
Columnwise Formulations

Because the fundamental idea of an optimization problem is to minimize or maximize a function of the decision variables, subject to constraints on them, AMPL is oriented toward explicit descriptions of variables and constraints. This is why `var` declarations tend to come first, followed by the `minimize` or `maximize` and `subject to` declarations that use the variables. A wide variety of optimization problems can be formulated with this approach, as the examples in this book demonstrate.

For certain kinds of linear programs, however, it can be preferable to declare the objective and constraints before the variables. Usually in these cases, there is a much simpler pattern or interpretation to the coefficients of a single variable down all the constraints than to the coefficients of a single constraint across all the variables. In the jargon of linear programming, it is easier to describe the matrix of constraint coefficients ~~columnwise~~ than ~~row-wise~~. As a result, the formulation is simplified by first declaring the constraints and objective, then listing the nonzero coefficients in the declarations of the variables.

One example of this phenomenon is provided by the network linear programs described in Chapter 15. Each variable has at most two nonzero coefficients in the constraints, a +1 and a -1. Rather than trying to describe the constraints algebraically, you may find it easier to specify, in each variable's declaration, the one or two constraints that the variable figures in. In fact, this is exactly what you do by using the special `node` and `arc` declarations introduced by Section 15.3. The `node` declarations come first to describe the nature of the constraints at the network nodes. Then the `arc` declarations define the network flow variables, using `from` and `to` phrases to locate their nonzero coefficients among the node constraints. This approach is particularly appealing because it corresponds directly to the way most people think about a network flow problem.

It would be impractical for AMPL to offer special declarations and phrases for every kind of linear program that you might want to declare by columns rather than by rows. Instead, additional options to the `var` and `subject to` declarations permit any linear program to be given a columnwise declaration. This chapter introduces AMPL's columnwise features through two contrasting examples ~~in~~ an input-output production model, and

a work-shift scheduling model and concludes with a summary of the language extensions that may be used in columnwise formulations.

16.1 An input-output model

In simple maximum-profit production models such as the examples in Chapter 1, the goods produced are distinct from the resources consumed, so that overall production is limited in an obvious way by resources available. In a more realistic model of a complex operation such as a steel mill or refinery, however, production is carried out at a series of units; as a result, some of a production unit's inputs may be the outputs from other units. For this situation we need a model that deals more generally with materials that may be inputs or outputs, and with production activities that may involve several inputs and outputs each.

We begin by developing an AMPL formulation in the usual row-wise (or constraint-oriented) way. Then we explain the columnwise (or variable-oriented) alternative, and discuss refinements of the model.

Formulation by constraints

The definition of our model starts with a set of materials and a set of activities:

```
set MAT;
set ACT;
```

The key data values are the input-output coefficients for all material-activity combinations:

```
param io {MAT,ACT};
```

If $io[i, j] > 0$, it is interpreted as the amount of material i produced (as an output) by a unit of activity j . On the other hand, if $io[i, j] < 0$, it represents minus the amount of material i consumed (as an input) by a unit of activity j . For example, a value of 10 represents 10 units of i produced per unit of j , while a value of ~~10~~ -10 represents 10 units consumed per unit of j . Of course, we can expect that for many combinations of i and j we will have $io[i, j] = 0$, signifying that material i does not figure in activity j at all.

To see why we want to interpret $io[i, j]$ in this manner, suppose we define $Run[j]$ to be the level at which we operate (run) activity j :

```
param act_min {ACT} >= 0;
param act_max {j in ACT} >= act_min[j];
var Run {j in ACT} >= act_min[j], <= act_max[j];
```

Then $io[i, j] * Run[j]$ is the total amount of material i produced (if $io[i, j] > 0$) or minus the amount of material i consumed (if $io[i, j] < 0$) by activity j . Summing over all activities, we see that

```

set MAT;           # materials
set ACT;           # activities
param io {MAT,ACT}; # input-output coefficients

param revenue {ACT};
param act_min {ACT} >= 0;
param act_max {j in ACT} >= act_min[j];

var Run {j in ACT} >= act_min[j], <= act_max[j];

maximize Net_Profit: sum {j in ACT} revenue[j] * Run[j];

subject to Balance {i in MAT}:
    sum {j in ACT} io[i,j] * Run[j] = 0;

