AMPL (A Mathematical Programming Language) at the University of Michigan

Documentation (**Version 2**)

D. Holmes `dholmes@engin.umich.edu`

23 June 1992

Updated 16 November, 1994

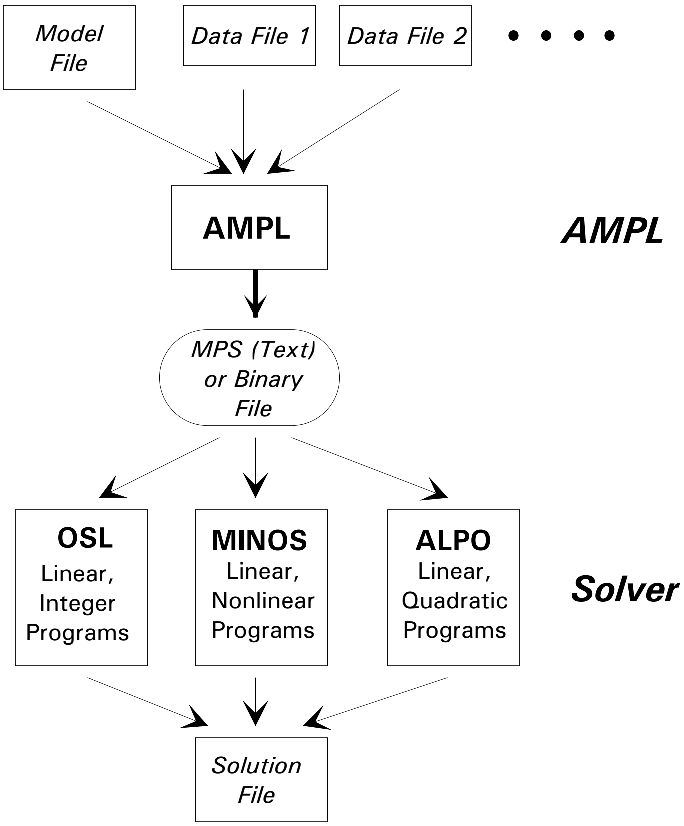
Updated 18 August, 1995

Contents

1 Introduction

AMPL (A Mathematical Programming Language) [6] is a high-level language for describing mathematical programs. AMPL allows a mathematical programming model to be specified independently of the data used for a specific instance of the model. AMPL's language for describing mathematical programs closely follows that used by humans to describe mathematical programs to each other. For this reason, modelers may spend more time improving the *model* and less time on the tedious details of data manipulation and problem solution.

A functional diagram of how AMPL is used is shown below. To start, AMPL needs a mathematical programming *model*, which describes variables, objectives and relationships without refering to specific data. AMPL also needs an instance of the data, or a particular data set. The model and one (or more) data files are fed into the AMPL program. AMPL works like a *compiler*: the model and input are put into an intermediate form which can be read by a *solver*. The solver actually finds an optimal solution to the problem by reading in the intermediate file produced by AMPL and applying an appropriate algorithm. The solver outputs the solution as a text file, which can be viewed directly and cross-referenced with the variables and constraints specified in the model file.



A commercial version of AMPL has been ported to most UNIX workstations at the University of Michigan's Computer Aided Engineering Network (CAEN). AMPL may be used with several different mathematical program solvers, including MINOS version 5.4 ([1]), ALPO ([8]), CPLEX citecplex) and OSL ([5]) version 2. AMPL may be used interactively or may be used in batch mode. Used interactively, a modeler may selectively modify both the model and its data, and see the changes in the optimal solution instantaneously. Used in batch mode, a modeler may obtain detailed and well-formatted solutions to previously defined mathematical programs. A shell script that may be used on any UNIX platform has been developed to simplify batch operation. Using this script, one can directly see the optimal solution to an AMPL problem without knowing UNIX or the details of the CAEN network.

This documentation is arranged into two sections. The first gives a brief introduction to the AMPL mathematical programming language. A more detailed description may be found in [6]. In addition, a tutorial by the authors of AMPL is available ([7]). The second section describes the use of AMPL and various solution packages on the CAEN network. Supporting technical details of the local AMPL interface are also presented.

2 AMPL: A Brief Description

AMPL stands for "A Mathematical Programming Language", and is a high level language that translates mathematical statements that describe a mathematical program into a format readable by most optimization software packages. The version of AMPL implemented here at the University of Michigan will currently model linear, mixed-integer, and nonlinear programs. The following description of the AMPL programming language presupposes some knowledge of the format and nature of mathematical programs.

A complete description of a specific mathematical program requires both a functional description of the relationships between problem data and the problem data itself. AMPL separates these two items into separate sections, the *model section* and the *data section*. These sections are usually in separate files (i.e. a separate model file and a data file), but may be combined into one or multiple files. The data section must always follow the model section.

NOTE: The following sections describe the most important features of AMPL. Many features of AMPL described in the referenced documentation are not described here. A complete reference may be found in the appendix of [6] available from the Engineering Library, call number QA402.5 F695

2.1 The Model Section

The general form for a linear program considered by AMPL is

$$\max \sum_{i \in I} c_i x_i \tag{1}$$

$$\text{st} \sum_{i \in I} a_{ij} x_i \leq b_j \qquad \forall j \in J \tag{2}$$

Here, c_i, a_{ij}, and b_j are all **parameters**, I and J are **Sets**, $\sum_{i \in I} c_i x_i$ is the **objective**, and $\sum_{i \in I} a_{ij} x_i \leq b_j$ are **constraints**.

