June 5, 2003

DESIGN PRINCIPLES AND NEW DEVELOPMENTS IN THE AMPL MODELING LANGUAGE

Robert Fourer
Department of Industrial Engineering and Management Sciences
Northwestern University
Evanston, IL, USA
4er@iems.northwestern.edu

David M. Gay
AMPL Optimization LLC
New Providence, NJ, USA
dmg@ampl.com

Brian W. Kernighan
Department of Computer Science
Princeton University
Princeton, NJ, USA
bwk@cs.princeton.edu

Abstract The design of the AMPL modeling language stresses naturalness of expressions, generality of iterating over sets, separation of model and data, ease of data manipulation, and automatic updating of derived values when fundamental values change. We show how such principles have guided the addition of database access, complementarity modeling, and other language features.

This paper will appear as chapter 7 of *Modeling Languages in Mathematical Optimization*, edited by Josef Kallrath, Kluwer Academic Publishers, 2003.

Keywords: AMPL, modeling languages, optimization, mathematical programming, model-data separation, sets, iteration, set manipulation, automatic updates, database access, complementarity, sparsity, nonlinearity

Introduction

AMPL is a little language that grew. Originally designed to express linear programming problems, it gradually expanded to encompass nonlinear programming problems — possibly with complementarity constraints and some integer variables — and it acquired a command environment for manipulating such problems. Although AMPL's presolve phase occasionally determines the solution to a problem, AMPL was never meant to solve problems itself; rather, it works with separate solvers that may even run on different machines.

The rest of this chapter provides more detail on AMPL's design and capabilities. AMPL's historical background has strongly affected its design, so section 1 gives more detail about AMPL's history. Section 2 presents a simple example, a variant of the venerable diet problem, to illustrate some aspects of AMPL's design. One of AMPL's strengths lies in the generality of its indexing and set expressions; section 3 demonstrates some of this by discussing an example of airline fleet assignment that uses sets of quadruples and slices.

In the decade between the appearances of the first and second editions of the AMPL book [12], we extended AMPL in various ways. Three further sections summarize these extensions, including some that are still underway. Section 4 deals with iterative schemes and other flow-of-control issues; section 5 considers new kinds of models — complementarity, combinatorial, and stochastic; and section 6 discusses communications with other systems — databases, Internet services, and solvers. Section 7 summarizes differences between the first and second AMPL book editions, and a brief conclusion appears in section 8.

1 Background and Early History

Several threads converged in the initial design of AMPL. In the early 1980s little languages were a topic of interest in the Computing Science Research Center of Bell Laboratories. These are special-purpose languages designed to simplify and facilitate computing in well focused application areas. Two nice examples are `grap` [3], a `troff` preprocessor that makes it easy to typeset various kinds of graphs, and `chem` [2], a `troff` preprocessor for typesetting chemical formulae.

Another aspect of AMPL's background was the hoopla surrounding Narendra Karmarkar's announcement of an efficient polynomial-time linear programming algorithm [20]. It was clear to us in the Computing Science Research Center that one needs more than just a powerful algorithm; one also needs a problem-formulation language that is natural for use by people yet suitable for processing by a computer, and one needs a convenient way to provide the relevant data and data structures to solvers. We believed that a mathematically natural language would help to make the audience for optimization technology much broader than it would otherwise be.

Bob Fourer and David Gay had first met when both worked at the National Bureau of Economic Research's Computer Research Center in Cambridge, Massachusetts, where Fourer worked on documentation for `sesame`, an LP solver that William Orchard-Hayes was developing for NBER. Fourer subsequently attended graduate school at Stanford, then joined the faculty at Northwestern University. Among other things, he wrote about the advantages of modeling languages over matrix generators [8]. Meanwhile Gay had moved to Bell Labs. When Gay and Fourer happened to meet at a conference, Fourer mentioned he would be coming up for sabbatical. In that way it developed that he was invited to visit Bell Labs for the 1985-86 academic year, with the expectation that the three of us (Fourer, Gay, Kernighan) would work on a modeling language for linear programming.

Our initial work led to a report [10] about the first version of AMPL. Models acceptable to that version would still work today, but commands would not: the first implementation did not recognize any commands, not even "`solve`". It simply read a model and subsequent data and wrote out a translated problem in a simple text format (that we no longer use). Much of the current data-specification syntax was provided by a separate preprocessor. The initial AMPL report eventually became our 1990 AMPL paper [11] in *Management Science*.

Many improvements ensued, as the following sections will show.

2 The McDonald's Diet Problem

As an introduction, imagine a student who must decide what foods to buy at a certain popular fast-food establishment so as to minimize cost while meeting some nutritional requirements. For concreteness, suppose the 9 foods and 7 nutrients shown in Table 1.1 are relevant. Suppose further that the food costs, nutrients, and nutrient requirements are as given in Table 1.2 (derived from the establishment's literature).

Table 1.1. McDonald's Diet Problem foods and nutrients.

Foods:		*Nutrients:*	
QP	Quarter Pounder	Prot	Protein
FR	Fries, small	Iron	Iron
MD	McLean Deluxe	VitA	Vitamin A
SM	Sausage McMuffin	Cals	Calories
BM	Big Mac	VitC	Vitamin C
1M	1% Lowfat Milk	Carb	Carbohydrates
FF	Filet-O-Fish	Calc	Calcium
OJ	Orange Juice		
MC	McGrilled Chicken		

Table 1.2. McDonald's Diet Problem data.

	QP	MD	BM	FF	MC	FR	SM	1M	OJ	
Cost	1.84	2.19	1.84	1.44	2.29	0.77	1.29	0.60	0.72	Need:
Protein	28	24	25	14	31	3	15	9	1	55
Vitamin A	15	15	6	2	8	0	4	10	2	100
Vitamin C	6	10	2	0	15	15	0	4	120	100
Calcium	30	20	25	15	15	0	20	30	2	100
Iron	20	20	20	10	8	2	15	0	2	100
Calories	510	370	500	370	400	220	345	110	80	2000
Carbo	34	35	42	28	42	26	27	12	20	350

Table 1.3. Concrete McDonald's Diet Problem.

Minimize

$$\begin{aligned} &1.84x_{QP} + 2.19x_{MD} + 1.84x_{BM} + 1.44x_{FF} \\ &\quad + 2.29x_{MC} + 0.77x_{FR} + 1.29x_{SM} + 0.60x_{1M} + 0.72x_{OJ} \end{aligned}$$

Subject to

$$\begin{aligned}
28x_{QP} &+ 24x_{MD} + 25x_{BM} + 14x_{FF} \\
&+ 31x_{MC} + 3x_{FR} + 15x_{SM} + 9x_{1M} + x_{OJ} \geq 55 \\
15x_{QP} &+ 15x_{MD} + 6x_{BM} + 2x_{FF} \\
&+ 8x_{MC} + 4x_{SM} + 10x_{1M} + 2x_{OJ} \geq 100 \\
6x_{QP} &+ 10x_{MD} + 2x_{BM} \\
&+ 15x_{MC} + 15x_{FR} + 4x_{1M} + 120x_{OJ} \geq 100 \\
30x_{QP} &+ 20x_{MD} + 25x_{BM} + 15x_{FF} \\
&+ 15x_{MC} + 20x_{SM} + 30x_{1M} + 2x_{OJ} \geq 100 \\
20x_{QP} &+ 20x_{MD} + 20x_{BM} + 10x_{FF} \\
&+ 8x_{MC} + 2x_{FR} + 15x_{SM} + 2x_{OJ} \geq 100 \\
510x_{QP} &+ 370x_{MD} + 500x_{BM} + 370x_{FF} \\
&+ 400x_{MC} + 220x_{FR} + 345x_{SM} + 110x_{1M} + 80x_{OJ} \geq 2000 \\
34x_{QP} &+ 35x_{MD} + 42x_{BM} + 38x_{FF} \\
&+ 42x_{MC} + 26x_{FR} + 27x_{SM} + 12x_{1M} + 20x_{OJ} \geq 350
\end{aligned}$$

Since the expressions for total food cost and resulting nutrients are linear, this problem has the form of a general linear programming problem,

$$\begin{array}{ll} \text{Minimize} & c^T x \\ \text{Subject to} & Ax = b \\ & x \geq 0 \end{array}$$

but such a formulation is too general for convenient manipulation. The problem also has the concrete form shown in Table 1.3, but this form is too specific —

Table 1.4. Abstract model for the McDonald's Diet Problem.

