Chapter #999

Conveying Problem Structure from an Algebraic Modeling Language to Optimization Algorithms

Robert Fourer and David M. Gay
Northwestern University; Bell Laboratories

Key words: Optimization, mathematical programming, linear programming, modeling languages

Abstract: Optimization algorithms can exploit problem structures of various kinds, such as sparsity of derivatives, complementarity conditions, block structure, stochasticity, priorities for discrete variables, and information about piecewise-linear terms. Moreover, some algorithms deduce additional structural information that may help the modeler. We review and discuss some ways of conveying structure, with examples from our designs for the AMPL modeling language. We show in particular how "declared suffixes" provide a new and useful way to express structure and acquire solution information.

1. INTRODUCTION

A modeling language can provide a useful way to express the elaborate optimization problems that often arise in practice. Many of these problems have structure that an optimization algorithm can exploit, such as sparsity of first and second derivatives, complementarity conditions, block structure, time-dependent stochasticity, priorities for discrete variables, and information about piecewise-linear terms. Moreover, some algorithms deduce additional structural information that may help the modeler — the person who formulated the problem — to understand whether it is the intended problem or how the formulation needs to be changed.

In this paper, we review some ways of conveying structure to and from solution algorithms and discuss experience with the AMPL modeling

language (Fourer, Gay and Kernighan 1990, 1993). In particular, declared suffixes, a recent addition to AMPL, provide a new and useful way to express structure and acquire solution information.

The next section of this paper briefly describes the general form of mathematical programming problems and some of their structure. Then §3 introduces builtin and declared suffix notations, and §4 provides several examples of declared suffixes. Some other kinds of structural information are discussed in §5, and the potential for future uses of the suffix feature is addressed by the concluding remarks in §6.

2. BASIC PROBLEM STRUCTURE

Mathematical programming problems can generally be stated in the following form: given smooth functions $f: \Re^n \to \Re$ and $c: \Re^n \to \Re^m$, m-vectors l and u, and n-vectors $\underline{x}$ and $\bar{x}$, with components $l_i, \underline{x}_i \in [-\infty, \infty)$, $u_i, \bar{x}_i \in (-\infty, \infty]$, $l_i \le u_i$, and $\underline{x}_i \le \bar{x}_i$,

$$\begin{aligned} &\text{minimize} && f(x) && (1a)\\ &\text{subject to} && l \le c(x) \le u && (1b)\\ &\text{and} && \underline{x} \le x \le \bar{x}, && (1c) \end{aligned}$$

with inequalities understood componentwise. Some components of x may also be required to be integers. Although the simple-bound constraints (1c) could be folded into the general constraints (1b), AMPL distinguishes these constraint types in the information it makes available to solution algorithms, since many algorithms gain efficiency by dealing separately with simple-bound constraints, and some only handle such constraints.

Algorithms for solving (1) often use the first derivatives of f and c, the gradient $\nabla f(x)$ and Jacobian matrix $J(x) = \nabla c(x)$ with $J_{ij} = \partial c_i / \partial x_j$. In large problems, it is important to know and exploit the sparsity structure of J (that is, the knowledge of which components of J are always zero).

Solvers — implementations of solution algorithms — are generally idiosyncratic in how one conveys problems to them. Often they require subroutines (or, in object-oriented parlance, methods) that compute f, c and the requisite derivatives, but they may also operate by reverse communication, returning to the caller with a request to be called again with arguments containing updated problem information, such as function and constraint values and derivatives. Because of the great variety of solver interfacing requirements, we have found it convenient to prepare an AMPL/solver interface library (Gay 1997) to assist in providing various kinds of problem informa-

tion. Solver-specific *drivers* obtain problem information from calls to the library, transform the information as necessary, and convey it to the solver.

