EXPRESSING COMPLEMENTARITY PROBLEMS IN AN ALGEBRAIC MODELING LANGUAGE AND COMMUNICATING THEM TO SOLVERS*

MICHAEL C. FERRIS†, ROBERT FOURER‡, AND DAVID M. GAY§

Abstract. Diverse problems in optimization, engineering, and economics have natural formulations in terms of complementarity conditions, which state (in their simplest form) that either a certain nonnegative variable must be zero or a corresponding inequality must hold with equality, or both. A variety of algorithms have been devised for solving problems expressed in terms of complementarity conditions. It is thus attractive to consider extending algebraic modeling languages, which are widely used for sending ordinary equations and inequality constraints to solvers, so that they can express complementarity problems directly. We describe an extension to the AMPL modeling language that can express the most common complementarity conditions in a concise and flexible way, through the introduction of a single new "complements" operator. We present details of an efficient implementation that incorporates an augmented presolve phase to simplify complementarity problems, and that converts complementarity conditions to a canonical form convenient for solvers.

Key words. complementarity, algebraic modeling languages, optimization

AMS subject classifications. 49J40, 65K10, 90C33

1. Introduction.

After equations and inequalities, complementarity conditions are one of the most common kinds of constraints formulated in terms of decision variables. In their simplest form, they state that either a certain nonnegative variable must be zero or a corresponding inequality must hold with equality, or both.

Complementarity conditions play a key role in the theory of convex optimization, being the natural form for optimality conditions in inequality-constrained problems. They also arise in a variety of applications from engineering to economics. As a result, various algorithms have been proposed to solve *complementarity problems* whose constraints consist partly or entirely of complementarity conditions. Several of these algorithms have been developed into large-scale, robust implementations of solvers for complementarity problems.

The work described in this paper is not concerned with the details of any particular algorithm for complementarity problems, but with the broader concern of helping people communicate such problems to a variety of solvers. We consider specifically the possibilities for extending algebraic modeling languages, which are widely used in communicating equality and inequality constraints, so as to express linear and nonlinear complementarity conditions. We show how the introduction of a "complements" operator enables a modeling language to express a variety of these conditions clearly and concisely for human modelers, while remaining amenable to efficient translation to forms required by solvers.

As a practical illustration of this approach, we describe its implementation in the AMPL modeling language. We touch upon a number of practical concerns, such as the extension of the presolve phase for simplifying problems, and the design of a

* Received by the editors AAA xx, 199y; accepted for publication (in revised form) BBB xx, 199y.

†Computer Sciences Dept., University of Wisconsin, Madison, WI 53706 (ferris@cs.wisc.edu).
This author's work was supported in part by National Science Foundation grant CCR-9619765.

‡Dept. of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL 60208-3119 (4er@iems.nwu.edu). This author's work was supported in part by National Science Foundation grants DMI-9414487 and DMI-9800077.

§Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974 (dmg@research.bell-labs.com).

canonical form for communicating problems to solvers.

Relevant background in complementarity and in modeling languages is summarized in §2 below. The kinds of complementarity conditions that we aim to represent are surveyed in §3, along with a critique of previous representations for complementarity in modeling languages. Our new AMPL representation is then presented in §4 and is evaluated with respect to specific design criteria.

The remainder of this paper addresses further aspects of our complementarity enhancements to the AMPL design, including extension of the presolve phase (§5), canonical forms for communication with solvers (§6), and extensions of related constraint notations and representations (§7). All of these features have been implemented as part of a recent release of the AMPL software; a demonstration version, including a link to the PATH solver, can be tried out through a Web interface as explained in our concluding remarks in §8.

2. Background. The significance of our topic stems from the existence of applications and algorithms for complementarity problems, together with modeling languages capable of expressing such problems. We begin by briefly reviewing each of these areas.

2.1. Applications. Complementarity relations arise in a variety of engineering and economics applications [17, 18, 26], most commonly to express an equilibrium of quantities such as forces or prices.

One standard application in engineering arises in contact mechanics, where complementarity expresses the fact that friction occurs only when two bodies are in contact. Other applications are found in structural mechanics, structural design, traffic equilibrium and optimal control [18].

Interest among economists in solving complementarity problems is due in part to increased use of computational general equilibrium models [33], where complementarity is used to express Walras' Law, and to the equivalence of various games to complementarity problems [10].

Some generalizations of nonlinear programming, such as multilevel optimization — in which auxiliary objectives are to be minimized — may be reformulated as problems with complementarity conditions [1, 2, 3, 14]. There is a growing literature on these and other mathematical programming problems with equilibrium constraints, or MPECs [28, 29].

2.2. Solvers. The demands of applications have motivated a variety of algorithms for complementarity problems [4]. Modelers currently have a choice of robust and efficient implementations such as MILES [32] and PATH [12, 16].

Recent research in this area can be divided into two general algorithmic approaches [4]. One approach transforms complementarity problems so that they can be solved using existing methods for differentiable optimization or equation-solving. The other generalizes existing methods — including Newton-type methods, path search methods, projection and proximal methods, and interior-point methods — to apply to complementarity problems of certain kinds. In particular, many standard techniques have been extended to deal with the special forms of nonsmoothness that naturally appear when formulating complementarity problems. No comprehensive survey of algorithms for complementarity problems is currently available, but extensive references to algorithms can be found in [17, 18, 26].

2.3. Modeling languages. Constructing problem descriptions suitable for solvers is a substantial task that can easily consume more time and expense than finding problem solutions. Modeling languages have become a popular means of streamlining this task. They allow people to work with general models expressed in a natural and convenient form, while leaving for the language processor the work of translating models and communicating problem instances to solvers.

We are concerned in particular with *algebraic* modeling languages, which describe expressions, equations and inequalities by use of familiar algebraic terms and operators. As an example, a collection of inequality constraints defined by

$$\sum_{r \in \mathcal{R}} T_{ru} \ \geq\ q_c^0 \prod_{j \in \mathcal{M}} (P_{ju}/p_j^0)^{e_{cj}}, \quad \text{for all } c \in \mathcal{C}, u \in \mathcal{U}$$

could be transcribed to the AMPL language [22, 23] as

```
subject to ineq1 {c in C, u in U}:
   sum {r in R} T[r,u] >=
      q0[c] * prod {j in M} (P[j,u] / p0[j]) ** e[c,j];
```

or, using somewhat more mnemonic identifiers, as

```
subject to CrudeSupply {cr in CRUDES, u in USERS}:
   sum {r in REFIN} Trans[r,u] >=
      q0[cr] * prod {co in COMOD} (P[co,u] / p0[co]) ** esub[cr,co];
```

Other AMPL statements define the index sets, numerical data, and variables that appear in such an expression, as seen in the illustration of an AMPL complementarity problem in Figure 4.1 of §4. Algebraic languages, such as AMPL, AIMMS [5], GAMS [6, 8], and LINGO [34], are currently the most popular type of modeling language for describing linear and nonlinear optimization problems.