Given
$\mathcal{F}$, a set of foods,
$\mathcal{N}$, a set of nutrients,

$a_{ij} \geq 0$, the units of nutrient i in one serving of food j,
for each $i \in \mathcal{N}$ and $j \in \mathcal{F}$,

$b_i \geq 0$, the units of nutrient i required in the diet, for each $i \in \mathcal{N}$,

$c_j \geq 0$, the cost per serving of food j, for each $j \in \mathcal{F}$,

Define $x_j \geq 0$ to be the number of servings of food j to be purchased,
for each $j \in \mathcal{F}$.

Minimize $\sum_{j \in \mathcal{F}} c_j x_j$,

Subject to $\sum_{j \in \mathcal{F}} a_{ij} x_j \geq b_i$ for each $i \in \mathcal{N}$.

Table 1.5. Diet Model in AMPL (`mcdiet.mod`).

```
set NUTR;     # nutrients
set FOOD;     # foods

param amt {NUTR,FOOD} >= 0;     # amount of nutrient in each food
param nutrLow {NUTR} >= 0;      # lower bound on nutrients in diet
param cost {FOOD} >= 0;         # cost of foods

var Buy {FOOD} >= 0 integer;    # amounts of foods to be purchased

minimize TotalCost: sum {j in FOOD} cost[j] * Buy[j];

subject to Need {i in NUTR}:
      sum {j in FOOD} amt[i,j] * Buy[j] >= nutrLow[i];
```

too hard to set up and maintain.

Between these extremes lies the general mathematical model for the diet problem shown in Table 1.4. AMPL was designed to permit easy transcription of mathematical models in such a form. An AMPL model for the diet problem is shown in Table 1.5. This model represents a whole class of diet problems; to obtain the particular instance corresponding to Tables 1.2 and 1.3, we must supply the relevant data. There are various ways to do this, but the simplest for small examples is to provide a file of AMPL data statements, such as the one in Table 1.6. With files `mcdiet.mod` and `mcdiet1.dat` containing the text shown in Tables 1.5 and 1.6, we obtain a continuous-variable solution to the problem if we use AMPL's default solver, MINOS, which ignores integrality restrictions on variables. This is shown in Table 1.7.

Table 1.6. AMPL data statements for McDonald's Diet Problem (`mcdiet1.dat`).

```
data;
param: FOOD:            cost :=
  "Quarter Pounder"    1.84        "Fries, small"        .77
  "McLean Deluxe"      2.19        "Sausage McMuffin"   1.29
  "Big Mac"            1.84        "1% Lowfat Milk"      .60
  "Filet-o-Fish"       1.44        "Orange Juice"        .72
  "McGrilled Chicken"  2.29 ;

param: NUTR: nutrLow :=
   Prot  55   VitA 100   VitC  100
   Calc 100   Iron 100   Cals 2000   Carb 350 ;

param amt (tr):          Cals  Carb  Prot  VitA  VitC  Calc  Iron :=
   "Quarter Pounder"      510    34    28    15     6    30    20
   "McLean Deluxe"        370    35    24    15    10    20    20
   "Big Mac"              500    42    25     6     2    25    20
   "Filet-o-Fish"         370    38    14     2     0    15    10
   "McGrilled Chicken"    400    42    31     8    15    15     8
   "Fries, small"         220    26     3     0    15     0     2
   "Sausage McMuffin"     345    27    15     4     0    20    15
   "1% Lowfat Milk"       110    12     9    10     4    30     0
   "Orange Juice"          80    20     1     2   120     2     2;
```

Table 1.7. Continuous-variable solution of McDonald's Diet Problem.

```
ampl: model mcdiet.mod;
ampl: data mcdiet1.dat;
ampl: solve;

MINOS 5.5: ignoring integrality of 9 variables
MINOS 5.5: optimal solution found.
7 iterations, objective 14.8557377
ampl: option omit_zero_rows 1;

ampl: display Buy;
Buy [*] :=
   '1% Lowfat Milk'  3.42213
     'Fries, small'  6.14754
  'Quarter Pounder'  4.38525
;
```

Table 1.8. Integer solution of McDonald's Diet Problem.

```
ampl: option solver cplex; solve;
CPLEX 8.0.0: optimal integer solution; objective 15.05
27 MIP simplex iterations
15 branch-and-bound nodes
ampl: display Buy;
Buy [*] :=
   '1% Lowfat Milk'  4
       Filet-o-Fish  1
     'Fries, small'  5
  'Quarter Pounder'  4
;
```

Table 1.9. McDonald's Diet for 63 foods, 12 nutrients.

```
ampl: reset data;
ampl: data mcdiet2.dat;
ampl: solve;
CPLEX 8.0.0: optimal integer solution; objective 0
0 MIP simplex iterations
0 branch-and-bound nodes
ampl: display Buy;
Buy [*] :=
             'Barbeque Sauce'  50
                     Croutons  55
          'Hot Mustard Sauce'  50
;
```

In reality, one cannot order fractional servings, so it is better to use a solver that respects integrality. This is shown in Table 1.8.

Solving with a larger data set is easy. Table 1.9 illustrates this and reveals a weakness in the model: one can satisfy all the nutritional requirements in the bigger data set by using free condiments. A more palatable solution requires additional constraints linking condiment amounts to purchases of related foods.

3 The Airline Fleet Assignment Problem

The McDonald's Diet Problem provides an introduction to many fundamental aspects of AMPL, but it illustrates only a few of the set indexing features that are essential in a good modeling language. AMPL's facilities for constructing and manipulating sets and for iterating over sets make it a powerful language that can readily express complex models. Whereas the diet example involves only simple one-dimensional sets (FOOD and NUTR), our example in this section — the Airline Fleet Assignment Problem — also relies on higher-dimensional sets and on *slices* through these sets, as well as indexed collection of sets. Multi-dimensional sets can be viewed as having members that are pairs, triples, or higher-order "tuples" of elements, in which case slices are subsets in which certain components of the tuples have prescribed values. A model for the Fleet Assignment Problem appears in Tables 1.10 and 1.11.

The model begins with declarations of three simple sets: FLEETS of airplanes, CITIES served by the airplanes, and discrete TIMES at which the airplanes take off or land. The TIMES set is circular, meaning that its members are ordered, with the first member considered to follow the last when computing the "next" member. Set FLEGS describes the schedule to be flown; it consists of 5-tuples (f,c1,t1,c2,t2) with f in FLEETS, c1, c2 in CITIES, and t1, t2 in TIMES, such that fleet f can provide an airplane that departs city c1 at time t1 and arrives in city c2 at time t2. Parameters leg_cost and fleet_size are indexed over these sets.

The model's first four sets are fundamental: their values must be provided before AMPL can generate a problem instance. The next three sets to be declared — LEGS, SERV_CITIES, and OP_TIMES — are all derived from the fundamental sets. LEGS is the set of all quadruples (c1,t1,c2,t2) such that (f,c1,t1,c2,t2) appears in set FLEGS for at least one f. SERV_CITIES is an indexed collection of sets, that is, a collection of similar sets that are distinguished from one another by subscripts (in square brackets): for each f in FLEETS, SERV_CITIES[f] is the set of cities served by fleet f. OP_TIMES is another indexed collection of sets: for each f in FLEETS and c in SERV_CITIES[f], OP_TIMES[f,c] is the ordered set of times at which an airplane from fleet f may take off or land at city c. The phrase circular by TIMES defines each set OP_TIMES[f,c] to be ordered in the same way as the fundamental set TIMES. Both setof expressions in the declaration for OP_TIMES iterate over slices of set FLEGS, the first slice being the set of all triples (t1,c2,t2) such that for a given pair (f,c), (f,c,t1,c2,t2) is in FLEGS, and the second similarly being the set of all triples (c1,t1,t2) such that for a given (f,c), (f,c1,t1,c,t2) is in FLEGS.