The form of an AMPL model file closely follows that of the mathematical program stated above. Specifically, a model file is arranged into the following sections.

- Declarations using the following keywords:

```
set
param
var
arc
```

- Objectives declared with:

```
maximize
minimize
```

- Constraints

```
subject to
node
```

A prototype and example of each item is given below, followed by a full example of an AMPL model file. For the purposes of explanation, we will consider a production planning model. Specifically, we wish to find an optimal production schedule for a group of products given known demands, known (linear) costs, and known requirements for raw materials for each unit of product produced. The example considered is provided with the AMPL programming package.

2.1.1 Sets

Sets are arbitrary collections of objects that are assigned to variables or constraints. For example, in a production planning example, a set may be a group of raw materials or products. AMPL also allows sets of indexed integers to be used for the purposes of describing constraints or objectives. The prototype is:

`set` *[set name]*

For example, a production planning model optimizes production of a *set* of products that are made from a *set* if raw materials. AMPL groups these products and raw materials using the declarations

```
set P;          # Products
set R;          # Raw Materials
```

Note that a `#` comments out the rest of the AMPL line.

Let *setexp* be any valid set expression. Optional phrases that may be used when describing sets are

Keyword	Meaning
`dimen` n	Defines the dimension of the set
`within` (*setexp*)	Checks that declared set is a subset of (set expr.)
`:=` (*setexp*)	Specifies a value for the set.
`default` (*setexp*)	Specifies default value for set.
`ordered [ by [ reversed ]` *set* `]`	The defined set has an order (possibly defined by *set*).
`circular [ by [ reversed ]` *set* `]`	The defined set has a circular order (possibly defined by *set*).

The set used in the `ordered` keyword described above may be a real valued interval or an integral interval, using the set description `interval[a,b]` or `integer[a,b]`. Several functions that define and operate on sets are available, including `union`, `diff`, `symdiff`, `inter`(section), `cross` (cartesian product), `first(S)`, `last(S)`, and `card(S)`.

More detailed examples of sets are given below.

`set U;`	`set A := 1..n;`
`set B := i..j by k;`	`set C ordered`
`set D:= {i in C: x[i] in U};`	`set E := D diff U`
`set V within interval[i,j);`	`set W ordered by integer(a,b);`

2.1.2 Parameters

Parameters are scalars, vectors, or matrices of known data. These may include limits of index sets, rhs coefficients, matrix entries, etc. The prototype declaration for parameters is

`param` *[name]* { *index1, index2, ...* } *attributes..* ;

where *[name]* is a required identifier, *index1, index2, ...* are *optional* sizes or ranges for subscripts on the data, and *attributes* states attributes the parameter must have, such as the possible range of values the parameter may take. AMPL allows parameters to be defined in terms of other (previously defined) parameters.

The attributes assigned to a parameter may be specified using the following keywords:

Keyword	Meaning
`binary`	Parameter must be either 0 or 1.
`integer`	Parameter must be an integer.
`symbolic`	Allows non-numeric parameters.
`< <= = != > >=` *expression*	Parameter must satisfy expression.
`default` *expression*	Default in case parameter not defined in data section.
`in` *setexp*	Parameter must be in *setexp*

Examples relevant to our production planning problem include:

```
param T > 0;           # number of production periods
param M > 0;           # Maximum production per period
param a{R,P} >= 0;     # units of raw material i to manufacture
                       # 1 unit of product j
param b{R} >= 0;       # maximum initial stock of raw material i
param c{P,1..T};       # estimated profit per unit of product in period t
param d{R};            # storage cost per period per unit of raw material
param f{R};            # estimated remaining value per unit of raw material
                       # after last period
```

Note that a set of indices $1, 2, \ldots, T$ may be written as `{1..T}`. Also note that the conditions `>= 0` may be used to restrict the ranges parameters may take.

Parameters may also be recursively defined, as long as a parameter definition only involves the results of previously defined parameters. For example, the number of combinations of n items taken k at a time, $\binom{n}{k}$, may be defined using

```
param comb {n in 0..N, k in 0..n} :=
     if k = 0 or k = n then 1  else comb[n-1,k-1] + comb[n-1,k];
```

Piecewise linear terms may also be specified using AMPL. The prototype is `<<` *breakpoints* `;` *slopes* `>>` *var*. There must be one more slope than the number of breakpoints, and the variable is defined as below. For more information, please see the AMPL reference manual ([6]).

2.1.3 Variables

The decision variables of a problem are specified using the same conventions as parameters. The prototype declaration is

`var` *[name]* { *index1, index2, ...* } {*attributes*

;

The attributes that a variable may have include those specified in the following table:

Keyword	Meaning
`binary`	Variable must be either 0 or 1.
`integer`	Variable must be an integer.
`<= >= =` *expression*	Variable must satisfy expression (bound).
`:=` *expression*	Variable has initial guess *expression*.
`coeff` *constraint*	Specifies coefficient for a previously defined constraint. Used for column generation. See AMPL reference manual ([6]).

