The AMPL Modeling Language — an Aid to Formulating and Solving Optimization Problems

David M. Gay

AMPL Optimization, Inc.
dmg@ampl.com
http://www.ampl.com

ABSTRACT

Optimization problems arise in many contexts. Sometimes finding a good formulation takes considerable effort. A modeling language, such as AMPL, facilitates experimenting with formulations and simplifies using suitable solvers to solve the resulting optimization problems. AMPL lets one use notation close to familiar mathematical notation to state variables, objectives, and constraints and the sets and parameters that may be involved. AMPL does some problem transformations and makes relevant problem information available to solvers. The AMPL command language permits computing and displaying information about problem details and solutions returned by solvers. It also lets one modify problem formulations and solve sequences of problems. AMPL addresses both continuous and discrete optimization problems and offers some constraint-programming facilities for the latter. More generally, AMPL permits stating and solving problems with complementarity constraints. For continuous problems, AMPL makes first and second derivatives available via automatic differentiation. The freely available AMPL/solver interface library (ASL) facilitates interfacing with solvers. This paper gives an overview of AMPL and its interaction with solvers and discusses some problem transformations and implementation techniques. It also looks forward to possible enhancements to AMPL.

This paper is based on a talk presented at the Third International Conference on Numerical Analysis and Optimization, which was held 5–9 January 2014 at Sultan Qaboos University in Muscat, Oman. This was written for possible inclusion in *Recent Developments in Numerical Analysis and Optimization*, to be published by Springer as part of the book series *Springer Proceedings in Mathematics and Statistics* and edited by Mehiddin Al-Baali, Lucio Grandinetti and Anton Purnama.

1. Introduction

Science is all about models and data — theories (models) that explain observed data and make predictions about data that may be observed later. Science makes engineering possible and has led to many developments that heavily influence modern human life. Many kinds of models are useful. Some involve mathematical structures, such as distributions or differential equations, to which one can devote lifetimes of study. Simpler models, involving only finite numbers of variables, equations, inequalities, and objectives and described by finitely many parameters and sets, have a surprisingly wide range of uses. When one studies a new area, choices for suitable models may be far from obvious, and it may be necessary to try many models. Statistics is largely about comparing candidate models and, particularly with exploratory data analysis, finding suitable ones.

Algebraic modeling languages, such as the AMPL language considered in this paper, facilitate formulating, comparing, changing, and deriving results from a subset of the class of ‘‘simpler’’ models outlined above in which equations, inequalities, objectives and derived sets and parameters are expressed algebraically. In short, AMPL is focused on *mathematical programming* problems, such as constrained optimization problems of the form

$$\text{minimize} \quad f(x) \tag{1a}$$

$$\text{s.t.} \quad \ell \le c(x) \le u, \tag{1b}$$

with $x \in \mathbb{R}^n$ and $c\colon \mathbb{R}^n \to \mathbb{R}^m$, possibly with some components of x restricted to integer values.

2. AMPL Design Principles

AMPL is meant to make it easy to transcribe models from mathematical notation, such as one might write by hand on paper or white board, to the AMPL language. We sought to make the language both close to elementary algebraic notation and easy to enter on an ordinary computer keyboard. As explained below, AMPL was created at Bell Labs, in the then Computing Science Research Center, where such languages as C [26, 27], C++ [30], and awk [1, 2] had been created, so AMPL uses some of the same notational conventions as these languages, such as square brackets for subscripts. Models often have sets of variables and constraints, and AMPL allows one to have various kinds of subscripted entities. In model entities, such as constraints and objectives, all subscripting is explicit, in part to make meaning of these entities clear. AMPL is a declare-before-use language, so one can read a model from top to bottom without worrying about the meaning of something whose properties are given later.

An AMPL model can represent a whole class of problems. For example, a linear objective might be specified by the declarations

```
set S;
var x{S} >= 0;
param p{S};
minimize Cost: sum{i in S} p[i]*x[i];
```

in which the objective is named ‘‘`Cost`’’ and is a transcription of

$$\sum_{i \in S} p_i x_i .$$

Thus a model can involve sets (such as `S`) over which entities, such as parameters and variables, e.g., `p` and `x`, are indexed, and can be stated without regard to the values that its sets and parameters will have in a particular problem to be solved, i.e., an *instance*. The AMPL language consists of three sub-languages: one for declarations, such as the `set`, `var`, `param`, and objective (`minimize`) declarations above, a simplified language in ‘‘data sections’’ for giving values to sets and parameters, and a command language for modifying values, solving problems, and writing results in various ways. While AMPL permits commingling declarations and instance data, AMPL also makes it easy to separate pure models from instance data.

AMPL does not solve problems by itself (except when AMPL’s problem simplifications — its *presolve* — result in a solved problem), but instead writes files with full details of the problem instances to be solved and invokes separate solvers. The AMPL/solver interface library [19], whose source is freely available, provides problem details to *solver interfaces*, which interact with particular solvers to find solutions and return them to AMPL.

Often one needs to solve sequences of problems, with the solution of one problem providing data used in the next problem. Sometimes this involves updating set and parameter values. AMPL only instantiates or recomputes problem entities as needed, effectively using lazy evaluation to help speed up processing.

While parts of the AMPL language are general purpose, other parts, such as the *presolve* phase and computation of reduced costs, are tailored to mathematical programming. AMPL is meant for use with both linear and nonlinear problems; its internal use of sparse data structures allows AMPL to be useful with some very large problem instances.