```

Figure 16-1: Input-output model by rows (iorow.mod).

```

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

```

represents the amount of material i produced in the operation minus the amount consumed. These amounts must balance, as expressed by the following constraint:

```

subject to Balance {i in MAT}:
    sum {j in ACT} io[i,j] * Run[j] = 0;

```

What about the availability of resources, or the requirements for finished goods? These are readily modeled through additional activities that represent the purchase or sale of materials. A purchase activity for material i has no inputs and just i as an output; the upper limit on $\text{Run}[i]$ represents the amount of this resource available. Similarly, a sale activity for material i has no outputs and just i as an input, and the lower limit on $\text{Run}[i]$ represents the amount of this good that must be produced for sale.

We complete the model by associating unit revenues with the activities. Sale activities necessarily have positive revenues, while purchase and production activities have negative revenues—that is, costs. The sum of unit revenues times activity levels gives the total net profit of the operation:

```

param revenue {ACT};
maximize Net_Profit: sum {j in ACT} revenue[j] * Run[j];

```

The completed model is shown in Figure 16-1.

A columnwise formulation

As our discussion of purchase and sale activities suggests, everything in this model can be organized by activity. Specifically, for each activity j we have a decision variable $\text{Run}[j]$, a cost or income represented by $\text{revenue}[j]$, limits $\text{act_min}[j]$ and $\text{act_max}[j]$, and a collection of input-output coefficients $\text{io}[i, j]$. Changes such as improving the yield of a unit, or acquiring a new source of supply, are accommodated by adding an activity or by modifying the data for an activity.

In the formulation by rows, the activities' importance to this model is somewhat hidden. While `act_min[j]` and `act_max[j]` appear in the declaration of the variables, `revenue[j]` is in the objective, and the `io[i,j]` values are in the constraint declaration. The columnwise alternative brings all of this information together, by adding `obj` and `coeff` phrases to the `var` declaration:

```
var Run {j in ACT} >= act_min[j], <= act_max[j],
    obj Net_Profit revenue[j],
    coeff {i in MAT} Balance[i] io[i,j];
```

The `obj` phrase says that in the objective function named `Net_Profit`, the variable `Run[j]` has the coefficient `revenue[j]`; that is, the term `revenue[j] * Run[j]` should be added in. The `coeff` phrase is a bit more complicated, because it is indexed over a set. It says that for each material `i`, in the constraint `Balance[i]` the variable `Run[j]` should have the coefficient `io[i,j]`, so that the term `io[i,j] * Run[j]` is added in. Together, these phrases describe all the coefficients of all the variables in the linear program.

Since we have placed all the coefficients in the `var` declaration, we must remove them from the other declarations:

```
maximize Net_Profit;
subject to Balance {i in MAT}: to_come = 0;
```

The keyword `to_come` indicates where the terms `io[i,j] * Run[j]` generated by the `var` declaration are to be added in. You can think of `to_come = 0` as a template for the constraint, which will be filled out as the coefficients are declared. No template is needed for the objective in this example, however, since it is exclusively the sum of the terms `revenue[j] * Run[j]`. Templates may be written in a limited variety of ways, as shown in Section 16.3 below.

Because the `obj` and `coeff` phrases refer to `Net_Profit` and `Balance`, the `var` declaration must come after the `maximize` and `subject to` declarations in the columnwise formulation. The complete model is shown in Figure 16-2.

```
set MAT;           # materials
set ACT;           # activities
param io {MAT,ACT}; # input-output coefficients

param revenue {ACT};
param act_min {ACT} >= 0;
param act_max {j in ACT} >= act_min[j];

maximize Net_Profit;

subject to Balance {i in MAT}: to_come = 0;

var Run {j in ACT} >= act_min[j], <= act_max[j],
    obj Net_Profit revenue[j],
    coeff {i in MAT} Balance[i] io[i,j];
```

Figure 16-2: Columnwise formulation (`iocoll.mod`).

