BAMS 506 & 508: Using AMPL

AMPL is an algebraic modeling language for linear and nonlinear optimization problems, in discrete or continuous variables. AMPL also denotes the software that allows you to submit optimization models, written in the AMPL language, to appropriate solvers and retrieve the solver output. More generally, the AMPL software interprets your AMPL models and commands and manages communication with the selected solvers.

The freely downloadable[1] **AMPL Free Demo** consists of the AMPL software together with some solvers. It is limited to 500 variables and 500 constraints and objectives for linear problems, and to 300 variables and 300 constraints and objectives for nonlinear problems, and might not support user-defined functions, but is otherwise identical to the full "professional edition" of AMPL. Refer to the Quick Start instructions (AMPL downloads page) to unpack the AMPL program and the enclosed solvers and tools. The AMPL software may be run on your computer as a command-line interactive program, whereby you interactively enter AMPL commands to build and process an optimization model.

It is recommended to prepare your models, data and commands on separate computer files, using an editor or other software, and submit these files to AMPL.

Help for AMPL

The authoritative reference on AMPL is *AMPL: A Modeling Language for Math Programming*, by Robert Fourer, David M. Gay, and Brian W. Kernighan. The AMPL web site has a wealth of information, such as AMPL: A Mathematical Programming Language. Quoting from that website: "AMPL is a comprehensive and powerful algebraic modeling language for linear and nonlinear optimization problems, in discrete or continuous variables. Developed at Bell Laboratories, AMPL lets you use common notation and familiar concepts to formulate optimization models and examine solutions, while the computer manages communication with an appropriate solver. AMPL's flexibility and convenience render it ideal for rapid prototyping and model development, while its speed and control options make it an especially efficient choice for repeated production runs."

Now, see here, if you see below something in "`quotes;`", which you are supposed to type into AMPL, you *know*, deep down inside, that you shouldn't type the quotes into AMPL. But don't forget the semicolon.

[1] The AMPL software may also be run online, using the *Try AMPL* service on the AMPL website at www.ampl.com.

Getting started with AMPL

Generally you should have three files, a model file, a data file, and a run file. A **model file** contains the AMPL formulation for the model (and is often given the extension “`.mod`”), with its sets, parameters, variables, objective function, and constraints. This should be independent of the data. For example, the following model file, named `Product_Mix.mod`, defines a general product mix (resource allocation) model. The actual data (products, resources, prices and costs, technological data, and available resource amounts) defining a particular instance will be specified in a data file, as shown later. (You may want to copy the following example to a file named `Product_Mix.mod`.)

```
# The following sets and parameter values to be defined in a (.dat) data file:
set PRODUCTS;
set RESOURCES;
param materials_cost  {PRODUCTS} >= 0;  # $/unit
param usage_rate {RESOURCES, PRODUCTS} >= 0;
param available  {RESOURCES}     >= 0;  # available amounts
param selling_price  {PRODUCTS}  >= 0;  # $/unit
param resource_costs {RESOURCES} >= 0;  # $/unit of resource
param overhead;                        # sunk cost, $

# decision variables:
var Make {j in PRODUCTS}  >= 0;  # nonnegative

maximize Profit : sum{ j in PRODUCTS}
  (selling_price [j] - materials_cost[j]
   - sum {i in RESOURCES}(resource_costs[i]*usage_rate[i,j]) )
  * Make[j] - overhead;

subject to Resource {i in RESOURCES} :
   sum{j in PRODUCTS} usage_rate[i, j] * Make[j] <= available[i];
```

A **data file** contains the data for one instance of the model (and is sometimes given the extension “`.dat`” or “`.txt`”). For example, you may want to solve 5 different instances of a product mix problem, each with a different sets of products and/or resources. You simply put each instance in a different data file. Example (file `Oak_Doors.dat`):

```
set PRODUCTS := Solid_Doors  French_Doors;
set RESOURCES := Assembly Finishing Glass_Panels;
param materials_cost := Solid_Doors 8.50  French_Doors 11.50 ;
param usage_rate:    Solid_Doors French_Doors:=
     Assembly         0.1   0.2
     Finishing        0.4   0.3
     Glass_Panels     0     1   ;
param available :=    Assembly 1300  Finishing 3700 Glass_Panels
5000;
param resource_costs := Assembly  10 Finishing 10 Glass_Panels 0;
param selling_price := Solid_Doors 17.50    French_Doors 21.50 ;
param overhead  := 38000;
```

A **run file** contains a list of commands for AMPL to follow in solving the model (and is often given the extension "`.run`"). The run file tells AMPL which model file and which data file to use. The run files are used for your convenience, to avoid the need to type the same commands over and over into AMPL. You can also put in other commands to have AMPL display the solution. You can even develop algorithms with AMPL's command language.