3. AMPL History

AMPL arose in part because of Karmarkar’s linear-programming algorithm [24]. At the time, there was much interest at the Computing Science Research Center in ‘‘little languages’’, e.g., for graphing data, solving least-squares problems, drawing figures, etc. While Karmarkar’s algorithm seemed to promise faster solutions of some linear programming problems, we thought a ‘‘little language’’ to express such problems would help make the algorithm useful in practice. I had known Bob Fourer since the mid 1970s, when we both worked at the NBER Computer Research Center in Cambridge, Massachusetts, where Bob had done his undergraduate work at MIT. He had subsequently obtained a Ph.D. at Stanford University under George Dantzig and had published a nice paper [10] arguing for modeling languages. Bob was now a professor at Northwestern University and, as I learned when I saw him at a meeting, was coming up for a sabbatical. My management arranged for Bob to spend his sabbatical at Bell Labs in the 1985-86 academic year, during which he, Brian Kernighan, and I worked on the first version of AMPL. (We were aware of GAMS [5], but GAMS was not yet generally available and, anyway, we thought we could do a better language design. Such other modeling languages as AIMMS [4] and MPL [28] came along later.) Brian wrote the first implementation of AMPL; I wrote a preprocessor to transform data sections to a simpler, now defunct, format for the original AMPL processor.

Our first technical report on AMPL [13] appeared in 1987. In revised form, it eventually appeared in *Management Science* [14]. By then, I had written a new implementation to facilitate various extensions we had in mind, such as handling nonlinearities.

Since the late 1970s I had been aware of Kedem's work [25] on forward automatic differentiation (AD), which provides a mechanical way to compute analytically correct derivatives, and I was thinking of adding such facilities to AMPL. I mentioned this to Andreas Griewank when I saw him in 1988 at the International Symposium on Mathematical Programming (ISMP) in Tokyo, and he told me about the more efficient ‘‘reverse’’ automatic differentiation. (He has subsequently written much more about AD; see [22] for pointers to AD history and [23] for more on AD in general.) Reverse AD computes a function and its gradient with work proportional to that of computing the function alone, whereas forward AD, like straightforward symbolic differentiation, can turn a function evaluation involving n arithmetic operations into a computation involving $O(n^2)$ operations. Both avoid the truncation errors inherent in finite differences. Ever since the Tokyo ISMP, I have been a fan of reverse AD. AMPL itself uses reverse AD to compute nonlinear reduced costs, but most AD happens in the solver interface library. See [17] for more on first derivative computations in this regard and [18] for some details of finding and exploiting partially separable structure when doing Hessian (second derivative) computations.

By the early 1990s we had enough material to write a book on AMPL [15]. We continued adding facilities to AMPL and added much new material to the second edition [16] of the book.

The ‘‘dot-com bubble burst’’ of 2001 threw a monkey wrench into AMPL development, but did cause creation of the AMPL Optimization company. Eventually I went to work at Sandia National Labs in Albuquerque, New Mexico, where I worked on AMPL support after hours (and without pay). Brian became a professor at Princeton. The three co-authors continued to interact via E-mail. When we got an NSF SBIR grant for some new work on AMPL, I left Sandia to work for the AMPL company (and get some pay). Bob Fourer retired somewhat later from Northwestern University and now also works full time for the AMPL company.

4. Some Simple Declarations and Commands

Here is a simple example of some declarations, commands, and a little data section:

```
param p;
param q = p + 10;
data; param p := 2.5;
display p, q;
```

The third line is the data section, which gives a value to `p` that is used in the ‘‘display’’ command, which produces output

```
p = 2.5
q = 12.5
```

Data sections are good for conveying single values as well as tables of data, but data sections have relaxed quoting rules and other simplifications that preclude the appearance of expressions. The ‘‘let’’ command, by contrast, can involve general expressions. For example,

```
let p := 17; display p, q;
```

gives

```
p = 17
q = 27
```

Notice that q was automatically recomputed.

AMPL can be used in batch interactive mode (reading from a file) or interactive mode (reading from the standard input). Prompts are given in interactive mode. Doing the above exercise in interactive mode, one would see

```
ampl: param p;
ampl: param q = p + 10;
ampl: data; param p := 2.5;
ampl: display p, q;
p = 2.5
q = 12.5
ampl: let p := 17; display p, q;
p = 17
q = 27
```

5. Simple Sets

To illustrate some simple sets and an error, here is a continuation of the above interactive-mode session.

```
ampl: set A; set B;
ampl: set C = p .. q;
ampl: display A;
Error executing "display" command:
        no data for set A
ampl: data; set A := a b c; set B := c d;
ampl data: display A, B, C;
set A := a b c;

set B := c d;

set C := 17 18 19 20 21 22 23 24 25 26 27;
```

The prompt ‘‘ampl data:’’ indicates data-section mode; the ‘‘display’’ command causes AMPL to revert to model/command reading mode. Here are examples of some set operations:

```
ampl: display A intersect B, A union B;
set A inter B := c;
set A union B := a b c d;

display A diff B, A symdiff B;
set A diff B := a b;

set A symdiff B := a b d;
```

