Numerical Issues and Influences in the Design of Algebraic Modeling Languages for Optimization

Robert Fourer
Northwestern University

David M. Gay
AMPL Optimization LLC

The idea of a modeling language is to describe mathematical problems symbolically in a way that is familiar to people but that allows for processing by computer systems. In particular the concept of an algebraic modeling language, based on objective and constraint expressions in terms of decision variables, has proved to be valuable for a broad range of optimization and related problems.

One modeling language can work with numerous solvers, each of which implements one or more optimization algorithms. The separation of model specification from solver execution is thus a key tenet of modeling language design. Nevertheless, several issues in numerical analysis that are critical to solvers are also important in implementations of modeling languages. So-called presolve procedures, which tighten bounds with the aim of eliminating some variables and constraints, are numerical algorithms that require carefully chosen tolerances and can benefit from directed roundings. Correctly rounded binary-decimal conversion is valuable in portably conveying problem instances and in debugging. Further rounding options offer tradeoffs between accuracy, convenience, and readability in displaying numerical results.

Modeling languages can also strongly influence the development of solvers. Most notably, for smooth nonlinear optimization, the ability to provide numerically computed, exact first and second derivatives has made modeling languages a valuable tool in solver development. The generality of modeling languages has also encouraged the development of more general solvers, such as for optimization problems with equilibrium constraints.

This paper draws from our experience in developing the AMPL modeling language [14, 15] to provide examples in all of the above areas. An associated set of slides [13] shows more completely the contexts from which our examples are taken.

1. Rounding and conversion

AMPL incorporates an interactive `display` command that is designed to produce usefully formatted results with a minimum of effort. Since numbers are often the results of interest, a variety of numerical issues arise.

When linear programs are solved by the simplex method, degenerate basic variables that would take values of zero in exact computation may come out instead having slightly nonzero values, as seen in this AMPL display of a variable named `Make`:

```
ampl: display Make;

:          bands       coils       plate     :=
CLEV      1.91561e-14   1950     3.40429e-14
GARY   1125             1750   300
PITT    775              500   500
```

To suppress these distracting and meaningless entries, AMPL offers the option of specifying that all values less than some "display epsilon" be represented as zeros:

```
ampl: option display_eps 1e-10;

ampl: display Make;

:     bands  coils plate :=
CLEV      0   1950     0
GARY   1125   1750   300
PITT    775    500   500
```

Since tiny magnitudes might have significance in some applications, however, the default display epsilon is left at zero, so that the substitution of zeros never occurs without specific instructions from the modeler.

AMPL shows by default 6 digits of precision, with trailing zeroes suppressed:

```
ampl: display Level;

P1a   450.7
 P3   190.124
P3c  1789.33
```

More or fewer significant digits can be requested:

```
ampl: option display_precision 4;
ampl: display Level;

P1a   450.7
 P3   190.1
P3c  1789

ampl: option display_precision 9;
ampl: display Level;

P1a   450.700489
 P3   190.123755
P3c  1789.33403
```

In some cases, such as where results represent amounts of money, it makes sense to round not to a given precision but to a given decimal place:

```
ampl: display Price;

AA1  16.7051
AC1   5.44585
BC1  48.909
BC2   8.90899


ampl: option display_round 2;
ampl: display Price;

AA1  16.71
AC1   5.45
BC1  48.91
BC2   8.91
```

These displays rely on numerical routines that provide correctly rounded conversions. *Correctly rounded decimal-to-binary* conversions produce the binary value "closest" to a given decimal number, for a given binary representation and rounding sense (up or down). Clinger [5] showed how to compute these conversions in IEEE double-extended arithmetic, and Gay [16] adapted them to require only double-precision arithmetic. *Correctly rounded binary-to-decimal* conversions produce a decimal representation that, among all those having a specified number of significant digits or digits after the decimal point, comes closest to the original binary value when correctly rounded in a specified rounding sense. Methods for this purpose were proposed by Steele and White [27] and implemented more efficiently by Gay [16]. Gay's work in both cases was motivated in part by AMPL's need for these conversions.

AMPL also incorporates a special option for conversion to a "maximum precision" decimal representations:

```
ampl: option display_precision 0;
ampl: display Level;

P1a   450.70048877657206
 P3   190.12375501709528
P3c  1789.334027055151
```

This conversion's result is the *shortest* decimal representation that will yield the original binary representation when correctly rounded back to binary. This option has several uses: to ensure that different numbers in index sets always display differently, to provide for equivalent text and binary forms of communications to solvers, and to assist in diagnosing unexpected behavior of rounding operations.

