19

Complementarity Problems

A variety of physical and economic phenomena are most naturally modeled by saying that certain pairs of inequality constraints must be *complementary,* in the sense that at least one must hold with equality. These conditions may in principle be accompanied by an objective function, but are more commonly used to construct *complementarity problems* for which a feasible solution is sought. Indeed, optimization may be viewed as a special case of complementarity, since the standard optimality conditions for linear and smooth nonlinear optimization are complementarity problems. Other kinds of complementarity problems do not arise from optimization, however, or cannot be conveniently formulated or solved as optimization problems.

The AMPL operator `complements` permits complementarity conditions to be specified directly in constraint declarations. Complementarity models can thereby be formulated in a natural way, and instances of such models are easily sent to special solvers for complementarity problems.

To motivate the syntax of `complements`, we begin by describing how it would be used to model a few simple economic equilibrium problems, some equivalent to linear programs and some not. We then give a general definition of the `complements` operator for pairs of inequalities and for more general ‘‘mixed’’ complementarity conditions via double inequalities. Where appropriate in these sections, we also comment on an AMPL interface to the PATH solver for ‘‘square’’ mixed complementarity problems. In a final section, we describe how complementarity constraints are accommodated in several of AMPL’s existing features, including presolve, constraint-name suffixes, and generic synonyms for constraints.

19.1 Sources of complementarity

Economic equilibria are one of the best-known applications of complementarity conditions. We begin this section by showing how a previous linear programming example in production economics has an equivalent form as a complementarity model, and how

```
set PROD;  # products
set ACT;   # activities

param cost {ACT} > 0;     # cost per unit of each activity
param demand {PROD} >= 0; # units of demand for each product
param io {PROD,ACT} >= 0; # units of each product from
                          # 1 unit of each activity

var Level {j in ACT} >= 0;

minimize Total_Cost:  sum {j in ACT} cost[j] * Level[j];

subject to Demand {i in PROD}:
   sum {j in ACT} io[i,j] * Level[j] >= demand[i];
```

Figure 19-1: Production cost minimization model (`econmin.mod`).

bounded variables are handled though an extension to the concept of complementarity. We then describe a further extension to price-dependent demands that is not motivated by optimization or equivalent to any linear program. We conclude by briefly describing other complementarity models and applications.

A complementarity model of production economics

In Section 2.4 we observed that the form of a diet model also applies to a model of production economics. The decision variables may be taken as the levels of production activities, so that the objective is the total production cost,

```
minimize Total_Cost:  sum {j in ACT} cost[j] * Level[j];
```

where `cost[j]` and `Level[j]` are the cost per unit and the level of activity `j`. The constraints say that the totals of the product outputs must be at least the product demands:

```
subject to Demand {i in PROD}:
   sum {j in ACT} io[i,j] * Level[j] >= demand[i];
```

with `io[i,j]` being the amount of product `i` produced per unit of activity `j`, and `demand[i]` being the total quantity of product `i` demanded. Figures 19-1 and 19-2 show this ‘‘economic’’ model and some data for it.

Minimum-cost production levels are easily computed by a linear programming solver:

```
ampl: model econmin.mod;
ampl: data econ.dat;

ampl: solve;
CPLEX 8.0.0: optimal solution; objective 6808640.553
3 dual simplex iterations (0 in phase I)
```

```
param: ACT:   cost  :=
        P1    2450      P1a   1290
        P2    1850      P2a   3700      P2b   2150
        P3    2200      P3c   2370
        P4    2170  ;

param: PROD:  demand :=
       AA1     70000
       AC1     80000
       BC1     90000
       BC2     70000
       NA2    400000
       NA3    800000  ;

param io (tr):
         AA1  AC1  BC1  BC2   NA2   NA3 :=
   P1     60   20   10   15   938   295
   P1a     8    0   20   20  1180   770
   P2      8   10   15   10   945   440
   P2a    40   40   35   10   278   430
   P2b    15   35   15   15  1182   315
   P3     70   30   15   15   896   400
   P3c    25   40   30   30  1029   370
   P4     60   20   15   10  1397   450 ;
```