Two examples that arise from a production planning problem are:

```
var x{P,1..T} >= 0, <= M; # units of product manufactured in period
var s{R,1..T+1} >= 0;     # stock of raw material at beg. of period
```

Networks may be modeled explicitly using AMPL. A special type of variable is the *arc*, which carries flow and connects two *nodes*. The prototype of an arc definition is

`arc` *[name]* { *index1,..* } `from` { *index1,..* } *node*
`to` { *index1,..* } *node expression* `obj` { *index1,..* } *objective expression*

where the *expression* after `to` and `obj` defines the change in capacity along the arc or the cost (or profit) of a unit of flow along that arc. For example, suppose we were managing a portfolio of investments and wanted to minimize variable transaction costs along an arc describing a flow of money from one investment held for one period to another in the following period. The arc may be described using

```
arc xaction {t in 1..T,  i in investments[t], j in investments[t+1]} >= 0
   from i to j (Return[i,t]) obj cost (XactionCost[i,j]*Discount[t]);
```

An example of the same arc with piecewise linear costs would be

```
arc xaction {t in 1..T,  i in investments[t], j in investments[t+1]} >= 0
   from i to j (Return[i,t]) obj cost << cutoff[i] ; (LoXactionCost[i,j]*
   Discount[t]), HiXactionCost[i,j]*Discount[t]) >> xaction ;
```

2.1.4 Objective

The objective in a linear program is an inner product describing the function to be optimized. The AMPL prototype *for a linear program* is:

`maximize` *[objective name]* `:` `sum` { *[index]* `in` *[Index Set]* } *[Parameter]* `*` *[Variable]* + ... ;

More generally, nonlinear objectives, or objectives formed via column-generation may be formed. As an example, consider

```
maximize profit:
    sum {t in 1..T} ( sum {j in P} c[j,t]*x[j,t] - sum {i in R} d[i]*s[i,t] )
    + sum {i in R} f[i] * s[i,T+1];
                      # total over all periods of estimated profit - storage costs
                      # + value of raw materials left after last period
```

2.1.5 Constraints

Constraints are specified in much the same way as the objective function. However, the names of the constraints may be subscripted. A prototype of a linear constraint declaration is:

`subject to` *[constraint name]* { *[index]* `in` *Index Set*:
`sum` { *[index]* `in` *[Index Set]* } *[Parameter]* `*` *[Variable]* + ... { `<=` , `>=` ,`=` } *[Parameter]*

The semantics of linear constraints are best seen by examples. Using the production planning example, a balance constraint relating the inventory in one period to the next would be

```
stock {i in R, t in 1..T}:
    s[i,t+1] = s[i,t] - sum {j in P} a[i,j] * x[j,t];
        # stock in next period = present period - raw materials
```

More generally, a prototype of a general constraint is

`subject to` *name indexing* `[` *initial dual* `]` *constraint expression*

where *initial dual* is a guess as to the initial dual value of the constraint. Any linear or nonlinear expression involving any number of variables or parameters may be used. Several built in functions are available, including all trigonometric functions, other transcendental functions, and random number generation functions. [1]

Network constraints may be specified using the `node` keyword. Most nodes are transshipment nodes, i.e. their flow in must equal their flow out. These nodes are simply declared using `node`. However, a source or sink node must be declared differently. A node with exogenous flow in or out must also be declared differently. Flow in and out of these nodes is identified using the `net_in` and `net_out` keywords. For example, a transshipment problem with one factory, two customers, and several transshipment points would be declared using

[1] Distributions available are Beta, Cauchy, Exponential, Gamma, UniformInteger$[0, 2^24)$, Normal, Poisson, and Uniform.

```
node Factory: 0 <= net_in <= Max_production;
node Xshipnode { XShip_pts };
node Customer_node{ CustomerSet } : min_req <= net_out ;
```

There are many other constructs in the AMPL language which are not covered here. For a complete reference, please consult [6].

2.1.6 An example

Putting the examples presented in the previous sections together, a complete AMPL model file describing the production planning model would be:

```
# From Bob Fourer's TOMS paper, June 1983
# A factory can manufacture some number of different products
# over the next T production periods.  Each product returns a
# characteristic estimated profit per unit, which varies from
# period to period.  The factory's size imposes a fixed upper
# limit on the total units manufactured per period.  Additionally,
# each product requires fixed characteristic amounts of certain
# raw materials per unit.
#
# Limited quantities of raw materials must be stored now for use in
# the next T periods.  Each raw material has a fixed characteristic
# storage cost per unit period.  Any material still unused after
# period T has a certain estimated remaining value.
#
# What products should be manufactured in what periods to maximize
# total expected profit minus total storage costs, adjusted for
# the remaining value of any unused raw materials?

set P;                # Products
set R;                # Raw materials

param T > 0;          # number of production periods
param M > 0;          # Maximum production per period
param a{R,P} >= 0;    # units of raw material i to manufacture
                      # 1 unit of product j
param b{R} >= 0;      # maximum initial stock of raw material i
param c{P,1..T};      # estimated profit per unit of product in period t
param d{R};           # storage cost per period per unit of raw material
param f{R};           # estimated remaining value per unit of raw material
                      # after last period
var x{P,1..T} >= 0;   # units of product manufactured in period
var s{R,1..T+1} >= 0; # stock of raw material at beginning of period

maximize profit:
    sum {t in 1..T} ( sum {j in P} c[j,t]*x[j,t] - sum {i in R} d[i]*s[i,t] )
    + sum {i in R} f[i] * s[i,T+1];
                      # total over all periods of estimated profit - storage costs
                      # + value of raw materials left after last period
subject to
prod {t in 1..T}:
    sum {j in P} x[j,t] <= M;
        # production in each period less than maximum
stock1 {i in R}:
```

```
    s[i,1] <= b[i];
        # stock in period 1 less than maximum
stock {i in R, t in 1..T}:
    s[i,t+1] = s[i,t] - sum {j in P} a[i,j] * x[j,t];
        # stock in next period = present period - raw materials
```