Information provided to drivers from AMPL/solver interface library calls includes problem statistics, sparsity, and routines for computing constraint and objective functions and their first and second derivatives. Detailed instructions for using the library appear in Gay (1997). Efficiently computing second derivatives (in the form of the Hessian of the Lagrangian function) explicitly or in the form of Hessian-vector products is a particular challenge. We use backwards automatic differentiation for both gradient and Hessian computations; see Gay (1991, 1996) for more details and references.

3. SUFFIXES

This section examines AMPL's suffix notations and their uses for conveying structure. AMPL suffixes are of two kinds, builtin and declared. Builtin suffixes have values derived from the current problem state, and are found in some form in most large-scale modeling languages. Declared suffixes may be assigned values by the modeler or by solvers, and are a relatively recent addition to the AMPL language; we are not aware of any attempts to extend other modeling languages with any comparable feature.

3.1 Motivation for declared suffixes

Over the course of AMPL's development, we have found it convenient to make various kinds of auxiliary information about variables and constraints available via "dot" or "suffix" notations, which involve appending a period ("dot") to the possibly subscripted name of a model entity, followed by a suffix name. For example, if `Buy[j]` denotes a variable that represents the amount of food `j` to be purchased, then `Buy[j].rc` refers to that variable's reduced cost; a display of the values and reduced costs of all such variables is produced by the command `display Buy, Buy.rc`. Suffixes may also appear on so-called generic synonyms for variables, so that `display _varname, _var, _var.rc` will produce a table of all variables, with their names in the first column, their values in the second, and their reduced costs in the third. Table 1 shows AMPL's builtin suffixes for problem and solution information associated with variables.

Builtin suffixes work in a similar way to denote values associated with constraints and with objectives. Complete tables of builtin suffix names appear in Fourer, Gay and Kernighan (1993), pp. 320–322, and in `http://www.ampl.com/ampl/NEW/suffbuiltin.html`.

Table 1. AMPL builtin suffixes for variables

Suffix	Associated value
`.lb .ub`	Lower, upper bound
`.lb0 .ub0`	Bounds from variable's definition (`var` statement)
`.lb1 .ub1`	Weaker bounds from presolve
`.lb2 .ub2`	Stronger bounds from presolve
`.init`	Most recent initial value
`.init0`	Original initial value
`.rc`	Reduced cost
`.lrc`	Lower bound reduced cost
`.urc`	Upper bound reduced cost
`.slack`	Slack (distance from bound)
`.lslack`	Lower bound slack
`.uslack`	Upper bound slack

Particular kinds of solvers can often make use of additional information associated with optimization model components. For example, solvers that employ the simplex method maintain a basis status for each variable and constraint of a problem. Communicating a good starting basis along with a problem can save these solvers considerable time. When solving a problem from scratch, a suitable starting basis may be hard to guess, but many applications involve the solution of a sequence of related problems, and the final basis for one problem may make a good initial basis for the next one. Thus it may be desirable to receive basis status information from a solver as well as to send it back later.

As another example, when some variables are restricted to integer values, the solver may need to carry out a branch-and-bound process, searching a tree of progressively more restricted cases so as to continually improve known solutions and bounds. Supplementary information associated with integer variables can enable the solver to make much more effective choices of the cases to be analyzed at each level of the search.

Auxiliary information can also arise as a result of applying a solver, in which case a means is needed for conveying it back to the modeling language's user. If a linear programming problem is unbounded, for example, it is sometimes useful to know a ray along which the objective can decrease without bound. The specification of the ray's direction associates an additional floating-point number with each variable.

We can never anticipate all the kinds of auxiliary information that solvers will find useful, either when solving a problem or when communicating information about the solution(s) it has found. Therefore we have chosen to permit declarations of suffixes, an open-ended scheme that adapts easily to the needs of various solvers and the problem classes they address. Each declared suffix associates one auxiliary value with each variable, constraint, objective, or problem subset of an AMPL model.

3.2 Declarations of suffixes

AMPL requires each model entity to be introduced by a declaration before it is used. Examples of such entities are parameters, sets, variables, constraints, and objective functions. Declared suffixes are a new entity that must be introduced by a declaration.