Figure 19-2: Data for production models (`econ.dat`).

```
ampl: display Level;
Level [*] :=
 P1      0
P1a  1555.3
 P2      0
P2a      0
P2b      0
 P3   147.465
P3c  1889.4
 P4      0
```

Recall (from Section 12.5) that there are also dual or marginal values — or ‘‘prices’’ — associated with the constraints:

```
ampl: display Demand.dual;
Demand.dual [*] :=
AA1  16.7051
AC1   5.44585
BC1  57.818
BC2   0
NA2   0
NA3   0
```

In the conventional linear programming interpretation, the price on constraint i gives, within a sufficiently small range, the change in the total cost per unit change in the demand for product i.

Consider now an alternative view of the production economics problem, in which we define variables Price[i] as well as Level[j] and seek an *equilibrium* rather than an optimum solution. There are two requirements that the equilibrium solution must satisfy.

First, for each product, total output must meet demand and the price must be nonnegative, and in addition there must be a *complementarity* between these relationships: where production exceeds demand the price must be zero, or equivalently, where the price is positive the production must equal the demand. This relationship is expressed in AMPL by means of the complements operator:

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

When two inequalities are joined by complements, they both must hold, and at least one must hold with equality. Because our example is indexed over the set PROD, it sets up a relationship of this kind for each product.

Second, for each activity, there is another relationship that may at first be less obvious. Consider that, for each unit of activity j, the value of the resulting product i output in terms of the model's prices is Price[i] * io[i,j]. The total value of all outputs from one unit of activity j is thus

```
sum {i in ACT} Price[i] * io[i,j]
```

At equilibrium prices, this total value cannot exceed the activity's cost per unit, cost[j]. Moreover, there is a complementarity between this relationship and the level of activity j: where cost *exceeds* total value the activity must be zero, or equivalently, where the activity is positive the total value must equal the cost. Again this relationship can be expressed in AMPL with the complements operator:

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

Here the constraint is indexed over ACT, so that we have a complementarity relationship for each activity.

Putting together the two collections of complementarity constraints, we have the linear *complementarity problem* shown in Figure 19-3. The number of variables and the number of complementarity relationships are equal (to activities plus products), making this a ‘‘square’’ complementarity problem that is amenable to certain solution techniques, though not the same techniques as those for linear programs.

Applying the PATH solver, for example, the complementarity problem can be seen to have the same solution as the related minimum-cost production problem:

```
set PROD;   # products
set ACT;    # activities

param cost {ACT} > 0;        # cost per unit of each activity
param demand {PROD} >= 0;    # units of demand for each product
param io {PROD,ACT} >= 0;    # units of each product from
                             # 1 unit of each activity

var Price {i in PROD};
var Level {j in ACT};

subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j] * Level[j] >= demand[i];

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

Figure 19-3: Production equilibrium model (`econ.mod`).

```
ampl: model econ.mod;
ampl: data econ.dat;
ampl: option solver path;
ampl: solve;
Path v4.5: Solution found.
7 iterations (0 for crash); 33 pivots.
20 function, 8 gradient evaluations.

ampl: display sum {j in ACT} cost[j] * Level[j];
sum{j in ACT} cost[j]*Level[j] = 6808640
```

Further application of `display` shows that `Level` is the same as in the production economics LP and that `Price` takes the same values that `Demand.dual` has in the LP.

Complementarity for bounded variables

Suppose now that we extend our models by placing bounds on the activity levels: `level_min[j] <= Level[j] <= level_max[j]`. The equivalence between the optimization problem and a square complementarity problem can be maintained, provided that the complementarity relationship for the activities is generalized to a ‘‘mixed’’ form. Where an activity’s cost is greater than its total value (per unit), the activity’s level must be *at its lower bound* (much as before). Where an activity’s level is *between its bounds,* its cost must equal its total value. And an activity’s cost may also be less than its total value, provided that its level is *at its upper bound.* These three relationships are summarized by another form of the `complements` operator:

```
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];
```

```
set PROD;   # products
set ACT;    # activities

param cost {ACT} > 0;      # cost per unit of each activity
param demand {PROD} >= 0;  # units of demand for each product
param io {PROD,ACT} >= 0;  # units of each product from
                           # 1 unit of each activity

param level_min {ACT} > 0; # min allowed level for each activity
param level_max {ACT} > 0; # max allowed level for each activity

var Price {i in PROD};
var Level {j in ACT};

subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j] * Level[j] >= demand[i];

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];
```

Figure 19-4: Bounded version of production equilibrium model (`econ2.mod`).

When a double inequality is joined to an expression by `complements`, the inequalities must hold, and either the expression must be zero, or the lower inequality must hold with equality and the expression must be nonnegative, or the upper inequality must hold with equality and the expression must be nonpositive.

A bounded version of our complementarity examples is shown in Figure 19-4. The PATH solver can be applied to this model as well:

```
ampl: model econ2.mod;
ampl: data econ2.dat;
ampl: option solver path;
ampl: solve;
Path v4.5: Solution found.
9 iterations (4 for crash); 8 pivots.
22 function, 10 gradient evaluations.

ampl: display level_min, Level, level_max;
:   level_min  Level level_max  :=
P1      240      240    1000
P1a     270     1000    1000
P2      220      220    1000
P2a     260      680    1000
P2b     200      200    1000
P3      260      260    1000
P3c     220     1000    1000
P4      240      240    1000
;
```

The results are the same as for the LP that is derived from our previous example (Figure 19-1) by adding the bounds above to the variables.

```
set PROD;   # products
set ACT;    # activities

param cost {ACT} > 0;       # cost per unit of each activity
param io {PROD,ACT} >= 0;   # units of each product from
                            # 1 unit of each activity

param demzero {PROD} > 0;  # intercept and slope of the demand
param demrate {PROD} >= 0; # as a function of price

var Price {i in PROD};
var Level {j in ACT};

subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j] * Level[j]
         >= demzero[i] - demrate[i] * Price[i];

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

Figure 19-5: Price-dependent demands (`econnl.mod`).

Complementarity for price-dependent demands

If complementarity problems only arose from linear programs, they would be of very limited interest. The idea of an economic equilibrium can be generalized, however, to problems that have no LP equivalents. Rather than taking the demands to be fixed, for example, it makes sense to view the demand for each product as a decreasing function of its price.

The simplest case is a decreasing linear demand, which could be expressed in AMPL as

```
demzero[i] - demrate[i] * Price[i]
```

where `demzero[i]` and `demrate[i]` are nonnegative parameters. The resulting complementarity problem simply substitutes this expression for `demand[i]`, as seen in Figure 19-5. The complementarity problem remains square, and can still be solved by PATH, but with clearly different results:

```
ampl: model econnl.mod;
ampl: data econnl.dat;

ampl: option solver path;

ampl: solve;
Path v4.5: Solution found.
11 iterations (3 for crash); 11 pivots.
12 function, 12 gradient evaluations.
```

```
ampl: display Level;
Level [*] :=
 P1  240
P1a  710.156
 P2  220
P2a  260
P2b  200
 P3  260
P3c  939.063
 P4  240
;
```

The balance between demands and prices now tends to push down the equilibrium production levels.

Because the `Price[i]` variables appear on both sides of the `complements` operator in this model, there is no equivalent linear program. There does exist an equivalent nonlinear optimization model, but it is not as easy to derive and may be harder to solve as well.

Other complementarity models and applications

This basic example can be extended to considerably more complex models of economic equilibrium. The activity and price variables and their corresponding complementarity constraints can be comprised of several indexed collections each, and both the cost and price functions can be nonlinear. A solver such as PATH handles all of these extensions, so long as the problem remains square in the sense of having equal numbers of variables and complementarity constraints (or being easily converted to such a form as explained in the next section).