The best way to get AMPL to solve is to create a run file, typically with just a few lines, a line for the model, a line for the data, and the solve command, as in the following example (file `Oak_Doors.run`):

```
reset;
model Product_Mix.mod;
data Oak_Doors.dat;
solve;
display Make;
```

This script tells AMPL

1. to erase all previous data;
2. to use the model described in Product_Mix`.mod` with the data in `Oak_Doors.dat`;
3. to solve the problem; and then
4. to show the values for the Make variables.

AMPL doesn't care about the file names or extensions. In fact, you could put everything into one file - model, data, and the "`solve;`" statement at the bottom. But putting the model, data, and commands into separate files will make understanding and communicating easier.

Basic AMPL syntax

- AMPL is case sensitive! That means that "`param y;`" and "`param Y;`" define two different parameters.
- AMPL ignores extra white spaces: "`param y;`" is the same as "`  param      y   ;`". You may want to use extra white spaces, indentation and tabs to structure your AMPL files and make them more readable.
- End statements with semicolons, "`;`".
- Watch your parentheses.
- You can comment out to the end of a line with a # comment.
- You can comment out a block with /* These commands are now ignored… */
- AMPL treats "=" and "`:=`" very differently, and it's important to understand when to use each of these:
  - Use "=" with "subject to" to define an equality constraint. Example: "`subject to Budget: x + y = 100;`"
  - Use "=" when specifing parts of sets. Example: "`subject to targetlevel {(c,v) in ROADS: c = 'Do_nothing'} sum{m in MAINT}`

`Area[c,v,m] <= initial[c,v];`". This tells AMPL to define the constraint `target_level` for all pairs `(c,v)`, where `c='Do_nothing'`.

- Use "`:=`" when specifying data. Example: "`param N := 16;`". This assigns the value 16 to the parameter `N`.
- Use "`:=`" when specifying parameters or variables in a script with the "`let`" keyword. Example: as part of a longer AMPL script, you might write "`let N := 16;`".

Commonly used commands

The **`reset`** command tells AMPL to erase any previous model and data so new ones can be read in.

The **`model`** command is used, at the AMPL command prompt or in a run file, to tell AMPL to use a file with a particular model in it. For example, "`model Product_Mix.mod;`" (at the AMPL command prompt or in your run file) tells AMPL that you intend to use the model in the file `Product_Mix.mod`. Frequently followed by a "`data`" command.

The **`data`** command is used to tell AMPL to use some data. The command can be used at least three different ways.

- First, you can put "`data;`" at the end of your model file, and follow it with the actual data you want to use, in the same file. This puts your model and data in the same file, which isn't so clever (but which does make the file compatible with the open-source GLPK solver package.)
- Second, you can type "`data Oak_Doors.dat;`" (if you have a product mix model) at the AMPL command line, usually right after the "`model`" command. This will tell AMPL that your product mix data is in the file `Oak_Doors.dat`.
- Third, you can type "`data Oak_Doors.dat;`" in your run file, again usually right after the "`model`" command, as in the "run file" discussion above. Again, this will tell AMPL that your product mix data is in the file `Oak_Doors.dat`.

The **`solve`** command tells AMPL to send a description of the optimization problem to a solver and to retrieve the results, in particular the optimal values of the decision variables. The best way to get AMPL to solve is to create a run file, typically with just 4 lines: a line with the `reset` command; a line for the model; one for the data; and one for the `solve` command. You may then add `display` and related commands and options, to see the optimum solution and related quantities, see below.

The **`display`** command tells AMPL to show the value of a parameter or variable, for example the solution found after a "`solve`" command. Example: "`display x;`" shows the value for the variable `x`. If `x` is a vector (e.g. was defined over a set), then AMPL will display all the values for `x`. Many of these will be zero, which isn't too interesting. If you

first give AMPL the command "`option omit_zero_cols 1;`" AMPL will show only the nonzeroes (see below).
Example: "`display x;`" shows the values for all components of the variable (vector) `x`.
Example: "`display x[1,2];`" shows the value for the variable `x[1,2]`.