AMPL allows one to specify a model in either of two ways:

- "row-wise" — declaring each objective or constraint all at once, through an algebraic expression; or
- "column-wise" — using each variable's declaration to indicate its contributions to various constraints and objectives.

AMPL's node and arc declarations are the most often used column-wise notation: a node declaration sets up a network balance-of-flow constraint to which variables declared with arc can contribute. Since airline fleet assignment has

Table 1.10. Airline Fleet Assignment Model, part 1 — column-wise specification of network flow costs and balance constraints.

```
set FLEETS;
set CITIES;
set TIMES circular;

set FLEGS within
   {f in FLEETS, c1 in CITIES, t1 in TIMES,
                 c2 in CITIES, t2 in TIMES: c1 <> c2 and t1 <> t2};

          # (f,c1,t1,c2,t2) represents the availability of fleet f
          # to cover the leg that leaves c1 at t1 and
          # whose arrival time plus turnaround time at c2 is t2

param leg_cost {FLEGS} >= 0;
param fleet_size {FLEETS} >= 0;
          # leg costs and sizes for each fleet

set LEGS = setof {(f,c1,t1,c2,t2) in FLEGS} (c1,t1,c2,t2);
          # the set of all legs that can be covered by some flight

set SERV_CITIES {f in FLEETS} =
   union {(f,c1,t1,c2,t2) in FLEGS} {c1,c2};

set OP_TIMES {f in FLEETS, c in SERV_CITIES[f]} circular by TIMES =
       setof {(f,c,t1,c2,t2) in FLEGS} t1  union
       setof {(f,c1,t1,c,t2) in FLEGS} t2;

          # for each fleet and city served by that fleet,
          # the set of active arrival & departure times at that city

minimize Total_Cost;

node Balance {f in FLEETS, c in SERV_CITIES[f], OP_TIMES[f,c]};

          # for each fleet and city served by that fleet,
          # a node for each possible time

arc Fly {(f,c1,t1,c2,t2) in FLEGS} >= 0  <= 1
   from Balance[f,c1,t1]  to Balance[f,c2,t2]
   obj Total_Cost leg_cost[f,c1,t1,c2,t2];
          # arcs for fleet/flight assignments

arc Sit {f in FLEETS, c in SERV_CITIES[f], t in OP_TIMES[f,c]} >= 0
   from Balance[f,c,t]  to Balance[f,c,next(t)];
          # arcs for planes on the ground
```

Table 1.11. Airline Fleet Assignment Model, part 2 — row-wise specification of flight coverage and fleet size limitations.

```
subject to Service {(c1,t1,c2,t2) in LEGS}:
   sum {(f,c1,t1,c2,t2) in FLEGS} Fly[f,c1,t1,c2,t2] = 1;
          # each leg must be served by some fleet

subject to Capacity {f in FLEETS}:
        sum {(f,c1,t1,c2,t2) in FLEGS:
               ord(t2,TIMES) < ord(t1,TIMES)} Fly[f,c1,t1,c2,t2]
      + sum {c in SERV_CITIES[f]} Sit[f,c,last(OP_TIMES[f,c])]
   <= fleet_size[f];

          # number of planes used is the number in the air
          # at day's end (arriving "earlier" than they leave)
          # plus the number on the ground in each city at day's end
```

the form of a network flow problem plus "side" constraints, it is convenient to use a mixture of row-wise and column-wise specifications. Table 1.10 shows the objective, `Total_Cost`, and the `Balance` constraints expressed column-wise; this part of the model sets up a separate network flow problem for each fleet. Table 1.11 shows the side constraints expressed row-wise. The `Capacity` constraints restrict the number of airplanes in each fleet network, and the `Service` constraints insure that exactly one fleet is assigned to each flight in the schedule. Slice notations again appear, this time in summations in these constraints.

4 Iterative Schemes

Many applications require solving sequences of problems. AMPL has various facilities that are useful in such applications. The following subsections discuss flow of control expressions and commands, named subproblems, and debugging facilities.

4.1 Flow of Control

AMPL offers two sorts of conditional computations: if-then-else expressions and flow-of-control commands. The former are occasionally useful in declarations, such as the following from a model in our first paper on AMPL [11]:

```
param minv 'minimum inventory' {p in prd, t in time}
   = dem[p,t+1] * (if pro[p,t+1] then pir else rir);
```

Here `pro[p,t+1]` is a logical parameter, that is, a parameter that takes only the value 0 (for false) or 1 (for true).

More generally, it is often convenient to solve sequences of problems or to compute certain sets and parameters using sequences of commands. For this purpose, AMPL provides several flow-of-control commands,

```
if lexpr then command [ else command ] ;
for  [ loopname ] { indexing } command
repeat [ loopname ] [ when lexpr ] { command } [ when lexpr ] ;
break [ loopname ] ;
continue [ loopname ] ;
```

where square brackets enclose optional parts, *lexpr* represents a logical expression, and *when* is either `while` or `until`. Each *command* is either a simple command or a sequence of commands within braces.

Table 1.12 is a simple sensitivity analysis script that uses a `for` loop to solve a sequence of diet problems, each with a larger limit on the amount of sodium allowed, until the constraint on sodium consumed is no longer binding. Table 1.13 shows output from running this script.

Table 1.12. Simple script for sensitivity analysis.

```
model diet.mod;
data diet2a.dat;

set NALOG default {};  # starts out empty
param NAobj {NALOG};
param NAdual {NALOG};

for {theta in 52000 .. 70000 by 1000} {
   let n_max["NA"] := theta;
   solve;
   let NALOG := NALOG union {theta};
   let NAobj[theta] := Total_Cost;
   let NAdual[theta] := Diet["NA"].dual;
   if Diet["NA"].dual > -1e-6 then break;
}
```

4.2 Named Subproblems

AMPL has commands `drop` to temporarily ignore specified constraints and objectives and `restore` to honor them again; and analogously `fix` to freeze specified variables at their current values and `unfix` to let them vary again. These commands are sometimes useful in solving sequences of related problems. But where it is desirable to switch between solving substantially different problems, usually it is clearer to give names to the problems to be solved. This

Table 1.13. Output from the sensitivity analysis script.

```
ampl: option solver_msg 0;
ampl: commands diet.run;
ampl: display NAobj, NAdual;
:        NAobj       NAdual    :=
52000   113.428   -0.0021977
53000   111.23    -0.0021977
54000   109.42    -0.00178981
55000   107.63    -0.00178981
56000   105.84    -0.00178981
57000   104.05    -0.00178981
58000   102.26    -0.00178981
59000   101.082   -0.000155229
60000   101.013   -5.27818e-19
;
```

is done via an AMPL `problem` declaration that lists the variables to be varied, the constraints to be enforced, and the objectives to be considered; these lists may involve iteration over sets.