More ambitious models may add an objective function and may mix equality, inequality and complementarity constraints in arbitrary numbers. Solution techniques for these so-called MPECs — mathematical programs with equilibrium constraints — are at a relatively experimental stage, however.

Complementarity problems also arise in physical systems, where they can serve as models of equilibrium conditions between forces. A complementarity constraint may represent a discretization of the relationship between two objects, for example. The relationship on one side of the `complements` operator may hold with equality at points where the objects are in contact, while the relationship on the other side holds with equality where they do not touch.

Game theory provides another class of examples. The Nash equilibrium for a bimatrix game is characterized by complementarity conditions, for example, in which the variables are the probabilities with which the two players make their available moves. For each move, either its probability is zero, or a related equality holds to insure there is nothing to be gained by increasing or decreasing its probability.

Surveys that describe a variety of complementarity problems in detail are cited in the references at the end of this chapter.

19.2 Forms of complementarity constraints

An AMPL complementarity constraint consists of two expressions or constraints separated by the `complements` operator. There are always *two inequalities,* whose position determines how the constraint is interpreted.

If there is one inequality on either side of `complements`, the constraint has the general form

> *single-inequality* `complements` *single-inequality* `;`

where a *single-inequality* is any valid ordinary constraint — linear or nonlinear — containing one `>=` or `<=` operator. A constraint of this type is satisfied if both of the *single-inequality* relations are satisfied, and at least one is satisfied with equality.

If both inequalities are on the same side of the `complements` operator, the constraint has instead one of the forms

> *double-inequality* `complements` *expression* `;`
> *expression* `complements` *double-inequality* `;`

where *double-inequality* is any ordinary AMPL constraint containing two `>=` or two `<=` operators, and *expression* is any numerical expression. Variables may appear nonlinearly in either the *double-inequality* or the *expression* (or both). The conditions for a constraint of this type to be satisfied are as follows:

- if the left side `<=` or the right side `>=` of the *double-inequality* holds with equality, then the *expression* is greater than or equal to 0;
- if the right side `<=` or the left side `>=` of the *double-inequality* holds with equality, then the *expression* is less than or equal to 0;
- if neither side of the *double-inequality* holds with equality, then the *expression* equals 0.

In the special case where the *double-inequality* has the form `0 <=` *body* `<= Infinity`, these conditions reduce to those for complementarity of a pair of single inequalities.

For completeness, the special case in which the left-hand side equals the right-hand side of the double inequality may be written using one of the forms

> *equality* `complements` *expression* `;`
> *expression* `complements` *equality* `;`

A constraint of this kind is equivalent to an ordinary constraint consisting only of the *equality*; it places no restrictions on the *expression*.

For the use of solvers that require ‘‘square’’ complementarity systems, AMPL converts to square any model instance in which the number of variables equals the number of complementarity constraints plus the number of equality constraints. There may be any number of additional inequality constraints, but there must not be any objective. Each equality is trivially turned into a complementarity condition, as observed above; each added inequality is made complementary to a new, otherwise unused variable, preserving the squareness of the problem overall.

19.3 Working with complementarity constraints

All of AMPL's features for ordinary equalities and inequalities extend in a straightforward way to complementarity constraints. This section covers extensions in three areas: expressions for related solution values, effects of presolve and related displays of problem statistics, and generic synonyms for constraints.

