the ASL does once to prepare for later numerical evaluations make such computations reasonably efficient, but, as illustrated in the above reverse-sweep loop and in Figure 4, some pointer chasing is still involved. With another kind of graph walk, that done by the *nlc* program described in [13], we can convert expression graphs into Fortran or C (C++), eliminating much of the pointer chasing and some unnecessary operations, e.g., addition of zero and multiplication by ± 1 .

The expression graphs that AMPL uses internally often involve loops, i.e., iterating something over a set, so dealing with loops in expression graphs is not hard. For simplicity in the ASL, the graphs that AMPL writes to “.nl” files to represent nonlinear programming problems are loop-free, with all operations explicit. Perhaps sometime this will change, as it somewhat limits problems that can be encoded in “.nl” files and sometimes makes them much larger than they might be if they were allowed to use looping nodes. This current limitation is somewhat mitigated by an imported-function facility that permits arbitrary functions to be introduced via shared libraries. When such functions are involved in gradient or Hessian computations, the functions must provide first or first and second partials with respect to their arguments, so the ASL can involve the functions in the derivative computations.

Some languages, such as Fortran and C++, allow operator overloading. With overloading, one can use the same arithmetic operators and functions in expressions involving new types; thus, after modifying source code by changing the types of some variables, one can leave the bulk of the source code unchanged and obtain a program with altered (ideally enhanced) behavior. Operator overloading in suitable languages provides another way to make AD computations conveniently available. An early, very general, and often used package for AD computations in C++ codes is ADOL-C [20], which operates by capturing an expression graph (called a “tape” in [20]) as a side effect of doing a computation of interest, then walking the graph to compute the desired derivatives. Because Sandia National Laboratories does much C++ computing and because more specialized implementations are sometimes more efficient, it has been worthwhile to develop our own C++ package, Sacado [2, 33], for AD. The reverse-AD portion of Sacado [15] does a reverse sweep whose core is equivalent to (2.1).

Computations based on operator overloading are very general and convenient, but present a restricted view of a calculation — somewhat like looking through a keyhole. As indicated by the timing results in Table 1 below, when a more complete expression graph is available, it can be used to prepare faster evaluations than are readily available from overloading techniques. Table 1 shows relative and absolute evaluation times for function and gradient evaluations of an empirical energy function for a protein-folding problem considered in [17]. This problem is rich in tran-

scendental function evaluations (such as $\cos()$, $\text{sqrt}()$, $\text{atan}()$), which masks some overhead. Computations were on a 3GHz Intel Xeon CPU; the times in the “rel.” column are relative to the time to compute $f(x)$ alone (the “Compiled C, no ∇f ” line), using a C representation obtained with the *nlc* program mentioned above. The “Sacado RAD” line is for C++ code that uses reverse-mode AD via operator overloading provided by Sacado. (Sacado also offers forward-mode AD, which would be considerably slower on this example.) The two ASL lines compare evaluations designed for computing $f(x)$ and $\nabla f(x)$ only (“ASL, fg mode”) with evaluations that save second partials for possible use in computing $\nabla^2 f(x)$ or $\nabla^2 f(x)v$. The “*nlc*” line is for evaluations using C from the *nlc* program; this line excludes time to run *nlc* and compile and link its output. Solving nonlinear mathematical programming problems often involves few enough evaluations that ASL evaluations make the solution process faster than would use of *nlc*, but for an inner loop repeated many times, preparing the inner-loop evaluation with *nlc* (or some similar facility) could be worthwhile.

Eval style	sec/eval	rel.
Compiled C, no ∇f	2.92e-5	1.0
Sacado RAD	1.90e-4	6.5
<i>nlc</i>	4.78e-5	1.6
ASL, fg mode	9.94e-5	3.4
ASL, pfg mode	1.26e-4	4.3

TABLE 1
Evaluation times for f and ∇f : protein folding ($n = 66$).

One lesson to draw from Table 1 is that while operator overloading is very convenient in large codes, in that one can significantly enhance computations by doing little more than changing a few types, there may be room to improve performance by replacing code involved in computational bottlenecks by alternate code.