With the specification of the objective omitted, algebraic modeling languages are equally useful for describing problems of finding feasible solutions to systems of equality and inequality constraints. We thus approach complementarity conditions as an additional kind of constraint to which modeling languages may be extended. The design of any such extension involves many tradeoffs between the goal of making the language natural and convenient for people, and the requirement that the language be processed with reasonable efficiency by a computer system. We have previously described the tradeoffs involved in various extensions to AMPL [19, 21]; similar considerations have influenced our extensions for complementarity, as we next explain.

3. Design issues. To motivate our choice of a modeling language representation for complementarity conditions, we first describe the variety of conditions that we want the language to be able to represent. We then take a critical look at representations that have been used previously in the GAMS and AMPL languages.

3.1. Forms of complementarity. A few fundamental forms account for almost all of the complementarity conditions that people want to use in models. The simplest of these forms can be written in terms of a variable x_j and an associated function $g_j(x)$, where x is a vector of variables that contains x_j.

The *classical* form of complementarity condition is the one described at the beginning of this paper. It requires that

$$\begin{aligned} &\text{either } x_j = 0 \text{ and } g_j(x) \geq 0, \\ &\quad\ \ \text{or } x_j > 0 \text{ and } g_j(x) = 0. \end{aligned} \tag{3.1}$$

This condition can be viewed as consisting of the inequalities $x_j \geq 0$ and $g_j(x) \geq 0$, together with the complementarity restriction that at least one of these must hold with equality. The complementarity restriction can be written equivalently as

$$x_j \cdot g_j(x) = 0,$$

or as the nonsmooth equation

$$\min(x_j, g_j(x)) = 0.$$

If conditions of this sort are imposed for every $j \in \mathcal{J}$, then they may also be written jointly as $x \geq 0$, $g(x) \geq 0$, and $x^T g(x) = 0$.

The more general *mixed* complementarity condition on a bounded variable $\ell_j \leq x_j \leq u_j$ and a function $g_j(x)$ states that

$$\begin{aligned} &\text{either } x_j = \ell_j \text{ and } g_j(x) \geq 0,\\ &\quad\text{or } x_j = u_j \text{ and } g_j(x) \leq 0,\\ &\quad\text{or } \ell_j < x_j < u_j \text{ and } g_j(x) = 0. \end{aligned} \tag{3.2}$$

This form generalizes the classical complementarity condition, which is the special case in which $\ell_j = 0$ and $u_j = \infty$. It can be expressed equivalently as the variational inequality problem of finding $x_j \in [\ell_j, u_j]$ such that

$$(y_j - x_j) \cdot g_j(x) \geq 0 \quad \text{for all } y_j \in [\ell_j, u_j],$$

or, where there is such a condition for each $j \in \mathcal{J}$, as the joint problem of finding $x \in [\ell, u]$ such that $(y - x)^T g(x) \geq 0$ for all $y \in [\ell, u]$. A mixed complementarity condition can be split into two of the classical conditions, but only through the addition of auxiliary variables. Thus it is desirable for a modeling language to represent mixed complementarity directly, rather than requiring that all mixed conditions be transformed to classical ones. The greater simplicity of classical complementarity (as in (3.1)) argues that it should also be represented directly, however, rather than having to be written as a special case of the mixed form with an infinite bound.

For completeness, our collection of fundamental complementarity conditions also includes the trivial case

$$x_j \text{ “free” and } g_j(x) = 0, \tag{3.3}$$

which can be seen to be another special case of mixed complementarity, with $\ell_j = -\infty$ and $u_j = +\infty$.

The above forms may be extended by substituting a function $f_j(x)$ for the individual variable x_j. Thus a *generalized classical* complementarity condition can be written

$$\begin{aligned} &\text{either } f_j(x) = 0 \text{ and } g_j(x) \geq 0,\\ &\quad\text{or } f_j(x) > 0 \text{ and } g_j(x) = 0, \end{aligned} \tag{3.4}$$

and a *generalized mixed* complementarity condition has the form

$$\begin{aligned} &\text{either } f_j(x) = \ell_j \text{ and } g_j(x) \geq 0,\\ &\quad\text{or } f_j(x) = u_j \text{ and } g_j(x) \leq 0,\\ &\quad\text{or } \ell_j < f_j(x) < u_j \text{ and } g_j(x) = 0. \end{aligned} \tag{3.5}$$

Complementarity conditions in these forms can be transformed to the simpler forms (3.1) and (3.2), but only by adding a variable and a defining equation. Thus it is desirable that a modeling language be able to directly represent these forms as well.

The above forms allow a modeler to express not only models that are well formed, solvable and stable, but also models that are poorly specified or badly behaved. For $g_j(x)$ as simple a function as $1 - x_j$, the complementarity condition (3.1) is equivalent to specifying that x_j can take only the values zero and one. This gives some indication of the difficulties associated with solving complementarity problems; the "tightness" requirement is combinatorial in nature and the solution set of a complementarity problem need not be convex or even connected.

It is possible to avoid undesirably hard cases by placing some restrictions on the functions involved. Just as there are classes of well behaved nonlinear optimization problems that involve convex functions, for complementarity problems there is a corresponding notion of a monotone function g_j, which satisfies

$$(y-x)^T(g_j(y)-g_j(x)) \geq 0$$

for all x and y [26, 31]. Current modeling languages largely avoid such restrictions, however, in the interest of keeping their design simple and general. Especially in working with nonlinear problems, a modeler is expected to be aware that solvers frequently have difficulties if the model is poorly specified or if the initial point is far from a solution. Some assistance may be provided by routines that test functions for desirable properties, but they are typically incorporated into individual solvers or related analysis tools such as MProbe [9].

