AMPL: a quick-start guide

Claudia D'Ambrosio
dambrosio@lix.polytechnique.fr

1 Introduction

AMPL is an algebraic modeling language for linear and nonlinear optimization problems, in discrete or continuous variables. It allows the development of models and algorithms. It is linked to the most widely used solvers for LP/MILP/MINLP programming, which can be called for the AMPL environment to solve a given instance of the problem coded in AMPL.

AMPL with a lincense for the course can be found here: `http://www.lix.polytechnique.fr/~dambrosio/teaching/`. The instructions on how to install AMPL are presented in Section 8. The AMPL book, a very detailed manual, can be found here `https://ampl.com/learn/ampl-book/`

1.1 AMPL files

The AMPL user writes the formulation of the problem (s)he wishes to solve in a file that as an extension `.mod`. Each instance of the formulation is stored in a data file, which extension is `.dat`. The script or commands to declare in which mod file AMPL should look for the formulation and in which dat file AMPL should look for the instance data are either typed in the AMPL environment/console, or written in a file, which extension is `.run`.

To summarize, each problem instance is coded in AMPL using three files:

- a model file (extension .mod): contains the mathematical formulation of the problem.
- a data file (extension .dat): contains the numerical values of the problem parameters.
- a run file (extension .run): specifies the solution algorithm (external and/or coded by the user in the AMPL language itself).

In the following, we present the different AMPL commands/instructions and how they are used in the three different files.

Lastly, we specify that in AMPL a comment start with the character `#`. In case a character `#` is encoutered, the rest of the line is a comment.

2 Model file

We start with the model file, where the formulation is defined. Similarly to when we write a formulation on paper, the main elements that we need to define are:

- parameters, lines starting with the keyword `param`
- sets, lines starting with the keyword `set`
- decision variables, lines starting with the keyword `var`
- objective function(s), lines starting with the keyword `minimize` or `maximize`
- constraints, lines starting with the keyword `subject to`

2.1 Parameters

In the model file, the user defines the parameters used in the formulation in an abstract way. The numerical values corresponding to each parameters will be found in the dat file, for each formulation instance.

The keyword `param` starts the line concerning parameters, followed by the name of the parameters itself together with its dimension. For example, if we wish to define a 1-dimensional parameters called n we can use the following command:

```
param n;
```

We can add some conditions on the values of the parameters, to be sure some assumptions are respected. For example, if the parameter n is strictly positive, we can write instead:

```
param n > 0;
```

Finally, for defining an n-dimensional, strictly positive vector of parameter called w, we can use the following command:

```
param w{1..n} > 0;
```

Note that the indices in AMPL starts from 1 and not from 0.

Clearly, we can define as well matrices, for example, we define n, m, and an $n \times m$ matrix a in the following:

```
param n > 0;
param m > 0;
param a{1..n, 1..m};
```

Finally, note that it is possible to define a default value as well, namely:

```
param n > 0, default 10;
```

The default value has to satisfy the defined conditions, in the example above it has to be a value > 0. If a value for n is present in the .dat file, then the default value is overwritten.

2.2 Sets

For ease of notation, we can define sets in the AMPL model file. For example, we can define the set $N = \{1, \ldots, n\}$ as follow:

```
set N := 1..n;
```

Once a set is defined, it can be used to define the dimension of a parameters or decision variables vector, for example. The definition of w presented in the previous section can be replaced with the following:

```
param w{N} > 0;
```

or, equivalently,

```
param w{j in N} > 0;
```

The last option is useful if the conditions depends on other parameters, for example:

```
param w{1..n} > 0;
param p{j in N} <= 10*w[j];
```

if parameter p_j have to be not greater than ten time w_j for all $j = 1, \ldots, n$.

2.3 Decision variables

The decision variables are defined in lines starting with the keyword `var`. Similarly to the parameters, their dimension is defined thanks to parameters or sets and come conditions can be defined on them. Unlikely in the case of parameters, the conditions on their value are simple bounds on the variables themselves that the solver has to respect, together with generic constraints. Moreover, the use could define the variables `binary` or `integer`, i.e.:

```
var x{j in 1..n} >= 0, <= 1, binary;
```

Clearly, as in the parameters case, decision variables can show more than one index.

Note that, when one defines a variable, it is possible to define a default value as well, namely:

```
var x{j in 1..n} >= 0, <= 100, default 10;
```

The default value can take any value between the lower and the uppe bound, in the example above, any value between 0 and 100. The default value could correspond to a parameter value as well, if needed.