As an example, we show how a cutting-stock problem would be solved in AMPL via Gilmore-Gomory column generation. Table 1.14 presents a model for optimizing over a given list of patterns, and Table 1.15 exhibits a model for generating a new pattern given dual prices on the desired widths. Associated `problem` declarations and commands to construct initial data appear in Table

Table 1.14. Model for cutting optimization with given patterns.

```
param roll_width > 0;         # width of raw rolls
set WIDTHS ordered;           # set of widths to be cut
param orders {WIDTHS} > 0;    # number of each width to be cut

param nPAT integer >= 0;      # number of patterns
set PATTERNS = 1..nPAT;       # set of patterns
param nbr {WIDTHS,PATTERNS} integer >= 0;

var Cut {PATTERNS} integer >= 0;   # rolls cut using each pattern

minimize Number:                   # minimize total raw rolls cut
   sum {j in PATTERNS} Cut[j];

subject to Fill {i in WIDTHS}:
   sum {j in PATTERNS} nbr[i,j] * Cut[j] >= orders[i];
```

Table 1.15. Model for new pattern generation.

```
param price {WIDTHS} default 0.0;
var Use {WIDTHS} integer >= 0;

minimize Reduced_Cost:
   1 - sum {i in WIDTHS} price[i] * Use[i];

subject to Width_Limit:
   sum {i in WIDTHS} i * Use[i] <= roll_width;
```

Table 1.16. Script for cutting-stock problem declarations and initialization.

```
option solver cplex;           # need an integer solver
option display_transpose -10;  # for nicer formatting
problem Cutting_Opt: Cut, Number, Fill;
option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost, Width_Limit;
option relax_integrality 0;

let nPAT := card(WIDTHS);
let {w in WIDTHS} nbr[w,ord(w)] := floor(roll_width/w);
let {w in WIDTHS, i in WIDTHS: i <> w} nbr[w,ord(i)] := 0;
```

Table 1.17. Script switching problems for cutting-stock optimization.

```
repeat {
   solve Cutting_Opt;
   let {i in WIDTHS} price[i] := Fill[i].dual;

   solve Pattern_Gen;
   if Reduced_Cost < -0.00001 then {
      let nPAT := nPAT + 1;
      let {i in WIDTHS} nbr[i,nPAT] := Use[i];
   }
   else break;
};
display nbr, Cut;
option Cutting_Opt.relax_integrality 0;
solve Cutting_Opt;
```

1.16, and Table 1.17 shows a script that uses the named problems to carry out the column-generating scheme.

4.3 Debugging

AMPL has a single-step mode that provides for stopping and examining data (perhaps by a `display` command) in the middle of a script. Relevant commands include `step` *n* to execute the next *n* commands (or just the next command if *n* is omitted), `next` *n* to execute the next *n* commands with compound commands (such as loops and if-then-else commands) counting as a single command, `skip` *n* to skip the next *n* commands, and `cont` to continue until the current compound command is finished. Table 1.18 illustrates some of these commands.

Among the output commands useful for debugging is `expand`, which shows portions of the problem instance that AMPL has generated: either the linear parts of specified constraints and objectives, or the linear contributions of specified variables to the constraints and objectives. Table 1.19 demonstrates this command.

Table 1.18. Using single-step mode with the cutting-stock script.

```
ampl: model cut.mod; data cut.dat;
ampl: option single_step 1;
ampl: commands cut.run;
cut.run:1(0) option ...
<2>ampl: step 6;
cut.run:12(337) repeat ...
<2>ampl: display nbr;
nbr [*,*]
:  1 2 3 4 5 :=
20 5 0 0 0 0
45 0 2 0 0 0
50 0 0 2 0 0
55 0 0 0 2 0
75 0 0 0 0 1
;
<2>ampl: step
cut.run:13(349) solve ...
<2>ampl: cont
```

5 Other Types of Models

AMPL has long dealt with linear, piecewise-linear, and nonlinear models. Linear models are the most frequently used, but nonlinear models are not uncommon and are easily expressed by means of nonlinear algebraic expressions.

Table 1.19. Expanding the cutting-stock constraints.

```
ampl: display nbr;
nbr [*,*]
:  1 2 3 4 5 6 7 8 :=
20 5 0 0 0 0 1 1 3
45 0 2 0 0 0 2 0 0
50 0 0 2 0 0 0 0 1
55 0 0 0 2 0 0 0 0
75 0 0 0 0 1 0 1 0
;
ampl: expand Fill;
subject to Fill[20]:
        5*Cut[1] + Cut[6] + Cut[7] + 3*Cut[8] >= 48;

subject to Fill[45]:
        2*Cut[2] + 2*Cut[6] >= 35;

subject to Fill[50]:
        2*Cut[3] + Cut[8] >= 24;

subject to Fill[55]:
        2*Cut[4] >= 10;

subject to Fill[75]:
        Cut[5] + Cut[7] >= 8;
```

Piecewise-linear terms are sometimes of interest, and AMPL offers special syntax for them. More recently, with Michael Ferris, we extended AMPL to handle complementarity constraints [7]. Lately we have been working to extend AMPL to handle some more general combinatorial [9] and stochastic programming problems. This section discusses some of these extensions.

5.1 Piecewise-Linear Terms

Piecewise-linear terms arise naturally in economic contexts, such as with graduated income taxes. They also may be useful as approximations for nonlinear terms; piecewise-linear approximations may permit using a linear rather than a nonlinear solver. AMPL offers the syntax

<< *bkpoint-list* ; *slope-list* >> *variable*

to indicate that a piecewise-linear term "multiplies" a variable. The term is zero when the variable is zero and has slopes given in the *slope-list* when the variable has a value between corresponding breakpoints in the *bkpoint-list*.

Piecewise-linear terms can be converted to equivalent linear expressions. In simple cases, such as a convex piecewise-linear term in an objective that is to be minimized or on the left-hand side of a $\leq$ constraint, it suffices to replace the term with suitable linear inequality constraints. Otherwise, linearization also involves the use of auxiliary constraints and binary variables, or equivalent Special Ordered Sets of Type 2 [1] that are recognized by many integer programming solvers. AMPL automatically provides the auxiliary constraints and variables if necessary, along with information that enables solvers that handle SOS2 sets to ignore these extra constraints and variables.

In general, AMPL permits entities in the lists of breakpoints and slopes to be iterated. For instance, Kallrath has posed a problem [19] with a nonlinear objective that could be expressed by the AMPL fragment shown in Table 1.20. He proposes approximating the nonlinear objective by a piecewise-linear one, to permit using an integer programming solver. Table 1.21 shows how AMPL would express the piecewise-linear objective, with a number `Nbkpts[r]` of breakpoints that depends on the reactor `r`. (Auxiliary linear constraints insure that `Vol[r]` is zero whenever binary variable `Active[r]` is zero, and that `Vol[r]` lies between the first and last breakpoints otherwise.)

Table 1.20. Nonlinear objective in Kallrath's problem.

```
set Reactors;
param costFix {Reactors};
param costInv {Reactors};

var Active {Reactors} binary;
var Vol {Reactors} >= 0;

minimize TotalCosts:
   sum {r in Reactors}
      (sqrt(costInv[r]*Vol[r]) + costFix[r]) * Active[r];
```

5.2 Complementarity Problems

A complementarity constraint specifies that a given pair of inequalities must be satisfied, at least one of them with equality. Among the applications that can be stated as collections of complementarity constraints — so-called complementarity problems — are equilibrium problems in economics and engineering, mechanical contact problems, and optimality conditions for nonlinear programs, bi-level linear programs, and bimatrix games.

As an example, consider the first-order necessary conditions for a smooth nonlinear programming problem involving inequality constraints: either an inequality is slack and its Lagrange multiplier is zero, or the inequality is tight

Table 1.21. Piecewise-linear objective in Kallrath's problem.

```
set Reactors;
param costFix {Reactors};
param costInv {Reactors};

var Active {Reactors} binary;
var Vol {Reactors} >= 0;

param Nbkpts {Reactors} integer > 0;
param Bpoint {r in Reactors, 1 .. Nbkpts[r]};

minimize TotalCosts:
  sum {r in Reactors} (
    << {q in 1 .. Nbkpts[r]-1} Bpoint[r,q] ;
       sqrt(costInv[r]*Bpoint[r,1]) / Bpoint[r,1],
       {q in 1 .. Nbkpts[r]-1}
         (sqrt(costInv[r]*Bpoint[r,q+1]) - sqrt(costInv[r]*Bpoint[r,q]))
           / (Bpoint[r,q+1] - Bpoint[r,q]) >> Vol[r]
  + costFix[r] * Active[r] );

subj to UseLo {r in Reactors}: Active[r] * Bpoint[r,1] <= Vol[r];
subj to UseHi {r in Reactors}: Vol[r] <= Active[r] * Bpoint[r,Nbkpts[r]];
```

and its multiplier must have the correct sign. All conventional (smooth) mathematical programming problems are thus complementarity problems, but the converse is not true: some complementarity problems cannot be expressed as constrained optimization problems.