3.2. Modeling language representations. The GAMS modeling language [6, 8] was the first (to our knowledge) to provide for specification of complementarity problems. As explained in [33], GAMS does not express complementarity through any modification to its constraint syntax, but rather by an extension to its model-defining statement. The list of constraints in its `model` statement is generalized to allow the specification of complementary constraint-variable pairs, as in the following example from `pies.gms` in MCPLIB [13]:

```
model pies / delc.c, delo.o, delct.ct, delot.ot, dellt.lt, delht.ht,
   dembal.p, cmbal.cv, ombal.ov, lmbal.lv, hmbal.hv, ruse.mu /;
```

The specification `delc.c`, for example, indicates that the constraints `delc`,

```
delc(creg,ctyp) ..
   ccost(creg,ctyp) + sum(R, cruse(R,creg,ctyp) * mu(R) =g= cv(creg);
```

are complementary to the variables `c` having lower bounds 0,

```
positive variables
   c(creg,ctyp), ...
```

and having upper bounds assigned from a data table,

```
c.up(creg,ctyp) = cmax(creg,ctyp);
```

From the fact that `c` has finite lower and upper bounds, GAMS infers that a certain mixed complementarity condition is intended; from the expression in the `delc` constraint statement, GAMS determines what we have been calling the function $g_j(x)$. Thus explicit conditions of the form (3.2) need not be added to the model. Analogous inferences allow variables that have only one finite bound to induce classical complementary conditions of the form (3.1).

These simple conventions provide sufficient expressiveness to describe a considerable variety of applications, as evidenced by the over 50 GAMS complementarity models collected in MCPLIB. The design of these extensions also promotes the reuse of equations previously declared, thus helping modelers to transform existing models into the complementarity framework.

Nevertheless, several aspects of the GAMS approach remain problematical. A full description of any one complementarity condition tends to be spread over several sections of the GAMS model, as seen in the example above. Generalized complementarity conditions can only be represented via transformations to simpler forms. Finally, and most seriously, the sense of the inequality in a complementary constraint is determined from the bounds on the corresponding variable, not by the inequality actually written in the statement of the constraint. As a result, both mixed and classical complementarity conditions may be interpreted by the GAMS processor in ways that are counterintuitive to modelers.

For the mixed case, the function $g_j(x)$ in (3.2) must be specified by means of a GAMS constraint declaration, even though it is not subject to any inequality. For example, although the `delc` statement above appears to define `=g=` ($\geq$) constraints, the implied complementarity condition allows the left-hand side of `delc(creg,ctyp)` to be $<$ the right-hand side when the corresponding variable `c(creg,ctyp)` is at upper bound. (The GAMS result listing marks a constraint as "redefined" if it is violated by the solution in this way.)

For the classical case, it is up to the modeler to correctly state the inequality on $g_j(x)$ in (3.1). As another example (also from `pies.gms`), the nonnegative variables `ct(creg,users)` are defined as being complementary to the constraints

```
delct(creg,users) ..
   ctcost(creg,users) + cv(creg) =g= p("C",users);
```

Because the `ct` variables have finite lower bounds but not finite upper bounds, the relational operator in this case *must* be `=g=`. The mathematically equivalent constraint

```
delct(creg,users) ..
   p("C",users) =l= ctcost(creg,users) + cv(creg);
```

is rejected as an error, because the relational operator `=l=` ($\leq$) is not compatible with complementary variables having only a finite lower bound. This distinction is hard to impress upon modelers, who see the above statements as two ways of saying the same thing.

A similar complementarity representation has been implemented in [11, chapter 2] for the AMPL modeling language [22, 23], though with some differences in the nature of the extension. Complementarity is indicated by writing a constraint in the equivalent multiplicative form $x_j \cdot g_j(x) = 0$, with bounds on the variable x_j specified in the declaration for the variable. Thus no new syntax is added to any part of the AMPL language (a key requirement of the design in [11]) and the existing AMPL translator can process the model and create a problem file in its usual format. Detection of complementarity conditions is left to the solver, or more accurately to the AMPL driver (or interface routines) for the solver. The driver examines the expression tree for each constraint to determine the variable x_j, and then generates an appropriate complementarity constraint for the solver, depending on which bounds of x_j are finite, using much the same logic as the GAMS implementation.

This design also has much the same drawbacks as the GAMS one. Most seriously, constraints that appear in the model are not necessarily enforced by the complementarity solver. The conditions actually enforced must be inferred from information that is partly in one place (a constraint) and partly in another (a variable's bounds). Generalized complementarity conditions must be handled by transformation to a simpler form.

4. A new representation. In creating a new form for complementarity conditions, we have sought to address the drawbacks of previous designs while preserving the existing strengths of the AMPL language. We begin this section by describing the representation that we ended up choosing. We then consider the extent to which our representation satisfies a range of design criteria.

Of particular importance for our discussion is the variety of arithmetic constraint expressions that AMPL recognizes. They can be summarized as

$expr_1$ `<=` $expr_2$
$expr_1$ `>=` $expr_2$
$expr_1$ `=` $expr_2$
$const_1$ `<=` $expr$ `<=` $const_2$
$const_1$ `>=` $expr$ `>=` $const_2$

where *expr* is any valid arithmetic expression, possibly involving variables (linearly or nonlinearly), and *const* is an arithmetic expression that does not contain variables.

Illustrations in this section are taken from `pies.mod`, the previous GAMS example's AMPL counterpart, which is shown in Figures 4.1 and 4.2. Additional AMPL complementarity models and corresponding data files can be found in MCPLIB [13] and at `http://www.ampl.com/ampl/NEW/COMPLEMENT/`.

4.1. Design specifics. The key to our design is the realization that the different complementarity forms (3.1), (3.2) and (3.3) have the same general structure. In each case, a variable is complementary, in some sense, to a function of variables; and in each case, exactly two inequalities are involved (counting one equality as two inequalities). The function can be defined by a modeling language expression, and the inequalities are corresponding modeling language constraints. The same observations apply to the generalized forms (3.4) and (3.5), except that the variable is replaced by a second function.

These observations suggest that all of the fundamental complementarity conditions identified in §3.1 can be represented by AMPL expressions of the form

$item_1$ `complements` $item_2$

The keyword `complements` is a new operator. Its operands $item_1$ and $item_2$ may be AMPL arithmetic expressions, or may be AMPL arithmetic constraints of any of the types listed above, provided that they contain together exactly two inequalities. A solution satisfies such an expression if it satisfies the constraints among the operands to `complements` and also the appropriate kind of complementarity between the operands.

If a constraint of this new kind has two single inequality operands, as in

```
subject to delct {c in CREG, u in USERS}:
   0 <= Ct[c,u]  complements  ctcost[c,u] + Cv[c] >= P["C",u];
```

then it specifies a classical complementarity condition. Both inequalities must hold, at least one with equality.