2.4 Objective function(s)

Usually, each formulation includes a unique objective function because standard mathematical optimization solvers can deal with just one objective function at time. However, AMPL provides the possibility to define more than one. The objective function declarations start with the `minimize` or `maximize` keyword as follows:

```
maximize total_profit:
  sum{j in N} p[j]*x[j];
```

where total_profit is the name of the objective function, sumj in N is the sum over j whose indices are in the set N and p[j]*x[j] is the simple product of the j-th element of parameter vector p and decision variables vector x.

2.5 Constraints

Constraints are defined thank to the keyword `subject to` followed by the name of the constraints and their quantifier. For example:

```
subject to capacity_constraint:
  sum{j in N} w[j]*x[j] <= c;
```

or

```
subject to random_constraint{j in 2..n}:
  w[j]*x[j] - w[j-1]*x[j-1]<= 1;
```

2.6 Example: the 01-Knapsack Problem

We present first the matematical optimization formulation of the 01-knapsack problem, followed by its AMPL model file.

$$\max \sum_{j=1}^{n} p_j x_j$$

$$\sum_{j=1}^{n} w_j x_j \leq c$$

$$x_j \in \{0,1\} \qquad j = 1, \ldots, n$$

```
param n > 0;        # number of items
set N := 1..n;
param p {j in N} > 0; # profits
param w {j in N} < 0; # weights
param c > 0;  # knapsack capacity


var x {j in N} >= 0, <= 1, binary; # variables

maximize total_profit: # objective function
  sum {j in N} p[j]*x[j];

subject to capacity_constraint: # constraint
  sum{j in N} w[j]*x[j] <= c;
```

3 Data file

A data file contains the data concerning an instance of a formulation. All the numerical values of the parameters and the sets have to be defined there. For a 1-dimensional parameter, it is possible to simply defining it as follows:

```
param n := 10;
```

As for the vector case, it is possible to define it as follows:

```
param: w :=
1        78.770199
2        77.468892
3        93.324757
4        96.180080
5        55.137398
6        40.101851
7        36.007819
8        5.317250
9        9.964929
10       60.265707
;
```

More details on how to define data files can be found in Chapter 9 of the AMPL book `https://ampl.com/learn/ampl-book/`. We conclude the session with an example of instance for the 01-knapsack problem.

```
param n := 10;
param c := 546.000000;

param: a :=
1        0.172274
2        0.134944
```

```
3       0.101030
4       0.163588
5       0.152350
6       0.196601
7       0.181208
8       0.126588
9       0.184087
10      0.187434
;

param: w :=
1       78.770199
2       77.468892
3       93.324757
4       96.180080
5       55.137398
6       40.101851
7       36.007819
8       5.317250
9       9.964929
10      60.265707
;

param: p :=
1       3.062328
2       43.280130
3       52.983122
4       62.101010
5       58.531125
6       47.574366
7       53.101406
8       6.902601
9       16.985577
10      62.576610
;
```

4 Run file

The run file in AMPL is basically the file containing the script that has to be run to solve the problem. It mainly contains the commands `model` and `data` which allow the user to define the model and the data file for a given formulation and instance. Then, it contains the reference to the solver which the user wishes to use to solve the instance of the formulation (`option solver` command) and the `solver` command that is needed for AMPL to call the solver.

Thus, a typical form of file looks like:

```
model myModel.mod;
data myDat.dat;
```

```
option solver gurobi;
solve;
```

where myModel.mod and myDat.dat are the model and the data files, respectively. Here, we assumed that both files are contained in the folder where ampl will be called. If this is not the case, a path can be always be defined as well, for example

```
model "../AMPL_example/myModel.mod";
```

In the following, we report other useful commands.

The keywords `reset` and `reset data` are useful when one runs several models/instances/run files one after the other in the AMPL environment. They usually appear at the beginning of the run file.

```
reset;
reset data;
```

Solver options: given a solver, specified thanks to the command `option solver`, we can modify its default options thanks to the command `option gurobi_options` before calling it, supposing the selected solver is gurobi. For example

```
option solver gurobi;
option gurobi_options "outlev 1";
solve;
```

For cplex, we will use `option cplex_options` and so on. The list of the available options can usually be found in the solver webpage (for example, you can google "ampl options solver cplex" for finding the list of options for the solver cplex).

Another useful command is `option relax_integrality`. The possible values are only 0 or 1: if it set to 1, then all the integrality requirements on the model are relaxed until its value is set again to 0, its default value. Suppose we are given an instance (coded in the file myInstance.dat) and a formulation (coded in the file myMILPmodel.mod) want to solve the continuous (or LP) relaxation of our instance and then the MILP problem we can do:

```
model myMILPmodel.mod;
data myInstance.dat;
option solver cplex;
option relax_integrality 1; # relaxing the integrality
    requirements on all the decision variables
solve;
option relax_integrality 0; # restoring the integrality
    requirements on all the decision variables
solve;
```

Finally, some commands concerning displaying/printing the values of the entities like parameters, sets, objectives, constraints, variables.