Table 1.22 shows an AMPL model (from [12]) for a production cost minimization problem stated as an equilibrium model. The keyword `complements` separates each pair of inequalities that are complementary in the sense that at least one must hold with equality.

Every complementarity condition involves exactly two inequalities. In the example of Table 1.22, there is one inequality on either side of the complements operator, and at least one of these inequalities must hold with equality. In a *mixed-complementarity* condition, the `complements` keyword separates an expression from a pair of inequalities, as in

expr1 <= *expr2* <= *expr3* `complements` *expr4*.

This condition is interpreted according to the following rules:

$$\begin{array}{ccc} expr1 = expr2 & \Longleftrightarrow & expr4 \geq 0 \\ expr1 < expr2 < expr3 & \Longleftrightarrow & expr4 = 0 \\ expr2 = expr3 & \Longleftrightarrow & expr4 \leq 0 \end{array}$$

As an example where mixed complementarity conditions arise, Table 1.23 is a variant of Table 1.22 corresponding to a minimization problem with lower and upper bounds on the production-level variables.

Table 1.22. Production cost minimization as an equilibrium model.

```
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 per unit of 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];
```

Table 1.23. Bounded-variable cost minimization as an equilibrium model.

```
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 per unit of 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];
```

Complementarity conditions are mainly of interest when they are part of problems that cannot be stated easily or at all as conventional optimization problems. For example, it may make sense to view the demand for each product as a decreasing function of the product's price, as in Table 1.24.

Some complementarity solvers require a "square" system, with

$$\begin{aligned}\text{\# of variables} = \text{\# of complementarity constraints} +\\ \text{\# of equality constraints}\end{aligned}$$

Table 1.24. Economic equilibrium with price-dependent demands.

```
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 per unit of 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];
```

For the convenience of such solvers, AMPL translates complementarity conditions into a canonical mixed-complementarity form of *constraint* `complements` *variable*. To help catch mistakes, AMPL complains by default if a complementarity problem is not square.

More generally, mathematical programs with equilibrium constraints, often called MPECs, have no restriction on the numbers of variables and constraints and may involve objectives. When working with MPECs, one can specify

```
      option compl_warn 0;
```

to suppress warnings about nonsquare complementarity systems.

5.3 Combinatorial Optimization

Combinatorial optimization problems arise in many contexts and are of considerable practical interest. Some are readily expressed as integer programs, but formulations in other terms are often more convenient. Constructs convenient to general-purpose combinatorial optimization can be added to conventional algebraic modeling languages in a way that lets one exploit conventional facilities, such as iteration over sets, while permitting links to new kinds of solvers.

Solvers incorporating the approach known as constraint programming [22], [21] are of particular interest in this context. They can operate directly on a variety of constructs that can be introduced into algebraic modeling languages, and they employ a tree-search approach (much like branch-and-bound for integer programming) that is optimal at least in principle. The tree search may fail in practice to find an optimum in a reasonable amount of time, but it can often be made practical through a judicious choice of search strategies. (Providing convenient ways to specify the search strategy is thus an important challenge.)

As an example, we contrast an AMPL integer programming formulation with a prospective constraint programming formulation for a problem of job sequencing with setups, simplified from [17]. We are given a set of jobs to be done, with a processing time and due time for each job. Jobs must be completed by their due times, and preferably as close to their due times as possible; hence there is also an early completion penalty for each job. A single machine carries out all jobs, one at a time, and readying the machine for a particular job involves a specified setup cost and time. The problem is to sequence the jobs so as to minimize the total setup costs plus earliness penalties.

In either an integer programming or a constraint programming formulation of this problem, it makes sense to define a variable `ComplTime[j]` for the completion time of job `j`. Since the constraints will prevent any job from finishing late, the total earliness penalty can be written as

```
sum {j in Jobs} duePen[j] * (dueTime[j] - ComplTime[j])
```

A typical integer programming formulation would introduce, for each pair of jobs `i` and `j`, a binary (zero-one) variable `Seq[i,j]` that is to equal one if and only if job `i` immediately precedes job `j`. The setup cost would then be

```
sum {i in Jobs, j in Jobs} setupCost[i,j] * Seq[i,j]
```

A more natural expression for the setup cost would introduce an array `JobForSlot` of job-valued variables, with `JobForSlot[k]` to equal the `k`th job handled by the machine. This kind of formulation, with fewer variables each having a larger domain, is particularly amenable to constraint programming techniques. It permits the setup cost to be written

```
sum {k in Slots} setupCost[JobForSlot[k],JobForSlot[k+1]]
```

with a special `JobForSlot[0]` initialized to represent the initial state of the machine. This expression's use of variables as "subscripts" to the `setupCost` parameter is another natural formulation feature that would not be accepted by an integer programming solver but that is handled directly by solvers for constraint programming.

In the model's constraints, an integer programming formulation would require that completion times respect the due times, and that completion times be consistent with the setup and processing times:

```
For each job i,
   ComplTime[i] <= dueTime[i]
For each job pair (i,j),
   ComplTime[i] + setupTime[i,j] + procTime[j] <=
   ComplTime[j] + BIG * (1 - Seq[i,j])
```

The use of the parameter `BIG` is a standard integer programming “trick” for insuring that only the sequencing constraints on adjacent jobs are significant. What we really want to say here is that a job’s completion time is bounded both by its due time and by the completion time of the next job less the appropriate processing and setup times. Since the completion time is bounded by just these two amounts, it can be constrained to equal the lesser of them:

```
For each slot k,
   ComplTime[JobForSlot[k]] = min (
      dueTime[JobForSlot[k]],
      ComplTime[JobForSlot[k+1]]
        - procTime[JobForSlot[k+1]]
        - setupTime[JobForSlot[k],JobForSlot[k+1]] )
```

A constraint programming solver does not attempt to bound the optimum by relaxing integer variables, and so can deal directly with operators such as `min` that are not strictly linear. It is the use of `min` that permits this to be an equality constraint; the equality has the advantage of fully defining the `ComplTime` variables, so that the solver can be instructed to search only over the `JobForSlot` variables.

It remains to constrain the variables to represent a valid sequencing. In the integer programming formulation, this is done through “assignment” constraints that allow only one `Seq[i,j]` to be 1 for each `i` and for each `j`:

```
For each job i,
   sum {j in Jobs} Seq[i,j] = 1
For each job j,
   sum {i in Jobs} Seq[i,j] = 1
```

The same restriction can be expressed more naturally as a single assertion that no job is repeated in the `JobForSlot` array. An AMPL iterated operator can serve to express such an assertion:

```
all_different {k in Slots} JobForSlot[k]
```

A constraint programming solver operates directly on this constraint, using an efficient matching procedure to reveal jobs that can be eliminated from the variables’ domains at nodes of the search tree.

Conventional linear and integer programs are most easily conveyed to solvers, which need to see only a matrix and some vectors. Conventional nonlinear programming problems are more complicated, in that solvers must be able to evaluate nonlinear expressions and their derivatives. AMPL converts conventional linear and nonlinear programs to a standard format, a so-called `.nl` file, which contains linear parts in a sparse-matrix format and expression graphs for nonlinear expressions. AMPL's solver-interface library provides facilities to read the .nl files and to evaluate expressions and their derivatives on request from a solver. First and second derivatives are computed efficiently by backwards automatic differentiation. (See [16] for more on automatic differentiation in general, and [14] and [15] for details on AMPL's use of it.) Source for AMPL's interface library is freely available; see `www.ampl.com/hooking.html` for more details.