Refinements of the columnwise formulation

The advantages of a columnwise approach become more evident as the model becomes more complicated. As one example, consider what happens if we want to have separate variables to represent sales of finished materials. We declare a subset of materials that can be sold, and use it to index new collections of bounds, revenues and variables:

```
set MATF within MAT;    # finished materials

param revenue {MATF} >= 0;

param sell_min {MATF} >= 0;
param sell_max {i in MATF} >= sell_min[i];

var Sell {i in MATF} >= sell_min[i], <= sell_max[i];
```

We may now dispense with the special sale activities previously described. Since the remaining members of ACT represent purchase or production activities, we can introduce a nonnegative parameter `cost` associated with them:

```
param cost {ACT} >= 0;
```

In the row-wise approach, the new objective is written as

```
maximize Net_Profit:
    sum {i in MATF} revenue[i] * Sell[i]
    - sum {j in ACT} cost[j] * Run[j];
```

to represent total sales revenue minus total raw material and production costs.

So far we seem to have improved upon the model in Figure 16-1. The composition of net profit is more clearly modeled, and sales are restricted to explicitly designated finished materials; also the optimal amounts sold are more easily examined apart from the other variables, by a command such as `display Sell`. It remains to fix up the constraints. We would like to say that the net output of material `i` from all activities, represented as

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

in Figure 16-1, must balance the amount sold, either `Sell[i]` if `i` is a finished material, or zero. Thus the constraint declaration must be written:

```
subject to Balance {i in MAT}:
    sum {j in ACT} io[i,j] * Run[j]
    = if i in MATF then Sell[i] else 0;
```

Unfortunately this constraint seems less clear than our original one, due to the complication introduced by the `if-then-else` expression.

In the columnwise alternative, the objective and constraints are the same as in Figure 16-2, while all the changes are reflected in the declarations of the variables:

```

set MAT;           # materials
set ACT;           # activities

param io {MAT,ACT}; # input-output coefficients

set MATF within MAT; # finished materials

param revenue {MATF} >= 0;

param sell_min {MATF} >= 0;
param sell_max {i in MATF} >= sell_min[i];

param cost {ACT} >= 0;
param act_min {ACT} >= 0;
param act_max {j in ACT} >= act_min[j];

maximize Net_Profit;

subject to Balance {i in MAT}: to_come = 0;

var Run {j in ACT} >= act_min[j], <= act_max[j],
    obj Net_Profit -cost[j],
    coeff {i in MAT} Balance[i] io[i,j];

var Sell {i in MATF} >= sell_min[i], <= sell_max[i],
    obj Net_Profit revenue[i],
    coeff Balance[i] -1;

```

Figure 16-3: Columnwise formulation, with sales activities (iocol2.mod).

```

var Run {j in ACT} >= act_min[j], <= act_max[j],
    obj Net_Profit -cost[j],
    coeff {i in MAT} Balance[i] io[i,j];

var Sell {i in MATF} >= sell_min[i], <= sell_max[i],
    obj Net_Profit revenue[i],
    coeff Balance[i] -1;

```

In this view, the variable `Sell[i]` represents the kind of sale activity that we previously described, with only material `i` as input and no materials as output. Hence the single coefficient of `1` in constraint `Balance[i]`. We need not specify all the zero coefficients for `Sell[i]`; a zero is assumed for any constraint not explicitly cited in a `coeff` phrase in the declaration. The whole model is shown in Figure 16-3.

This example suggests that a columnwise approach is particularly suited to refinements of the input-output model that distinguish different kinds of activities. It would be easy to add another group of variables that represent purchases of raw materials, for instance.

On the other hand, versions of the input-output model that involve numerous specialized constraints would lend themselves more to a formulation by rows.

16.2 A scheduling model

In Section 2.4 we observed that the general form of a blending model was applicable to certain scheduling problems. Here we describe a related scheduling model for which the columnwise approach is particularly attractive.

Suppose that a factory's production for the next week is divided into fixed time periods, or shifts. You want to assign employees to shifts, so that the required number of people are working on each shift. You cannot fill each shift independently of the others, however, because only certain weekly schedules are allowed; for example, a person cannot work five shifts in a row. Your problem is thus more properly viewed as one of assigning employees to schedules, so that each shift is covered and the overall assignment is the most economical.