6. Iterated and Recursive Expressions

Often it is useful to use iterated expressions, such as iterated sums. Here are some iterated expressions and a recursive definition, illustrated with the help of ‘‘print’’ commands.

```
ampl: print sum {i in 1..4} i;
10
ampl: print prod {i in 1..4} i;
24
ampl: param fac{ i in 1..9 }
ampl? = if i == 1 then 1 else i*fac[i-1];
ampl: print max{i in 1..9}
ampl? abs(fac[i] - prod{j in 2..i} j);
0
ampl: display fac, {i in 1..9} prod{j in 2..i} j;
:    fac   prod{j in 2 .. i} j    :=
1        1             1
2        2             2
3        6             6
4       24            24
5      120           120
6      720           720
7     5040          5040
8    40320         40320
9   362880        362880
;
```

7. Example Model: diet.mod

The diet model in the AMPL book [16] provides a short but complete example of a model for choosing what foods to buy. The model involves sets `NUTR` and `FOOD` of nutrients and foods, subscripted parameters `f_min`, `f_max`, and `cost` that specify minimum and maximum amounts of each food to buy and how much one unit of each food costs, a doubly subscripted parameter `amt` that tells how many units of each nutrient are provided by one unit of each food, and subscripted parameters `n_min` and `n_max` that give lower and upper bounds on the amounts of each nutrient that the foods we buy are to provide. The objective is to satisfy the nutritional requirements at minimal cost by choosing suitable values for the decision variables `Buy`.

```
set NUTR;
set FOOD;

param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];

param amt {NUTR,FOOD} >= 0;

var Buy {j in FOOD} >= f_min[j], <= f_max[j];

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

subject to Diet {i in NUTR}:
   n_min[i] <= sum{j in FOOD} amt[i,j]*Buy[j] <= n_max[i];
```

The above model describes a class of problems. Here is an example of a data section (called ‘‘diet.dat’’ in the AMPL book) that provides data for a particular problem instance.

```
data; set NUTR := A B1 B2 C ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:   cost  f_min  f_max :=
  BEEF   3.19    0     100
  CHK    2.59    0     100
  FISH   2.29    0     100
  HAM    2.89    0     100
  MCH    1.89    0     100
  MTL    1.99    0     100
  SPG    1.99    0     100
  TUR    2.49    0     100 ;

param:   n_min  n_max :=
   A      700   10000
   C      700   10000
   B1     700   10000
   B2     700   10000 ;

param amt (tr):
          A    C   B1   B2 :=
   BEEF   60   20   10   15
   CHK     8    0   20   20
   FISH    8   10   15   10
   HAM    40   40   35   10
   MCH    15   35   15   15
```

```
          MTL    70   30   15   15
          SPG    25   50   25   15
          TUR    60   20   15   10 ;
```

The data section above illustrates some tabular input formats. AMPL also has ‘‘table’’ declarations and ‘‘read table’’ and ‘‘write table’’ commands for reading data from, and writing data to, external repositories, such as data bases and spreadsheets.

8. Sample Session

Here is an example of solving the above problem instance.

```
ampl: model diet.mod; data diet.dat;
ampl: solve;
MINOS 5.51: optimal solution found.
6 iterations, objective 88.2
ampl: display Buy;
Buy [*] :=
BEEF   0
 CHK   0
FISH   0
 HAM   0
 MCH  46.6667
 MTL   1.57618e-15
 SPG   8.42982e-15
 TUR   0
;
```

The resulting menu is not very satisfactory: 46 and 2/3 packages of macaroni and cheese (‘‘MCH’’). We probably want to buy only whole packages, which we can do by using integer variables:

```
ampl: redeclare var Buy{j in FOOD}
ampl? integer >= f_min[j] <= f_max[j];
ampl: solve;
MINOS 5.51: ignoring integrality of 8 variables
MINOS 5.51: optimal solution found.
4 iterations, objective 88.2
```

Since MINOS (the default solver) does not deal with integer variables, we need to use a solver that only allows integer variables to have integer values. Many solvers can do this; here we use CPLEX:

```
ampl: option solver cplex; solve;
CPLEX 12.6.0.0: optimal integer solution; objective 88.44
4 MIP simplex iterations
0 branch-and-bound nodes
ampl: display Buy;
```

```
Buy [*] :=
BEEF   0
 CHK   2
FISH   0
 HAM   0
 MCH  43
 MTL   1
 SPG   0
 TUR   0
;
```

9. Analyzing Infeasibility

Formulating a good model is often an iterative process: we repeatedly try a formulation, examine its consequences, then modify it. As a simple example, the diet above is still not very satisfactory, so we could change the data to provide positive lower bounds on the amounts of each food bought. Here is file ‘‘diet2.dat’’ from the AMPL book:

```
set NUTR := A B1 B2 C NA CAL ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:   cost  f_min  f_max :=
  BEEF   3.19    2     10
  CHK    2.59    2     10
  FISH   2.29    2     10
  HAM    2.89    2     10
  MCH    1.89    2     10
  MTL    1.99    2     10
  SPG    1.99    2     10
  TUR    2.49    2     10  ;

param:   n_min  n_max :=
   A       700   20000
   C       700   20000
   B1      700   20000
   B2      700   20000
   NA        0   40000
   CAL  16000   24000 ;

param amt (tr):
           A    C   B1   B2    NA   CAL :=
   BEEF   60   20   10   15   938   295
   CHK     8    0   20   20  2180   770
   FISH    8   10   15   10   945   440
   HAM    40   40   35   10   278   430
   MCH    15   35   15   15  1182   315
   MTL    70   30   15   15   896   400
   SPG    25   50   25   15  1329   370
```

```
            TUR    60   20   15   10  1397   450 ;
```