When combinatorial constraints are employed as described in this subsection, complications of an additional kind can arise. Some solvers need to be given the `.nl` file's actual expression graphs in particular formats. This can be achieved by means of recursive "tree-walks" of the expression graphs after they have been read from the .nl file; more details appear in [9], with illustrations from the C++ interface to ILOG Solver.

5.4 Stochastic Programming

Many practical planning problems involve uncertainty. Tomorrow's prices, demands, and resources are seldom known precisely, so it is generally necessary to make educated guesses. Sometimes historical data can guide those guesses and even suggest probabilities for them. Other situations may require a more Bayesian approach in which one simply guesses at probabilities for various scenarios.

Stochastic programming deals with uncertainties of several kinds. Assuming probability distributions on some input parameters, one might minimize expected cost or maximize expected profit, perhaps as penalized by a variance term. Sometimes it suffices — or is only feasible — to require that a constraint hold with some probability, for example, that the probability of a particular kind of failure be sufficiently low or that some measure of success be met with sufficiently high probability. There is a large and growing literature on stochastic programming; see for example [18] or [4].

Many kinds of algorithms have been proposed for stochastic programs. In simple cases involving discrete random variables, perhaps in the form of scenarios, one can simply state a "deterministic equivalent", a conventional mathematical programming problem equivalent to the stochastic program. If continuous distributions are involved, one can approximate the problem by sampling, and perhaps again deal with a deterministic equivalent. Unfortunately, it is all too easy to find examples where the deterministic equivalent is too large to solve. Decomposition algorithms, perhaps with dynamic sampling, such as importance sampling, may then appear to be the only reasonable computational alternatives. In some such situations, it is desirable for the solver to be able to sample the relevant distributions as necessary.

We are working on extensions to AMPL that would facilitate dealing with at least some stochastic programming problems. One of our goals is to give suitable solvers the ability to do their own sampling. To this end, we will permit declaration of random parameters. Because of how they would be conveyed to solvers and to accord with the conventional notion of "random variables", these quantities may in fact be declared as variables with the `random` attribute. Just as conventional parameters may either be fundamental or derived, with fundamental ones being supplied values after the model has been stated and derived ones computed from expressions in the model, so random variables may be either fundamental or derived. For example, in the declarations

```
param avail_mean >= 0;
param avail_var >= 0;
var avail {1..T} random = Normal (avail_mean, avail_var);
```

`avail` is a derived random variable whose distribution is determined by the fundamental parameters `avail_mean` and `avail_var`. Fundamental random variables could be assigned distributions by the `let` command, as illustrated in Table 1.25. Both fundamental and derived random variables would be sent to solvers as expression graphs, in much the same way as AMPL's "defined variables".

Table 1.25. Assigning Distributions to Random Variables.

```
set PROD; param T integer > 0;
param mktbas {PROD} >= 0;
param grow_min {PROD} >= 0;
param grow_max {PROD} >= 0;
var Market {PROD,1..T} random;
.......
let {p in PROD} Market[p,1] := mktbas[p];
let {p in PROD, t in 2..T} Market[p,t] :=
     Market[p,t] + Uniform (grow_min[p], grow_max[p]);
```

New kinds of expressions would help in stating some problems. For example, a discrete distribution would be expressed by a sequence of (*probability*, *value*) arguments to a new `Discrete` built-in function, as in

```
Discrete (1/3, 20,  1/3, 50,  1/3, 175)
Discrete ( {s in SCEN} (prob[s],demand[s]) )
```

(In general, function arguments in AMPL can be iterated, as in the latter `Discrete` invocation.) New built-in functions `Expected_value` and `Variance` would permit use of expected values and variances in objectives and constraints, and the new built-in function `Probability (`*logical-expression*`)` would permit stating reliability constraints.

In recourse problems, one makes a decision now — in stage 1 — and makes corrections, i.e., takes recourse, later — in subsequent stages, after new information becomes available. For such problems, a declared suffix `.stage` could be employed to distinguish the different stages of recourse variables; see the discussion of suffixes in section 6.3. Solvers could learn which constraints involve random entities from the sparsity pattern of the Jacobian matrix, which is available in the `.nl` file.

`Discrete`, `Uniform`, and other (half-)bounded distributions will also offer new opportunities for AMPL's presolve phase to simplify the problem after deducing bounds on certain expressions.

6 Communicating with Other Systems

In the following subsections, we consider a variety of ways that AMPL can communicate with other entities, such as databases, compute servers, and solvers.

6.1 Relational Database Access

In many practical applications, it is convenient to maintain data — both input and certain forms of output — in the form of relational tables in database or spreadsheet files. A relatively recent addition to AMPL is its general mechanism for communicating with these external data repositories. A `table` declaration establishes connections between AMPL entities, such as sets, parameters and variables, and their external representations. Subsequent `read table` commands copy data from external repositories to the AMPL session, while `write table` commands copy values back from the AMPL session to the external repositories. The AMPL model itself remains strictly independent of the data. Special *table handlers* communicate with the external representations; an open interface makes it possible for anyone to provide additional table handlers that deal with new kinds of external data representations.

For example, the AMPL book [12] defines a variant of Table 1.5 above in which there are also lower and upper bounds `f_min[j]` and `f_max[j]` on the amount `Buy[j]` of food `j` that is bought, and similarly lower and upper bounds `n_min[i]` and `n_max[i]` on the amounts of nutrient `i` consumed. Table 1.26 is a script that obtains data for `diet.mod` from a Microsoft Access database, solves the problem, and writes some results to another Access table.

As seen in this script, the first part of a table declaration generally provides some quoted strings; these identify the table handler and provide handler-dependent details about the external representation. In the `table dietFoods`

Table 1.26. Reading and writing database tables for the diet problem.

```
model diet.mod;
table dietFoods IN "ODBC" "diet1.mdb" "Foods":
   FOOD <- [foods], cost, f_min, f_max;
table dietNutrs IN "ODBC" "diet1.mdb" "Nutrs":
   NUTR <- [nutrients], n_min, n_max;
table dietAmts IN "ODBC" "diet1.mdb" "Amounts":
   [nutrs, foods] amt;
read table dietFoods;
read table dietNutrs;
read table dietAmts;
solve;
table dietResults OUT "ODBC" "diet1.mdb" "Scen3":
   [foodlist], Buy, Buy.rc ~ BuyRC,
   {j in FOOD} Buy[j]/f_max[j] ~ BuyFrac;
write table dietResults;
```

declaration, for instance, `"ODBC"` identifies AMPL's ODBC table handler, which works on Microsoft systems with various representations, including Access and Excel. The next string, `"diet1.mdb"`, is the name of the database file, and `"Foods"` is the name of a table within it. The syntax `FOOD <- [foods]` causes set `FOOD` to be assigned the set of names in external database column `"foods"`; the AMPL parameters `cost`, `f_min`, and `f_max` are also assigned from database columns having the corresponding names.

Sometimes the external (database) and internal (AMPL) names differ. For instance, the `table dietResults` declaration specifies a new table, with external column names `"foodlist"`, `"Buy"`, `"BuyRC"`, and `"BuyFrac"`, that is written after a solution is determined. As this declaration illustrates, specification of database columns can involve iteration over sets; in fact, columns can also be iterated, and tables can be subscripted. Many more details appear in chapter 10 of [12].