Related solution values

AMPL's built-in suffixes for values related to a problem and its solution extend to complementarity constraints, but with two collections of suffixes — of the form *cname*`.L`*suf* and *cname*`.R`*suf* — corresponding to the left and right operands of `complements`, respectively. Thus after `econ2.mod` (Figure 19-4) has been solved, for example, we can use the following `display` command to look at values associated with the constraint `Lev_Compl`:

```
ampl: display Lev_Compl.Llb, Lev_Compl.Lbody,
ampl?         Lev_Compl.Rbody, Lev_Compl.Rslack;
```

```
: Lev_Compl.Llb Lev_Compl.Lbody Lev_Compl.Rbody Lev_Compl.Rslack :=
P1        240           240      1392.86              Infinity
P1a       270          1000      -824.286             Infinity
P2        220           220       264.286             Infinity
P2a       260           680         5.00222e-12       Infinity
P2b       200           200       564.286             Infinity
P3        260           260       614.286             Infinity
P3c       220          1000      -801.429             Infinity
P4        240           240       584.286             Infinity
;
```

Because the right operand of `Lev_Compl` is an expression, it is treated as a ‘‘constraint’’ with infinite lower and upper bounds, and hence infinite slack.

A suffix of the form *cname*`.slack` is also defined for complementarity constraints. For complementary pairs of single inequalities, it is equal to the lesser of *cname*`.Lslack` and *cname*`.Rslack`. Hence it is nonnegative if and only if both inequalities are satisfied and is zero if the complementarity constraint holds exactly. For complementary double inequalities of the form

> *expr* `complements` *lbound* `<=` *body* `<=` *ubound*
> *lbound* `<=` *body* `<=` *ubound* `complements` *expr*

cname`.slack` is defined to be

`min(`*expr*`,` *body* `-` *lbound*`)`	if *body* `<=` *lbound*
`min(`–*expr*`,` *ubound* `-` *body*`)`	if *body* `>=` *ubound*
`-abs(`*expr*`)`	otherwise

Hence in this case it is always nonpositive, and is zero when the complementarity constraint is satisfied exactly.

If *cname* for a complementarity constraint appears unsuffixed in an expression, it is interpreted as representing *cname*`.slack`.

Presolve

As explained in Section 14.1, AMPL incorporates a presolve phase that can substantially simplify some linear programs. In the presence of complementarity constraints, several new kinds of simplifications become possible.

As an example, given a constraint of the form

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

if presolve can deduce that $expr_1$ is strictly positive for all feasible points — in other words, that it has a positive lower bound — it can replace the constraint by $expr_2$ `= 0`.

Similarly, in a constraint of the form

lbound `<=` *body* `<=` *ubound* `complements` *expr*

there are various possibilities, including the following:

If presolve can deduce for all feasible points that	Then the constraint can be replaced by
body `<` *ubound*	*lbound* `<=` *body* `complements` *expr* `>= 0`
lbound `<` *body* `<` *ubound*	*expr* `= 0`
expr `< 0`	*body* `=` *ubound*

Transformations of these kinds are carried out automatically, unless `option presolve 0` is used to turn off the presolve phase. As with ordinary constraints, results are reported in terms of the original model.

By displaying a few predefined parameters:

`_ncons`	the number of ordinary constraints before presolve
`_nccons`	the number of complementarity conditions before presolve
`_sncons`	the number of ordinary constraints after presolve
`_snccons`	the number of complementarity conditions after presolve

or by setting `option show_stats 1`, you can get some information on the number of simplifying transformations that presolve has made:

```
ampl: model econ2.mod; data econ2.dat;
ampl: option solver path;
ampl: option show_stats 1;
ampl: solve;
```

```
Presolve eliminates 16 constraints and 2 variables.
Presolve resolves 2 of 14 complementarity conditions.
Adjusted problem:
12 variables, all linear
12 constraints, all linear; 62 nonzeros
12 complementarity conditions among the constraints:
        12 linear, 0 nonlinear.
0 objectives.

Path v4.5: Solution found.
7 iterations (1 for crash); 30 pivots.
8 function, 8 gradient evaluations.

ampl: display _ncons, _nccons, _sncons, _snccons;
_ncons = 28
_nccons = 14
_sncons = 12
_snccons = 12
```

When first instantiating the problem, AMPL counts each complementarity constraint as two ordinary constraints (the two arguments to complements) and also as a complementarity condition. Thus _nccons equals the number of complementarity constraints before presolve, and _ncons equals twice _nccons plus the number of any non-complementarity constraints before presolve. The presolve messages at the beginning of the show_stats output indicate how much presolve was able to reduce these numbers.