Experiments reported in [16] showed that the computational times for typical correctly-rounded binary-to-decimal conversions are competitive with the times required by the then available standard C library routines — which did not produce correctly-rounded results. Even conversion to maximum precision is not prohibitively slow, requiring usually between 3 and 10 times the effort of correct binary-to-decimal conversion to 6 places.

As an example of the diagnostic function of maximum precision, consider these results, from a scheduling model, that show how many people are to work each of several numbered schedules:

```
ampl: option display_eps .000001;

ampl: display Work;

 10 28.8    73 28
 18  7.6    87 14.4
 24  6.8   106 23.2
 30 14.4   109 14.4
 35  6.8   113 14.4
 66 35.6   123 35.6
 71 35.6
```

To get implementable results, we might round each fractional number of people up to the next highest integer. Then the total required workforce should be given by

```
ampl: display sum {j in SCHEDS} ceil(Work[j]);
sum{j in SCHEDS} ceil(Work[j]) = 273
```

When we compute the same total explicitly, however, it comes out with two people fewer:

```
ampl: display 29+8+7+15+7+36+36+28+15+24+15+15+36;
29 + 8 + 7 + 15 + 7 + 36 + 36 + 28 + 15 + 24 + 15 + 15 + 36 = 271
```

The anomaly can be resolved by displaying all nonzero values at maximum precision:

```
ampl: option display_eps 0, display_precision 0;

ampl: display Work;

 10 28.799999999999997        73 28.000000000000018
 18  7.599999999999998        87 14.399999999999995
 24  6.79999999999999         95 -5.876671973951407e-15
 30 14.40000000000001        106 23.200000000000006
 35  6.799999999999995       108  4.685288280240683e-16
 55 -4.939614313857677e-15   109 14.4
 66 35.6                     113 14.4
 71 35.599999999999994       123 35.59999999999999
```

We can see here that `Work[73]` and `Work[108]`, which appeared to have values 28 and 0 when rounded to AMPL's default 6 digits of precision, are in fact slightly greater than those integer values and so are being unexpectedly rounded up.

For situations of this kind, AMPL provides a function `round` that rounds its first argument to the number of places after the decimal point specified by the second argument. Using this function in our example, the correct sum can be obtained by

```
ampl: display sum {j in SCHEDS} ceil(round(Work[j],6));
sum{j in SCHEDS} ceil(round(Work[j], 6)) = 271
```

Alternatively, the modeler may set an option `solution_round` to specify that *all* values returned by the solver be automatically rounded to a certain number of decimal places, when AMPL retrieves them from the solver. Setting this option to 6, for example, would cause our original expression `sum {j in SCHEDS} ceil(Work[j])` to come out to 271 as expected.

2. Presolve

A presolve phase attempts to reduce the numbers of variables and constraints in a problem instance before sending it to a solver. The underlying motivation for including a presolve phase in AMPL — rather than leaving this operation to each solver — is to provide consistent handling of simple bounds, which most solvers can treat specially to advantage. Explicit simple bounding constraints are removed and the bounds are folded into the definitions of the variables, so that it makes no difference whether a model declares

```
var Sell {PROD,1..T} >= 0;
subject to MLim {p in PROD, t in 1..T}: Sell[p,t] <= market[p,t];
```

or

```
var Sell {p in PROD, t in 1..T} >= 0, <= market[p,t];
```

As a result a modeler need not appreciate the importance of simple bounds to gain the computational advantages of them.

AMPL also incorporates a more powerful presolve, due to Brearley, Mitra and Williams [3], that uses known bounds together with linear constraints in order to deduce tighter bounds. The idea is simple. If we know for example that $l_j \le x_j \le u_j$ for $j = 1, \dots, n$, and if we have a constraint $\sum_{j=1}^{n} a_{rj} x_j \le b_r$ with $a_{rs} > 0$, then we can deduce

$$x_s \le \frac{1}{a_{rs}} \left(b_r - \sum_{\substack{j \in \mathcal{P} \\ j \ne s}} a_{rj} l_j - \sum_{\substack{j \in \mathcal{N} \\ j \ne s}} a_{rj} u_j \right),$$

where

$$\mathcal{P} \equiv \{ j = 1, \dots, n : \ a_{rj} > 0 \}, \quad \mathcal{N} \equiv \{ j = 1, \dots, n : \ a_{rj} < 0 \}.$$

Other cases are handled similarly, as described in [12]. Whenever any such inferred bound is tighter than a known bound, it replaces the known bound. Thus progressively tighter bounds are sometimes achieved by iterating several times through the nonzero coefficients a_{ij} of the relevant constraints.