6.2 Internet Optimization Services

Network services for optimization have grown in sophistication along with the Internet and the World Wide Web [13]. One successful experiment of this sort is the NEOS project, which makes various facilities related to mathematical programming freely available over the Internet. The NEOS solver facility offers many solvers for a growing list of problem areas, including those shown in Table 1.27. Many of these solvers accept AMPL input, as can be seen from the detailed listing at the NEOS web site, `www-neos.mcs.anl.gov/neos/`.

The central NEOS Server can accept problem submissions via web forms, e-mail, or a specialized client (the NEOS "submission tool") that provides a

Table 1.27. Problem types accepted by the NEOS Server.

Linear programming
Linear network optimization
Linear integer programming
Nonlinear programming
Nonlinear integer programming
Nondifferentiable & global optimization
Stochastic optimization
Complementarity problems
Semidefinite programming

graphical user interface. The Server has various workstations, in other Internet locations, at its disposal; upon receiving a problem submission, it selects an available workstation and sends it the problem. While the problem is being worked on, NEOS may provide periodic progress reports. Once a problem is solved, NEOS returns the solution or information about it; the details depend on the solver and means of submission.

For submissions via web, e-mail, or submission tool, NEOS provides what is essentially batch processing. This is fine for some purposes, but for applications where one must manipulate the computed solution, a closer connection to AMPL's interactive environment is highly desirable. Such a connection is provided via NEOS's Kestrel client. Tables 1.28 and 1.29 show Kestrel in a typical use. To an AMPL session running locally, Kestrel appears to be a locally installed solver. When a problem is passed to Kestrel, it is not solved locally, however, but is instead sent by the Kestrel routines to the NEOS Server. Eventually the completed solution from the NEOS Server is passed back, and is

Table 1.28. Kestrel example, part 1: sending a problem instance to a remote solver.

```
ampl: model gridneta.mod; data gridneta.dat;
ampl: option solver kestrel, kestrel_options 'solver=knitro';
ampl: option knitro_options 'hessopt=6 feastol=1.0e-5 iprint=2';
ampl: solve;

Job has been submitted to Kestrel
Kestrel/NEOS Job number    : 269646
Kestrel/NEOS Job password  : QAYkBqEW
Check the following URL for progress report :
    http://www-neos.mcs.anl.gov/neos/neos-cgi/
    check-status.cgi?job=269646&pass=QAYkBqEW
In case of problems, e-mail :
    neos-comments@mcs.anl.gov
```

Table 1.29. Kestrel example, part 2: receiving the solver listing and displaying results from the remote solver.

```
Intermediate Solver Output:
Checking the AMPL files
Executing algorithm...

Nondefault Options
------------------
 hessopt     =  6
 feastol     =  1.00E-05

 Iter   Res   Objective    Feas err  Opt err   ||Step||  CG its     mu
------  ---  ------------  --------  --------  --------  ------  --------
     0       6.733333E-01  1.00E+01
     1  Acc  1.050234E+00  4.78E+00  1.05E-01  5.23E+00       2  1.00E-01
     2  Acc  2.512883E+00  1.78E+00  1.21E-01  6.70E+00       2
     3  Acc  5.126285E+00  5.77E-15  1.11E-01  5.87E+00       2
     4  Acc  5.072647E+00  3.11E-15  2.78E-02  7.87E-01       2  2.00E-02
     5  Acc  5.042755E+00  1.78E-15  6.83E-03  8.79E-01       3  2.00E-04
     6  Acc  5.038339E+00  6.66E-16  1.13E-03  3.95E-01       4
     7  Acc  5.038202E+00  4.44E-16  5.03E-04  7.04E-02       6
     8  Acc  5.038191E+00  8.88E-16  2.52E-04  1.31E-02       7
     9  Acc  5.038189E+00  6.66E-16  6.48E-05  7.40E-03       7  4.00E-05
    10  Acc  5.038188E+00  8.88E-16  1.48E-05  2.86E-03       7  4.00E-07
    11  Acc  5.038188E+00  4.44E-16  5.02E-06  6.20E-04       6
    12  Acc  5.038188E+00  1.33E-15  1.84E-06  3.93E-04       5
    13  Acc  5.038188E+00  4.44E-16  7.21E-07  9.79E-05       5

EXIT: OPTIMAL SOLUTION FOUND.
Results completed on remote station.

ampl: display Cost;
Cost = 5.03819

ampl: display {(i,j,0) in A} x[i,j,0];
x[i,j,0] [*,*] (tr)
:      0          1          2          3          4       :=
0   5.01906    3.57129    2.08151    1.48575    0.826271
1   2.0354     1.42124    1.44185    0.891306   0.336723
2   0.663005   1.39555    1.26844    1.06157    0.838679
3   0.761372   0.632623   0.991102   0.966883   0.673966
4   0.250817   0.96421    1.19955    1.41281    1.59119
5   1.27034    2.01509    3.01755    4.18168    5.73317
;
```

accessible through the AMPL command environment as if it had been computed locally. Kestrel's calls to the remote server employ the CORBA protocols, as summarized in [6] and explained more fully in [5].

Kestrel allows for a simple sort of parallel processing. The idea is to use AMPL scripts `kestrelsub` to submit multiple solve requests and `kestrelret` to retrieve the results in the order of submission. The NEOS server distributes the requests to multiple workstations, queuing some of the requests if necessary. The `kestrelret` script does not return control to AMPL until the results of the corresponding `kestrelsubmit` are available. Table 1.30 shows an example from a decomposition script.

Table 1.30. Kestrel parallel processing.

```
for {p in PROD} {
        problem subI[p];
        commands kestrelsub;
}
for {p in PROD} {
        problem subI[p];
        include kestrelret;
        display Artif_Reduced_Cost[p];
        ...
}
```

Table 1.30 also illustrates both AMPL's `commands` command and its `include` facility. The former reads and processes the indicated file, `kestrelsub`, anew during each iteration of the loop. The rereading would be important if `kestrelsub` were updated elsewhere in the loop. Since neither `kestrelsub` nor `kestrelret` is modified by the loop in which it appears, it suffices to read them with AMPL's `include` facility, which simply interpolates text from the indicated file during parsing of the loop, so the file is read only once no matter how many times the loop body is executed.

Kestrel uses some powerful machinery (CORBA and NEOS) to run solvers on remote machines, but simpler approaches are possible where the AMPL user is also the owner of a copy of a solver on a different machine. The key is that AMPL runs solvers as separate processes that can do whatever they want, such as cause the problem to be solved remotely. On Unix and Linux systems, one can easily use a shell script as a solver, and such a solver script can use `ssh` or (when security is not an issue) `rsh` to invoke a remote solver, perhaps after copying the `.nl` file to the user's directory at the remote location.

6.3 Communication with Solvers via Suffixes

AMPL represents auxiliary information associated with model components by appending to a component's name a construction of the form `.`*suffix-name*. For a variable, the auxiliary information includes the variable's lower and upper bounds and, after a solve, its reduced cost. In Table 1.26, for instance, `Buy.rc` denotes the reduced cost of variable `Buy`.

Some suffixes are built-in, including `.sstatus` for exchanging basis information with simplex-based solvers, but AMPL also lets one declare new suffixes, both for arbitrary use in AMPL scripts and for exchanging auxiliary information with suitable solvers. Mixed-integer programming solvers, for example, often use priorities on integer variables to help decide which variable to branch on next. A properly written AMPL interface to such a solver would recognize suffix `.priority` on integer variables and act accordingly. After declaring a suffix, one can assign values to it with a `let` command, as in

```
model multmip1.mod; data multmip1.dat;
suffix priority;
let {i in ORIG, j in DEST} Use[i,j].priority
   := sum {p in PROD} demand[j,p];
```

One can also specify suffix values in a variable's declaration:

```
suffix priority;
var Use {ORIG, j in DEST} binary
   suffix priority sum {p in PROD} demand[j,p];
```

Solvers can return information in suffixes and even declare new suffixes if necessary. For instance, on request, when a problem is infeasible, some solvers return in suffix `.iis` an indication of which variables and constraints belong to an irreducible infeasible subset, a minimal set of mutually inconsistent variable bounds and constraints. An IIS can help pinpoint a cause of infeasibility.