2.2 The Data section

Since the same model may be used for many different instances of data, AMPL reads data specific to a problem from a separate section, denoted by the `data;` keyword. Data required by AMPL includes the names of all defined sets and the values of each parameter declared in the model section. The general form for an AMPL data file is

`data;`
`set` *[set name]* `:=` *[item 1], { item 2}, {item 3 } ...* `;`
`param` *[parameter name]* `:=` *[value1] ...* `;`
`end;`

As an example of a set declaration, suppose three different types of beer (lite, bud, and mich) were to be produced in the above production planning example. The data file statement to declare these items is

```
set P := lite bud mich ;
set R := malt hops ;
```

Parameter data may be given to AMPL in many convenient formats. A parameter may be specified as a scalar, a vector, a matrix, or a "matrix slice".

Consider the production planning example described above. We wish to describe the parameters `a` and `b`, which give the number of raw material units necessary for the production of each unit of finished product and their respective limits. Suppose the data is:

	Finished Product			
Raw Material	Lite	Bud	Mich	Limit
Malt	5	3	1	400
Hops	1	2	3	275

This data may be given to AMPL using the following formats:

1. **Scalars:** Each parameter element may be enumerated using its full identifier enclosed within brackets. For example, the parameter `a` may be given as:

```
param a [malt,lite] 5 ;
param [malt,bud] 3 ;
param [malt,mich] 1 ;
param [hops,lite] 1 ;
param [hops,bud] 2 ;
param [hops,mich] 3 ;
param b [malt] 400 [hops] 275 ;
```

2. **Vectors:** Parameters indexed by a single subscript may be declared using one parameter statement. For example, the parameter `b` may be declared as

```
param b [malt] 400 [hops] 275 ;
```

3. **Matrices:** Perhaps the most convenient method for supplying matrix coefficients is through a table. Tables of data may be entered directly into a data file using an editor or may be generated using a spread sheet. Tables are organized into columns and rows, with the relevant index names adjacent to each. Using the above data, the parameter `a` may be given as

```
param a : lite  bud  mich :=

   malt    5     3     1
   hops    1     2     3  ;
```

   It is also possible to describe the transpose of a matrix by including the `(tr)` modifier after the parameter name. This option is most useful when rows of a matrix are too long to work with conveniently. For example, the parameter `a` may also be specified as

```
param a (tr) :
         malt   hops :=
 lite     5       1
 bud      3       2
 mich     1       3  ;
```

4. **Matrix "Slices":** Matrices of dimensions greater than 2 are difficult to specify conveniently using the approach outlined above. So, it is possible to specify data along a "slice" of a matrix, or a specific entry along one (or two) of the matrix's dimensions. AMPL recognizes these slices by substituting an index name for a place holder (*) in the parameter name. The parameter `a` in the production planning problem may be specified as

```
param a :=

[malt,*] lite 5 bud 3 mich 1
[hops,*] lite 1 bud 2 mich 3 ;
```

   More than one place holder may be used in the declaration of a parameter. For example, `a` also could have been specified as

```
param a :=

[*,*]   malt,lite 5 malt,bud 3 malt,mich 1
        hops,lite 1 hops,bud 2 hops,mich 3 ;
```

Many other options for the description of data are available with the AMPL package. For a complete description, see the AMPL documentation ([6]). A complete data file for the production planning problem given above might look like:

```
data;

set P := lite bud mich ;
set R := malt hops ;

param T 3 ;
param M 40 ;
param a [malt,lite] 5 [malt,bud] 3 [malt,mich] 1
        [hops,lite] 1 [hops,bud] 2 [hops,mich] 3 ;
param b [malt] 400 [hops] 275 ;
```

```
param c [lite,1] 25 [lite,2] 20 [lite,3] 10
        [bud,1] 50 [bud,2] 50 [bud,3] 50
        [mich,1] 75 [mich,2] 80 [mich,3] 100 ;
param d [malt] 0.5 [hops] 2.0 ;
param f [malt] 15 [hops] 25 ;

end;
```

2.3 Using AMPL interactively

AMPL also includes many commands which may be used to organize and solve a model, and to customize AMPL's operation. When used interactively, AMPL's commands may be used to display the results of intermediate calculations or intermediate solutions to a sequence of mathematical programming problems.