By using a ‘‘reset data’’ command, we can keep the current model but associate a fresh set of data with it.

```
ampl: reset data; data diet2.dat;
ampl: solve;
CPLEX 12.6.0.0: integer infeasible.
1 MIP simplex iterations
0 branch-and-bound nodes
No basis.
```

There are various approaches to diagnosing infeasibility. Sometimes it is helpful just to see which constraints are infeasible and what variables are at lower or upper bound at the variable values where the solver detected infeasibility. For example,

```
ampl: option solver minos; solve;
MINOS 5.51: ignoring integrality of 8 variables
MINOS 5.51: infeasible problem.
9 iterations
ampl: display Diet.lb, Diet.body, Diet.ub, Diet.slack;
:   Diet.lb   Diet.body Diet.ub     Diet.slack      :=
A       700    1993.09    20000   1293.09
B1      700     841.091   20000    141.091
B2      700     601.091   20000    -98.9086
C       700    1272.55    20000    572.547
CAL   16000   17222.9     24000   1222.92
NA        0   40000       40000      7.27596e-12
;
```

Here, `Diet.lb`, `Diet.body` and `Diet.ub` correspond to ℓ, $c(x)$ and u in (1b), and the constraint slack `Diet.slack` corresponds to $\min(u - c(x),\, c(x) - \ell)$. Most of the constraints are satisfied as inequalities (i.e., they have positive slacks), but the B2 constraint has a decidedly negative slack, while the NA (sodium) constraint is essentially satisfied as an equality (with a slack of about 7.3×10^{-12}) and `Diet.body` essentially at its upper bound. Increasing the upper bound on the sodium constraint might help:

```
ampl: let n_max['NA'] := 50000; solve;
MINOS 5.51: ignoring integrality of 8 variables
MINOS 5.51: optimal solution found.
5 iterations, objective 118.0594032
```

so allowing more sodium is one way to remove the infeasibility.

Another way to diagnose infeasibility is by finding an *irreducible infeasible set* (IIS) of constraints and variable bounds that are mutually inconsistent; see [29, 7] and references therein for more details. Some solvers nowadays have facilities for finding an IIS. With CPLEX, for example,

```
option cplex_options 'iisfind=1'; solve;
```

would also implicate the B2 and sodium constraints.

10. A Nonlinear Example

AMPL allows general nonlinear expressions in constraints and objectives. The ‘‘largest small hexagon’’ problem [21] provides a small example of an interesting nonlinear optimization problem. Here is a lightly edited variant of a little AMPL model, ‘‘pgon.mod’’, that describes the problem and has long been available as http://www.netlib.org/ampl/models/pgon.mod:

```
# Maximum area for unit-diameter polygon of N sides.
# The following model started as a GAMS model by Francisco J. Prieto.

param N integer > 0 default 6;
set I = 1..N;

param pi = 4*atan(1.);

var rho{i in I} <= 1, >= 0  # polar radius (distance to fixed vertex)
                := 4*i*(N + 1 - i)/(N+1)**2;

var theta{i in I} >= 0  # polar angle (measured from fixed direction)
                := pi*i/N;

subject to cd{i in I, j in i+1 .. N}:
      rho[i]**2 + rho[j]**2 - 2*rho[i]*rho[j]*cos(theta[j]-theta[i])
      <= 1;

subject to ac{i in 2..N}:
        theta[i] >= theta[i-1];

subject to fix_theta: theta[N] = pi;
subject to fix_rho:   rho[N] = 0;

maximize area:
      .5*sum{i in 2..N} rho[i]*rho[i-1]*sin(theta[i]-theta[i-1]);
```

The # character introduces a comment that extends to the end of the line. The ‘‘`:=` *expression*’’ phrases specify initial guesses for the variables. Perhaps surprisingly, the solution is not the regular N-gon. Figure 1 depicts a solution for N = 6.

11. Slices

AMPL’s basic indexing notation introduces one new dummy variable for each component of the tuples that comprise a set. For example,

```
set S dimen 2;
```

declares a set of pairs, and

```
{(i,j) in S}
```

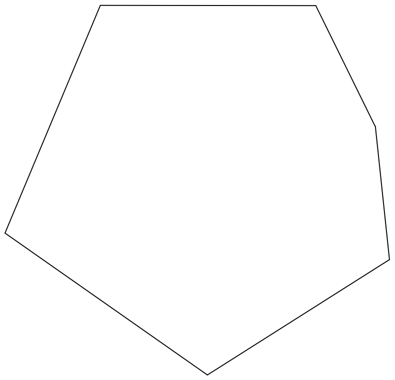

Figure 1: solution of `pgon.mod` for N = 6.

is an ‘‘indexing’’ in which dummy variables `i` and `j` assume the values of the first and second components of each pair in the set. Sometimes one wants a ‘‘slice’’ of a set of tuples, i.e., an indexing in which some components are given by expressions valid in the context of the indexing. For example,

```
s.t. c{a in A}:
    sum{(i,j) in S: i == a} x[i,j] == 1;
```

is a constraint declaration with a sum that effectively involves a slice. AMPL’s slice notation allows one to put desired values directly into the indexing notation. The above example has the same effect as

```
s.t. c{a in A}:
    sum{(a,j) in S} x[a,j] == 1;
```

but the latter is easier to read and can be much faster, since internally S is split into a set of one-dimensional sets. For a set S of n members, this can turn an $O(n^2)$ computation into an $O(n)$ computation. A few years ago I saw an example in which changing the former to the latter reduced problem instantiation time from four hours to a minute.