Table 1.31 demonstrates finding an IIS and illustrates some details of suffixes and generic synonyms. In response to the first infeasible `solve`, this solver returns in suffix `.dunbdd` a direction of dual unboundedness. Since `.dunbdd` has not yet been declared as a suffix, AMPL creates and exhibits a `suffix` declaration statement automatically. The following `display` command involves the generic synonyms `_conname` for the names of all constraints and `_con` for the constraints themselves; in this case, `_con.dunbdd` shows the `.dunbdd` values of all the constraints. The output suggests an inconsistency between the bounds on sodium (NA) and vitamin B2.

For further confirmation, a suitable `option` command and second `solve` request an IIS. Following the `solve`, AMPL creates and exhibits another `suffix` declaration, for the suffix `.iis`. Since this is a `symbolic` suffix, the solver has provided symbolic synonyms for the `.iis` values, shown in the `option` `iis_table`

Table 1.31. Retrieval of auxiliary solver information via solver-defined suffixes.

```
ampl: model diet.mod; data diet2.dat;
ampl: option solver cplex; solve;
CPLEX 8.0.0:  infeasible problem.
4 dual simplex iterations (0 in phase I)
constraint.dunbdd returned

suffix dunbdd OUT;
ampl:  display _conname, _con.dunbdd;
:     _conname      _con.dunbdd    :=
1   "Diet['A']"      0
2   "Diet['B1']"     0
3   "Diet['B2']"     0.322951
4   "Diet['C']"      0
5   "Diet['NA']"    -0.00409836
6   "Diet['CAL']"    0
;

ampl: option cplex_options 'iisfind=1'; solve;
CPLEX 8.0.0:  iisfind=1
CPLEX 8.0.0:  infeasible problem.
0 simplex iterations (0 in phase I)
Returning iis of 7 variables and 2 constraints.
constraint.dunbdd returned

suffix iis symbolic OUT;

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

ampl: display _varname, _var.iis, _conname, _con.iis;
:     _varname    _var.iis     _conname    _con.iis    :=
1   "Buy['BEEF']"   non      "Diet['A']"     non
2   "Buy['CHK']"    low      "Diet['B1']"    non
3   "Buy['FISH']"   low      "Diet['B2']"    low
4   "Buy['HAM']"    upp      "Diet['C']"     non
5   "Buy['MCH']"    low      "Diet['NA']"    upp
6   "Buy['MTL']"    upp      "Diet['CAL']"   non
7   "Buy['SPG']"    low            .            .
8   "Buy['TUR']"    low            .            .
;
```

command. (Solvers always deal with numeric suffix values; in the AMPL session, the numeric values are available in the `.iis_num` suffix.) Finally, the `display` output shows that, in this example, the IIS consists of the bounds on several `Buy` variables plus the lower limit on vitamin B2 and the upper limit on sodium. This can be interpreted as indicating that, for instance, if it is essential to keep all `Buy` amounts within their specified bounds, then the infeasibility can be alleviated only by relaxing either the lower limit on B2 or the upper limit on sodium.

More information about declaring and using suffixes appears in chapter 14 of [12].

7 Updated AMPL Book

The first edition of the AMPL book appeared in late 1992, with a 1993 copyright. The second edition [12] appeared a decade later (late 2002, with a 2003 copyright). It describes many extensions made during that decade, with new chapters on modeling commands, database access, command scripts, complementarity problems, display commands, and interactions with solvers. The other chapters and the reference manual in the appendix are extensively revised. More details about the book appear at `www.ampl.com/ampl/BOOK/`.

8 Concluding Remarks

Since its inception in the mid-1980s, AMPL has evolved in many ways, but several goals and principles have guided its evolution from the beginning:

- Stay close to conventional mathematical notation, but use notation that is easy to enter on an ordinary keyboard.
- Use a consistent style in designing notation.
- Permit entities to be iterated whenever this makes sense.
- Make most indexing explicit, to avoid surprise interpretations and to allow complicated relationships to be expressed.
- Encourage separation of model and data.
- Automatically update derived entities as needed after fundamental data values have changed.
- Provide reasonable default values and behaviors and make specifying alternatives easy.
- Use open interfaces where possible, for example to solvers, databases, and user-defined functions.

References

[1] E. M. L. Beale and J. A. Tomlin. Special facilities in a general mathematical system for non-convex problems using ordered sets of variables. In J. Lawrence, editor, *Proceedings of the Fifth International Conference on Operational Research*, pages 447–454. Tavistock Publications, London, 1970.

[2] Jon L. Bentley, Lynn W. Jelinski, and Brian W. Kernighan. CHEM — A language for phototypesetting chemical structure diagrams. *Computers and Chemistry*, 11(4):281–297, 1987.

[3] Jon L. Bentley and Brian W. Kernighan. GRAP — A language for typesetting graphs. *Communications of the ACM*, 29(8):782–792, 1986.

[4] John R. Birge and François Louveaux. *Introduction to Stochastic Programming*. Springer Verlag, 1997.

[5] Elizabeth D. Dolan, Robert Fourer, Jean-Pierre Goux, and Todd S. Munson. Kestrel: An interface from modeling systems to the neos server. Technical report, Argonne National Laboratory, 2002. URL = `http://www-neos.mcs.anl.gov/neos/ftp/kestrel2.pdf`.

[6] Elizabeth D. Dolan and Todd S. Munson. The kestrel interface to the neos server. Technical report, Argonne National Laboratory, 2002. URL = `http://www-neos.mcs.anl.gov/neos/ftp/kestrel.pdf`.

[7] Michael Ferris, Robert Fourer, and David M. Gay. Expressing complementarity problems in an algebraic modeling language and communicating them to solvers. *SIAM J. Optimization*, 9(4):991–1009, 1999.

[8] Robert Fourer. Modeling languages versus matrix generators for linear programming. *ACM Trans. Math. Software*, 9(2):143–183, 1983.

[9] Robert Fourer and David M. Gay. Extending an algebraic modeling language to support constraint programming. *INFORMS J. Computing*, 14(4):322–344, 2002.

[10] Robert Fourer, David M. Gay, and Brian W. Kernighan. AMPL: A mathematical programming language. Technical Report Computing Science Technical Report No. 133, AT&T Bell Laboratories, Murray Hill, NJ, USA, Jan. 1987. revised June 1989.

[11] Robert Fourer, David M. Gay, and Brian W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990.

[12] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press/Brooks/Cole Publishing Co., second edition, 2003.

[13] Robert Fourer and Jean-Pierre Goux. Optimization as an internet resource. *Interfaces*, 31(2):130–150, 2001.

[14] D. M. Gay. Automatic differentiation of nonlinear AMPL models. In A. Griewank and G. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 61–73. SIAM, 1991.

[15] David M. Gay. More AD of nonlinear AMPL models: Computing Hessian information and exploiting partial separability. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, 1996.

[16] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic*. SIAM, 2000.

[17] Carsten Jordan and Andreas Drexl. A comparison of constraint and mixed-integer programming solvers for batch sequencing with sequence-dependent setups. *INFORMS J. Computing*, 7(2):160–165, 1995.

[18] Peter Kall and Stein W. Wallace. *Stochastic Programming*. John Wiley & Sons, Chichester, 1994.

[19] Josef Kallrath. Exact computation of global minima of a nonconvex portfolio optimization problem. In C. A. Floudas and P. M. Pardalos, editors, *Frontiers in Global Optimization*. Kluwer Academic Publishers, 2003.

[20] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[21] Irvin J. Lustig and Jean Francois Puget. Program does not equal program: Constraint programming and its relationship to mathematical programming. *Interfaces*, 31(6):29–53, 2001.

[22] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.