If a constraint of this kind has instead one double inequality operand and one expression operand, as in

```
subject to delc {c in CREG, t in CTYP}:
   0 <= C[c,t] <= cmax[c,t]  complements
   ccost[c,t] + sum {res in R} cruse[res,c,t] * Mu[res] - Cv[c];
```

then it specifies a mixed complementarity condition. Again both inequalities must

FIG. 4.1. *An AMPL model of a complementarity problem, part 1: declarations of sets, numerical data, and decision variables.*

```
set COMOD := {"C","L","H"};   # coal and light and heavy oil

set R;        # resources
set CREG;     # coal producing regions
set OREG;     # crude oil producing regions
set CTYP;     # increments of coal production
set OTYP;     # increments of oil production
set REFIN;    # refineries
set USERS;    # consumption regions

param rmax {R};               # maximum resource usage
param cmax {CREG,CTYP};       # coal prod. limits
param omax {OREG,OTYP};       # oil prod. limits
param rcost {REFIN};          # refining cost
param q0 {COMOD};             # base demand for commodities
param p0 {COMOD};             # base prices for commodities
param demand {COMOD,USERS};   # computed at optimality
param output {REFIN,COMOD};   # % output of light/heavy oil
param esub {COMOD,COMOD};     # cross-elasticities of substitution
param cruse {R,CREG,CTYP};    # resource use in coal prod
param oruse {R,OREG,OTYP};    # resource use in oil prod
param ccost {CREG,CTYP};      # coal production cost
param ocost {OREG,OTYP};      # oil production cost
param ctcost {CREG,USERS};    # coal transportation costs
param otcost {OREG,REFIN};    # crude oil transportation costs
param ltcost {REFIN,USERS};   # light oil transportation costs
param htcost {REFIN,USERS};   # heavy oil transportation costs

var C {CREG, CTYP};      # coal production
var O {OREG, OTYP};      # oil production
var Ct {CREG, USERS};    # coal transportation levels
var Ot {OREG, REFIN};    # crude oil transportation levels
var Lt {REFIN, USERS};   # light transportation levels
var Ht {REFIN, USERS};   # heavy transportation levels
var P {COMOD, USERS};    # commodity prices
var Mu {R};              # dual to ruse: marginal utility
var Cv {CREG};           # dual to cmbal
var Ov {OREG};           # dual to ombal
var Lv {REFIN};          # dual to lmbal
var Hv {REFIN};          # dual to hmbal
```

hold, but the nature of the complementarity is somewhat different. Either the lower inequality holds with equality and the expression is nonnegative, or the upper inequality holds with equality and the expression is nonpositive, or neither inequality holds with equality and the expression is zero.

A single equality constraint may take the place of the double inequality:

```
subject to cmbal {c in CREG}:
   Cv[c]  complements
   sum {t in CTYP} C[c,t] = sum u in USERS Ct[c,u];
```

This form of constraint merely imposes the equality. It has no effect on the expression

FIG. 4.2. *An AMPL model of a complementarity problem, part 2: declarations of complementarity conditions.*

```
subj to delc {c in CREG, t in CTYP}:
  0 <= C[c,t] <= cmax[c,t] complements
  ccost[c,t] + (sum {res in R} cruse[res,c,t] * Mu[res]) - Cv[c];

subj to delo {o in OREG, t in OTYP}:
  0 <= O[o,t] <= omax[o,t] complements
  ocost[o,t] + (sum {res in R} oruse[res,o,t] * Mu[res]) - Ov[o];

subj to delct {c in CREG, u in USERS}:
  0 <= Ct[c,u] complements
  ctcost[c,u] + Cv[c] >= P["C",u];

subj to delot {o in OREG, r in REFIN}:
  0 <= Ot[o,r] complements
  otcost[o,r] + rcost[r] + Ov[o] >=
     output[r,"L"] * Lv[r] + output[r,"H"] * Hv[r];

subj to dellt {r in REFIN, u in USERS}:
  0 <= Lt[r,u] complements
  ltcost[r,u] + Lv[r] >= P["L",u];

subj to delht {r in REFIN, u in USERS}:
  0 <= Ht[r,u] complements
  htcost[r,u] + Hv[r] >= P["H",u];

subj to dembal {co in COMOD, u in USERS}:  # excess supply of product
  .1 <= P[co,u] complements
  (if co = "C" then sum {c in CREG} Ct[c,u]) +
  (if co = "L" then sum {r in REFIN} Lt[r,u]) +
  (if co = "H" then sum {r in REFIN} Ht[r,u]) >=
     q0[co] * prod {cc in COMOD} (P[cc,u]/p0[cc])**esub[co,cc];

subj to cmbal {c in CREG}:                 # coal material balance
  Cv[c] complements
  sum {t in CTYP} C[c,t] = sum {u in USERS} Ct[c,u];

subj to ombal {o in OREG}:                 # oil material balance
  Ov[o] complements
  sum {t in OTYP} O[o,t] = sum {r in REFIN} Ot[o,r];

subj to lmbal {r in REFIN}:                # light material balance
  Lv[r] complements
  sum {o in OREG} Ot[o,r] * output[r,"L"] = sum {u in USERS} Lt[r,u];

subj to hmbal {r in REFIN}:                # heavy material balance
  Hv[r] complements
  sum {o in OREG} Ot[o,r] * output[r,"H"] = sum {u in USERS} Ht[r,u];

subj to ruse {res in R}:                   # resource use constraints
  0 <= Mu[res] complements
  rmax[res] >=
     sum {c in CREG, t in CTYP} C[c,t] * cruse[res,c,t] +
     sum {o in OREG, t in OTYP} O[o,t] * oruse[res,o,t];
```

operand, which could just as well be omitted (along with the `complements` operator). It does have some value in exhibiting "square" models (like `pies.mod`), where each constraint is paired with a different complementary variable; squareness is required by some solvers, as discussed further in §6.2.

Although the first operand to `complements` in the above examples involves only a single variable, our definitions make no mention of this fact, and indeed it is not a requirement of our representation. So long as the total number of inequality operators is two, our representation allows each operand to be any arithmetic expression or constraint.

4.2. Design criteria. Our representation's introduction of the `complements` operator is valuable in several respects. Its presence clearly distinguishes complementarity constraints from other types, and its operands contain all of the information necessary to define a complementarity condition. The definition of an AMPL complementarity constraint (or indexed collection of constraints) thus appears all in one place, rather than being divided among different statements as in earlier designs.

There is also no question as to which kind of complementarity is intended by our representation, since the classical and mixed forms are readily distinguished by the position of the inequalities relative to the `complements` operator. Nor is there any question as to which two inequalities are implied, since both appear explicitly as operands to `complements`. Earlier designs' practice of inferring such information from the number of finite bounds is avoided entirely. At the same time, the interpretation of existing AMPL constraint forms — ones that do not contain the `complements` operator — is left unchanged, and existing models are unaffected.