12. Iterated Unions

In various contexts, it is useful to construct sets by iterating over computed set expressions and forming their union. For example, given the declaration

```
set A dimen 2;
```

of a set of pairs, the declaration

```
set J = union{(i,j) in A} {j};
```

forms the set J of second components of the pairs in A. For forming iterated unions of singleton sets, such as `{j}` above, the `setof` operator provides simpler syntax that achieves the same effect:

```
set J = setof{(i,j) in A} j;
```

As an example where `setof` is useful, here is a little model for choosing a convex combination of classifiers that is ‘‘best’’ in a least-squares sense.

```
set A dimen 2;               # (observation, classifier) pairs
param E{A};                  # signed, weighted predictions
set I = setof {(i,j) in A} i; # observations
set J = setof {(i,j) in A} j; # classifiers
param y{I} in {1,-1};        # y[i] = 1 ==> "yes", -1 ==> "no"
var x{J} >= 0;               # weights on classifiers
set B = {(i,j) in A: y[i]*E[i,j] < 0}; # mis-classified pairs

minimize errsq: sum{i in I} (sum{(i,j) in B} y[i]*E[i,j]*x[j])ˆ2;
s.t. convex: sum{i in J} x[i] = 1;
```

More elaborate iterated unions are sometimes useful. For example, the following fragment from a mesh-untangling model declares the set of directed edges of some boxes.

```
set P;    # points
set Boxes within {P,P,P,P,P,P,P,P};
set Edges = union {(a,b,c,d,e,f,g,h) in Boxes} {
                        (a,b), (a,d), (a,e),
                        (b,c), (b,a), (b,f),
                        (c,d), (c,b), (c,g),
                        (d,a), (d,c), (d,h),
                        (e,h), (e,f), (e,a),
                        (f,e), (f,g), (f,b),
                        (g,f), (g,h), (g,c),
                        (h,g), (h,e), (h,d)};
```

Products of matrices appear surprisingly rarely in the mathematical programming problems one sees in practice, but the sparse product of sparse matrices is easily expressed with the help of an iterated union (via *setof*) and slice notation:

```
set IJ dimen 2; param A{IJ};
set JK dimen 2; param B{JK};
set IK = setof{(i,j) in IJ, (j,k) in JK} (i,k);
param C{(i,k) in IK} =
     sum{(i,j) in IJ: (j,k) in JK} A[i,j]*B[j,k];
```

13. AMPL Flexibility Goals

We have sought to make AMPL useful in various contexts. For developing models, it can be useful to use AMPL interactively, typing commands at it. For longer computations, ‘‘batch’’ mode, in which AMPL reads everything from specified files, can be convenient. We have long had some experimental graphical user interfaces (GUIs) and have recently put considerable effort into developing a new ‘‘integrated development environment’’ (IDE); see http://ampl.com/products/ide/.

The AMPL language itself is primitive recursive, but AMPL has facilities for importing libraries of functions implemented in other languages. A README file about these facilities and some examples appear in http://www.netlib.org/ampl/solvers/funclink. (At times, http://www.ampl.com/netlib/ampl/solvers/funclink may be more up to date.) A library that includes more than 300 functions from the GNU Scientific Library (http://www.gnu.org/software/gsl/) is available in source and binary form at http://ampl.com/resources/extended-function-library/. AMPL's imported function facilities also allow AMPL to import ''table handlers'' for reading data from and writing data to external repositories, such as spreadsheets and data bases. Details on using the table facilities appear in http://ampl.com/resources/database-and-spreadsheet-table-handlers/. Details on writing your own table handlers are in http://ampl.com/NEW/TABLES/. The *tableproxy* table handler permits accessing data on remote machines and facilitates mixing 32- and 64-bit versions of AMPL and data providers on the same machine. See http://ampl.com/NEW/TABLEPROXY/.

In various ways, we have sought to make it convenient for AMPL to interact with its host environment (operating system). A general ''shell'' command allows one to run arbitrary programs. AMPL's printing commands (`print` for unformatted printing, `printf` for formatted printing, and `display` for labeled printing) can have their output directed to files, which may either be created afresh or appended to. The `remove` command is for deleting files. ''Pipe'' functions provide a simple way for AMPL to interact with external programs: AMPL writes function arguments to the standard input of an external program, and the program returns the function value by writing to its standard output. A program implementing a ''pipe'' function must flush its output buffers before reading new function arguments, which can be awkward.

The currently popular operating systems all provide an ''environment'' of name-value pairs that programs can see and manipulate. The names are ''environment variables''. AMPL's ''option'' command operates on these environment variables and exports them to solvers (which are invoked as separate processes) and ''shell'' commands (which also are invoked as separate processes). AMPL's behavior is affected by some options. When starting execution, AMPL acquires values for these options from the incoming environment if present there and provides default values for them if not. Most solvers also are affected by environment variable values. Conventionally, the AMPL interface to a solver named `mysolver` would look at the environment variable named `mysolver_options`, which could be specified in an AMPL session by ''`option mysolver_options`'' commands, such as

```
option cplex_options 'advance=2 lpdisplay=1 \
                      prestats = 1 \
                      primalopt'
                     " aggregate=1 aggfill=20";

option solver knitro,
              knitro_options "maxit=30";
```

Strings may be quoted by single or double quotes. For option values, adjacent strings are concatenated.