In this case the reason for the reduction can be seen by comparing each product's demand to the minimum possible output of that product — the amount that results from setting each Level[j] to level_min[j]:

```
ampl: display {i in PROD}
ampl?    (sum{j in ACT} io[i,j]*level_min[j], demand[i]);
:   sum{j in ACT} io[i,j]*level_min[j] demand[i]    :=
AA1                  69820                   70000
AC1                  45800                   80000
BC1                  37300                   90000
BC2                  29700                   70000
NA2                1854920                   4e+05
NA3                 843700                   8e+05
;
```

We see that for products NA2 and NA3, the total output exceeds demand even at the lowest activity levels. Hence in the constraint

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

the right-hand argument to complements never holds with equality for NA2 or NA3. Presolve thus concludes that Price["NA2"] and Price["NA3"] can be fixed at zero, removing them from the resulting problem.

Generic synonyms

AMPL's generic synonyms for constraints (Section 12.6) extend to complementarity conditions, mainly through the substitution of `ccon` for `con` in the synonym names.

From the modeler's view (before presolve), the ordinary constraint synonyms remain:

`_ncons`	number of ordinary constraints before presolve
`_conname`	names of the ordinary constraints before presolve
`_con`	synonyms for the ordinary constraints before presolve

The complementarity constraint synonyms are:

`_nccons`	number of complementarity constraints before presolve
`_cconname`	names of the complementarity constraints before presolve
`_ccon`	synonyms for the complementarity constraints before presolve

Because each complementarity constraint also gives rise to two ordinary constraints, as explained in the preceding discussion of presolve, there are two entries in `_conname` corresponding to each entry in `_cconname`:

```
ampl: display {i in 1..6} (_conname[i], _cconname[i]);
:       _conname[i]            _cconname[i]        :=
1   "Pri_Compl['AA1'].L"   "Pri_Compl['AA1']"
2   "Pri_Compl['AA1'].R"   "Pri_Compl['AC1']"
3   "Pri_Compl['AC1'].L"   "Pri_Compl['BC1']"
4   "Pri_Compl['AC1'].R"   "Pri_Compl['BC2']"
5   "Pri_Compl['BC1'].L"   "Pri_Compl['NA2']"
6   "Pri_Compl['BC1'].R"   "Pri_Compl['NA3']"
;
```

For each complementarity constraint *cname*, the left and right arguments to the `complements` operator are the ordinary constraints named *cname*`.L` and *cname*`.R`. This is confirmed by using the synonym terminology to expand the complementarity constraint `Pri_Compl['AA1']` and the corresponding two ordinary constraints from the example above:

```
ampl: expand Pri_Compl['AA1'];
subject to Pri_Compl['AA1']:
        Price['AA1'] >= 0
   complements
        60*Level['P1'] + 8*Level['P1a'] + 8*Level['P2'] +
        40*Level['P2a'] + 15*Level['P2b'] + 70*Level['P3'] +
        25*Level['P3c'] + 60*Level['P4'] >= 70000;

ampl: expand _con[1], _con[2];
subject to Pri_Compl.L['AA1']:
        Price['AA1'] >= 0;
subject to Pri_Compl.R['AA1']:
        60*Level['P1'] + 8*Level['P1a'] + 8*Level['P2'] +
        40*Level['P2a'] + 15*Level['P2b'] + 70*Level['P3'] +
        25*Level['P3c'] + 60*Level['P4'] >= 70000;
```

From the solver's view (after presolve), a more limited collection of synonyms is defined:

`_sncons`	number of all constraints after presolve
`_snccons`	number of complementarity constraints after presolve
`_sconname`	names of all constraints after presolve
`_scon`	synonyms for all constraints after presolve

Necessarily `_snccons` is less than or equal to `_sncons`, with equality only when all constraints are complementarity constraints.