This presolving procedure can reduce problem size in several ways. If the inferred bounds $l_s = u_s$, then x_s can be fixed and eliminated from the problem; if $l_s > u_s$ then no feasible solution is possible. If

$$\sum_{j \in \mathcal{P}} a_{rj} u_j + \sum_{j \in \mathcal{N}} a_{rj} l_j \le b_r,$$

then the rth constraint is redundant and can be dropped; if

$$\sum_{j \in \mathcal{P}} a_{rj} l_j + \sum_{j \in \mathcal{N}} a_{rj} u_j > b_r,$$

then no feasible solution is possible, and the same with $\ge$ implies that the constraint can be "fixed" to $\sum_{j=1}^{n} a_{rj} x_j = b_r$. (Again there are other cases handled similarly, and described in [12].) These are numerical tests, so in practice they may have to be carried out with respect to tolerances. In fact a variety of tolerance options have proved necessary to get AMPL's presolve to act the way modelers want and expect in a range of circumstances.

As a simple example, consider an AMPL model that specifies the data values as

```
set PROD;  # products

param avail >= 0;           # production hours available in a week
param commit {PROD} >= 0;  # lower limit on tons sold in a week
param market {PROD} <= 0;  # upper limit on tons sold in a week
```

and for which the variables `Make[p]` and single constraint `Time` are defined as follows:

```
var Make {p in PROD} >= commit[p], <= market[p];

subject to Time: sum {p in PROD} (1/rate[p]) * Make[p] <= avail;
```

If we set hours available to 13, then AMPL's presolve phase reports that no feasible solution is possible:

```
ampl: let avail := 13;
ampl: solve;

presolve: constraint Time cannot hold:
    body <= 13 cannot be >= 13.2589; difference = -0.258929
```

The reason for this infeasibility is not hard to see. If in the constraint we use the lower bounds `commit[p]` as the values of the variables, we see that hours available must be at least 13.2589 (to six digits) to allow the minimum possible production:

```
ampl: display sum {p in PROD} (1/rate[p]) * commit[p];
sum{p in PROD} 1/rate[p]*commit[p] = 13.2589
```

AMPL's previous message that "`body <= 13 cannot be >= 13.2589`" is reporting this same observation.

Here are AMPL's responses to three more values of `avail`:

```
ampl: let avail := 13.2589;
ampl: solve;

presolve: constraint Time cannot hold:
    body <= 13.2589 cannot be >= 13.2589; difference = -2.85714e-05

ampl: let avail := 13.25895;
ampl: solve;

MINOS 5.5: optimal solution found.
0 iterations, objective 61750.10714

ampl: let avail := 13.258925;
ampl: solve;

presolve: constraint Time cannot hold:
    body <= 13.2589 cannot be >= 13.2589; difference = -3.57143e-06
Setting $presolve_eps >= 4.29e-06 might help.
```

In the first case we see that 13.2589 is not quite enough; evidently it is rounded down when it is converted to six significant decimal digits. In the second case, a slightly higher value, 13.25895, proves sufficient to allow a feasible solution, and hence an optimal value can be reported. For the third case, the value has been taken between the previous two, at 13.258925. Again presolve reports no feasible solution, but it suggests a new tolerance setting that might make a difference.

There are actually two tolerances at work in this example. First, infeasibility is detected in the rth constraint if (continuing our example)

$$b_r - \sum_{j \in \mathcal{P}} a_{rj} l_j - \sum_{j \in \mathcal{N}} a_{rj} u_j$$

is less than the negative of the value in the AMPL option `presolve_eps`. Once infeasibility has been detected, presolve also calculates a (necessarily larger) hypothetical value of `presolve_eps` that might lead to a determination of feasibility instead. This value is reported if it is considered small enough to be meaningful — specifically, if it is no larger than the value in AMPL option `presolve_epsmax`. Presolve reports that this increase "might" make a difference because actions of other presolve steps may have additional effects that cannot be quickly foreseen.