The incorporation of existing AMPL expression and constraint forms into the new representation is also valuable. By allowing operands to `complements` to be any AMPL arithmetic expressions and constraints, subject only to the two-inequality restriction, we keep our language rules simple to apply and easy to remember. The constraint `delct` above, for example, may be written in any obviously equivalent fashion, such as

```
subject to delct {c in CREG, u in USERS}:
   Ct[c,u] >= 0 complements ctcost[c,u] + Cv[c] >= P["C",u];
```

or

```
subject to delct {c in CREG, u in USERS}:
   0 >= P["C",u] - ctcost[c,u] - Cv[c] complements Ct[c,u] >= 0;
```

We also make no special distinction for inequalities that happen to be bounds on individual variables. As a result, the generalized complementarity forms (3.4) and (3.5) are specified as easily as their more restricted counterparts (3.1) and (3.2), without any of the transformations required by earlier designs.

Our representation does require the modeler or reader to remember the rules for deriving a complementarity condition from `complements` and its operands. In this respect, `complements` is a primitive operator, like `+` or `<=`, whose meaning must be furnished from a user's knowledge of the modeling language. The alternative would be to write out the complementarity requirement more explicitly, perhaps in forms (3.1) or (3.2) or their generalizations. As an example, the constraint `delc` introduced above might be declared equivalently in the following form motivated by (3.2):

```
var Cdual {c in CREG, t in CTYP}
   = ccost[c,t] + sum {res in R} cruse[res,c,t] * Mu[res];
```

```
subject to delc {c in CREG, t in CTYP}:
   C[c,t] = 0 and Cdual[c,t] >= Cv[c]  or
   C[c,t] = cmax[c,t] and Cdual[c,t] <= Cv[c]  or
   0 <= C[c,t] <= cmax[c,t] and Cdual[c,t] = 0;
```

This representation clearly states the entire complementarity condition using only existing AMPL operators. However, its adoption would require that AMPL be extended to allow variables in the operands to boolean operators (such as `and` and `or`). Such an extension would introduce a great variety of constraint types unrelated to complementarity, making complementarity constraints much harder for the modeler (and AMPL processor) to recognize. We could further modify the design to preserve recognizability, but only by introducing complex new rules on the use of variables with boolean operators. In light of this and similar examples, we have decided that the drawbacks of having a primitive `complements` operator are greatly outweighed by the advantages.

5. Extending presolve. Often it is worthwhile to simplify an optimization problem before sending it to a solver. Brearley, Mitra and Williams [7] describe a set of simplification techniques based on iteratively tightening the bounds on variables and constraint expressions. These "presolve" techniques have been found to work well for linear programs, and are provided as an option by many commercial linear programming solvers.

The AMPL modeling language processor also incorporates a primal presolve phase [20] that applies the ideas of [7] to linear constraints. (Nonlinearities are handled, but in a naive way. Because AMPL may send several objectives to the solver, we have not yet exploited the opportunities described in [7] to use dual information.) An integrated presolver is useful to a modeling language system in several respects. By identifying constraints involving only one variable, the presolver makes it irrelevant whether one states bounds on a variable in the variable's declaration or in a separate constraint declaration. Presolving sometimes results in a significantly smaller problem to convey to the solver, reducing communication time. Presolving on the modeling language side can also benefit any solver that does not have its own presolve phase.

Complementarity constraints introduce new information that we can exploit in AMPL's presolve routines. For instance, given a constraint of the form

$expr_1$ `>= 0  complements` $expr_2$ `>= 0`,

if we can deduce a positive lower bound on $expr_1$, then we can infer that it is strictly positive for all feasible points, and we can replace the constraint by

$expr_2$ `= 0`.

Similarly, for a constraint of the form

$const_1$ `<=` $expr_2$ `<=` $const_3$ `complements` $expr_4$,

there are various possibilities. If we can deduce that, say, $expr_2 < const_3$ for all feasible points, then we can replace the constraint by

$const_1$ `<=` $expr_2$ `complements` $expr_4$ `>= 0`.

If we can deduce that always $const_1 < expr_2 < const_3$, then we can replace the constraint by

$expr_4$ `= 0`.

Conversely, if we can deduce that $expr_4 < 0$, then we can replace the constraint by

$expr_2$ = $const_3$,

and so forth.

Each of these deductions can be triggered by presolve's manipulations of variable and constraint bounds. There are many combinations to be considered, but they are straightforward to enumerate and fast to check. As a simple illustration, consider the model

```
var x1;
var x2;
var x3;

subj to f1:  x1 >= 0  complements  x1 + 2*x2 + 3*x3 >= 1;
subj to f2:  x2 >= 0  complements  x2 - x3 >= -1;
subj to f3:  x3 >= 0  complements  x1 + x2 >= -1;
```

proposed by Todd S. Munson [private communication] and called `munson1.mod` in MCPLIB [13]. The first inequalities in the complementarity constraints imply that all the variables are nonnegative. Then the second constraint in `f3` must always be slack, which implies that `x3` $= 0$, whence the second constraint in `f2` must always be slack, which implies that `x2` $= 0$. The second constraint in `f1` now reduces to `x1` ≥ 1, which implies that the first inequality in `f1` must always be slack, which implies `x1` $= 1$, and the presolver completely determines the solution. Our results in §6 identify two larger test problems for which presolve's simplifications are significant.

Presolve folds together all bounds on a variable, whether specified in an ordinary `var` or `subj to` declaration or as an operand to `complements`. The user has the option of turning off most of presolve's logic, in which case separate bounds on some variables may remain separate. However, regardless of the presolve setting, AMPL detects when bounds given in a `var` declaration are redundant due to the same or tighter bounds being given as an argument to `complements` in a subsequent constraint. For example, to enhance the definition of variable `Ct` in Figure 4.1, we could add bounds,

```
var Ct {CREG, USERS} >= 0;
```

even though Figure 4.2 defines the same bounds in a subsequent complementarity constraint:

```
subj to delct {c in CREG, u in USERS}:
   0 <= Ct[c,u] complements ctcost[c,u] + Cv[c] >= P["C",u];
```

The redundant bounds do not make any difference to the form of the problem seen by the solver.

6. Communicating problems to solvers. Since a modeling language is designed for the convenience of human modelers, the language processing software must do a certain amount of transformation to put problems into the forms required by efficient solvers. We describe in this section the transformations performed by AMPL's language processor to yield a canonical complementarity form useful for a variety of solvers. We then briefly comment on specific drivers for the PATH solver and for solvers written in MATLAB.