You can also use "`display`" to show the reduced costs of nonnegative variables and the dual prices (Lagrange multipliers) and slack values of constraints in a linear or nonlinear program. (Remember, dual prices and reduced costs are not available in an integer program. The command might work, but the values should be considered rubbish.)
Example: "`display x.rc;`" shows the reduced costs of variable `x`.
Example: "`display Budget.dual;`", or simply "`display Budget;`", shows the dual price (Lagrange multiplier) for (all) the `Budget` constraint(s).
Example: "`display Budget[2017].slack;`" shows the slack of the `Budget` constraint for year 2017.

Some important AMPL key words

The **`set`** key word is used to tell AMPL that you have a bunch of related objects, such as cities, nodes, arcs, etc. You use sets most often in a model file. Once you have defined some sets, you can use the sets to define parameters and variables. See also the set-oriented functions below.

Example: to define in your model file a set of cities for a vehicle routing problem, you might write "`set Cities := {1..20};`". But it's better to put the "`20`" in a data file, so you can change the number of cities without changing the model file, like this: "`param N; set Cities := {1..N};`". Now you can define arcs between different cities (representing the possibility for a vehicle to go directly from city `i` to city `j`) as "`set Arcs := {i in Cities, j in Cities: i <> j};`" where the condition "`: i <> j`" is translated as "such that `i` not equal to `j`". If you have a symmetric problem, where the cost from `i` to `j` is the same as the cost from `j` to `i`, and thus the direction of travel does not matter, you could write "` : i < j`".

The **`param`** key word is used to define a parameter that is used for data. It is not a decision variable, and will not be calculated by the solver. Rather, a parameter is data that you give the model.

Example: "`param N;`" defines a parameter named `N`. Example: "`param Y >= 1 default 5;`" tells AMPL to define a parameter named `Y` with a value of 5, and that `Y` must always be bigger than or equal to 1. If you do not use the "`default`" key word, you should assign it a value somewhere (e.g. in a data or run file). Otherwise AMPL may complain that `Y` has not been assigned a value. Use the "`>=`" to make sure that your data is reasonable. If your script or data assigns it a data that violates this, AMPL will let you know.

If you assign the value to the parameter at the same time that you define the parameter (rather than in a data section, or with a "`let`" command), AMPL will give an error if you try to change the parameter elsewhere in your model or script. So specifying a parameter's value when the parameter is defined makes it permanent.

To avoid your own confusion, consider using 2 or more characters in every parameter, and consider using just one character in set indices. Example: you define “`param i default 0;`” and later write “`let y := sum {i in Foods} cost[i]*Buy[i];`”. This is bad, because AMPL will use the parameter `i` which is probably fixed to 0, when you really want `i` to be an ordinary counter. You should make your parameter names a bit more descriptive, anyway!

You do not have to define a parameter for an index, such as the `i` or `j` in “`minimize TotalCost: sum {(i,j) in Arcs} cost[i,j] * x[i,j];`”

The **`var`** key word is used to define a variable that will be calculated by the solver. A variable is not data that you give the model. Rather, it is represents a decision or value that you want to know, and are expecting AMPL to tell you when it has finished solving. Example: “`var w >=0;`” tells AMPL to define a non-negative decision variable named `w`. Example: “`var x >=-4 <= 5 integer;`” tells AMPL to define a decision variable named `x`, with a lower bound of `-4`, an upper bound of 5, and that `x` must be integer. In place of “`integer`”, you can use the keyword “`binary`” to restrict the variable to 0 or 1.

The **`minimize`** key word is used to declare a minimizing objective function. You must name your objective function. Example: “`minimize Pain: z;`”. AMPL can solve over only one objective at a time, so your mod file will usually have one objective. If you have a complex algorithm, where you want to change the objective during the algorithm, you may wish to use the **`problem`** key word, to define a named model.

The **`maximize`** key word is used to declare a maximizing objective function. You must name your objective function. Example: “`maximize Happiness: z;`”.

Don't confuse the “`minimize`” or “`maximize`” keywords with the “`min()`” or “`max()`” functions.

The **`subject to`** or **`s.t.`** key phrase is used to declare a constraint. You must name each constraint. Example: “`subject to Limit: x+y <= 5;`” or equivalently, “`s.t. Limit: x+y <= 5;`”. Often, you will want to declare a constraint over a set, and you specify the set as part of the constraint name. For example, in a symmetric vehicle routing problem, to require that each node (or city) be the endnode of exactly two arcs chosen by the (binary) variable `x`, i.e., has “degree 2” in this solution, type: “`subject to Degree {i in Nodes}: sum {(i,j) in Arcs} x[i,j] + sum {(j,i) in Arcs} x[j,i] = 2;`”. In this case, one constraint is defined for each `i` in `Nodes`. Cool, huh?