To simplify the problem description that is sent to the solver, AMPL converts every complementarity constraint into one of the following canonical forms:

expr `complements` *lbound* `<=` *var* `<=` *ubound*
expr `<= 0 complements` *var* `<=` *ubound*
expr `>= 0 complements` *lbound* `<=` *var*

where *var* is the name of a *different* variable for each constraint. (Where an expression more complicated than a single variable appears on both sides of `complements`, this involves the introduction of an auxiliary variable and an equality constraint defining the variable to equal one of the expressions.) By using `solexpand` in place of `expand`, you can see the form in which AMPL has sent a complementarity constraint to the solver:

```
ampl: solexpand Pri_Compl['AA1'];
subject to Pri_Compl['AA1']:
   -70000 + 60*Level['P1'] + 8*Level['P1a'] + 8*Level['P2'] +
   40*Level['P2a'] + 15*Level['P2b'] + 70*Level['P3'] +
   25*Level['P3c'] + 60*Level['P4'] >= 0
 complements
   0 <= Price['AA1'];
```

A predefined array of integers, `_scvar`, gives the indices of the complementing variables in the generic variable arrays `_var` and `_varname`. This terminology can be used to display a list of names of such variables:

```
ampl: display {i in 1..3} (_sconname[i],_svarname[_scvar[i]]);
:       _sconname[i]     _svarname[_scvar[i]]    :=
1   "Pri_Compl['AA1'].R"   "Price['AA1']"
2   "Pri_Compl['AC1'].R"   "Price['AC1']"
3   "Pri_Compl['BC1'].R"   "Price['BC1']"
;
```

When constraint `i` is an ordinary equality or inequality, `_scvar[i]` is 0. The names of complementarity constraints in `_sconname` are suffixed with `.L` or `.R` according to whether the *expr* in the constraint sent to the solver was derived from the left or right argument to `complements` in the original constraint.

Bibliography

Richard W. Cottle, Jong-Shi Pang, and Richard E. Stone, *The Linear Complementarity Problem*, Academic Press (San Diego, CA, 1992). An encyclopedic account of linear complementarity problems with a nice overview of how these problems arise.

Steven P. Dirkse and Michael C. Ferris, ‘‘MCPLIB: A Collection of Nonlinear Mixed Complementarity Problems.’’ Optimization Methods and Software **5**, 4 (1995) pp. 319–345. An extensive survey of nonlinear complementarity, including problem descriptions and mathematical formulations.

Michael C. Ferris and Jong-Shi Pang, ‘‘Engineering and Economic Applications of Complementarity Problems.’’ SIAM Review **39**, 4 (1997) pp. 669–713. A variety of complementarity test problems, originally written in the GAMS modeling language but now in many cases translated to AMPL.

Exercises

19-1. The economics example in Section 19.1 used a demand function that was linear in the price. Construct a nonlinear demand function that has each of the characteristics described below. Define a corresponding complementarity problem, using the data from Figure 19-2 as much as possible.

Use a solver such as PATH to compute an equilibrium solution. Compare this solution to those for the constant-demand and linear-demand alternatives shown in Section 19.1.

(a) For price `i` near zero the demand is near `demzero[i]` and is decreasing at a rate near `demrate[i]`. After price `i` has increased substantially, however, both the demand and the rate of decrease of the demand approach zero.

(b) For price `i` near zero the demand is approximately constant at `demzero[i]`, but as price `i` approaches `demlim[i]` the demand drops quickly to zero.

(c) Demand for `i` actually rises with price, until it reaches a value `demmax[i]` at a price of `demarg[i]`. Then demand falls with price.

19-2. For each scenario in the previous problem, experiment with different starting points for the `Level` and `Price` values. Determine whether there appears a unique equilibrium point.

19-3. A *bimatrix game* between players A and B is defined by two m by n ‘‘payoff’’ matrices, whose elements we denote by a_{ij} and b_{ij}. In one round of the game, player A has a choice of m alternatives and player B a choice of n alternatives. If A plays (chooses) i and B plays j, then A and B win amounts a_{ij} and b_{ij}, respectively; negative winnings are interpreted as losses.