Currently under development is ‘‘AMPL API’’, another way for AMPL to interact with external programs. See http://ampl.com/products/apibeta/.

14. Interaction with Solvers

AMPL’s ‘‘`solve`’’ command proceeds by writing a ‘‘.nl file’’ (a file whose name ends with ‘‘.nl’’) containing

- problem statistics
- coefficients for linear expressions
- expression graphs for nonlinear expressions
- initial guesses (primal and dual)
- suffixes (builtin or user declared).

Solvers return solution results to AMPL by writing a ‘‘.sol’’ file for AMPL to read. This file contains a ‘‘solve_message’’ and status code and may contain updated primal and dual variable values. It may also contain suffix values, which are auxiliary values associated with individual variables, constraints, objectives and problems, such as basis status for variables and constraints.

15. Problem Transformations

AMPL’s presolve phase [11] derives and propagates bounds with directed roundings and may fix variables, remove constraints (e.g., inequalities that are never tight), resolve complementarities, turn nonlinear expressions into linear expressions (after fixing relevant variables), simplify convex piecewise-linear expressions, and convert nonconvex piecewise-linear expressions into equivalent systems of integer variables and SOS-2 [3] constraints. It also processes ‘‘defined variables’’, which in effect are named common expressions. For example, the declarations

```
param N integer > 0;
set I = 1 .. N;
var x{I}; var y{I};
var dot = sum{i in I} x[i]*y[i];
```

declares independent variables x and y and defined-variable dot, which is the inner product of x and y. Constraints and objectives could involve dot, but the solver would only see x and y as independent variables.

16. Spline Example

A referee asked about splines. I do not recall anyone wanting to use splines with AMPL, but the following illustration of constructing a spline approximation provides an example of using some of the facilities sketched above. We will use an imported function called `bspline` that, given a spline degree, a set of breakpoints and weights on B-spline basis functions (see chapter X of [6]) and a point *x* sufficiently within the breakpoints that all relevant basis functions are defined, computes the value of the spline at x and the first derivatives of this value with respect to *x*, the weights, and the breakpoints. The derivatives facilitate choosing the weights to fit specified data. The derivatives are handled by the ASL and do not explicitly appear in the following model.

```
param N default 3;   # degree of splines
param ND;            # ND+1 = number of data points
set SD = 0 .. ND;    # indices of data points
param xd{SD};        # ordinates of data points
param fd{SD};        # function values at data points

check{i in 1 .. ND}: xd[i-1] < xd[i];

param NI >= 1;           # number of intervals for
                         # x in bspline(n,x,...)
set SK = -N .. NI + 3;   # indices of knots
set SW = 1 .. NI + N;    # indices of B-spline weights
param wrange = xd[ND] - xd[0];
param b0{i in SK} := xd[0] + i*wrange/NI;
var b{i in SK} := b0[i]; # spline knots
var w{i in SW};          # spline weights

function bspline;
var s{i in SD} =
    bspline(N, xd[i], {j in SK} b[j], {j in SW} w[j]);

minimize ssq: sum{i in SD} 0.5*(fd[i] - s[i])ˆ2;

s.t. resid{i in SD}: s[i] == fd[i];

problem SSQ: b, w, ssq;
problem NLS: b, w, resid; option presolve 0;
```

It might be good to add constraints that would keep the breakpoints ordered, but for the solvers used in the sample session shown below, this turns out not to be unnecessary. To find values for `b` and `w` so `bspline(n,`xd[i],...) approximates `fd[i]` in a least-squares sense, we can either use an unconstrained solver with problem SSQ or a least-squares solver with problem NLS; least-squares solvers, such as `nl2` (discussed in [19] and based on NL2SOL [8]) solve equations in a least-squares sense. For such solving, it is often necessary to turn AMPL's presolve off to prevent it from satisfying some equations exactly.

For an example session, let us fit a cubic spline to the sine function. Suppose the above model appears in file `bspline.mod` and that file `sine.fit` contains

```
model splined.mod;
param pi = 4*atan(1);
```

```
data;
param ND := 21;  param NI := 5;

let{i in SD} xd[i] := 2*pi*(i/ND);
let{i in SD} fd[i] := sin(xd[i]);
fix{i in -N .. -1} b[i];
fix{i in NI+1 .. NI+N} b[i];
```

Here is a session fitting the data both ways with the above model and setup files:

```
ampl: include spline.fit
ampl: load bspline.dll;
ampl: option solver nl2; solve;
nl2:    Relative Function Convergence; function = 5.40485704e-06
        RELDX = 8.12e-05; PRELDF = 1.94e-11; NPRELDF = 1.94e-11
        19 func. evals; 16 grad. evals
ampl: printf "%.3g\n", max{i in SD} abs(s[i] - fd[i]);
0.00109
ampl: problem SSQ;
ampl: option reset_initial_guesses 1, solver snopt;
ampl: solve;
SNOPT 7.2-8 : Optimal solution found.
80 iterations, objective 5.404937356e-06
Nonlin evals: obj = 67, grad = 66.
ampl: printf "%.3g\n", max{i in SD} abs(s[i] - fd[i]);
0.00109
```

Both solvers achieved about the same residual sum of squares and maximum fit error on the set of sample points.