We can conveniently represent the schedules for this problem by an indexed collection of subsets of shifts:

```
set SHIFTS;                # shifts
param Nsched;              # number of schedules;
set SCHEDS = 1..Nsched;    # set of schedules

set SHIFT_LIST {SCHEDS} within SHIFTS;
```

For each schedule j in the set SCHEDS, the shifts that a person works on schedule j are contained in the set SHIFT_LIST[j]. We also specify a pay rate per person on each schedule, and the number of people required on each shift:

```
param rate {SCHEDS} >= 0;
param required {SHIFTS} >= 0;
```

We let the variable Work[j] represent the number of people assigned to work each schedule j , and minimize the sum of rate[j] * Work[j] over all schedules:

```
var Work {SCHEDS} >= 0;
minimize Total_Cost: sum {j in SCHEDS} rate[j] * Work[j];
```

Finally, our constraints say that the total of employees assigned to each shift i must be at least the number required:

```
subject to Shift_Needs {i in SHIFTS}:
    sum {j in SCHEDS: i in SHIFT_LIST[j]} Work[j]
    >= required[i];
```

On the left we take the sum of Work[j] over all schedules j such that i is in SHIFT_LIST[j]. This sum represents the total employees who are assigned to schedules that contain shift i , and hence equals the total employees covering shift i .

The awkward description of the constraint in this formulation motivates us to try a columnwise formulation. As in our previous examples, we declare the objective and constraints first, but with the variables left out:

```
minimize Total_Cost;
subject to Shift_Needs {i in SHIFTS}: to_come >= required[i];
```

The coefficients of `Work[j]` appear instead in its `var` declaration. In the objective, it has a coefficient of `rate[j]`. In the constraints, the membership of `SHIFT_LIST[j]` tells us exactly what we need to know: `Work[j]` has a coefficient of 1 in constraint `Shift_Needs[i]` for each `i` in `SHIFT_LIST[j]`, and a coefficient of 0 in the other constraints. This leads us to the following concise declaration:

```
var Work {j in SCHEDS} >= 0,
    obj Total_Cost rate[j],
    coeff {i in SHIFT_LIST[j]} Shift_Needs[i] 1;
```

The full model is shown in Figure 16-4.

As a specific instance of this model, imagine that you have three shifts a day on Monday through Friday, and two shifts on Saturday. Each day you need 100, 78 and 52 employees on the first, second and third shifts, respectively. To keep things simple, suppose that the cost per person is the same regardless of schedule, so that you may just minimize the total number of employees by setting `rate[j]` to 1 for all `j`.

As for the schedules, a reasonable scheduling rule might be that each employee works five shifts in a week, but never more than one shift in any 24-hour period. Part of the data file is shown in Figure 16-5; we don't show the whole file, because there are 126 schedules that satisfy the rule! The resulting 126-variable linear program is not hard to solve, however:

```
ampl: model sched.mod; data sched.dat; solve;
MINOS 5.5: optimal solution found.
19 iterations, objective 265.6

ampl: option display_eps .000001;
ampl: option omit_zero_rows 1;
ampl: option display_lcol 0, display_width 60;

ampl: display Work;
Work [*] :=
  10 28.8    30 14.4    71 35.6    106 23.2    123 35.6
  18 7.6     35 6.8     73 28      109 14.4
  24 6.8     66 35.6    87 14.4    113 14.4
;
```

As you can see, this optimal solution makes use of 13 of the schedules, some in fractional amounts. (There exist many other optimal solutions to this problem, so the results you get may differ.) If you round each fraction in this solution up to the next highest value, you get a pretty good feasible solution using 271 employees. To determine whether this is the best whole-number solution, however, it is necessary to use integer programming techniques, which are the subject of Chapter 20.

The convenience of the columnwise formulation in this case follows directly from how we have chosen to represent the data. We imagine that the modeler will be thinking in terms of schedules, and will want to try adding, dropping or modifying different schedules to see what solutions can be obtained. The subsets `SHIFT_LIST[j]` provide a convenient and concise way of maintaining the schedules in the data. Since the data are then organized by schedules, and there is also a variable for each schedule, it proves to be

```

set SHIFTS;                # shifts
param Nsched;              # number of schedules;
set SCHEDS = 1..Nsched;    # set of schedules
set SHIFT_LIST {SCHEDS} within SHIFTS;
param rate {SCHEDS} >= 0;
param required {SHIFTS} >= 0;
minimize Total_Cost;
subject to Shift_Needs {i in SHIFTS}: to_come >= required[i];
var Work {j in SCHEDS} >= 0,
    obj Total_Cost rate[j],
    coeff {i in SHIFT_LIST[j]} Shift_Needs[i] 1;