Parameters in AMPL are named data values, introduced by `param` declarations. Often it is natural for a parameter to assume only numeric values, but sometimes it is convenient to have string-valued parameters. Since numeric values are more common, AMPL's convention is that a parameter is restricted to numeric values unless the keyword `symbolic` appears in its declaration.

Declared suffixes are introduced by a `suffix` declaration. For example,

```
suffix priority;
```

introduces suffix `.priority`. For consistency with AMPL's declarations of parameters, and because we expect that numeric-valued declared suffixes will be more common than string-valued ones, a declared suffix is also restricted to having numeric values unless `symbolic` appears in its declaration.

In addition to suffixes declared explicitly by a modeler, AMPL provides a few "predeclared" suffixes for common purposes, such as handling the previously mentioned basis statuses. Solvers are also provided with a mechanism for causing suffixes to be defined. Examples of these options appear in subsequent sections.

For convenience in communicating with solvers (by explicitly transmitting only nonzero suffix values), all declared suffix values are initially zero. We originally intended to allow a default-value clause in suffix declarations, analogous to the existing default-value clause in parameter declarations. The expression for the default value may involve model parameters, however, whose values may be changed by assignments and various other mechanisms. In some of these cases, it is not clear when the default expression should be evaluated, and any of the reasonable resolutions of this issue would complicate the specification and implementation of suffixes. (Similarly, for simplicity, derived parameters — whose values are permanently defined by expressions in their declarations — may not depend on declared suffixes.)

3.3 Symbolic suffix tables

Although many kinds of auxiliary information are naturally numeric, some information is more conveniently considered to be symbolic. Solvers that maintain a basis, for instance, generally place each variable and constraint slack into one of a small number of categories such as “basic” or “nonbasic at upper bound”. It is convenient to associate short symbolic names with these categories, and symbolic declared suffixes are valuable for this purpose.

In the uses we have so far seen for symbolic declared suffixes, only a relatively few symbolic values are of interest. For convenience and efficiency in communicating with solvers, we use a kind of “super sparsity” to maintain symbolic suffix values and exchange them with solvers: symbolic suffixes always have an underlying numeric value that is associated with a string value by a “suffix table”. For example, `.sstatus` is predeclared to serve as a symbolic suffix for recording the basis status of variables and of (the slack or artificial variables associated with) constraints. The default value for the suffix table associated with `.sstatus` is

```
0  none  no status assigned
1  bas   basic
2  sup   superbasic
3  low   nonbasic <= (normally =) lower bound
4  upp   nonbasic >= (normally =) upper bound
5  equ   nonbasic at equal lower and upper bounds
6  btw   nonbasic between bounds
```

Numeric values in the first column are associated with string values in the second column, and the rest of each line is an optional description of the value. Thus the default numeric value 0 is associated with `none`, 1 is associated with `bas`, 2 with `sup`, and so forth. The numeric values themselves can be accessed by appending `_num` to the name of a symbolic suffix. For example,

```
ampl: model diet.mod;
ampl: data diet2a.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032
```

```
ampl: display _varname, _var,
ampl?   _var.sstatus, _var.sstatus_num;
:  _varname        _var  _var.sstatus _var.sstatus_num :=
1  "Buy['BEEF']"    5.36061   bas              1
2  "Buy['CHK']"     2         low              3
3  "Buy['FISH']"    2         low              3
4  "Buy['HAM']"    10         upp              4
5  "Buy['MCH']"    10         upp              4
6  "Buy['MTL']"    10         upp              4
7  "Buy['SPG']"     9.30605   bas              1
8  "Buy['TUR']"     2         low              3
;
```

The correspondence between symbolic suffixes and suffix tables is maintained through AMPL *environments,* which are collections of name-value pairs. Some common operating systems, such as Unix, MS-DOS, and Microsoft Windows, maintain an environment, and AMPL inherits its initial environment from the operating system (and supplies default values for some environment variables if they do not appear in the operating system's environment). AMPL's `option` command can display or change environment values, and these values can also be accessed in commands by a "dollar" notation, a `$` followed by the name of the environment variable.