The problem of choosing `b` and `w` to fit best in a least-squares sense is a separable nonlinear least-squares problem [20], as the `w` variables appear linearly, and a separable solver probably would be faster and somewhat more robust. At any rate, after determining `b` and `w`, we could fix them (causing them to retain their current values and be treated as parameters) and deal with some application where the spline just found would be useful.

Source for `bspline.dll` is too long to include with this paper, but is available as
`http://www.ampl.com/netlib/ampl/solvers/examples/bspline.c`

17. Implementation Techniques

AMPL’s implementation is an exercise in practical computer science. Parsing proceeds via the venerable Unix tools *lex* and *yacc*, which build up expression graphs that are subsequently manipulated. Declared names are associated with unique ‘‘symbols’’ found by hashing. Hashing is also used in a ‘‘compile’’ phase to find common expressions. The compile phase lifts invariant subexpressions out of inner loops. With the help of dependency graphs, entities are only instantiated or updated when needed — lazy evaluation. When appropriate, cleanup routines are registered, so they can be invoked either when an operation completes normally or when it is interrupted by an error, such as an invalid subscript or missing data. Error handling proceeds via *longjump*. Some things are reference-counted, and sparse-matrix techniques make processing large, sparse models feasible. AMPL is written (and debugged) in C++, but for porting to various platforms, the AMPL source code is converted to portable C with the help of *cfront* (the original C++ ‘‘compiler’’).

18. Wish List

There are many improvements we hope to make to AMPL and its associated ASL (solver-interface library). Just when and whether these improvements will be available remains to be seen. Functions expressed directly in AMPL would turn AMPL from a primitive-recursive language to a Turing-complete language. When conveyed to solvers via the ASL, they would allow providing callbacks to solvers, e.g., for influencing branching decisions in integer programming. They would also find some use in AMPL models. Ordered sets of tuples would sometimes be useful. While AMPL already facilitates solving sequences of related problems, updating entities could sometimes be done more efficiently. AMPL has long had some facilities for constraint programming, but allowing variables in subscripts remains to be done. When there is just one objective (for multi-objective optimization, AMPL allows one to declare several objectives, including indexed collections of objectives), AMPL’s presolve could exploit duality. (It already does reductions for complementarity.) AMPL has long permitted some declarations related to stochastic programming, but corresponding extensions to the ASL need to be completed and examples of their use need to be created. Facilities supporting semidefinite programming and multi-level optimization would be useful. We have long wanted AMPL to be able to carry on two-way conversations with solvers, so after a problem has been solved, a slightly modified problem could be conveyed just by telling the solver of changes to the existing problem. Units (of distance, time, charge, etc.) might help catch or avoid some mistakes. For some mathematical research, such data types as rational, complex, and complex rational could be helpful. Facilities for parallel evaluations in the ASL would be useful. Constructs for parallelism might also be useful in AMPL itself.

19. Other AMPL Facilities

This paper provides an overview of AMPL, but gives little or no detail about various useful AMPL facilities:

- drop, restore (affecting what constraints and objectives a solver sees)
- fix, unfix (affecting the variables a solver sees)
- named problems and environments
- suffixes
- tables and table handlers
- column-generation syntax (e.g., node and arc)

- complementarity constraints [9]
- subscripted sets versus tuples
- constraint programming [12]

The AMPL web site (http://www.ampl.com) provides pointers to more detail on the above topics, including

- the AMPL book (and free PDF files for it)
- examples (models and data)
- descriptions of new facilities
- a new IDE
- a new API
- *Try AMPL!* and NEOS for free web-based use
- course licenses
- trial licenses
- downloads
  - □ student binaries
  - □ ASL (solver-interface library) source
  - □ example solver interfaces
  - □ ‘‘standard’’ table handler (binaries, source)
  - □ papers, reports, talk slides

20. Concluding Remarks

Mathematical programming models, such as (1), are useful in many contexts. Formulating good models is often an iterative process: you test a formulation, assess how well it works, modify it and test again. The AMPL modeling language can assist in this endeavor. Its associated interface library (ASL) provides automatically derived details to solvers, such as sparsity information and derivatives.

21. REFERENCES

[1] A. V. AHO, P. J. WEINBERGER, AND B. W. KERNIGHAN, ‘‘AWK — a Pattern Scanning and Processing Language,’’ *Software—Practice and Experience* (July 1978).

[2] A. V. AHO, P. J. WEINBERGER, AND B. W. KERNIGHAN, *The AWK Programming Language,* Addison-Wesley, 1988.

[3] E. M. L. BEALE AND J. A. TOMLIN, ‘‘Special Facilities in a General Mathematical System for Non-Convex Problems Using Ordered Sets of Variables,’’ pp. 447–454 in *Proceedings of the Fifth International Conference on Operational Research*, ed. J. Lawrence, Tavistock Publications, London (1970).

[4] JOHANNES BISSCHOP AND ROBERT ENTRIKEN, *AIMMS, The Modeling System,* Paragon Decision Technology, 1993.