The tolerances `presolve_eps` and `presolve_epsmax` relate to determining infeasibility through inconsistent inferred bounds on variables or constraints. Analogous tolerances `presolve_fixeps` and `presolve_fixepsmax` play the same roles in the determination of whether bounds are close enough that a variable should be fixed or an inequality constraint should be made an equality. And also the rth constraint is dropped as redundant when

$$b_r - \sum_{j \in \mathcal{P}} a_{rj} u_j - \sum_{j \in \mathcal{N}} a_{rj} l_j$$

is at least as large as a tolerance given by `constraint_drop_tol`.

All three of `presolve_eps`, `presolve_fixeps`, and `constraint_drop_tol` are set by default to zero, allowing in effect no tolerance for computational imprecision. Experience suggests that, when the presolve routines can be written to use the *directed rounding* — lower bounds toward $-\infty$ and upper bounds toward $+\infty$ — that is available with IEEE arithmetic [24], then decisions on infeasibility, equality, and redundancy are made correctly without any assistance from explicit tolerances (as long as the problem involves exactly known data). The tolerance options are retained, however, for use on platforms that do not offer directed rounding and with problems having data of limited precision.

A final set of tolerances relate to variables that are allowed to take only integer values. When a new lower bound is inferred for an integer variable, that bound may be immediately rounded up to the next highest integer. Similarly, an inferred upper bound may be immediately rounded down. In these situations the AMPL presolver must again take care not to round up a bound that is only slightly greater than an integer, or to round down a bound that is only slightly less than an integer. Presolve thus only rounds a lower bound l_j up or an upper bound u_j down if $l_j - \mathrm{floor}(l_j)$ or $\mathrm{ceil}(u_j) - u_j$ is greater than a given tolerance. The tolerance value is taken from the value of the AMPL option `presolve_inteps`, which is set to `1e-6` by default. Messages suggesting that a slightly higher value of the tolerance might make a difference are governed by an option `presolve_intepsmax` that works with `presolve_inteps` in the same way that `presolve_epsmax` works with `presolve_eps`.

Presolve operations can be performed very efficiently. For a run that reported

```
Presolve eliminates 1769 constraints and 8747 variables.

19369 variables, all linear
3511 constraints, all linear; 150362 nonzeros
1 linear objective; 19369 nonzeros
```

AMPL's processing required only about 2 seconds in total on a fast PC, of which about $1/2$ second was in presolve. For a larger run that reported

```
Presolve eliminates 32989 constraints and 54819 variables.

327710 variables, all linear
105024 constraints, all linear
1 linear objective; 317339 nonzeros
```

AMPL's processing time was 23 seconds, of which only 4 were in presolve.

3. Modeling language influences on solvers

In addition to linear and mixed-integer programs, algebraic modeling languages can express a broad variety of nonlinear programs. The expressive abilities of modeling languages have in fact at times gone beyond the computational abilities of available solvers. Extensions to solver technology have come about as a result.

We summarize here two cases in which the generality of AMPL has encouraged new solver developments. One involves higher-order derivatives, and the other an extension for specifying equilibrium conditions.

Second derivatives. Expressing a nonlinear program in an algebraic modeling language involves nothing more than using the available operators to form nonlinear expressions, possibly in conjunction with elementary nonlinear functions. The following is for example a nonlinear objective function in AMPL:

```
minimize Energy:
   sum {(j,k) in PAIRS} x[j,k] *
      (c[j,k] + log (x[j,k] / sum {m in J: (m,k) in PAIRS} x[m,k]));
```

Nonlinear constraints are written using the same sorts of expressions along with relational operators.

Given a nonlinear model and a data instance, AMPL generates an expression graph for the nonlinear part of each objective and constraint. The AMPL version of a nonlinear solver incorporates a *driver* that sets up a data structure to represent the expression graphs. The solver later calls AMPL library routines that use the graph data structures to evaluate the nonlinear functions; specifically, the solver passes a current iterate to the AMPL routines, and gets back the values of the nonlinear function at that iterate. A detailed explanation of this process is given by Gay [20]. (Solvers that work directly with expression graphs can also be accommodated by AMPL, but we are concerned here only with conventional solvers that address nonlinearities through function evaluations.)