Inspecting your model

The **`show`** command lists the names of all components (sets, parameters, variables, constraints, objective) of your model.

Using
```
option show_stats 1;
```
before a solve cammand will produce summary statistics on the size of the optimization problem that AMPL generates. This may differ from the intended size of your model as AMPL uses a ‘‘presolve’’ phase that tries to make it easier for the solver. This may sometimes greatly reduce the size of your model, and sometimes even detect that it is infeasible and so it does not bother sending your model to the solver.

The **`xref`** command lists all model components that depend on a specified component, either directly (by referring to it) or indirectly (by referring to its dependents). If more than one component is given, the dependents are listed separately for each. Example:
```
xref Make;
```
produces the output
```
# 3 entities depend on Make:
Initial
Profit
Resource
```
where, by default, `Initial` refers to the current problem (unless you have defined a named problem).

The **`expand`** command tells AMPL to show an objective or constraint in long form. Example: suppose your model has the objective “`minimize TotalCost: sum {(i,j) in Arcs} cost[i,j] * x[i,j];`”. At the AMPL command prompt (or in your run file), you could type “`expand TotalCost;`”. AMPL would then display something like this:

```
minimize TotalCost: 210.117*x[1,2] + 1663.01*x[1,3] + 3845.68*x[1,4]
        +   2809.79*x[1,5]   +   3455.81*x[1,6]   +   1142.42*x[1,7]   +
4307.28*x[1,8] + 2503.7*x[1,9]
        + ...
        +  857.03*x[16,19]  +  611.945*x[16,20]  +  3185.03*x[17,18]  +
3379.88*x[17,19]
        +  4054.48*x[17,20]  +  393.102*x[18,19]  +  935.548*x[18,20]  +
675.246*x[19,20];
```

Only it wouldn't have the three dots, but would show the whole thing.

Example: suppose your model has the constraint “`subject to Degree {i in Nodes}: sum {(i,j) in Arcs} x[i,j] + sum {(j,i) in Arcs} x[j,i] = 2;`”. At the AMPL command prompt (or in your run file), you could type “`expand Degree[3];`”. AMPL would then display something like this:

```
subject to Degree[3]: x[1,3] + x[2,3] + x[3,4] + x[3,5] + x[3,6] +
x[3,7]
      + x[3,8] + x[3,9] + x[3,10] + x[3,11] + x[3,12] + x[3,13] +
x[3,14]
      + x[3,15] + x[3,16] + x[3,17] + x[3,18] + x[3,19] + x[3,20] = 2;
```

If you typed "`expand Degree;`", you would get all of the `Degree` constraints, one for each node.

Common problems with running AMPL

Problem: AMPL says *Error executing "commands" command: …is already defined*. **Solution**: all the sets and variables from your previous run are still in memory. Just type "`reset;`" before running your model to clear all the previous information.

Problem: AMPL says, *syntax error*.

Solution: AMPL should point to the line with the problem. Check the statement closely. Many times, a semicolon is missing, a parenthesis or bracket is missing, or you have misspelled a term. Remember that AMPL is case sensitive.

Problem: when you run AMPL, AMPL says, *error processing set ...: no value for ...* or *error processing param ...: no value for ....*

Solution: AMPL can't find your data, as it sees only the model file. Run AMPL from within a run file, not from a model or data file.

Problem: AMPL says that a parameter or variable *is undefined*.

Solution: you are using a parameter, variable, or subscript that AMPL doesn't understand. Example: "`minimize TotalCost: sum {(i,k) in Arcs} cost[i,j] * x[i,j];`" results in "*j is undefined*", because the `sum{}` statement has subscripts `i` and `k`, but no `j`. This could be corrected either as "`minimize TotalCost: sum {(i,j) in Arcs} cost[i,j] * x[i,j];`" or "`minimize TotalCost: sum {(i,k) in Arcs} cost[i,k] * x[i,k];`".