To run AMPL interactively, copy the file `ampl_interactive` from the IOE area to your current directory by typing

```
azure% cp /afs/engin.umich.edu/group/engin/ioe/ampl/ampl_interactive .
```

and type `ampl_interactive`. You should receive an `ampl:` prompt. At this prompt, either quit ampl using `quit;` or you can use one of the following commands:

Keyword	Meaning
`model;`	Forces AMPL to accept model definitions.
`data;`	Forces AMPL to accept data.
`end;`	Forces AMPL to accept no more input for the current model.
`include` *filename*	Read in a separate file *filename*.
`display, print` *arglist*	Display or print a list of expressions.
`option` *envname, envvalue*	Change an option defined by *envname* to *envvalue*. See AMPL reference manual for list of options.
`solve`	Solve the currently defined problem.
`solution` *filename*	Loads a solver's solution from *filename*.
`write;`	Writes an MPS file or AMPL stub files.
`drop, add` *cons. name*	Instructs AMPL not to transmit (transmit) given constraint or objective.
`shell` ' *command line* ' ;	Performs shell command.
`reset;`	Clear all model declarations and data.
`quit`	Exit without writing any files.

Other commands may be found in the appendix of [6]. The `display` may be used to display or print any data from either the model or the solution. Pieces of constraints and solutions may be obtained using *dot notation*, which is very similar to structure notation in C. The dot notation for data is generically given as *constraint name.suffix* or *variable name.suffix*. Suffixes for variables include `init`, `lb`, `ub`, `val`, `rc`, and `slack`, while suffixes for constraints include `body`, `dinit`, `dual`, `lb`, `lslack`, and `uslack`. The slacks are always defined with respect to the bounds on the variables and on the constraints. For example, any constraint may be described as $lb \leq body \leq ub$, so *body*`.lslack` is $body - lb$. These values may also be used within the context of column or row generation schemes.

2.4 Controlling the solver from within AMPL

Solving particularly large or difficult problems may require a new algorithm (solver) or changes to the particular parameters used by a solver. To change solvers (from the default) type

`ampl: option solver` *solver* `;`

where solver is usually one of the following options:

Hardware	Program	Used for	Solver name to input
RS/6000	OSL	Integer, Linear Programs	*$ioe*/ampl/solvers_AIX/osl
RS/6000	MINOS	Nonlinear, Linear Programs	*$ioe*/ampl/solvers_AIX/minos
Sun	CPLEX	Linear, Integer Programs	*$ioe*/ampl/solvers_SunOS/cplex
Sun	MINOS	Nonlinear, Linear Programs	*$ioe*/ampl/solvers_SunOS/minos
HP	CPLEX	Linear, Integer Programs	*$ioe*/ampl/solvers_HP-UX/cplex [2]

where *$ioe* is `/afs/engin.umich.edu/group/engin/ioe`.

Most solvers are very flexible. Almost all of the parameters they use can be changed while using AMPL interactively. From the `ampl:` prompt (or within the model or data file), solver parameters can be changed by typing (for example)

`option osl_options "` *option 1, option 2,...,* `";`

Options are dependent on the particular solver used. For example, to limit the number of iterations used by OSL and to show solution times, the AMPL input line should be used:

```
ampl: option osl_options "maxiter 1000 timing 1";
```

Options for OSL and MINOS can be found in the following files:

```
/afs/engin.umich.edu/group/engin/ioe/ampl/doc/README.osl
/afs/engin.umich.edu/group/engin/ioe/ampl/doc/README.minos
/afs/engin.umich.edu/group/engin/ioe/ampl/doc/README.cplex
```

3 AMPL at the University of Michigan

AMPL is available for two popular workstations on the CAEN network (Suns and IBMS). This section will first describe a system that automates the use of AMPL. We will then describe how to run AMPL interactively or as stand alone package, and then how to use the solvers available with AMPL to find optimal solutions to the AMPL models. This section will be of particular interest to those who do not wish to bother with the operational details of the AMPL package or of the solver packages. However, familiarity with the CAEN system and some Unix exposure is assumed.

3.1 Using the IOE AMPL/Solver package AMPLUM

For those who do not wish to concern themselves with the technicalities of AMPL or of a particular optimization package, a shell script has been written to automate the generation of an optimal solution from a AMPL model and data files. This shell does the following:

1. calls AMPL to process the model and data file into an intermediate format.
2. calls a *solver* to actually find an optimal solution or determine infeasibility in the model.
3. calls a script which translates the solution obtained by the solverback to the original AMPL variable and constraint names.

(The technical details of this implementation are given in the next section.)

The shell script is named `amplum` and resides in the AFS directory `afs/engin.umich.edu/group/engin/ioe/ampl`. The prototype for the shell script `amplum` may be obtained by simply typing `amplum` without any arguments. Doing so gives:

```
amplum ampl_files.. output_file [-m] [-s solver] [-o output] [-d debug]

where:
ampl_files.. is a list of [readable] model and/or data files.
output_file  is an output file. [must not previously exist].
```

```
[solver]
         = minos : Default on all others. Solves LPs, NLPs.
         = alpo  : Available on all platforms. Solves LPs.
         = cplex : Available on suns.
         = ip    : OSL Integer Programming
         = lpsens      :  OSL Linear sensitivity analysis
         = lpintpt     :  OSL LP w/ Barrier Method & Simplex
         = lpintptnosw :  OSL LP w/ Barrier Method

[output] = 0 : No solution file, keep stub files.
         = 1 : Solution file, remove stub files [default].
         = 2 : Solution file, keep stub files.
         = 3 : Solution file w/ Sensitivity analysis, remove stub files. [OSL only]
         = 4 : Solution file w/ Sensitivity analysis, keep stub files. [OSL only]

[debug]  = 0 : AMPLUM passes critical information and soln [default]
         = 1 : AMPLUM passes all information from solver

 -m : MPS files and stub files only.
         NOTE: stub files are used by AMPL's  output processor.
shank%
```