Many solvers require gradients as well as function values. AMPL's evaluation routines apply *automatic differentiation* [21] to compute gradient values at the same time as function values. As described by Gay [17], AMPL employs in particular an approach known as "backward" AD, which makes two passes through the graph for each nonlinear expression $f(x)$:

- ▷ a forward sweep computes $f(x)$ and saves information on $\partial o/\partial a_i$ for each argument a_i to each operation o;
- ▷ a backward sweep recursively computes $\partial f/\partial o$ for each operation o, and hence ultimately $\nabla f(x)$.

The work of computing the gradient vector can be shown to be bounded by a small multiple of the work of computing the function value alone, though possibly at a large cost in additional space in the expression-graph data structure. Backward AD is more accurate and efficient than finite differencing, and has substantially better worst-case efficiency than straightforward symbolic differentiation.

The concepts of automatic differentiation can be applied as well to computing second-derivative information. Hessian-vector products $\nabla^2 f(x)v$ can be determined by applying backward AD to compute $v^T\nabla f(x)$, or equivalently to compute

$\nabla_x(df(x+\tau v)/d\tau|_{\tau=0})$. The entire Hessian can thus be determined through the Hessian-vector products $\nabla^2 f(x)e_j$ with the n unit vectors. Alternatively, if the function of interest can be determined to have a group partially separable structure [6, 22, 23],

$$f(x) \;=\; \sum_{i=1}^{q} \theta_i \left(\sum_{j=1}^{r_i} \phi_{ij}(U_{ij}x) \right) \;=\; \sum_{i=1}^{q} \theta_i \left(\Phi_i(x) \right),$$

where U_{ij} is $m_{ij} \times n$ with $m_{ij} \ll n$, $\phi_{ij} \colon \Re^{m_{ij}} \mapsto \Re$, and $\theta_i \colon \Re \mapsto \Re$, then the gradient and Hessian also have simplified structures,

$$\begin{aligned} \nabla f(x) &= \sum_{i=1}^{q} \theta_i'(\Phi_i(x)) \nabla \Phi_i(x), \\ \nabla^2 f(x) &= \sum_{i=1}^{q} \left(\theta_i'(\Phi_i(x)) \nabla^2 \Phi_i(x) + \theta_i''(\Phi_i(x)) \nabla \Phi_i(x) \nabla \Phi_i(x)^T \right) \end{aligned}$$

where

$$\nabla \Phi_i(x) = \sum_{j=1}^{r_i} U_{ij}^T \nabla \phi_{ij}(U_{ij}x), \;\; \nabla^2 \Phi_i(x) = \sum_{j=1}^{r_i} U_{ij}^T \, \nabla^2 \phi_{ij}(U_{ij}x) \, U_{ij}.$$

Thus the Hessian can be computed from sums of outer products involving the generally much smaller Hessians of the component functions ϕ_{ij}.

The AMPL function-evaluation routines have been extended as described in Gay [18, 19] to provide Hessians in dense or sparse form, as a full (symmetric) matrix or just the lower triangle. Hessian-vector products are also available, to be used for example in iteratively solving the linear equations that define the steps taken by some algorithms. Given Lagrange multipliers as well as the current iterate, the routines return the Hessian of the Lagrangian or its product with a vector. AMPL's library routines also perform an initial walk of the expression graphs to detect group partially separable structure, using a hashing scheme to spot common subexpressions; the modeler need not identify this structure.

How has this feature been critical to nonlinear solver development? Its addition to AMPL encouraged researchers' interest in extending interior-point methods to nonlinear optimization problems. These methods can be viewed as solving a nonlinear problem such as

$$\begin{array}{ll} \text{Minimize} & f(x) \\ \text{Subject to} & h_i(x) \ge 0, \quad i = 1, \ldots, m \end{array}$$

by applying a modified Newton's method to the "centered" Karush-Kuhn-Tucker optimality conditions,

$$\nabla f(x) = \nabla h(x)^T y, \;\; h(x) = w, \;\; WYe = \mu e,$$

where $W = \operatorname{diag}(w)$, $Y = \operatorname{diag}(y)$. The result is a Newton linear system of the form

$$\left[\begin{array}{cc} -\nabla^2 \left(f(x) - h(x)^T y \right) & \nabla h(x)^T \\ \nabla h(x) & WY^{-1} \end{array} \right] \left(\begin{array}{c} \Delta x \\ \Delta y \end{array} \right) = \cdots$$

that incorporates the Hessian of the Lagrangian in the upper left-hand block.