Commonly used options

The "`option`" command is used at the AMPL command prompt, in a run file, or at the beginning of a model file, to give any of a variety of instructions to AMPL. These include which solver to use, how output is displayed, and precision of calculation:

- Tell AMPL which solver to use, e.g.:
  `option solver cplex`; # This tells AMPL to use the linear and integer programming solver CPLEX.
  `option solver gurobi;` # Tells AMPL to use the LP and IP solver Gurobi.
  `option solver minos;` # Tells AMPL to use the nonlinear programming solver MINOS (this is the default solver).
  `option solver` *`your_ampl_compatible_solver_here`*`;` /* Perhaps you have another AMPL-compatible solver available, such as LP_Solve or IPOPT. */

- Show or omit the variables that are zero:
  `option omit_zero_cols 1;`
  `option omit_zero_rows 1;` # The default is 0, which shows everything.
  `option display_eps .000001;` # Treats every number between -`0.000001` and `0.000001` as zero in display commands.
- Round all output to *d* decimal digits after the decimal point:
  `option display_round 4;` # Rounds to 4 decimal digits
- Solve the continuous LP relaxation of an integer program. Zero is the default, so if you don't specify anything, models with variables defined as integer or binary are naturally solved as IPs.
  `option relax_integrality 0;` # Solve the integer program.
  `option relax_integrality 1;` # Solve the LP relaxation.
- Show output as a wide table rather than a tall table. By default, "`display`" tries to make a table tall rather than wide, e.g.,
  ```
  x x
  x x
  x x
  x x
  ```
  You may use
  `option display_transpose 0;` # This is the default setting. A wide table with 2 rows and 4 columns will be transposed, as shown above.
  `option display_transpose 1000;` # Makes most tables come out tall.
  `option display_transpose -1000;` # Makes them come out wide.
  In general, if (number of rows) - (number of columns) < `display_transpose` value, the table will be transposed.

- `option show_stats 1;` # Tells AMPL to show the number of variables and constraints in the model.
- `option presolve_eps 1e-9;` # Tells AMPL to use a precision of 10^{-9} when running its presolve.

Formatting data, and data files

Getting data formatted properly to satisfy AMPL can take some patience. In general, do it by following examples. Most often, you will want to define one, two, or three dimensional lists of parameters. We'll use the Traveling Salesman problem, or TSP (i.e., find the shortest closed route that visits every given node exactly once) as an example, with `N` nodes. Assume our model file has the following declarations:

```
  param N; # Number of nodes (or cities)
  set Nodes := {1..N};
  set Arcs := {i in Nodes, j in Nodes: i < j}; # Symmetric TSP: (i,j)is
the same as (j,i), so don't use both.
  param xcoord {Nodes}; # x and y coordinates of the nodes.
  param ycoord {Nodes};
  param length {(i,j) in Arcs} := sqrt((xcoord[i] - xcoord[j])^2 +
  (ycoord[i] - ycoord[j])^2); # Euclidean distances.
```

To declare `N`, `xcoord`, and `ycoord` in a data file, our data file might look like this:

```
data;
param N := 10;
param xcoord :=
1       0
2       0
3       3
4       6
5       5
6       3
7       1
8       6
9       4
10      2;
param ycoord :=
1       4
2       0
3       0.5
4       0
5       1.75
6       2.25
7       1.75
8       4
9       3.75
10      3.6;
```

Note that both `xcoord` and `ycoord` are defined over the list of `Nodes`, 1..10.

If we wish, we can use a more concise format, as shown below. Note the extra colon after the second occurrence of "`param`", to specify values for multiple parameters:

```
data;
param N := 10;
param: xcoord  ycoord :=
1       0       4
2       0       0
3       3       0.5
4       6       0
5       5       1.75
6       3       2.25
7       1       1.75
8       6       4
9       4       3.75
10      2       3.6;
```

If we don't care about x,y coordinates, but instead want to give a list of arc length, then the model and data may look like this:

```
  param N; # Number of nodes
  set Nodes := {1..N};
  set Arcs := {i in Nodes, j in Nodes: i < j}; # Symmetric TSP:
                     # (i,j)is the same as (j,i), so don't use both.
  param length {(i,j) in Arcs};
```

Then we need to declare `N` and `length` in a data file. Watch your colons! The dots are placeholders, that tell AMPL that this value is undefined.

```
data;
param N := 10;
param length [*,*]:
        2       3       4       5       6       7       8       9       10       :=
1       5       8       11      11      8       5       6       4       2
2       .       3       6       12      9       6       11      9       7
3       .       .       3       9       12      9       8       10      10
4       .       .       .       6       9       12      5       7       9
5       .       .       .       .       3       6       5       7       9
6       .       .       .       .       .       3       8       10      10
7       .       .       .       .       .       .       11      9       7
8       .       .       .       .       .       .       .       2       4
9       .       .       .       .       .       .       .       .       2
10      .       .       .       .       .       .       .       .       . ;
```

Now, if you wanted to have a 10,000 city TSP, the `length` data would be unwieldy in this format. If you wish, you can put data in "arc format", listing from-node, to-node, and length, like this:

```
  data;
  param N := 10000;
  param length :=
  1 2 5
  1 3 8
… and so on…
  1 10000 2
  2 3 3
… and so on…
  9999 10000 2;
```