NOTE: Some solvers may not be able to solve all types problems. In addition, not all solvers are available on all machine types. If a solver is chosen which is not available on the machine you are logged into, the following message is shown:

```
amplum mod.mod mod.dat oo -s osl

Option Not supported...
................. UM AMPL solver shell ................
Sorry, your choices are not currently supported. The
available solvers are given in the following table.
 Machine       Solver->   osl    minos   alpo   cplex
 -----------------------------------------------------
 RS/6000                LP,MIP  LP,NLP   LP
 SunOS                          LP,NLP   LP   LP,MIP
 -----------------------------------------------------
 Valid output options    0 - 4   0 - 2   0 - 2
 Please rerun shell with appropriate combinations
```

`amplum` *attempts* to insure that the appropriate solver is called for the type of problem being solved, but cannot distinguish difficult (i.e. nonlinear) models. For example, is there are binary or integer variables in the formulation, CPLEX or OSL are the *only* solvers which may be used. Likewise, MINOS is the only solver which will solve nonlinear problems.

For illustrative purposes, consider the production planning example given above and suppose the model and data sections are given in the files `mod.mod` and `mod.dat`, respectively. Then a solution to the AMPL file may be obtained by emulating (here, from a Sun) the sample shown below. Note that the CPLEX solver is being used.

```
shank% amplum mod.mod mod.dat mod.out

................. UM AMPL solver shell ................
 Solver: cplex
 Stub  : a_418.xxx
 Output file: mod.out
 Ampl files : mod.mod mod.dat
```

```
 ........... AMPL of 03Aug1995 times ..............
 .... AMPL: Processing MPS Files for formatter ....
 ................ LP/IP: CPLEX times .....................
 umcplex: Input file: a_418.mps      Output file a_418.cpx
        1.9 real         0.0 user         0.2 sys
shank%
cat mod.out

Selected  Objective Sense: MINIMIZE
Selected  Objective  Name: R0012
Selected  RHS        Name: B
 Problem Name     a_418.mps
 Objective Value  -140
 Status           Optimal
 Iteration        2

 Objective       profit                       (MIN)
 RHS              B
 Ranges
 Bounds
 ROWS

  NUMBER  ......ROW......  AT  ...ACTIVITY...  SLACK ACTIVITY  ..LOWER LIMIT.  ..UPPER LIMIT.  .DUAL ACTIVITY

       1 profit                BS        -140         140   -1.01E+75    1.01E+75           1
       2 prod[1]               BS           0          40   -1.01E+75          40          -0
       3 prod[2]               BS           0          40   -1.01E+75          40          -0
       4 prod[3]               UL          40           0   -1.01E+75          40         3.5
       5 stock1[malt]          BS         200         200   -1.01E+75         400          -0
       6 stock1[hops]          BS          40         235   -1.01E+75         275          -0
       7 stock[malt,1]         EQ           0           0           0           0        -0.5
       8 stock[malt,2]         EQ           0           0           0           0          -1
       9 stock[malt,3]         EQ           0           0           0           0        -1.5
      10 stock[hops,1]         EQ           0           0           0           0          -2
      11 stock[hops,2]         EQ           0           0           0           0          -4
      12 stock[hops,3]         EQ           0           0           0           0          -6
COLUMNS

  NUMBER  .....COLUMN....  AT  ...ACTIVITY...  ..INPUT COST..  ..LOWER LIMIT.  ..UPPER LIMIT.  .REDUCED COST.

      13 x[lite,1]         LL           0           25           0    1.01E+75        20.5
      14 x[lite,2]         LL           0           20           0    1.01E+75          11
      15 x[lite,3]         BS          40           10           0    1.01E+75           0
      16 x[bud,1]          LL           0           50           0    1.01E+75        44.5
      17 x[bud,2]          LL           0           50           0    1.01E+75          39
      18 x[bud,3]          LL           0           50           0    1.01E+75          37
      19 x[mich,1]         LL           0           75           0    1.01E+75        68.5
      20 x[mich,2]         LL           0           80           0    1.01E+75          67
      21 x[mich,3]         LL           0          100           0    1.01E+75          84
      22 s[malt,1]         BS         200         -0.5           0    1.01E+75           0
      23 s[malt,2]         BS         200         -0.5           0    1.01E+75           0
      24 s[malt,3]         BS         200         -0.5           0    1.01E+75           0
      25 s[malt,4]         LL           0           15           0    1.01E+75        13.5
      26 s[hops,1]         BS          40           -2           0    1.01E+75           0
```

```
      27 s[hops,2]            BS          40          -2           0    1.01E+75            0
      28 s[hops,3]            BS          40          -2           0    1.01E+75            0
      29 s[hops,4]            LL           0          25           0    1.01E+75           19
shank%
```

As can be seen above, the shell script substantially simplifies the modeling process, and provides quite readable output. Anyone is free to make changes to `amplum` for their own purposes, but does so at their own risk. AMPL is governed by a site license with AT&T (Scientific Press), however, so is there are any questions concerning AMPL's use or quoting its results, please check with the IOE department (`ckonrad@engin.umich.edu`).

As always, any comments or improvements to the shell script given above or any other aspect of the use of AMPL at the University of Michigan are greatly appreciated.