We can use display with any of the entities listed above, for example:

```
display n, c;
display N;
display w, p;
```

can result into something like:

```
n = 7
c = 19

set N := 1 2 3 4 5 6 7;

:    w     p      :=
1    11    10
2     6     3
3     6     4
4     5     5
5     5     6
6     4     7
7     1     2
;
```

Thus, the values corresponding to the defined instance are shown. The use of the `display` command on the decision variables makes sense only after a solver is run and it shows the values of the variables corresponding to the best solution find by the solver during the last `solve` command. The same for the objective function, `display` shows its value for the best solution find by the solver during the last `solve` command. For example:

```
display x;
display cost;
```

can result into something like:

```
x [*] :=
1  0.363636
2  0
3  0
4  1
5  1
6  1
7  1
;

cost = 23.6364
```

Similarly, when called wit a constraint name, it shows the value of the left-hand-side for the best solution find by the solver during the last `solve` command. For example:

```
display capacity_constraint;
```

can result into something like:

```
capacity_constraint = 0.909091
```

In order to show explicitly the constraints for a given instance, we can use the command `expand`, for example:

```
expand capacity_constraint;
```

can result into something like:

```
subject to capacity_constraint:
        11*x[1] + 6*x[2] + 6*x[3] + 5*x[4] + 5*x[5] + 4
```

An alternative to display is `printf`, thank to which we can define precisely the printing format.

```
printf "param n := %d;\n", n;
printf "\n";

# param c
printf "param c :";
for {j in N} {
  printf "\t%d", j;
}
printf "\t:=\n";
for {i in N} {
  printf "\t%d", i;
  for {j in N} {
    printf "\t%d", c[i,j];
  }
  printf "\n";
}
printf ";\n\n";
```

where `%d` allows us to print an integer number, while `%f` allows us to print a real number. The reader is referred again to the AMPL book for more details and options, also concerning strings manimulations.

We end the section by reporting an example of run file for the 01-knapsack problem:

```
#  Author: Claudia D'Ambrosio
#  Date: 20190121
#  nlkp.run

reset;
reset data;
# model file
model nlkp.mod;
```

```
# data file
data "/mypath/nlkp.dat";

option solver gurobi;
option gurobi_options "outlev 1";

# call the solver and provide it with
# the formulation and the instance data
solve > nlkp.out;
```

5 Other useful operators

There are plenty of other useful operators/keywords/commands in AMPL. We list here the most widely-used ones and refer the reader to the AMPL book for a more detailed list.

Conditional commands like `if then else`:

```
if x[1]== 0 then {
...
}
else {
...
}
```

The user might need to repeat instructions. The main looping commands are `for`, `repeat while`, `repeat until`

```
for {j in 1..n} {
...
}
```

or

```
repeat {
 . . .
  }
until x[n] > 0;
```

6 How to run AMPL to solve an istance of a mathematical optimization formulation

Now that we know how to create dat, mod, and run files, we discuss how we can run AMPL and call a mathematical optimization solver to solve an instance of a formulation represented by a pair dat/mod file.

We have two main options:

- use the AMPL environment/IDE
- use a command-line interface.

6.1 AMPL environment/IDE

If we have a run file where all the commands we wish to run are stored, then we can simply use the `include` keyword as follows:

```
ampl: include "../myfolder/myrunfile.run";
```

Note that to enter in the AMPL environment from the command-line interface we need to type `ampl`. To exit the AMPL environment we need to type `quit;`

6.2 Command-line interface

How to call AMPL from the command line:

```
Claudias-MacBook-Pro-8: ampl myrunfile.run
```

or

```
Claudias-MacBook-Pro-8: ampl myrunfile.run > myoutputfi
```

In the second case, all the screen output will be stored in the file called myoutputfile.out – which will be created if it does not exist.