6.1. Transformation to canonical form. To simplify the task of presenting complementarity problems to solvers, we have arranged for the AMPL processor to transform general complementarity constraints to the form

$$(6.1) \quad \ell_1 \le \mathit{expr} \le u_1 \quad \texttt{complements} \quad \ell_2 \le \mathit{variable} \le u_2,$$

where the complemented *variable*s are all distinct, and exactly two of the constants ℓ_1, u_1, ℓ_2, u_2 are finite. Ignoring the infinite bounds, this representation clearly describes a classical or mixed complementarity condition by the rules previously given.

This canonical form has the advantage of allowing the left operand and right operand of `complements` to be communicated to the solver as an ordinary constraint and an ordinary variable, respectively, as described in [24]. The complementarity extension can then be implemented by sending the solver only one new array, `cvar`, which pairs constraints with variables. Specifically, if the `i`th constraint seen by the solver has arisen by a complementarity relationship of the form (6.1) with the `j`th variable, then `cvar[i]` is set to `j`. Otherwise, the `i`th constraint has not arisen from any complementarity relation, and `cvar[i]` is set to an index that does not correspond to any variable.

The form of the transformation to (6.1) is straightforward, though it sometimes involves adding a new variable and an equality constraint defining the new variable. An expression complementing a general double-inequality constraint, for example, is transformed by

$$\begin{aligned} & \mathit{expr}_1 \ \texttt{complements} \ \ell \le \mathit{expr}_2 \le u \\ \Longrightarrow\ & -\infty \le \mathit{expr}_1 \le +\infty \ \texttt{complements} \ \ell \le z \le u, \quad z = \mathit{expr}_2, \end{aligned}$$

where z is the new variable. In the common case of a bounded variable v complementing a single inequality, it is unnecessary to introduce a new variable and equality constraint, so long as v has not already been used as the canonical *variable* in another complementarity constraint. For example,

$$\begin{aligned} & v \ge 0 \ \texttt{complements} \ \mathit{expr} \ge 0 \\ \Longrightarrow\ & 0 \le \mathit{expr} \le +\infty \ \texttt{complements} \ 0 \le v \le +\infty. \end{aligned}$$

But if v is used in two such constraints, then it can serve as the canonical variable for the first, but a new variable w must be introduced as the canonical variable of the second:

$$\begin{aligned} & v \ge 0 \ \texttt{complements} \ \mathit{expr}_1 \ge 0, \ \ v \ge 0 \ \texttt{complements} \ \mathit{expr}_2 \ge 0 \\ \Longrightarrow\ & 0 \le \mathit{expr}_1 \le +\infty \ \texttt{complements} \ 0 \le v \le +\infty, \\ & 0 \le \mathit{expr}_2 \le +\infty \ \texttt{complements} \ 0 \le w \le +\infty, \quad w = v. \end{aligned}$$

Other cases are similarly straightforward. All of AMPL's transformations to canonical form preserve the property of monotonicity described in §3.1, ensuring that the complementarity problem sent to a solver will tend to be as well behaved as the problem originally formulated by the modeler.

6.2. Interface to PATH. Some current solvers, such as PATH [12, 16], want to see only complementarity conditions. If a problem is "square" in the sense that the number of variables equals the number of equality constraints plus the number of canonical complementarity conditions (6.1), then it is straightforward to create an equivalent pure complementarity problem in which all constraints have the form (6.1).

First, each finite bound on an unassociated variable (one not yet associated with a canonical complementarity constraint) is removed from the variable and added as a separate inequality constraint instead. Then each equality constraint can be converted to a complementarity condition by the transformation

$$\begin{array}{ll} & expr = const \\ \Longrightarrow & const \le expr \le const \quad \texttt{complements} \quad -\infty \le v \le +\infty, \end{array}$$

where v is any unassociated variable. The squareness of the problem insures that every equality can be covered by a different variable in this way. Finally, a new variable is associated with each of the inequality constraints (including the aforementioned constraints created from variable bounds), after which the inequalities can also be converted to complementarity conditions. For example,

$$\begin{array}{ll} & expr \ge const \\ \Longrightarrow & const \le expr \le +\infty \quad \texttt{complements} \quad 0 \le z \le +\infty, \end{array}$$

where z is the new variable. The other inequality forms are handled similarly.

We have implemented an interface (or driver) that, when compiled with PATH, produces an AMPL solver `path` for square complementarity problems. It can be used in the AMPL command environment in the same way as other solvers:

```
ampl: model pies.mod;
ampl: data pies.dat
ampl: option solver path;
ampl: solve;
PATH 3.0: Solution found.
14 iterations (1 for crash); 28 pivots.
30 function, 16 gradient evaluations.
```

The driver reads a problem in the canonical form (6.1) and applies the manipulations described above, to produce a problem consisting entirely of complementarity conditions as the PATH solver requires. Instructions and C source for this driver are freely available from `ftp://netlib.bell-labs.com/netlib/ampl/solvers/path`.

Table 6.1 shows the results of running `path` on some AMPL problems from MCPLIB [13]. Certain problems are supplied with several starting guesses, as distinguished in the *start* column. Results are given both with ("yes") and without ("no") deduction of bounds by AMPL's presolver, in the two cases (*choi* and *pies*) where presolving makes a difference. The columns headed *nv*, *ncc*, and *nsc* give the numbers of variables, complementarity constraints (6.1), and side constraints seen by the solver (before it makes the previously described manipulations). The *nfunc* and *ngrad* columns report the numbers of function and gradient (Jacobian) evaluations.

6.3. Interface to MATLAB. Often it is convenient to use MATLAB [25, 30] implementations to experiment with algorithms. The examples associated with [24] include source for MATLAB *mex* functions that provide various information about optimization problems expressed in AMPL, such as dimensions, bounds, starting guesses, and function, gradient (or Jacobian matrix), and Lagrangian Hessian values. To encourage experiments with complementarity algorithms, we have extended these *mex* functions to also make available the `cvar` array of complementarity relations (as defined in §6.1).

7. Related notations. The complementarity extensions to AMPL constraints necessitate corresponding extensions to notations for referring to constraints. This section briefly describes extensions to the "dot suffix" notation for constraint-related quantities, and to "synonyms" for constraint names.