Various environment variables affect AMPL's behavior. For example, `$solver` determines which solver AMPL invokes in response to a `solve` command. Solvers run as separate programs, and when AMPL invokes a solver or other program, it passes along the current environment. Solvers often respond to particular environment variables. For example, by convention appending `_options` to the name of a solver gives an associated environment variable that can be used to specify values for operational parameters and switches. Thus the value of `$minos_options`, for example, affects the behavior of solver `minos`.

The situation with suffix tables for symbolic suffixes is analogous: appending `_table` to the suffix name gives the name of an environment variable that contains the suffix table. For example, the AMPL command `print $sstatus_table` causes the table shown above to be displayed (so long as `$sstatus_table` has its default value).

What about `_num` values that do not appear in the suffix table? The suffix table mechanism is also used in situations where it is convenient for a range of numeric values to correspond to a single symbolic value. For example, the results of executing a solver may fall into one of the six broad categories shown in AMPL's default `$solve_result_table`:

```
0     solved
100   solved?
200   infeasible
300   unbounded
400   limit
500   failure
```

Solvers can return a value for the builtin parameter `solve_result_num` that gives more detail about how they fared. Often it suffices to base a test on the broad category, however, which is available in the builtin symbolic parameter `solve_result`. The value that AMPL assigns to `solve_result` is taken from the last line of `$solve_result_table` whose numerical value is less than or equal to `solve_result_num`. (For `solve_result_num` < 0, `solve_result` is `"?"`.)

Suffix tables for declared symbolic suffixes work analogously to `$solve_result_table`, except that `_num` values less than the first value in the suffix table are simply rendered as the `_num` value itself.

3.4 Declarations and manipulations

AMPL offers various facilities for consistency checking. For example, a parameter's declaration may specify inequalities and set memberships that the parameter must satisfy. A suffix declaration can similarly specify inequalities and set memberships that the suffix must satisfy. For instance, the `.priority` suffix could be limited to nonnegative values less than ten thousand by declaring it as

```
suffix priority >= 0, < 10000;
```

One of the keywords `IN`, `OUT`, `INOUT`, or `LOCAL` may also appear in a suffix declaration to indicate whether the suffix should be only sent to solvers (`IN`), only returned by solvers (`OUT`), both sent to and returned by solvers (`INOUT`, the default), or neither sent to nor returned by solvers (`LOCAL`).

Standard commands for manipulating AMPL model components also work for declared suffixes: one can remove them with a `delete` command, reset all values for a particular suffix or for all suffixes to 0 with a `reset suffix` command, modify a suffix's declaration with a `redeclare suffix` statement, and exhibit the current declaration for a suffix with a `show suffix` command.

4. EXAMPLES OF DECLARED SUFFIXES

To exhibit the usefulness of the declared suffix concept, we next describe a few specific cases in more detail. We first consider suffixes (of type `IN`) defined by the modeler and subsequently recognized by certain solvers. (Solvers may simply ignore any suffixes they receive but are not prepared to recognize.) We then describe some uses of suffixes (of type `OUT`) that may be declared by solvers so as to return auxiliary information relating to a solution.

4.1 Suffixes declared by the modeler

For solvers that use a branch-and-bound algorithm to handle integer variables, the fundamental operation is to "branch" on a variable by dividing its domain in two. Often a human modeler can offer useful guidance about the order in which to branch on integer variables and about which branch to try first. For example, it is usually a good idea to first branch on a variable that controls whether a facility is built, and only later to branch on variables that specify details of the facility. If the former variable takes the value 1 or 0 when the facility is or isn't built, then it may be better to branch first in the "up" direction, so that a value of 1 is tried initially.