3.2 Using solvers directly

For particularly large or difficult problems, it may be desirable to exercise more control over a solver or to change the slgorithm (or solver) used. Most solvers use the *MPS* file format, which is an industry standard text-based format for specifying linear, integer, quadratic and stochastic programming problems.

To get an MPS file from an AMPL model, use `amplum` with the `-m` option. This file can be input directly with any solver that accepts MPS format files. (See the figure on Page 2.) For example, there is a completely customizeable front-end for OSL [3] which can be used with an MPS file. For the remainder of this section, let *$ioe* be `/afs/engin.umich.edu/group/engin/ioe`.

A front end also exists for MINOS. On line documentation for these front ends are in:

$ioe`/osl/doc/localosl.ps`
$ioe`/minos/minos.README`

Further technical details regarding the AMPL-solver interface are given in the next section.

Variable and constraint names in MPS files are limited to 8 characters each. So, AMPL replaces each "natural language" variable in the model file with a generic column name of the form (e.g.) `C0000001`. (Rows are also treated in this way). Any solution given by a solver will also refer to these pseudo-column names. To translate the row and column names in a solution file back to AMPL names, copy the files shown below to your working directory

Solver	Directory	File
OSL	*$ioe*`/ampl/perls`	`format_osl.perl`
MINOS	*$ioe*`/ampl/perls`	`format_minos.perl`
CPLEX	*$ioe*`/ampl/perls`	`format_gen.perl`

and type (e.g.)

`buteos% perl format_m.perl` ¡*Output file*¿ ¡*stub name*¿ `>` ¡*output filename*¿

where *stub name* is the name used as an output file when `amplum` was called. (Note: `format_m.perl` should also work for most other solvers.)

4 Technical Details

This section will describe some technical details necessary to customize AMPL or understand (or modify) the scripts associated with `amplum`.

4.1 How AMPL interfaces with solvers

AMPL has been linked directly to several production-quality mathematical programming solvers, including `cplex`, `osl`, `minos` (version 5.4), and `alpo`. AMPL may also be used to generate industry-standard MPS-format input files for use with other solvers that haven't been explicitly linked to AMPL.

When the solver is called with the `solve` command or as a separate program, AMPL writes the translated model and data into several *stub files*, each of which contains data describing the mathematical program. Stub files have the form *stub.xxx*, and are among those listed below:

File	Description	File	Description
stub.adj	constant adde to objective values.	*stub*.col	AMPL variable names.
stub.env	environment data.	*stub*.fix	Eliminated (fixed) variables.
stub.spc	MINOS "specs" file.	*stub*.row	AMPL row names.
stub.slc	Eliminated constraints.	*stub*.unv	Unused variables.
stub.mps	MPS model file.	*stub*.nl	Nonlinear data.

Directing AMPL which files to produce is covered in the next section.

Once the stub files have been written, AMPL passes the *stub* name followed by `-AMPL` to the solver. The solver is then expected to write a file *stub*`.sol` containing the optimal primal and dual solutions to the problem. AMPL then reads these in for further processing.

4.2 The AMPL package

AMPL is available on all Sun and RS/6000 AFS machines. The program is called `amplx`, and is located in one of the following directories.

Platform	Location
RS/6000	`afs/engin.umich.edu/group/engin/ioe/ampl/ampl_AIX`
Sun	`afs/engin.umich.edu/group/engin/ioe/ampl/ampl_SunOS`

NOTE: If you want to run AMPL repeatedly, putting the line (e.g.)
`setenv ampl = afs/engin.umich.edu/group/engin/ioe/ampl/ampl_AIX` in your `.cshrc` file allows `amplx` to be run by typing `$ampl/amplx`.

`amplx` may be run with several options. These options may be seen at any time from the UNIX prompt by typing `amplx "-?"`, and are shown below.

```
dice% cp /afs/engin.umich.edu/group/engin/ioe/ampl/ampl_AIX/amplx ./
dice% amplx "-?"
Usage: amplx [options] [file [file...]]
No file arguments means read from standard input, as does - by itself.
Options:        * = sets option keyletter_op (e.g. C_op for -C)
        -Cn {0 = suppress Cautions; 1 = default; 2 = treat as error;}*
        -D {print data read in (debug option)}
        -F {force generation of := sets (debug option)}
        -G {print generated data (debug option)}
        -L {fully eliminate linear definitional constraints and var = decls}*
        -M {print model (debug option)}
        -O {print compiled model (debug option)}
        -P {skip presolve -- same as "option presolve 0;" }
        -S {substitute out definitional constraints (var = expression)}*
        -T {show genmod times for each item}*
        -enn {exit at nn-th error: default 10 or, for stdin, 0 (no exit)}*
        -f {do not treat unavailable functions of constant args as variable}*
        -ooutopt {specify -o? for details}*
        -pnn {use nn decimal places in converting numbers to symbols}
        -q {always quote output literals (under -D, -G, -M, -O)}
        -s[seed] {seed for random numbers; -s means current time}*
        -t {show times}*
        -v {show version; -v? shows other -v options}
        -z {lazy mode -- treat := as default (evaluate only as needed)}*
dice%
```