TABLE 6.1
Tests of the `path` *solver for AMPL.*

problem	*start*	*presolve*	*nv*	*ncc*	*nsc*	*iters*	*pivots*	*nfunc*	*ngrad*
bertsek	1	—	15	10	5	5	13	25	6
bertsek	2	—	15	10	5	4	5	10	6
bertsek	3	—	15	10	5	6	12	14	8
bertsek	4	—	15	10	5	5	13	25	6
bertsek	5	—	15	10	5	4	3	10	6
bertsek	6	—	15	10	5	5	13	25	6
choi	1	no	14	13	2	4	7	10	6
choi	1	yes	13	13	0	4	3	10	6
ehl_def	1	—	101	100	1	5	6	12	7
ehl_kost	1	—	101	100	1	5	6	12	7
josephy	1	—	4	4	0	8	18	29	10
josephy	2	—	4	4	0	10	16	27	12
josephy	3	—	4	4	0	16	22	34	18
josephy	4	—	4	4	0	5	4	13	7
josephy	5	—	4	4	0	3	2	8	5
josephy	6	—	4	4	0	10	31	26	12
josephy	7	—	4	4	0	10	16	25	12
josephy	8	—	4	4	0	2	1	6	4
kojshin	1	—	4	4	0	10	21	26	12
kojshin	2	—	4	4	0	13	34	68	16
kojshin	3	—	4	4	0	16	36	34	18
kojshin	4	—	4	4	0	1	0	4	3
kojshin	5	—	4	4	0	5	6	12	7
kojshin	6	—	4	4	0	15	29	39	17
kojshin	7	—	4	4	0	10	27	25	12
kojshin	8	—	4	4	0	4	5	10	6
nash	1	—	10	10	0	6	5	14	8
nash	2	—	10	10	0	6	5	14	8
nash	3	—	10	10	0	5	4	12	7
nash	4	—	10	10	0	3	2	8	5
obstacle	1	—	2500	2500	0	7	1	14	9
pies	1	no	42	34	16	14	150	30	16
pies	1	yes	42	34	8	14	28	30	16

7.1. Suffixes. As an aid to evaluating and understanding computed solutions, it is convenient to have notations for quantities such as lower and upper bounds, slack values (distances from bounds), and reduced costs associated with variables and constraints. The AMPL language admits various .*suffix* notations to denote these quantities. In particular, AMPL puts each constraint into the canonical form $\ell \le body \le u$, in which ℓ and u are constants (possibly $-\infty$ and $+\infty$), with $\ell = u$ for equality constraints, after which the most frequently used .*suffix* options can be defined as shown in Table 7.1.

For dealing with complementarity constraints, we extend the .*suffix* notations in several ways. A complementarity constraint `Goo` may be viewed as consisting of a "left" and "right" constraint, `Goo.L` and `Goo.R`, with a complementarity condition between them. To indicate quantities associated with `Goo`'s left and right constraints, we introduce the notations `Goo.L`*suf* and `Goo.R`*suf*, where *suf* is any suffix permitted

TABLE 7.1
Most frequently used suffixes for a constraint `Foo`, *in terms of the canonical form* $\ell \le body \le u$.

notation	*meaning*
`Foo.body`	*body*
`Foo.lb`	ℓ
`Foo.ub`	u
`Foo.lslack`	$body - \ell$
`Foo.uslack`	$u - body$
`Foo.slack`	$\min(\texttt{Foo.lslack}, \texttt{Foo.uslack})$

for an ordinary constraint. For showing how close a complementarity condition is to holding, we also introduce the notation `Goo.slack`, whose meaning depends on `Goo`'s nature. If `Goo.L` and `Goo.R` involve one explicit inequality each, then

$$\texttt{Goo.slack} = \min(\texttt{Goo.Lslack}, \texttt{Goo.Rslack}).$$

Otherwise `Goo` has one of the forms

`Goo.Lbody complements` ℓ `<= Goo.Rbody <=` u,
ℓ `<= Goo.Lbody <=` u `complements Goo.Rbody.`

In the former case,

$$\texttt{Goo.slack} = \begin{cases} \min(\texttt{Goo.Lbody}, \texttt{Goo.Rbody} - \ell) & \text{if } \texttt{Goo.Rbody} \le \ell, \\ \min(-\texttt{Goo.Lbody}, u - \texttt{Goo.Rbody}) & \text{if } \texttt{Goo.Rbody} \ge u, \\ -|\texttt{Goo.Lbody}| & \text{otherwise;} \end{cases}$$

the latter case is defined analogously. Clearly `Goo.slack` is zero when the complementarity condition is satisfied. If `Goo.L` and `Goo.R` involve one inequality each, `Goo.slack` can be positive (if both constraints are strictly satisfied) or negative (if at least one is violated), so its sign conveys some information. In the other cases `Goo.slack` is always nonpositive.

7.2. Synonyms. Models are usually most conveniently described in terms of several kinds of differently named (and indexed) constraints, as seen in Figure 4.2. But sometimes it is helpful to address the variables and constraints with a uniform notation. For this purpose, AMPL offers generic *synonyms* for constraints (as well as variables and objectives). The synonym `_con[`i`]` denotes the ith constraint as the modeler sees constraints (before presolve), for $i = 1, \ldots,$ `_ncons`, and `_scon[`i`]` denotes the ith constraint that the solver sees (after presolve), for $i = 1, \ldots,$ `_sncons`. The notations `_conname[`i`]` and `_sconname[`i`]` denote the corresponding names of these constraints.

After considering several possibilities, we have found it most convenient to introduce separate synonyms for complementarity constraints, reserving `_con` and other existing synonyms for "ordinary" constraints (including each of the pair of constraints involved in a complementarity constraint declaration). The new synonyms are `_ccon[`i`]` for the ith complementarity constraint before presolve, and `_cconname[`i`]` for its name, both for $i = 1, \ldots,$ `_nccons`. We also define `_scvar[`i`]` as the index of the complementing variable associated with constraint i in the canonical form (6.1) sent to the solver. As an example of the use of these synonyms, one can see the extent to which the current solution satisfies the constraints of a complementarity problem by issuing the AMPL command

```
display max {i in 1.._nccons} abs(_ccon[i].slack),
        min {i in 1.._ncons} _con[i].slack;
```

to show the maximum complementarity violation and, over all constraints, the maximum constraint violation (negative values of `.slack` indicating violations).

8. Concluding remarks. Modeling languages make it easy for people to go from a familiar mathematical formulation to the solution of a specific problem instance, without worrying about computer programming details such as the data structures that solvers require. Thus modelers can focus on choosing the right model instead of worrying over lower-level aspects of implementation. Modeling languages have hitherto been used mainly for expressing conventional linear and nonlinear programs. The present work describes an extension to a wider class, including complementarity problems and mathematical programming problems with equilibrium constraints.

We hope that experience with and reaction to the present work will guide us in designing other useful extensions. One obvious possibility concerns expressing bi-level and multi-level optimization problems. These can be transformed to complementarity problems of the kind we have addressed, but only by means of a cumbersome conversion that requires one to write hand-coded derivative expressions in the complementarity constraints. It might be possible instead to introduce a simple extension that allows a constraint to reference the values of lower-level objectives.