LP Sensitivity Analysis

Some LP solvers, such as CPLEX or Gurobi allow you to perform sensitivity analysis beyond the reduced costs, dual variables and slack values, i.e., to also display the objective coefficients and RHS **ranges** for linear programs. For example, for the product mix model with decision variables `Make` and `Resource` constraints:

```
reset;
model Product_Mix.mod;
data Oak_Doors.dat;
option solver cplex;
option cplex_options 'sensitivity';
solve;
option display_eps .000001;
display {j in 1.._nvars} (_varname[j],_var[j],_var[j].rc,
      _var[j].current,_var[j].up,_var[j].down);
display {i in 1.._ncons} (_conname[i],_con[i].body,_con[i],
      _con[i].current,_con[i].up,_con[i].down);
```

produces the following output:

```
# $4 = _var[j].current
:       _varname[j]     _var[j] _var[j].rc  $4 _var[j].up _var[j].down
:=
1  "Make['Solid_Doors']"   7000       0       4   6.66667       2.5
2  "Make['French_Doors']"  3000       0       5   8             3
;

# $4 = _con[i].current
# $6 = _con[i].down
:      _conname[i]        _con[i].body _con[i]    $4  _con[i].up    $6
 :=
1  "Resource['Assembly']"        1300      16    1300     1550      925
2  "Resource['Finishing']"       3700       6    3700     5200     2700
3   "Resource['Glass_Panels']"   3000       0       0        0        0
;
```

which is similar to the Excel Solver sensitivity report, except that

1. CPLEX produces actual ranges instead of allowable changes; and
2. it does not correctly report the RHS and ranges for nonbinding constraints[2]

(Note also that some column headers are abbreviated as `$[column number]`, defined just before the corresponding table.)

You may also produce sensitivity analysis reports for selected model elements, e.g.,

```
  printf '\n'; printf 'Variables\n';
  display Make, Make.rc, Make.current, Make.up, Make.down;
  printf 'Constraints\n';
  display {i in RESOURCES : abs(Resource[i].slack) < 0.000001}
    (Resource[i].body, Resource[i], Resource[i].current,
     Resource[i].up, Resource[i].down);
```

which produces the following output:

```
Variables
:              Make Make.rc Make.current   Make.up Make.down    :=
Solid_Doors    7000     0         4        6.66667     2.5
French_Doors   3000     0         5        8           3
;

Constraints
# $3 = Resource[i].body
# $4 = Resource[i].current
:             $3   Resource[i]   $4  Resource[i].up Resource[i].down :=
Assembly    1300       16      1300       1550             925
Finishing   3700        6      3700       5200            2700
;
```

Formatting may be fine-tuned using the `printf` command.

[2] This is due to the fact that AMPL actually considers that every constraint has both a lower bound and an upper bound (each possibly infinite), and treats a constraint differently (as "inactive") when its value ("body") is strictly between these bounds.

The **`let`** command, "`let <name>:=<value>;`" assigns the variable or parameter named `<name>` the specified value. It is useful for making small changes to the LP data. Example:

```
  let available["Assembly"] := 1800;
  display available;
  solve;
  display Make;
```

For larger changes, use "`reset data; data <new data-file>;`".

Scripts and algorithms

For more information about AMPL's scripting, see:
Writing scripts in the AMPL command language,
Implementing algorithms through AMPL scripts,
Index to examples, looping and testing 1.
Index to examples, looping and testing 2.

The **`let`** key word is used in a script to assign a value to a parameter. Example: "`let N := 16;`". You can also use "`let`" to assign a value to a variable, but usually we want the solver to assign values to variables.

Flow control key words: **`for, repeat, while, break, if then else, printf`**

Named models: **`problem`**.

File input and output

Use the "<" and ">" symbols to temporarily connect a file to be input to AMPL, or output from AMPL to a file. Example: "`display x > myoutput.txt;`" sends the current value of `x` to the file `myoutput.txt`. The first command that specifies "`> myoutput.txt`" creates a new file by that name (or overwrites any existing file with the same name). Subsequent commands add to the end of the file, until the end of the session or a matching close command, "`ampl: close multi.out;`". To open a file and append output to whatever is already there (rather than overwriting) use ">>" instead of ">". Once a file is open, subsequent uses of ">" and ">>" have the same effect.