To provide for such guidance, some solvers recognize the declared suffixes `.priority` and `.direction`. Here is a simple illustration that settings of these suffixes can make a difference:

```
ampl: model multmip3.mod; data multmip3.dat;
ampl: option solver cplex;

solve;
CPLEX 6.0.1: optimal integer solution; objective 235625
602 simplex iterations
91 branch-and-bound nodes

ampl: reset; model multmip3.mod; data multmip3.dat;
ampl: suffix priority;
ampl: let {i in ORIG, j in DEST} Use[i,j].priority :=
ampl?    sum {p in PROD} demand[j,p];
ampl: suffix direction;
ampl: let Use["GARY","FRE"].direction := -1;

solve;
CPLEX 6.0.1: optimal integer solution; objective 235625
447 simplex iterations
64 branch-and-bound nodes
```

(With the subsequent version 6.5 of this solver, the above choices of `.priority` and `.direction` require more rather than fewer nodes and iterations than the default settings. In general the best choice depends not only on the problem, but also on the implementation of the solution algorithm.)

Suffixes also seem a natural candidate for conveying certain auxiliary block structure to solvers. The implementation of this idea is a project for the future, however, as indicated in §6.

4.2 Suffixes declared by solvers

Information frequently returned by solvers is handled by AMPL's builtin suffixes. Declared suffixes provide a complementary mechanism for returning auxiliary information in particular circumstances.

The AMPL/solver interface library permits solvers to declare and return arbitrary suffixes to AMPL, and to supply corresponding `_table` options for symbolic suffixes. When reading a solution file that introduces a new suffix, AMPL echoes the new declaration to inform the modeler about it.

For example, if a linear programming problem is unbounded, the solver may find a direction of infeasibility — a ray along which the objective function can decrease without bound. Such a direction is essential to iterative schemes, such as Benders and Dantzig-Wolfe decomposition, that generate a certain column or cut based on each ray returned from an unbounded subproblem. Several solvers can send AMPL the direction vector for such a ray by declaring a new suffix, `.unbdd`, on variables. The solver introduces this suffix into an AMPL session only if an unbounded problem is encountered.

When a problem is infeasible, the modeler may want help in diagnosing the source(s) of infeasibility. One helpful technique offered by some solvers is to identify an irreducible infeasible subset (or IIS) of constraints and variable bounds (Chinneck and Dravnieks 1991). Since this computation may be time-consuming, and since an IIS is not always wanted, the usual arrangement is that the solver only computes an IIS on request. When it does identify an IIS, the solver can return it to AMPL via a symbolic suffix on variables and constraints, conventionally named `.iis`. In the following example from an AMPL session, a `solve` command first reveals that the problem is infeasible:

```
ampl: model diet.mod; data diet2.dat;
ampl: option solver osl; solve;
OSL 2.0: primal infeasible; objective 164.8854098
8 dual simplex iterations
```

Then the solver directive `iisfind=1` is set, causing a second `solve` to return an IIS:

```
ampl: option osl_options 'iisfind=1'; solve;
OSL 2.0: iisfind=1
OSL 2.0: primal infeasible; objective 164.8854098
0 dual simplex iterations
Returning iis of 7 variables and 2 constraints.

suffix iis symbolic OUT;

option iis_table '\
0     non   not in the iis\
1     low   at lower bound\
2     fix   fixed\
3     upp   at upper bound\
';
```

(The second `solve` requires 0 simplex iterations because AMPL has sent it the previously described `.sstatus` values that were returned by the first `solve`.)

The `.iis` suffix can now be used to view the IIS. If we are not willing to consider changes to the bounds on the variables, then it is sufficient to examine the constraints whose IIS status is other than `non`:

```
ampl: display {i in 1.._ncons: _con[i].iis != "non"}
ampl?    (_conname[i],_con[i].iis);

:   _conname[i]  _con[i].iis  :=
3   "diet['B2']"   low
5   "diet['NA']"   upp
;
```

Here we can conclude that, to achieve a feasible solution, we will at least have to relax either the lower limit constraint on `B2` or the upper limit constraint on `NA` in the diet.