```

Figure 16-4: Columnwise scheduling model (sched.mod).

```

set SHIFTS := Mon1 Tue1 Wed1 Thu1 Fri1 Sat1
            Mon2 Tue2 Wed2 Thu2 Fri2 Sat2
            Mon3 Tue3 Wed3 Thu3 Fri3 ;

param Nsched := 126 ;

set SHIFT_LIST[ 1] := Mon1 Tue1 Wed1 Thu1 Fri1 ;
set SHIFT_LIST[ 2] := Mon1 Tue1 Wed1 Thu1 Fri2 ;
set SHIFT_LIST[ 3] := Mon1 Tue1 Wed1 Thu1 Fri3 ;
set SHIFT_LIST[ 4] := Mon1 Tue1 Wed1 Thu1 Sat1 ;
set SHIFT_LIST[ 5] := Mon1 Tue1 Wed1 Thu1 Sat2 ;

(117 lines omitted)

set SHIFT_LIST[123] := Tue1 Wed1 Thu1 Fri2 Sat2 ;
set SHIFT_LIST[124] := Tue1 Wed1 Thu2 Fri2 Sat2 ;
set SHIFT_LIST[125] := Tue1 Wed2 Thu2 Fri2 Sat2 ;
set SHIFT_LIST[126] := Tue2 Wed2 Thu2 Fri2 Sat2 ;

param rate default 1 ;

param required := Mon1 100 Mon2 78 Mon3 52
                  Tue1 100 Tue2 78 Tue3 52
                  Wed1 100 Wed2 78 Wed3 52
                  Thu1 100 Thu2 78 Thu3 52
                  Fri1 100 Fri2 78 Fri3 52
                  Sat1 100 Sat2 78 ;

```

Figure 16-5: Partial data for scheduling model (sched.dat).

simpler ~~to~~ and for larger examples, more efficient ~~to~~ specify the coefficients by variable.

Models of this kind are used for a variety of scheduling problems. As a convenience, the keyword `cover` may be used (in the manner of `from` and `to` for networks) to specify a coefficient of 1:

```

var Work {j in SCHEDS} >= 0,
    obj Total_Cost rate[j],
    cover {i in SHIFT_LIST[j]} Shift_Needs[i];

```

Some of the best known and largest examples are in airline crew scheduling, where the variables may represent the assignment of crews rather than individuals, the shifts become flights, and the requirement is one crew for each flight. We then have what is known as a set covering problem, in which the objective is to most economically cover the set of all flights with subsets representing crew schedules.

16.3 Rules for columnwise formulations

The algebraic description of an AMPL constraint can be written in any of the following ways:

```

arith-expr <= arith-expr
arith-expr = arith-expr
arith-expr >= arith-expr

const-expr <= arith-expr <= const-expr
const-expr >= arith-expr >= const-expr

```

Each *const-expr* must be an arithmetic expression not containing variables, while an *arith-expr* may be any valid arithmetic expression though it must be linear in the variables (Section 8.2) if the result is to be a linear program. To permit a columnwise formulation, one of the *arith-exprs* may be given as:

```

to_come
to_come + arith-expr
arith-expr + to_come

```

Most often a ~~template~~ constraint of this kind consists, as in our examples, of *to_come*, a relational operator and a *const-expr*; the constraint's linear terms are all provided in subsequent *var* declarations, and *to_come* shows where they should go. If the template constraint does contain variables, they must be from previous *var* declarations, and the model becomes a sort of hybrid between row-wise and columnwise forms.

The expression for an objective function may also incorporate *to_come* in one of the ways shown above. If the objective is a sum of linear terms specified entirely by subsequent *var* declarations, as in our examples, the expression for the objective is just *to_come* and may be omitted.