Hessian-vector computations provide a more dramatic contrast between evaluations done with operator overloading (in C++) and evaluations prepared with the entire expression graph in view. Table 2 shows timings for Hessian-vector computations done several ways on the Hessian of a 100×100 dense quadratic form, $f(x) = \frac{1}{2}x^T Qx$. (Such evaluations only involve additions and multiplications and are good for exposing overhead.) The kinds of evaluations in Table 2 include two ways of nesting the forward (FAD) and reverse (RAD) packages of Sacado, a custom mixture (“RAD2”) of forward- and reverse AD that is also in Sacado, and the “interpreted” evaluations of the AMPL/solver interface library (ASL) prepared by the ASL’s `pfg_read` routine. The computations were again on a 3GHz Intel Xeon CPU.

Eval style	sec/eval	rel.
RAD \circ FAD	4.70e-4	18.6
FAD \circ RAD	1.07e-3	42.3
RAD2 (Custom mixture)	2.27e-4	9.0
ASL, pfgH mode	2.53e-5	1.0

TABLE 2
Times for $\nabla^2 f(x)v$ with $f = \frac{1}{2}x^T Qx$, $n = 100$.

For much more about AD in general, see Griewank’s book [19] and the “autodiff” web site [1], which has pointers to many papers and packages for AD.

3. Bound computations. Computing bounds on a given expression can be helpful in various contexts. For nonlinear programming in general and mixed-integer nonlinear programming in particular, it is sometimes useful to “branch”, i.e., divide a compact domain into the disjoint union of two or more compact subdomains that are then considered separately. If we find a feasible point in one domain and can compute bounds showing that any feasible points in another subdomain must have a worse objective value, then we can discard that other subdomain.

Various kinds of bound computations can be done by suitable expression graph walks. Perhaps easiest to describe and implement are bound computations based on interval arithmetic [24]: given interval bounds on the operands of an operation, we compute an interval that contains the results of the operation. For example, for any $a \in [\underline{a}, \bar{a}]$ and $b \in [\underline{b}, \bar{b}]$, the product $ab = a \cdot b$ satisfies

$$ab \in [\min(\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}), \max(\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b})].$$

(It is only necessary to compute all four products when $\max(\underline{a}, \underline{b}) < 0$ and $\min(\bar{a}, \bar{b}) > 0$, in which case $ab \in [\min(\underline{a}\bar{b}, \bar{a}\underline{b}), \max(\underline{a}\underline{b}, \bar{a}\bar{b})$.) When computing with the usual finite-precision floating-point arithmetic, we can use directed roundings to obtain rigorous enclosures.

Unfortunately, when the same variable appears several times in an expression, interval arithmetic treats each appearance as though it could have any value in its domain, which can lead to very pessimistic bounds. More elaborate interval analysis (see, e.g., [25, 26]) can give much tighter bounds. For instance, *mean-value forms* [25, 28] have an excellent outer approximation property that will be explained shortly. Suppose domain $X \subset \mathbb{R}^n$ is the Cartesian product of compact intervals, henceforth called an *interval vector*, i.e.,

$$X = [\underline{x}_1, \bar{x}_1] \times \cdots \times [\underline{x}_n, \bar{x}_n].$$

Suppose $f : X \rightarrow \mathbb{R}$ and we have a point $c \in X$ and another interval vector $S \subset \mathbb{R}^n$ with the property that

$$(3.1) \quad \begin{array}{l} \text{for any } x \in X, \text{ there is an } s \in S \\ \text{such that } f(x) = f(c) + s^T(x - c); \end{array}$$

if $f \in C^1(\mathbb{R})$, i.e., f is continuously differentiable, it suffices for S to enclose $\{\nabla f(x) : x \in X\}$. Then an enclosure of