The **`close`** command tells AMPL to close the file. This releases the operating system lock on the file, so the file can be used by another program, such as AMPL's "`read`" command. It's always a good idea to close a file after writing to it or reading from it.

Trying to read a file that hasn't been closed may give an error such as "`unexpected end of file`".[3]

The **`read`** command can be used to read data from a file. Example:

```
 read N < TSPinput.txt; close TSPinput.txt;
```

if you have a file called "`TSPinput.txt`" with a single number, and you want AMPL to read the number from the file and put it into the parameter `N`. (Also, note the use of the "`close`" command.)

You can use the "`read`" command to make AMPL slightly interactive, so it asks you for a value. Use a minus sign for the file, and AMPL will read from the keyboard, waiting for you to type. Example:

```
  printf "How many nodes are in this TSP problem?\n";
          read N < - ;
```

Some commonly used AMPL functions

For help with strings, see Character Strings on AMPL.com.

Set oriented functions

`card``(set)` Returns the number of elements in the set. Example: "`let N := card(Nodes);`" sets the parameter `N` to be the number elements in the set `Nodes`.

`diff` Set subtraction. Example: "`let NotInTree := Nodes diff {v, u};`" assigns the set `NotInTree` to be the same as `Nodes`, but leaving out elements `v` and `u`.

`first``(set)` Returns the first element in the set.

`in` This operator specifies that the left hand side is contained in the right. Example: "`subject to Degrees {i in Nodes}: sum {(i,j) in Arcs} x[i,j] + sum {(j,i) in Arcs} x[j,i] = 2;`". This tells AMPL that the constraint is defined over all elements `i` that are contained in the set `Nodes`.

`A` **`inter`** `B` returns the intersection of `A` and `B`, which is the set of elements that are both in `A` and in `B`.

`last``(set)` Returns the last element in the set.

`member``(j, set)` Returns the element in position j of the set.

The **`max`** and **`min`** functions may be used over scalars `()` or over sets `{}`. Don't confuse these wee functions with the minimize and maximize key words, which are used to define objective functions.

[3] You can sometimes get away without closing files, because AMPL might close the file automatically (for example, if you exit from AMPL). But if your script writes to a file, then later reads from the file, you will have to close it immediately after writing to it or reading from it.

`max``(x,y[,z,...])` Returns the largest of the arguments. Example: "`let biggest := max(x[a,b], y[a,b]);`" which assigns to biggest the largest of two values.

`min``(x,y[,z,...])` Returns the smallest of the arguments. Example: "`smallest_length := min {(i,j) in Arcs} length[i,j];`" which scans the set of `Arcs`, and assigns to `smallest_length` the length of the smallest arc.

`next``(t, S, n)` Returns the element that is `n` positions after element `t` in the set `S`.

`ord``(t, S)` Returns the position of element `t` in the set `S`.

`ord0``(t, S)` Returns the position of element `t` in the set `S`, and returns 0 if `t` is not in the set `S`.

`prev``(t, S, n)` Returns the element that is `n` position before element `t` in the set `S`.

`union` This operator does set "addition". Example: "`let Nodes := Nodes union {i,j};`" adds elements `i` and `j` to set `Nodes`. Note the curly brackets, which tell AMPL that `i` and `j` are set elements. If you add sets with some common elements, the resulting set will not have duplicates.

`within` This operator specifies that one set is contained in another. Example: "`set Destinations within Cities;`" tells AMPL that each element in the set `Destinations` is also in the set `Cities`. If your data or script does not satisfy this condition, then AMPL will let you know.

Numerical functions

`abs``(x)` Returns the absolute value of `x`. Example: "`let y := abs(-3);`" will set `y` to 3.

`acos``(x)`,**`acosh`**`(x)`, **`asin`**`(x)`, **`asinh`**`(x)`, **`atanh`**`(x)` Inverse trigonometric and hyperbolic functions.

`ceil``(x)` Rounds `x` up to the nearest higher integer.

`cos``(x)` Returns the cosine of `x`.

`ctime``()` Prints the current time as a string. Try using "`printf "Start time %s.\n", ctime();`".

`exp``(x)` Returns e^x.

`floor``(x)` Rounds `x` down to the nearest lower integer.

`log``(x)` Natural log of `x`.

`log10``(x)` Log base 10 of `x`.

`a` **`mod`** `b` Returns the remainder of `a/b`. Ex.: "`let c := 17 mod 5;`" sets `c` to 2.

`sin``(x)` Returns the sine of `x`.

`sqrt``(x)` Returns $x^{0.5}$ (square root)

`tan``(x)` Returns the tangent of `x`.