In a *var* declaration, constraint coefficients may be specified by one or more phrases consisting of the keyword *coeff*, an optional indexing expression, a constraint name, and an *arith-expr*. If an indexing expression is present, a coefficient is generated for each member of the indexing set; otherwise, one coefficient is generated. The indexing expression may also take the special form *{if logical-expr}* as seen in Section 8.4 or 15.3, in which case a coefficient is generated only if the *logical-expr* evaluates to true. Our simple examples have required just one *coeff* phrase in each *var* declaration, but

```

set CITIES;
set LINKS within (CITIES cross CITIES);
set PRODS;

param supply {CITIES,PRODS} >= 0; # amounts available at cities
param demand {CITIES,PRODS} >= 0; # amounts required at cities

    check {p in PRODS}:
        sum {i in CITIES} supply[i,p] = sum {j in CITIES} demand[j,p];

param cost {LINKS,PRODS} >= 0;      # shipment costs/1000 packages
param capacity {LINKS,PRODS} >= 0; # max packages shipped
param cap_joint {LINKS} >= 0;      # max total packages shipped/link

minimize Total_Cost;

node Balance {k in CITIES, p in PRODS}:
    net_in = demand[k,p] - supply[k,p];

subject to Multi {(i,j) in LINKS}:
    to_come <= cap_joint[i,j];

arc Ship {(i,j) in LINKS, p in PRODS} >= 0, <= capacity[i,j,p],
    from Balance[i,p], to Balance[j,p],
    coeff Multi[i,j] 1.0,
    obj Total_Cost cost[i,j,p];

```

Figure 16-6: Columnwise formulation of Figure 15-13 (netmcol.mod).

in general a separate `coeff` phrase is needed for each different indexed collection of constraints in which a variable appears.

Objective function coefficients may be specified in the same way, except that the keyword `obj` is used instead of `coeff`.

The `obj` phrase in a `var` declaration is the same as the `obj` phrase used in `arc` declarations for networks (Section 15.4). The constraint coefficients for the network variables defined by an `arc` declaration are normally given in `from` and `to` phrases, but `coeff` phrases may be present as well; they can be useful if you want to give a columnwise description of ~~the~~ constraints that apply in addition to the balance-of-flow constraints. As an example, Figure 16-6 shows how a `coeff` phrase can be used to rewrite the multicommodity flow model of Figure 15-13 in an entirely columnwise manner.

Bibliography

Gerald Kahan, ~~Walking~~ Through a Columnar Approach to Linear Programming of a Business. ~~1/2~~ Interfaces **12**, 3 (1982) pp. 328-39. A brief for the columnwise approach to linear programming, with a small example.

Exercises

- 16-1.** (a) Construct a columnwise formulation of the diet model of Figure 2-1.
 (b) Construct a columnwise formulation of the diet model of Figure 5-1. Since there are two separate collections of constraints in this diet model, you will need two `coeff` phrases in the `var` declaration.
- 16-2.** Expand the columnwise production model of Figure 16-3 to incorporate variables `Buy[i]` that represent purchases of raw materials.
- 16-3.** Formulate a multiperiod version of the model of Figure 16-3, using the approach introduced in Section 4.2: first replicate the model over a set of weeks, then introduce inventory variables to tie the weeks together. Use only columnwise declarations for all of the variables, including those for inventory.
- 16-4.** The roll trim or cutting stock problem has much in common with the scheduling problem described in this chapter. Review the description of the roll trim problem in Exercise 2-6, and use it to answer the following questions.
 (a) What is the relationship between the available cutting patterns in the roll trim problem, and the coefficients of the variables in the linear programming formulation?
 (b) Formulate an AMPL model for the roll trim problem that uses only columnwise declarations of the variables.
 (c) Solve the roll trim problem for the data given in Exercise 2-6. As a test of your formulation, show that it gives the same optimal value as a row-wise formulation of the model.
- 16-5.** The set covering problem, mentioned at the end of Section 16.2, can be stated in a general way as follows. You are given a set S , and certain subsets T_1, T_2, \dots, T_n of S ; a cost is associated with each of the subsets. A selection of some of the subsets T_j is said to *cover* S if every member of S is also a member of at least one of the selected subsets. For example, if
- $$S = \{1, 2, 3, 4\}$$
- and
- $$T_1 = \{1, 2, 4\} \quad T_2 = \{2, 3\} \quad T_3 = \{1\} \quad T_4 = \{3, 4\} \quad T_5 = \{1, 3\}$$
- the selections (T_1, T_2) and (T_2, T_4, T_5) cover S , but the selection (T_3, T_4, T_5) does not. The goal of the set covering problem is to find the least costly selection of subsets that covers S . Formulate a columnwise linear program that solves this problem for any given set and subsets.