Sensitivity analysis is a common topic in courses that cover the simplex method for linear programming. Given an optimal basis, how much can one change a single right-hand side value or a single cost coefficient while keeping the basis optimal? The usefulness of this information for practical purposes is unclear, but modelers continue to ask for it, perhaps because they are familiar with it from their studies. Accordingly, on request, some solvers will compute this information and return it in suffixes `.down` (for the lowest value), `.up` (for the highest value), and `.current` (for the current value of the right-hand side or cost value).

5. OTHER STRUCTURAL INFORMATION

The AMPL processor manipulates several other kinds of structural information, sometimes exchanging it with solvers by the declared suffix mechanism, as discussed in the following subsections.

5.1 Statuses

The modeler's view and the solver's view of an optimization problem may differ for many reasons: because some variables are currently fixed (such as by AMPL's `fix` command), because some constraints are being ignored (such as through use of AMPL's `drop` command), because of AMPL's presolve phase (Fourer and Gay 1994), because of manipulations to convey complementarity constraints (Ferris, Fourer and Gay 1999), because of linearization of piecewise-linear terms (discussed below), or because of manipulations to express "defined variables" (Fourer, Gay and Kernighan 1993, pp. 337–338). Accordingly, there are two variants of many builtin suffixes, one for each view. The builtin suffix `.astatus` reflects AMPL's view of a variable, constraint, or objective, as summarized in the default `$astatus_table`:

```
0    in      normal state (in problem)
1    drop    removed by drop command
2    pre     eliminated by presolve
3    fix     fixed by fix command
4    sub     defined variable, substituted out
5    unused  not used in current problem
```

As mentioned above, solvers that maintain a basis have their own notion of variable and constraint statuses. Conventionally, such solvers can take incoming status values from the `.sstatus` ("solver status") suffix and return updated status values to AMPL in the same suffix.

Modelers are usually most interested in `.sstatus` values for variables and constraints that are seen by the solver, but in `.astatus` values for other variables and constraints. Accordingly, the builtin suffix `.status` assumes the `.sstatus` value if the `.astatus` value is `in` and assumes the `.astatus` value otherwise.

5.2 Special ordered sets and piecewise-linear terms

Special ordered sets (Beale and Tomlin 1970, Beale and Forrest 1976) are useful for expressing problems that contain piecewise-linear functions, variables restricted to a discrete set of values, and (in conjunction with some

other auxiliary variables and constraints) semicontinuous variables (which must be either 0 or at least a given positive value). There are two flavors of special ordered sets: in a type 1 (SOS1) set, exactly one member of an ordered set of variables may be nonzero; in a type 2 (SOS2) set, at most two adjacent variables may be nonzero.

AMPL has special syntax for expressing piecewise-linear terms. For example, if `x` is a variable, then

```
<<{i in 1..n} b[i]; {i in 0..n} s[i]>> x
```

denotes the piecewise-linear function whose slope is `s[i]` for `b[i]` ≤ x ≤ `b[i+1]` (regarding `b[0]` as –∞ and `b[n+1]` as +∞), and whose value is 0 at `x` = 0. AMPL linearizes piecewise-linear terms that appear linearly in objectives or constraints. In some cases — for example, if the term is convex and appears in the objective of a minimization problem — introducing some new inequality constraints is sufficient for linearization and no integer variables or special ordered sets are required. But in hard cases, such as where a nonconvex piecewise-linear term appears in an objective to be minimized, AMPL uses a special ordered set in expressing the linearized term.

Since only some solvers recognize SOS variables, AMPL introduces auxiliary binary (zero-one) variables and constraints to enforce each SOS restriction, and also uses the suffix mechanism to convey information about these variables and constraints. Drivers for solvers that do recognize SOS members may make use of a recent addition to the AMPL/solver interface library, a function `suf_sos()`, which removes the auxiliary variables and constraints and makes the SOS information available directly.