Output files (see the section "How AMPL interfaces with solvers" above) are written by specifying an `-o` option to the `amplx` program. At any time, the output options may be obtained by typing `amplx "-o?"`.

```
dice% amplx "-o?"
        -o0 {no output}
        -o! {no genmod and no output}
        -obstub {generic binary format -- line -og, but binary
                after the first 10 lines}
        -oestub {nonlinear equations format (to be withdrawn -- superceded by -o b and -og)}
        -ogstub {generic ASCII format: no MPS file, no .spc file,
                full Jacobian in .nl file, otherwise like -omstub}
        -om {MPS format to stdout, no other files written}
        -omstub {MPS format to stub.mps,
                names to stub.row & stub.col,
                fixed vars to stub.fix,
                unused vars to stub.unv
                slack constraints to stub.slc
                objective adjustments to stub.adj,
                nonlinearities to stub.nl
                MINOS SPECS to stub.spc}
        -onstub {MPS format to <stdout>, otherwise like -omstub}
dice%
```

`amplx` may be run either interactively or in batch mode. However, a solver must be defined before problems can be solved. A list of the available solvers is given below.

Platform	Location	Solvers
RS/6000	afs/engin.umich.edu/group/engin/ioe/ampl/solvers_AIX	OSL, MINOS, ALPO
Sun	afs/engin.umich.edu/group/engin/ioe/ampl/solvers_SunOS	CPLEX[3] , MINOS, ALPO
HP	afs/engin.umich.edu/group/engin/ioe/ampl/solvers_HP-UX	MINOS, ALPO

The best way to inform AMPL as to which solver to use is to use the `setenv` command. For example, if an RS/6000 is being used with OSL, the command

```
setenv solver afs/engin.umich.edu/group/engin/ioe/ampl/solvers_AIX/osl
```

may be used to specify osl as the solver of choice for all future `amplx` runs. A sample interactive session that solves the problem developed in the previous sections is shown below. Comments are also provided.

```
dice% amplx                                  ( Set solver, note quotes )
ampl: option solver "/afs/engin.umich.edu/group/engin/ioe/ampl/solvers_AIX/osl"‘
ampl: include mod.mod                        ( Read in model file )
ampl: include mod.dat                        ( Read in data file )
ampl: solve;
OSL: optimal solution
primal objective 10564
14 simplex iterations
ampl: display prod[1].dual;                  ( Display parts of the solution )
prod[1].dual = 0

ampl: display stock.dual;
stock.dual :=
hops 1   25
hops 2   27
hops 3   29
malt 1   11
malt 2   11.5
malt 3   12
;
```

```
ampl: quit
dice%
```

The best way to get a feel for how this works is to obtain the reference documentation and to try it out.

4.3 AMPLUM details.

`amplum` is a shell script which completely processes AMPL input files using an appropriate solver. Assuming that enough files are specified on `amplum`'s command line, the script performs the following actions (in order):

1. Determine which files are input files and which is meant to be the output file. (Generally speaking the output file is always the last file in the command line list). Also process other arguments.
2. Determine the most appropriate solver to call, depending on the architecture `amplum` is being run on and the nature of the problem. (Note: this code in the script is a bit messy).
3. Prepend `option auxfiles rc;` to the model file and call the relevant version of `amplx`. If the `-m` option is specified, stop with AMPL MPS output.
4. Call the relevant solver (OSL LP, OSL IP, MINOS, ALPO) for the architecture being run on. [4]
5. Call an output Perl script (`format_osl.perl` or `format_minos.perl`, in `.../ioe/ampl/perls`) to translate the column and row names assigned by AMPL (for the solver) back to the "natural language" variable and cconstraint names specified in the model file.

All intermediate files (e.g. stub files) are of the form `a_`*xxxx*, where *xxxx* is the process number of the amplum script (determined at runtime). Unless the `-o 2` option is specified on the `amplum` command line, these files are deleted after the model has been solved.

As mentioned in the previous section, tranlating the solver results to natural languange (AMPL) identifiers is accomplished using Perl scripts located in `.../ioe/ampl/perls`. These scripts read in the relevant `.col` and `.row` stub files and substitute in the names found in those files where necessary in the solver output file. Further details can be found in the comments in each perl script.

References

[1] B.A. Murtaugh and M.A. Saunders, 1983. "MINOS 5.1 User's Guide," Technical Report SOL-83-20R, Systems Optimization Laboratory, Stanford University, Stanford, California.

[2] CPLEX User's Manual, 1995. CPLEX Corp., Incline Villange, NV.

[3] D. Holmes, 1991. "Optimization Subroutine Library at the University of Michigan (Documentation)."

[4] International Business Machines Corporation, 1990. "Optimization Subroutine Library," Licensed Program Numbers 5688-137, 5601-469, 5621-013.

[5] International Business Machines Corporation, 1990. "Optimization Subroutine Library Guide and Reference Manual," Publication SC23-0519.

[6] R. Fourer, D. Gay, and B. Kernighan, 1990. **AMPL: A Mathematical Programming Language**, Scientific Press, San Francisco, CA.

[7] D. Gay, 1992. "A Production Model: Maximizing Profits," tutorial available with AMPL package (Print using makedoc in the AMPL subdirectory).

[8] R. J. Vanderbei, "ALPO: Another Linear Programming Optimizer," ORSA J. Computing, to appear.

[4] Note:(11 November 1994). OSL does not appear to recognize general integer variables without specifying them in the `BOUNDS` section. The Perl script `fixip.perl` is called prior to problem solution to fix the MPS file to match the intent of the modeler.-dfh