$$(3.2) \quad \{f(c) + s^T(x - c) : x \in X, s \in S\}$$

also encloses $f(X) \equiv \{f(x) : x \in X\}$. For an interval vector V with components $[\underline{v}_i, \bar{v}_i]$, define the width $w(V)$ by $w(V) = \max\{\bar{v}_i - \underline{v}_i : 1 \leq i \leq n\}$, and for an interval enclosure $F = [\underline{F}, \bar{F}]$ of $f(X)$, define $\inf\{f(X)\} = \inf\{f(x) : x \in X\}$, $\sup\{f(X)\} = \sup\{f(x) : x \in X\}$, and $\epsilon(F, f(X)) = \max(\inf\{f(X)\} - \underline{F}, \bar{F} - \sup\{f(X)\})$, which is sometimes called the *excess width*. If $f \in C^1(\mathbb{R})$ and $S = \nabla f(X) \equiv \{\nabla f(x) : x \in X\}$, and $F = \{f(c) + s^T(x - c) : s \in S, x \in X\}$, then $\epsilon(F, f(X)) = O(w(X)^2)$, and this remains true if we use interval arithmetic (by walking an expression graph for f) to compute an outer approximation S of $\nabla f(X)$ and compute F from S by interval arithmetic. If $\nabla f(c) \neq 0$, then for small enough $h > 0$ and $X = [c_1 - h, c_1 + h] \times \cdots \times [c_n - h, c_n + h]$, the relative excess width $(w(F) - w(f(X)))/w(X) = O(h)$, which is the excellent approximation property mentioned above. This means that by dividing a given compact domain into sufficiently small subdomains and computing bounds on each separately, we can achieve bounds within a factor of $(1 + \tau)$ of optimal for a specified $\tau > 0$.

We can do better by computing *slopes* [23, 28] rather than interval bounds on ∇f . Slopes are divided differences, and interval bounds on them give an S that satisfies (3.1), so an enclosure of (3.2) gives valid bounds. For $\phi \in C^1(\mathbb{R})$ and $\xi, \zeta \in \mathbb{R}$, the slope $\phi[\xi, \zeta]$ is uniquely defined by

$$\phi[\xi, \zeta] = \begin{cases} (\phi(\xi) - \phi(\zeta))/(\xi - \zeta) & \text{if } \xi \neq \zeta, \\ \phi'(\zeta) & \text{if } \xi = \zeta. \end{cases}$$

Slopes for functions of n variables are n -tuples of bounds on divided differences; they are not uniquely defined, but can be computed (e.g., from an expression graph) operation by operation in a way much akin to forward AD. The general idea is to compute bounds on $f(X) = \{f(x) : x \in X\}$ by choosing a nominal point z , computing $f(z)$, and using slopes to bound $f(x) - f(z)$ for $x \in X$:

$$f(X) \subseteq \left\{ f(z) + \sum_{i=1}^n s_i(x_i - z_i) : x \in X, s_i \in f[X, z]_i \right\}$$

where the i -th component $f[X, z]_i$ of an interval slope is a bound on $(f(x + \tau e_i) - f(x))/\tau$ (which is taken to be $\partial f(x)/\partial x_i$ if $\tau = 0$) for $x \in X_i$ and

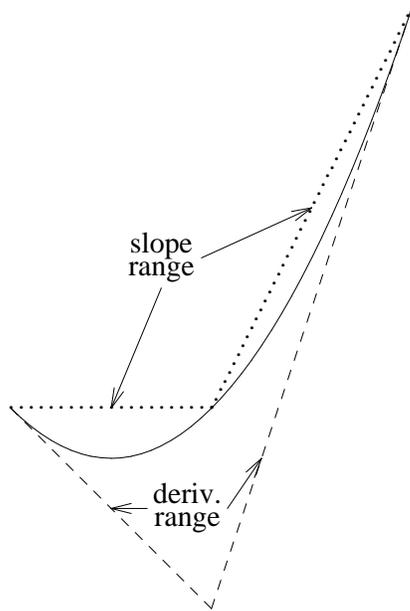


FIG. 5. Slopes and derivatives for $\phi(x) = x^2$, $x \in [-.5, 1.5]$, $c = 0.5$.

τ such that $x + \tau e_i \in X$. Most simply $X_i = X$ for all i , but we can get tighter bounds [32] at the cost of more memory by choosing

$$X_i = [x_1, \bar{x}_1] \times \cdots \times [x_i, \bar{x}_i] \times \{c_{i+1}\} \times \cdots \times \{c_n\},$$

e.g., for $c_i \approx \frac{1}{2}(x_i + \bar{x}_i)$. With this scheme, permuting the variables may result in different bounds; deciding which of the $n!$ permutations is best might not be easy.

Figure 5 indicates why slopes can give sharper bounds than we get from a first-order Taylor expansion with an interval bound on the derivative. Bounds on $\phi'(X)$ give $S = [-1, 3]$, whereas slope bounds give $S = [0, 2]$.