7 Non linear knapsack problem

File nlkp.out obtained:

```
Log started (V12.9.0.0) Tue Apr 13 13:42:28 2021

Problem 'S1A.lp' read.
Read time = 0.01 sec. (0.22 ticks)
Tried aggregator 2 times.
MIP Presolve eliminated 363 rows and 20 columns.
MIP Presolve modified 289 coefficients.
Aggregator did 26 substitutions.
Reduced MIP has 1433 rows, 745 columns, and 8290 nonzeros.
Reduced MIP has 426 binaries, 0 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.02 sec. (7.86 ticks)
Found incumbent of value -87047.062500 after 0.13 sec. (17.36 ticks)
Probing fixed 12 vars, tightened 14 bounds.
Probing time = 0.01 sec. (3.80 ticks)
Tried aggregator 1 time.
MIP Presolve eliminated 27 rows and 24 columns.
MIP Presolve modified 5 coefficients.
Reduced MIP has 1406 rows, 721 columns, and 8020 nonzeros.
Reduced MIP has 414 binaries, 0 generals, 0 SOSs, and 0 indicators.
Presolve time = 0.01 sec. (4.48 ticks)
Probing time = 0.01 sec. (2.40 ticks)
Clique table members: 2770.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 4 threads.
Root relaxation solution time = 0.04 sec. (26.27 ticks)

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

*     0+    0                       -87047.0625   196841.1648          326.13%
      0     0   146592.8674   128   -87047.0625   146592.8674      774  268.41%
*     0+    0                       -18856.5625   146592.8674          877.41%
```

```
      0     0   143226.7335   115   -18856.5625     Cuts: 328      946  859.56%
      0     0   141330.8970    95   -18856.5625     Cuts: 156     1076  849.51%
      0     0   141078.6961    91   -18856.5625      Cuts: 82     1170  848.17%
      0     0   140976.2661    67   -18856.5625      Cuts: 51     1275  847.62%
*     0+    0                         67624.4514   140976.2661           108.47%
*     0+    0                         91823.6758   140976.2661            53.53%
      0     0   140962.7183    80    91823.6758      Cuts: 55     1341   53.51%
      0     0   140955.1845    87    91823.6758      Cuts: 35     1391   53.51%
      0     0   140954.4377    86    91823.6758      Cuts: 29     1428   53.51%
*     0+    0                        124609.1795   140954.4377            13.12%
      0     2   140954.4377    86   124609.1795   140954.4377     1428   13.12%
Elapsed time = 1.06 sec. (329.84 ticks, tree = 0.02 MB, solutions = 5)
*    11+    1                        124870.6640   140952.4371            12.88%
*   104+   66                        135835.4756   140910.7472             3.74%
*   362+  172                        136560.6915   140890.5641             3.17%
*   426   189      integral     0   136613.3802   140890.5641    12085    3.13%
*   806+  420                        136746.9103   140350.7946             2.64%
*   872+  460                        136849.0125   140302.2631             2.52%
*  1129+  564                        136964.1629   140195.7784             2.36%
   1138   598   137746.3561    43   136964.1629   140195.7784    25126    2.36%
*  1161+  564                        137008.0519   140195.7784             2.33%
*  1174+  564                        137027.8104   140195.7784             2.31%
*  1482+  741                        137198.4695   140039.3186             2.07%

Clique cuts applied:  21
Cover cuts applied:  2
Implied bound cuts applied:  64
Flow cuts applied:  26
Mixed integer rounding cuts applied:  69
Lift and project cuts applied:  6
Gomory fractional cuts applied:  32

Root node processing (before b&c):
  Real time             =    1.05 sec. (329.11 ticks)
Parallel b&c, 4 threads:
  Real time             =    2.79 sec. (643.23 ticks)
  Sync time (average)   =    0.62 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    3.84 sec. (972.34 ticks)

Solution pool: 16 solutions saved.

MIP - Aborted, integer feasible:  Objective =  1.3719846949e+05
Current MIP best bound =  1.3965664119e+05 (gap = 2458.17, 1.79%)
Solution time =    3.84 sec.  Iterations = 36640  Nodes = 1770 (891)
Deterministic time = 972.35 ticks  (252.96 ticks/sec)
```

8 To install AMPL

- Download one of the following .zip files:
  - `http://www.lix.polytechnique.fr/~dambrosio/teaching/ampl_linux-intel64.tgz`
  - `http://www.lix.polytechnique.fr/~dambrosio/teaching/ampl_macos64.tgz`
  - `http://www.lix.polytechnique.fr/~dambrosio/teaching/ampl_mswin64.zip`
- Follow installation instructions that can be found here.
- Available solvers: baron, conopt, gurobi, ilogcp, knitro, lgo, loqo, minos, snopt, xpress

9 References

- Modeling languages like ampl: `ampl.com`
  or gams: `www.gams.com` or jump `https://jump.dev/JuMP.jl/`

- AMPL Book: `https://ampl.com/learn/ampl-book/`
- AMPL examples: `https://ampl.com/learn/ampl-book/example-files/`
- Open source solvers like scip: `scip.zib.de`
- NEOS Server, State-of-the-Art Solvers for Numerical Optimization: `www.neos-server.org/neos/`