[5] J. BISSCHOP AND A. MEERAUS, ‘‘Selected Aspects of a General Algebraic Modeling Language,’’ pp. 223–233 in *Optimization Techniques, Part 2*, ed. K. Iracki, K. Malanowski, and S. Walukiewicz, Springer-Verlag, Berlin (1980). Proceedings of the 9th IFIP Conference on Optimization Techniques, Warsaw, Sept. 4–8, 1979

[6] C. DE BOOR, *A Practical Guide to Splines*, Springer-Verlag, 1978.

[7] J. W. CHINNECK AND E. W. DRAVNIEKS, ‘‘Locating Minimal Infeasible Constraint Sets in Linear Programs,’’ *ORSA J. Computing* **3** #2 (1991), pp. 157–168.

[8] J. E. DENNIS, JR., D. M. GAY, AND R. E. WELSCH, ‘‘An Adaptive Nonlinear Least-Squares Algorithm,’’ *ACM Trans. Math. Software* **7** (1981), pp. 348–368.

[9] MICHAEL C. FERRIS, ROBERT FOURER, AND DAVID M. GAY, ‘‘Expressing Complementarity Problems in an Algebraic Modeling Language and Communicating Them to Solvers,’’ *SIAM Journal on Optimization* **9** #4 (1999), pp. 991–1009.

[10] R. FOURER, ‘‘Modeling Languages Versus Matrix Generators for Linear Programming,’’ *ACM Trans. Math. Software* **9** #2 (1983), pp. 143–183.

[11] ROBERT FOURER AND DAVID M. GAY, ‘‘Experience with a Primal Presolve Algorithm,’’ pp. 135–154 in *Large Scale Optimization: State of the Art*, ed. W. W. Hager, D. W. Hearn, and P. M. Pardalos, Kluwer Academic Publishers (1994).

[12] ROBERT FOURER AND DAVID M. GAY, ‘‘Extending an Algebraic Modeling Language to Support Constraint Programming,’’ *INFORMS Journal on Computing* **14** #4 (2002), pp. 322–344.

[13] R. FOURER, D. M. GAY, AND B. W. KERNIGHAN, ‘‘AMPL: A Mathematical Programming Language,’’ Computing Science Technical Report No. 133 (Jan. 1987 (revised June 1989)), AT&T Bell Laboratories, Murray Hill, NJ 07974.

[14] R. FOURER, D. M. GAY, AND B. W. KERNIGHAN, ‘‘A Modeling Language for Mathematical Programming,’’ *Management Science* **36** #5 (1990), pp. 519–554.

[15] ROBERT FOURER, DAVID M. GAY, AND BRIAN W. KERNIGHAN, *AMPL: A Modeling Language for Mathematical Programming,* Duxbury Press/Wadsworth, 1993. ISBN: 0-89426-232-7.

[16] ROBERT FOURER, DAVID M. GAY, AND BRIAN W. KERNIGHAN, *AMPL: A Modeling Language for Mathematical Programming,* Duxbury Press / Brooks/Cole Publishing Company, 2003. ISBN: 0-534-38809-4.

[17] DAVID M. GAY, ‘‘Automatic Differentiation of Nonlinear AMPL Models,’’ pp. 61–73 in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, ed. A. Griewank and G. F. Corliss, SIAM (1991).

[18] D. M. GAY, ‘‘More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability,’’ in *Computational Differentiation: Applications, Techniques, and Tools*, ed. George F. Corliss, SIAM (1996).

[19] DAVID M. GAY, ‘‘Hooking Your Solver to AMPL,’’ Technical Report 97-4-06 (April, 1997), Computing Sciences Research Center, Bell Laboratories. See http://-www.ampl.com/ampl/REFS/hooking2.ps.gz.

[20] G. H. GOLUB AND V. PEREYRA, ‘‘The Differentiation of Pseudo-Inverses and Nonlinear Least-Squares Problems Whose Variables Separate,’’ *SIAM J. Numer. Anal.* **10** (1973), pp. 413–432.

[21] R. L. GRAHAM, ‘‘The Largest Small Hexagon,’’ *J. Combinatorial Theory (A)* **18** (1975), pp. 165–170.

[22] A. GRIEWANK, ‘‘On Automatic Differentiation,’’ pp. 83–107 in *Mathematical Programming*, ed. M. Iri and K. Tanabe, Kluwer Academic Publishers (1989).

[23] ANDREAS GRIEWANK AND ANDREA WALTHER, *Evaluating Derivatives,* SIAM, 2008.

[24] N. KARMARKAR, ‘‘A New Polynomial-time Algorithm for Linear Programming,’’ *Combinatorica* **4** (1984), pp. 373–395.

[25] GERSHON KEDEM, ‘‘Automatic Differentiation of Computer Programs,’’ *ACM Trans. Math. Software* **6** #2 (1980), pp. 150–165.

[26] B. W. KERNIGHAN AND D. M. RITCHIE, *The C Programming Language,* Prentice-Hall, 1978.

[27] B. W. KERNIGHAN AND D. M. RITCHIE, *The C Programming Language,* Prentice-Hall, 1988. Second Edition

[28] BJARNI KRISTJANSSON, *MPL — Modelling System Quick Guide,* Maximal Software, Reykjavik, Iceland, 1991.

[29] J. VAN LOON, ‘‘Irreducibly Inconsistent Systems of Linear Inequalities,’’ *European J. Operational Research* **8** (1981), pp. 283–288.

[30] B. STROUSTRUP, *The C++ Programming Language*, Addison-Wesley, 1986.