Sometimes we can obtain still tighter bounds by using second-order slopes [39, 36, 37], i.e., slopes of slopes. The general idea is to compute a slope matrix H such that an enclosure of

$$f(c) + \nabla f(c)(x - c) + (x - c)^T H(x - c)$$

for $x \in X$ gives valid bounds on $f(X)$. (To be rigorous in the face of roundoff error, we must compute interval bounds on $\nabla f(c)$.) Bounds computed this way are sometimes better than those from the mean-value form (3.2). In general, whenever there are several ways to compute bounds, it is usually best to compute all of them and compute their intersection; this is common practice in interval analysis.

As described in [16], I have been modifying some ASL routines to do bound computations from first- and second-order slopes. The computations are similar to forward AD, which greatly simplifies the .nl file reader in that it does not have to set up a reverse sweep. One small novelty is my exploitation of sparsity where possible; the preliminary expression-graph walk done by the .nl reader sets things up so evaluations can proceed more quickly. Table 3 summarizes the bound computations available at this writing, and Table 4 shows the widths of two sets of resulting bounds. In Table 3, F denotes an outer approximation of f . See [16] for details and more references. Explicit expression graphs are convenient for this work, in which the main focus is on properties of particular operations.

interval	$F(X) \supset f(X)$
Taylor 1	$f(z) + F'(X)(X - z)$
slope 1	$f(z) + F[X, z](X - z)$
slope 2	$f(z) + f'(z)(X - z)$ $+ F[X, z, z](X - z)^2$
slope 2*	slope 2 plus Theorem 2 in [16]

TABLE 3
Bound computations

Method	Barnes	Sn525
interval	162.417	0.7226
Taylor 1	9.350	0.3609
slope 1	6.453	0.3529
slope 2	3.007	0.1140
slope 2*	2.993	0.1003
true	2.330	0.0903

TABLE 4
Bound widths

4. Presolve and constraint propagation. Often it is worthwhile to spend a little time trying to simplify a mathematical programming problem before presenting it to a solver. With the help of suitable bound computations, sometimes it is possible to fix some variables and remove some constraints. Occasionally a problem can be completely solved this way, but more likely the solver will run faster when presented with a simpler problem.

For linear constraints and objectives, computing bounds is straightforward but best done with directed roundings [6] to avoid false simplifications.

For nonlinear problems, we can use general bounding techniques, such

as those sketched in §3, along with specific knowledge about some nonlinear functions, such as that $\sin(x) \in [-1, 1]$ for all x .

Another use of bounding techniques is to reduce variable domains. If we have rigorous bounds showing that part of the nominal problem domain is mapped to values, all of which violate problem constraints, then we can discard that part of the nominal problem domain. In the constraint-propagation community, this is called “constraint propagation”, but it can also be regarded as “nonlinear presolve”. See [35] for more discussion of constraint propagation on expression graphs.

In general, a presolve algorithm may repeatedly revisit bound computations when new information comes along. For instance, if we have upper and lower bounds on all but one of the variables appearing in a linear inequality constraint, we can deduce a bound on the remaining variable; when another deduction implies a tighter bound on one of the other variables, we can revisit the inequality constraint and tighten any bounds deduced from it. Sometimes this leads to a sequence of deductions tantamount to solving linear equations by an iterative method, so it is prudent to limit the repetitions in some way. Similar comments apply when we use nonlinear bounding techniques.

As mentioned in §3, it is sometimes useful to branch, i.e., divide the domain into disjoint subdomains that are then considered separately, perhaps along with the addition of “cuts”, i.e., inequalities that must be satisfied (e.g., due to the requirement that some variables assume integer values). Any such branching and imposition of cuts invites revisiting relevant presolve deductions, which might now be tightened, so an expression-graph representation of the problem can be attractive and convenient.

5. Convexity detection. Convexity is a desirable property for several reasons: it is useful in computing bounds; convex minimization (or concave maximization) problems can be much easier to solve globally than nonconvex ones; and convexity enables use of some algorithms that would otherwise not apply. It is thus useful to know when an optimization problem is convex (or concave).

An approach useful for some problems is to use a problem specification that guarantees convexity; examples include CVXMOD [5] and Young’s recent Ph.D. thesis [38]. More generally, a suitable expression-graph walk [10, 27, 29, 30] can sometimes find sufficient conditions for an expression to be convex or concave. As a special case, some solvers make special provisions for quadratic objectives and sometimes quadratic constraints. Walking a graph to decide whether it represents a constant, linear, quadratic, or nonlinear expression is straightforward; if quadratic, we can attempt to compute a Cholesky factorization to decide whether it is convex.