Given this extension to AMPL, a solver developer need only add an AMPL driver to have full access to the necessary second-derivative information. Hundreds of standard test problems — as well as any of the developer's own benchmarks — are immediately available, without any need to write and debug new code for Hessian computations. This facility has contributed to rapid development of new interior-point codes such as LOQO [28], KNITRO [4], and MOSEK [1]. The success of these codes has in turn stimulated developers of other kinds of codes, such as CONOPT [2] and PATH [9], to pursue possibilities for taking advantage of second-derivative information.

Complementarity problems. A classical complementarity condition says that two inequalities must hold, at least one with equality. Collections of complementarity conditions, known as complementarity problems, arise as equilibrium conditions for problems in economics and in engineering, and represent optimality conditions for nonlinear programs, bi-level programs, and bi-matrix games.

To support solvers designed specifically for complementarity problems, the operator `complements` has been added to the AMPL language [10]. Thus it is possible to state a collection of conditions such as

```
subject to Lev_Compl {j in ACT}:
   Level[j] >= 0 complements
      sum {i in PROD} Price[i] * io[i,j] <= cost[j];
```

AMPL also provides support for "mixed" complementarity conditions, which hold between a double inequality and an expression:

```
subject to Lev_Compl {j in ACT}:
   level_min[j] <= Level[j] <= level_max[j] complements
      cost[j] - sum {i in PROD} Price[i] * io[i,j];
```

This condition is satisfied if the double-inequality holds with equality at its lower limit and the expression is ≥ 0, or if the double-inequality holds with equality at its upper limit and the expression is ≤ 0, or otherwise if the expression $= 0$.

The initial motivation for complementarity conditions in AMPL was to support codes such as PATH [7] that solve "square" systems of such conditions — where the number of variables equals the number of complementarity conditions plus the number of equality constraints. Once a convenient notation for complementarity conditions was available, however, there was nothing to stop modelers from writing non-square systems and adding objective functions. Indeed, uses for these "mathematical programs with equilibrium constraints" — or MPECs — are of growing interest.

Thus the availability of complementarity conditions in AMPL encouraged development of algorithmic approaches to solving general MPECs, especially through the application of nonlinear optimization methods to modifications of the MPEC conditions [8, 11, 25, 26]. The necessary modifications can be carried out by a solver driver, so that AMPL's convenient representation of the complementarity conditions is maintained at the modeler's level.

References

[1] E.D. Andersen and K.D. Andersen, "The MOSEK Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm." In *High Performance Optimization,* H.Frenk, K.Roos, T.Terlaky, and S.Zhang, eds., Kluwer Academic Publishers (2000) pp. 197–232.

[2] ARKI Consulting & Development A/S, "The CONOPT Algorithm." At `www.conopt.com/Algorithm.htm`.

[3] A.L. Brearley, G. Mitra and H.P. Williams, "Analysis of Mathematical Programming Problems Prior to Applying the Simplex Method." *Mathematical Programming* **8** (1975) 54–83.

[4] R. Byrd, M.E. Hribar, and J. Nocedal, "An Interior Point Method for Large Scale Nonlinear Programming." *SIAM Journal on Optimization* **9** (1999) 877–900.

[5] W.D. Clinger, "How to Read Floating Point Numbers Accurately." In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation,* White Plains, NY, June 20-22, 1990, pp. 92–101.

[6] A.R. Conn, N.I.M. Gould, and Ph.L. Toint, *LANCELOT, a Fortran Package for Large-Scale Nonlinear Optimization (Release A).* Springer Series in Computational Mathematics 17, Springer-Verlag (1992).

[7] S.P. Dirkse and M.C. Ferris, "The PATH Solver: A Non-Monotone Stabilization Scheme for Mixed Complementarity Problems." *Optimization Methods and Software* **5** (1995) 123–156.

[8] M.C. Ferris, S.P. Dirkse and A. Meeraus, "Mathematical Programs with Equilibrium Constraints: Automatic Reformulation and Solution via Constrained Optimization." Numerical Analysis Group Research Report NA-02/11, Oxford University Computing Laboratory, Oxford University (2002), `web.comlab.ox.ac.uk/oucl/publications/natr/na-02-11.html`.