An implementation of AMPL that includes the new complementarity extensions can be accessed through Web interfaces at either of the following:

```
http://www.ampl.com/ampl/TRYAMPL/
http://www.mcs.anl.gov/home/otc/Server/
```

Although they differ in details, both of these interfaces accept AMPL models, data, and commands for execution on a remote computer, with PATH as one option for the choice of solver. Both then generate a web page showing the results. Thus it is not necessary to have AMPL running locally to experiment with the new complementarity features.

Acknowledgements. We thank Brian Kernighan, Margaret Wright, and the anonymous refereees for helpful comments on the manuscript.

REFERENCES

[1] J.F. Bard, An algorithm for solving the general bilevel programming problem. *Mathematics of Operations Research* **8** (1983) 260–272.

[2] J.F. Bard, Optimality conditions for the bilevel programming problem. *Naval Research Logistics Quarterly* **31** (1984) 13–26.

[3] J.F. Bard, Convex two-level optimization. *Mathematical Programming* **40** (1988) 15–27.

[4] S.C. Billups, S.P. Dirkse and M.C. Ferris, A comparison of large scale mixed complementarity problem solvers. *Computational Optimization and Applications* **7** (1997) 3–25.

[5] J.J. Bisschop and R. Entriken, *AIMMS: The Modeling System.* Paragon Decision Technology (1993).

[6] J. Bisschop and A. Meeraus, On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study* **20** (1982) 1–29.

[7] 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.

[8] A. Brooke, D. Kendrick and A. Meeraus, *GAMS: A User's Guide, Release 2.25.* Scientific Press/Duxbury Press (1992).

[9] J.W. Chinneck, "MProbe: Software for exploring nonlinear models." Presented at CORS '97, Ottawa, May 26–28, and at Optimization Days 1997, Montreal, May 12–14. See also `http://www.sce.carleton.ca/faculty/chinneck/mprobe.html`.

[10] R.W. Cottle, J.-S. Pang and R.E. Stone, *The Linear Complementarity Problem.* Academic Press (1992).

[11] S.P. Dirkse, Robust solution of mixed complementarity problems. Mathematical Programming Technical Report 94-12, Computer Sciences Department, University of Wisconsin, Madison (1994); `ftp://ftp.cs.wisc.edu/math-prog/tech-reports/94-12.ps.Z`.

[12] 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.

[13] S.P. Dirkse and M.C. Ferris, MCPLIB: a collection of nonlinear mixed complementarity problems. *Optimization Methods and Software* **5** (1995) 319–345. See also `ftp://ftp.cs.wisc.edu/math-prog/mcplib/`.

[14] S.P. Dirkse and M.C. Ferris, Modeling and solution environments for MPEC: GAMS & MATLAB. In *Nonsmooth, Piecewise Smooth, Semismooth and Smoothing Methods,* M. Fukishima and L. Qi, eds., Kluwer (1998). See also `ftp://ftp.cs.wisc.edu/math-prog/tech-reports/97-09.ps.Z`.

[15] S.P. Dirkse, M.C. Ferris, P.V. Preckel and T.F. Rutherford, The GAMS callable program library for variational and complementarity solvers. Mathematical Programming Technical Report 94-07, Computer Sciences Department, University of Wisconsin, Madison (1994); `ftp://ftp.cs.wisc.edu/math-prog/tech-reports/94-07.ps.Z`.

[16] M.C. Ferris, and T.S. Munson, Interfaces to PATH 3.0: design, implementation and usage. Mathematical Programming Technical Report 97-12, Computer Sciences Department, University of Wisconsin, Madison (1994); `ftp://ftp.cs.wisc.edu/math-prog/tech-reports/97-12.ps.Z`. Forthcoming in *Computational Optimization and Applications.*

[17] M.C. Ferris and J.-S. Pang, *Complementarity and Variational Problems: State of the Art.* SIAM (1997).

[18] M.C. Ferris and J.-S. Pang, Engineering and economic applications of complementarity problems. *SIAM Review* **39** (1997) 669–713.

[19] R. Fourer, Extending a general-purpose algebraic modeling language to combinatorial optimization: a logic programming approach. In *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search: Interfaces in Computer Science and Operations Research,* D.L. Woodruff, ed., Kluwer Academic Publishers (1998) 31–74.

[20] 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) 135–154.

[21] R. Fourer and D.M. Gay, Expressing special structures in an algebraic modeling language for mathematical programming. *ORSA Journal on Computing* **7** (1995) 166–190.

[22] R. Fourer, D.M. Gay and B.W. Kernighan, A modeling language for mathematical programming. *Management Science* **36** (1990) 519–554.

[23] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming.* Scientific Press/Duxbury Press (1993).

[24] D.M. Gay, Hooking your solver to AMPL. Technical Report 97-4-06, Computing Sciences Research Center, Bell Laboratories, Lucent Technologies (1997); `http://www.ampl.com/ampl/REFS/hooking2.ps.gz`.

[25] D.C. Hanselman and B.C. Littlefield, *Mastering MATLAB 5: A Comprehensive Tutorial and Reference.* Prentice Hall (1997).

[26] P.T. Harker and J.-S. Pang, Finite-dimensional variational inequality and nonlinear complementarity problems: a survey of theory, algorithms and applications. *Mathematical Programming* **48** (1990) 161–220.

[27] M.M. Kostreva, Elasto-hydrodynamic lubrication: a non-linear complementarity problem. *International Journal for Numerical Methods in Fluids* **4** (1984) 377–397.

[28] Z.-Q. Luo, J.-S. Pang and D. Ralph, *Mathematical Programs with Equilibrium Constraints.* Cambridge University Press (1996).

[29] Z.-Q. Luo, J.-S. Pang, D. Ralph and S.-Q. Wu, Exact penalization and stationarity conditions of mathematical programs with equilibrium constraints. *Mathematical Programming* **75** (1996) 19–76.

[30] D. Redfern and C. Campbell, *MATLAB Handbook.* Springer Verlag (1998).

[31] R.T. Rockafellar, *Convex Analysis.* Princeton University Press (1970).

[32] T.F. Rutherford, MILES: a Mixed Inequality and nonLinear Equation Solver. Working paper, Dept. of Economics, University of Colorado (1993); `http://robles.colorado.edu/~tomruth/milesdoc/milesdoc.htm`.

[33] T.F. Rutherford, Extensions of GAMS for complementarity problems arising in applied economic analysis. *Journal of Economic Dynamics & Control* **19** (1995) 1299–1324.

[34] L. Schrage, *Optimization Modeling with LINGO.* LINDO Systems, Inc. (1998).