6. Outer approximations. Finding outer approximations — convex underestimates and concave overestimates — for a given expression can be useful. By optimizing the outer approximations, we obtain bounds

on the expression, and if we compute linear outer approximations (i.e., sets of linear inequalities satisfied by the expression), we can use a linear programming solver to compute bounds, which can be fast or convenient. It is conceptually straightforward to walk an expression graph and derive rigorous outer approximations; see, e.g., [11, 22] for details. Linear outer approximations are readily available from first-order slopes; see §7 of [35].

7. Concluding remarks. Expression graphs are not convenient for users to create explicitly, but are readily derived from high-level representations that are convenient for users. Once we have expression graphs, it is possible to do many useful sorts of computations with them, including creating other representations that are faster to evaluate, carrying out (automatic) derivative computations, computing bounds and outer approximations, detecting convexity, and recognizing problem structure, and simplifying problems. Internal use of expression graphs can be a boon in optimization (and other) algorithms in general, and in mixed-integer nonlinear programming in particular.

Acknowledgment. I thank an anonymous referee for helpful comments.

REFERENCES

- [1] *Autodiff Web Site*, <http://www.autodiff.org>.
- [2] ROSCOE A. BARTLETT, DAVID M. GAY AND ERIC T. PHIPPS, *Automatic Differentiation of C++ Codes for Large-Scale Scientific Computing*. In Computational Science – ICCS 2006, Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot and Jack Dongarra (eds.), Springer, 2006, pp. 525–532.
- [3] MARTIN BERZ, *Differential Algebraic Description of Beam Dynamics to Very High Orders*, Particle Accelerators **24** (1989), p. 109.
- [4] MARTIN BERZ, YOKO MAKINO, KHODR SHAMSEDDINE, GEORG H. HOFFSTÄTTER AND WEISHI WAN, *COSY INFINITY and Its Applications in Nonlinear Dynamics*, SIAM, 1996.
- [5] STEPHEN P. BOYD AND JACOB MATTINGLEY, *CVXMOD — Convex Optimization Software in Python*, <http://cvxmod.net/>, accessed July 2009.
- [6] R. FOURER AND D. M. GAY, *Experience with a Primal Presolve Algorithm*. In Large Scale Optimization: State of the Art, W. W. Hager, D. W. Hearn and P. M. Pardalos (eds.), Kluwer Academic Publishers, 1994, pp. 135–154.
- [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 Co., 2nd edition, 2003.
- [9] ROBERT FOURER, JUN MA AND KIPP MARTIN, *An Open Interface for Hooking Solvers to Modeling Systems*, slides for DIMACS Workshop on COIN-OR, 2006, <http://dimacs.rutgers.edu/Workshops/COIN/slides/osil.pdf>.
- [10] R. FOURER, C. MAHESHWARI, A. NEUMAIER, D. ORBAN AND H. SCHICHL, *Convexity and Concavity Detection in Computational Graphs*, manuscript, 2008, to appear in INFORMS J. Computing.
- [11] EDWARD P. GATZKE, JOHN E. TOLSMAN AND PAUL I. BARTON, *Construction of Convex Function Relaxations Using Automated Code Generation Techniques* Optimization and Engineering **3**, 2002, pp. 305–326.