We can allow for ‘‘mixed’’ strategies in which A plays i with probability p_i^A and B plays j with probability p_j^B. Then the expected value of player A’s winnings is:

$$\sum_{j=1}^{n} a_{ij} \times p_j^B, \text{ if A plays } i$$

and the expected value of player B’s winnings is:

$$\sum_{i=1}^{m} b_{ij} \times p_i^A, \text{ if B plays } j$$

A ‘‘pure’’ strategy is the special case in which each player has one probability equal to 1 and the rest equal to 0.

A pair of strategies is said to represent a *Nash equilibrium* if neither player can improve his expected payoff by changing only his own strategy.

(a) Show that the requirement for a Nash equilibrium is equivalent to the following complementarity-like conditions:

> for all i such that $p_i^A > 0$, A's expected return when playing i equals A's maximum expected return over all possible plays
>
> for all j such that $p_j^B > 0$, B's expected return when playing j equals B's maximum expected return over all possible plays

(b) To build a complementarity problem in AMPL whose solution is a Nash equilibrium, the parameters representing the payoff matrices can be defined by the following `param` declarations:

```
param nA > 0;   # actions available to player A
param nB > 0;   # actions available to player B

param payoffA {1..nA, 1..nB};   # payoffs to player A
param payoffB {1..nA, 1..nB};   # payoffs to player B
```

The probabilities that define the mixed strategies are necessarily variables. In addition it is convenient to define a variable to represent the maximum expected payoff for each player:

```
var PlayA {i in 1..nA};   # player A's mixed strategy
var PlayB {j in 1..nB};   # player B's mixed strategy

var MaxExpA;   # maximum expected payoff to player A
var MaxExpB;   # maximum expected payoff to player B
```

Write AMPL declarations for the following constraints:

- The probabilities in any mixed strategy must be nonnegative.
- The probabilities in each player's mixed strategy must sum to 1.
- Player A's expected return when playing any particular i
  must not exceed A's maximum expected return over all possible plays
- Player B's expected return when playing any particular j
  must not exceed B's maximum expected return over all possible plays

(c) Write an AMPL model for a square complementarity system that enforces the constraints in (b) and the conditions in (a).

(d) Test your model by applying it to the ''rock-scissors-paper'' game in which both players have the payoff matrix

```
 0   1  -1
-1   0   1
 1  -1   0
```

Confirm that an equilibrium is found where each player chooses between all three plays with equal probability.

(e) Show that the game for which both players have the payoff matrix

```
-3   1   3  -1
 2   3  -1  -5
```

has several equilibria, at least one of which uses mixed strategies and one of which uses pure strategies.

Running a solver such as PATH will only return one equilibrium solution. To find more, experiment with changing the initial solution or fixing some of the variables to 0 or 1.

19-4. Two companies have to decide now whether to adopt standard 1 or standard 2 for future introduction in their products. If they decide on the same standard, company A has the greater payoff because its technology is superior. If they decide on different standards, company B has the greater payoff because its market share is greater. These considerations lead to a bimatrix game whose payoff matrices are

```
A = 10   3     B = 4   6
     2   9         7   5
```

(a) Use a solver such as PATH to find a Nash equilibrium. Verify that it is a mixed strategy, with A's probabilities being 1/2 for both standards and B's probabilities being 3/7 and 4/7 for standards 1 and 2, respectively.

Why is a mixed strategy not appropriate for this application?

(b) You can see what happens when company A decides on standard 1 by issuing the following commands:

```
ampl: fix PlayA[1] := 1;
ampl: solve;
presolve, constraint ComplA[1].L:
        all variables eliminated, but upper bound = -1 < 0
```

Explain how AMPL's presolve phase could deduce that the complementarity problem has no feasible solution in this case.

(c) Through further experimentation, show that there are no Nash equilibria for this situation that involve only pure strategies.