[9] M.C. Ferris and K. Sinapiromsaran, "Formulating and Solving Nonlinear Programs as Mixed Complementarity Problems." In *Optimization,* V.H. Nguyen, J.J. Strodiot and P. Tossings, eds., volume 481 of *Lecture Notes in Economics and Mathematical Systems,* Springer-Verlag (2000) pp. 132–148.

[10] M.C. Ferris, R. Fourer and D.M. Gay, "Expressing Complementarity Problems in an Algebraic Modeling Language and Communicating Them to Solvers." *SIAM Journal on Optimization* **9** (1999) 991–1009.

[11] R. Fletcher and S. Leyffer, "Numerical Experience with Solving MPECs as NLPs." University of Dundee Report NA/210 (August 2002). At `www.mcs.anl.gov/~leyffer/MPEC-num-2.ps.Z`.

[12] 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, Dordrecht, 1994, pp. 135–154.

[13] R. Fourer and D.M. Gay, "Numerical Issues and Influences in the Design of Algebraic Modeling Languages for Optimization." Slides for presentation at the 20th Biennial Conference on Numerical Analysis, University of Dundee, Scotland, 24-27 June 2003. At `www.ampl.com/REFS`.

[14] R. Fourer, D.M. Gay and B.W. Kernighan, "A Modeling Language for Mathematical Programming." *Management Science* **36** (1990) 519–554.

[15] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming,* 2nd edition. Duxbury Press, Belmont, CA (2003).

[16] D.M. Gay, "Correctly Rounded Binary-Decimal and Decimal-Binary Conversions." Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, Murray Hill, NJ (1990). At `www.ampl.com/REFS/rounding.pdf`.

[17] D.M. Gay, "Automatic Differentiation of Nonlinear AMPL Models." In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application,* A. Griewank and G. Corliss, eds., SIAM, Philadelphia, PA (1991) pp. 61–73.

[18] D.M. Gay, "More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability." In *Computational Differentiation: Techniques, Applications, and Tools,* M. Berz, C. Bischof, G. Corliss and A. Griewank, eds., SIAM, Philadelphia, PA (1996) pp. 173–184.

[19] D.M. Gay, "Automatically Finding and Exploiting Partially Separable Structure in Nonlinear Programming Problems." Technical report, Bell Laboratories, Murray Hill, NJ (1996). At `www.ampl.com/REFS/psstruc.pdf`.

[20] D.M. Gay, "Hooking Your Solver to AMPL." Technical report, Bell Laboratories, Murray Hill, NJ (1993; revised 1994, 1997). At `www.ampl.com/REFS/hooking2.pdf` or `www.ampl.com/REFS/HOOKING`.

[21] A. Griewank, "On Automatic Differentiation." In *Mathematical Programming: Recent Developments and Applications,* M. Iri and K. Tanabe, eds., Kluwer Academic Publishers, Dordrecht (1989) pp. 83–107.

[22] A. Griewank and Ph.L. Toint, "On the Unconstrained Optimization of Partially Separable Functions." In *Nonlinear Optimization 1981,* M.J.D. Powell, ed., Academic Press (1982) pp. 301–312

[23] A. Griewank and Ph.L. Toint, "Partitioned Variable Metric Updates for Large Structured Optimization Problems." *Numerische Mathematik* **39** (1982) 119–137.

[24] *IEEE Standard for Binary Floating-Point Arithmetic,* Institute of Electrical and Electronics Engineers, New York, NY (1985). ANSI/IEEE Standard 754-1985.

[25] S. Leyffer, "Mathematical Programs with Complementarity Constraints." Argonne National Laboratory Preprint ANL/MCS-P1026-0203 (February 2003). At `www.mcs.anl.gov/~leyffer/mpec-survey.ps.gz`.

[26] S. Leyffer, "Complementarity Constraints as Nonlinear Equations: Theory and Numerical Experiences." Argonne National Laboratory Preprint ANL/MCS-P1054-0603 (June 2003). At `www.mcs.anl.gov/~leyffer/MPEC-NCP-02.ps.gz`.

[27] G.L. Steele and J.L. White, "How to Print Floating-Point Numbers Accurately." In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation,* White Plains, NY, June 20-22, 1990, pp. 112–126.

[28] R.J. Vanderbei, "LOQO: an Interior Point Code for Quadratic Programming." *Optimization Methods and Software* **11-12** (1999) 451–484.