- [12] DAVID M. GAY, *Automatic Differentiation of Nonlinear AMPL Models*. In Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. Corliss (eds.), SIAM, 1991, pp. 61–73.
- [13] DAVID M. GAY, *Hooking Your Solver to AMPL*, AT&T Bell Laboratories, Numerical Analysis Manuscript 93-10, 1993 (revised 1997). <http://www.ampl.com/REFS/hooking2.pdf>.
- [14] DAVID M. GAY, *More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability*. In Computational Differentiation: Techniques, Applications, and Tools, Martin Berz, Christian Bischof, George Corliss and Andreas Griewank (eds.), SIAM, 1996, pp. 173–184.
- [15] DAVID M. GAY, *Semiautomatic Differentiation for Efficient Gradient Computations*. In Automatic Differentiation: Applications, Theory, and Implementations, H. Martin Bücker, George F. Corliss, Paul Hovland and Uwe Naumann and Boyana Norris (eds.), Springer, 2005, pp. 147–158.
- [16] DAVID M. GAY, *Bounds from Slopes*, report SAND-1010xxxx, to be available as <http://www.sandia.gov/~dmgay/bounds10.pdf>.
- [17] D. M. GAY, T. HEAD-GORDON, F. H. STILLINGER AND M. H. WRIGHT, *An Application of Constrained Optimization in Protein Folding: The Poly-L-Alanine Hypothesis*, *Forefronts* **8**(2) (1992), pp. 4–6.
- [18] ANDREAS GRIEWANK, *On Automatic Differentiation*. In Mathematical Programming: Recent Developments and Applications, M. Iri and K. Tanabe (eds.), Kluwer, 1989, pp. 83–108.
- [19] , ANDREAS GRIEWANK, *Evaluating Derivatives*, SIAM, 2000.
- [20] A. GRIEWANK, D. JUEDES AND J. UTKE, *Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++*, *ACM Trans. Math Software* **22**(2) (1996), pp. 131–167.
- [21] R. BAKER KEARFOTT *An Overview of the GlobSol Package for Verified Global Optimization*, talk slides, 2002, <http://www.mat.univie.ac.at/~neum/glopt/mss/Kea02.pdf>.
- [22] , PADMANABAN KESAVAN, RUSSELL J. ALLGOR, EDWARD P. GATZKE AND PAUL I. BARTON, *Outer Approximation Algorithms for Separable Nonconvex Mixed-Integer Nonlinear Programs*, *Mathematical Programming* **100**(3), 2004, pp. 517–535.
- [23] R. KRAWCZYK AND A. NEUMAIER, *Interval Slopes for Rational Functions and Associated Centered Forms*, *SIAM J. Numer. Anal.* **22**(3) (1985), pp. 604–616.
- [24] R. E. MOORE, *Interval Arithmetic and Automatic Error Analysis in Digital Computing*, Ph.D. dissertation, Stanford University, 1962.
- [25] RAMON E. MOORE, *Methods and Applications of Interval Analysis*, SIAM, 1979.
- [26] RAMON E. MOORE, R. BAKER KEARFOTT AND MICHAEL J. CLOUD, *Introduction to Interval Analysis*, SIAM, 2009.
- [27] IVO P. NENOV, DANIEL H. FYLSTRA AND LUBOMIR V. KOLEV, *Convexity Determination in the Microsoft Excel Solver Using Automatic Differentiation Techniques*, extended abstract, 2004, <http://www.autodiff.org/ad04/abstracts/Nenov.pdf>.
- [28] ARNOLD NEUMAIER, *Interval Methods for Systems of Equations*, Cambridge University Press, 1990.
- [29] DOMINIQUE ORBAN, *Dr. AMPL Web Site*, <http://www.gerad.ca/~orban/drAMPL/>, accessed July 2009.
- [30] DOMINIQUE ORBAN AND ROBERT FOURER, *Dr. AMPL, A Meta Solver for Optimization*, CORS/INFORMS Joint International Meeting, 2004, <http://users.iems.northwestern.edu/~4er/SLIDES/ban0406h.pdf>.
- [31] *Polish notation*, http://en.wikipedia.org/wiki/Polish_notation, accessed July 2009.
- [32] S. M. RUMP, *Expansion and Estimation of the Range of Nonlinear Functions*, *Mathematics of Computation* **65**(216) (1996), pp. 15031512.
- [33] *Sacado Web Site*, <http://trilinos.sandia.gov/packages/sacado/>.

- [34] Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This manuscript (SAND2009-5066C) has been authored by a contractor of the U.S. Government under contract DE-AC04-94AL85000. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.
- [35] HERMANN SCHICHL AND ARNOLD NEUMAIER, *Interval Analysis on Directed Acyclic Graphs for Global Optimization* Journal of Global Optimization **33**(4) (2005), pp. 541–562.
- [36] MARCO SCHNURR, *Steigungen hoeherer Ordnung zur verifizierten globalen Optimierung*, Ph.D. dissertation, Universität Karlsruhe, 2007.
- [37] MARCO SCHNURR, *The Automatic Computation of Second-Order Slope Tuples for Some Nonsmooth Functions*, Electronic Transactions on Numerical Analysis **30** (2008), pp 203–223.
- [38] JOSEPH G. YOUNG *Program Analysis and Transformation in Mathematical Programming*, Ph.D. dissertation, Rice University, 2008.
- [39] SHEN ZUHE AND M. A. WOLFE, *On Interval Enclosures Using Slope Arithmetic*, Applied Mathematics and Computation **39** (1990), pp. 89105.