Occasionally a modeler finds it useful to indicate SOS1 or SOS2 sets explicitly. The function `suf_sos()` also recognizes user-defined suffixes `.sosno` and `.ref` for this purpose. All variables with the same nonzero `.sosno` value are put into the same SOS, which is of type 1 if the `.sosno` value is positive and of type 2 if it is negative. The `.ref` values describe the discrete values or breakpoint values (`b[i]` above) for the variables. This representation is sufficient except in the (seemingly rare) case where the sets overlap.

5.3 Complementarity

The optimality conditions for (1) are a special case of a complementarity problem: when *f* and *c* are smooth and no components of *x* are required to be integers, the first-order necessary conditions for solving (1) state that either an inequality constraint is satisfied as an equality, or its associated Lagrange multiplier is zero. Various other problems involve explicit complementarity

conditions. Thus, as described in Ferris, Fourer and Gay (1999), we have found it helpful to add explicit syntactic forms to AMPL, including some new builtin suffixes, for purposes of expressing complementarity problems.

To make complementarity information available to solvers, the AMPL processor puts complementarity problems into a standard form in which each complementarity condition pairs an inequality constraint with a variable. An array of these pairings (with a special value to indicate ordinary constraints) is all we have had to add to the data structures in the AMPL/solver interface library. More details appear in Ferris, Fourer and Gay (1999).

6. CONCLUSION

Various problem structures are useful to solvers, including derivatives, complementarity conditions, and a variety of details that can be conveyed with declared suffixes. Declared suffixes also permit solvers to return various kinds of auxiliary information.

The suffix scheme that we describe is not the most general, and so does not cover all cases of interest. Sometimes, for example, it might be desirable to have a whole vector of auxiliary information for each variable or constraint. Nevertheless, the mechanism that we have described is able to handle many common situations.

Many optimization problems have a block structure of some kind, corresponding for example to a series of time periods or a collection of scenarios. This information is clear to the modeler, but solvers may have a hard time recovering it. Suffixes seem a natural candidate for conveying some of this auxiliary block information to solvers. The case of stochastic programming seems particularly promising for this approach, as there are a variety of specialized stochastic solvers that require detailed structural information.

The flexibility of declared suffixes might allow such a stochastic programming feature to be added without any further change to the AMPL language. More generally, we expect that suffixes will eventually be used for many purposes that we did not have in mind when the suffix feature was originally designed.

REFERENCES

Beale, E.M.L. and Forrest, J.J..H. (1976), "Global Optimization Using Special Ordered Sets," *Mathematical Programming,* vol. 10, pp. 52–69.

Beale, E.M.L. and Tomlin, J.A. (1970), "Special Facilities in a General Mathematical System for Non-Convex Problems Using Ordered Sets of Variables," in *Proceedings of the Fifth*

International Conference on Operational Research, J. Lawrence, ed., Tavistock Publications, London, pp. 447–454.

Chinneck, J.W. and Dravnieks, E.W. (1991), "Locating Minimal Infeasible Constraint Sets in Linear Programs," *ORSA Journal on Computing,* vol. 3, pp. 157–168.

Ferris, M.C., Fourer, R. and Gay, D.M. (1999), "Expressing Complementarity Problems in an Algebraic Modeling Language and Communicating Them to Solvers," forthcoming in *SIAM Journal on Optimization*.

Fourer, R. and Gay, D.M. (1994), "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, The Netherlands, pp. 135–154.

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

Fourer, R., Gay, D.M. and Kernighan, B.W. (1993), *AMPL: A Modeling Language for Mathematical Programming,* Duxbury Press-Brooks/Cole Publishing, Pacific Grove, CA.

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

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

Gay, D.M. (1997), "Hooking Your Solver to AMPL," Technical Report 97-4-06, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ. See `http://www.ampl.com/ampl/REFS/`.