`tanh``(x)` Returns tanh of `x`.

`Uniform01``()` Returns a pseudo-random number, uniformly distributed between 0 and 1. Example: "`let xcoord := 1000*Uniform01();`". Often used in a script, to assign random numbers to parameters.

Frequently asked questions

See also the AMPL FAQ page

Q: The solver runs very slowly, or the numerical result is inaccurate. What could be wrong?
A: You may be using the Minos solver, which is used by default. You can use CPLEX instead by adding this command to your run file:

```
option solver cplex;
```

Q: How do I get AMPL to display just the nonzero values of a variable x?
A: At the AMPL prompt, type:

```
ampl: option omit_zero_rows 1, omit_zero_cols 1;
ampl: display x;
```

Q: How can I view the reduced cost of a variable in AMPL?
A: To view the reduced costs of all variables named `x`, type

```
  display x.rc;
```

To view the reduced cost of a single variable, such as `x[5,7]`, type

```
  display x[5,7].rc;
```

Q: How do I get AMPL to take a `.RUN` file, such as `CUT.RUN`, from the AMPL command prompt?
A: At the ampl command prompt you may type

```
  ampl: commands cut.run;
```

Don't forget the semicolon.

Q: Is there a function in AMPL to print the output to a file?
A: To send output from the display command to a file, include in your `.RUN` file:

```
display x > myoutput.txt;
```

This will put the `x` values in the file `myoutput.txt`. Include the file absolute or relative address if it is not located in the current folder, e.g.

```
  display x > C:\MyAmplWork\myoutput.txt;
  display x > ..\..\myoutput.txt;
```

Q: `printf` command: What does the following line mean? I understand it prints the solution `x`, but I don't understand the `%i` and `\n`.

```
printf "x[%i] = %g\n", i, sum {k in 1..K} theta[k]*x_fxd[k,i];
```

A: The `printf` statement prints a string to the screen. For example,

```
  printf "hello";
```

does the obvious. On the other hand,

```
  printf "hello";
  printf "hello";
```
will show
```
  hellohello
```
on the screen. To get it to do a carriage return within "`hellohello`", we need to use
```
  printf "hello\n";
  printf "hello\n";
```
The "`\n`" tells AMPL to insert a carriage return.

We would also like to print data in the statement sometimes. If we want to print an integer such as 2005, we could use `%i` as a place-holder, to indicate where we want the data to go:
```
  let year := 2005;
  printf "The year is %i.\n", year;
```
AMPL should print
```
  The year is 2005.
```

If we want to print a floating point number, such as the number pi, we can use `%f` or `%g` as a place-holder to indicate where the data should go.
```
  let my_pi := 3.1415;
  printf "Today, I feel like my_pi = %g.\n", my_pi;
```
AMPL should print
```
  Today, I feel like my_pi = 3.1415.
```

If you try to use `%i` for a floating point number or vice versa, AMPL is likely to print rubbish.

More Advanced Questions:

Q: When I run the cutting stock problem, AMPL complains that my parameter fails an integrality check. What is wrong?

A: AMPL is probably using MINOS to solve the problem. MINOS does not solve integer programs.

You need to have AMPL call CPLEX instead. To make AMPL call CPLEX, add the following line at the beginning of your .RUN file:
```
  option solver cplex;
```

Q: When solving the decomposition, the master and subproblem don't converge, and so the algorithm keeps repeating. What could be wrong?

A: Check the objective value calculations. You could always solve the original linear program, so you know what the correct value is. That would tell you which one is wrong (subproblem or master). The use the display command to show the values, so you can see them as the algorithm runs. This will help determine when an error occurs.

Q: How do I get AMPL to make an MPS file?

A: Let's say your model is “`yourmodel.mod`” and your data is “`yourdata.dat`”. Then at the AMPL prompt, you would type:

```
  ampl: model yourmodel.mod;
  ampl: data yourdata.dat;
  ampl: write yourmps;
```

This will write an MPS file called “`yourmps.mps`” to the local directory. Unfortunately, all the row and column name information is lost. You can get AMPL to at least write out a file with a list of the names:

```
  ampl: model yourmodel.mod;
  ampl: data yourdata.dat;
  ampl: option auxfiles rc;
  ampl: write yourmps;
```

Now you get two extra files, “`yourmps.col`” and “`yourmps.row`”, which list the names of the columns and rows. You would have to match up the numbers with the names.

Based on documents prepared by Jon Lee (IBM T.J. Watson Research Center) and John F. Raffensperger (University of Canterbury, NZ).