Dealing with MPS formats in Ampl, Cplex and Matlab environment

Stefano Nasini
Dept. of Statistics and Operations Research
Universitat Politècnica de Catalunya

1 Introduction

Mathematical Programming System (MPS) is a file format to represent LP and MILP which is generally accepted by all commercial LP solvers. With the acceptance of algebraic modeling languages MPS usage is declining, though its utility when simultaneously working with differen programming languages might still be high, as it is shown in this notes.

MPS is column-oriented (as opposed to entering the model as equations), and all model components (variables, rows, etc.) receive names. Sections of an MPS file are marked by so-called header cards, which are distinguished by their starting in column 1.

When working with AMPL, you can use AMPL's write command to create an MPS file. Because MPS form limits the row (constraint or objective) and column (variable) names to 8 characters, AMPL substitutes artificial names such as R0001 and C0007. The ordering of the names in these files corresponds to their numbering in the MPS file.

An MPS file contains only the nonzero values that define one instance of your model. Thus an MPS file generated by AMPL is mainly useful as input to solvers that do not yet have a direct AMPL interface; because MPS form has been in use for a longer time than any comparable format, it is recognized by more solvers than any other file type.

2 An illustrative example using the knapsack problem

Given a set of items, each with a size and associated payoff, the knapsack problem consist in determining the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

We are required to find a subset of the specified items such that the total size of the subset does not exceed the knapsack capacity, while maximizing the sum of the payoffs associated with the items. More formally, let $U = \{u_1 \dots u_2\}$ be a set of n items where each item u_i has a *size* s_i and a *cost* c_i associated with it. Let C, the capacity of the knapsack, also be specified as a part of the input. The goal is to find a feasible solution $U' \subseteq U$ such that $\sum_{u_i \in U'} s_i \geq C$ while minimize the total cost $\sum_{u_i \in U'} c_i$.

The problem often arises in resource allocation where there are financial constraints.

2.1 AMPL implementation

AMPL is an algebraic modeling language to describe and solve Mathematical Programming problems. The natural approach to program this described knapsack problem in AMPL is entered using syntax that is very close to the algebraic expressions of the LP formulation above. To start you need to create a `file.mod`, for example `knapsack.mod`, in which you should write the AMPL code, as shown below:

```
# AMPL model for the knapsack problem

param n;
set S := 1..n;

param s {S};
param c {S};

param C;

var x {S} binary;

minimize total_weight: sum {i in S} c[i] * x[i];

subject to capacity: sum {i in S} s[i] * x[i] >= C;

# data section

data;

param n = 8;
param C = 60;

param s :=
1 30    2 24
3 11    4 35
5 29    6  8
7 31    8 18;

param c :=
1 3     2 2
3 2     4 4
5 5     6 4
7 3     8 1;

write mknapsack;
```

The command `write mknapsack` allows to generate a MPS file containing all the information of the specified write knapsack problem. AMPL interprets "write m..." as indicating that you want to write an MPS file, and creates the filename by appending ".mps" to the letters after the "m". Thus in our example The file `knapsack.mps` will be created in the working directory. Calling CPLEX from AMPL the optimal solution is find as follows:

```
ampl: model knapsack.mod;
ampl: solve;
CPLEX 12.5.0.0: optimal integer solution; objective 6
2 MIP simplex iterations
0 branch-and-bound nodes
ampl: display x;
x [*] :=
1  1
2  1
3  0
4  0
5  0
6  0
7  0
8  1
;
ampl:
```

The optimal solution found states that only the first, second and eighth commodities must be included.

2.2 MPS format

Let us consider the file `knapsack.mps`, generated by AMPL. The main things to know about fixed MPS format are that it is column oriented (as opposed to entering the model as equations), and everything (variables, rows, etc.) gets a name.

```
NAME          knapsack
ROWS
 G  R0001
 N  R0002
COLUMNS
    INT1      'MARKER'                 'INTORG'
    C0001     R0001     30
    C0001     R0002     3
    C0002     R0001     24
    C0002     R0002     2
    C0003     R0001     11
    C0003     R0002     2
    C0004     R0001     35
    C0004     R0002     4
    C0005     R0001     29
    C0005     R0002     5
    C0006     R0001     8
    C0006     R0002     4
    C0007     R0001     31
    C0007     R0002     3
    C0008     R0001     18
    C0008     R0002     1
    INT1END   'MARKER'                 'INTEND'
RHS
    B         R0001     60
BOUNDS
 UP BOUND     C0001     1
 UP BOUND     C0002     1
 UP BOUND     C0003     1
 UP BOUND     C0004     1
 UP BOUND     C0005     1
 UP BOUND     C0006     1
 UP BOUND     C0007     1
 UP BOUND     C0008     1
ENDATA
```

The `NAME` record can have any value, starting in column 15. The `ROWS` section defines the names of all the constraints; entries in column 2 or 3 are E for equality rows, L for less-than (¡=) rows, G for greater-than (¿=) rows, and N for non-constraining rows (the first of which would be interpreted as the objective function). The order of the rows named in this section is unimportant.

The COLUMNS section contains the entries of the A-matrix. All entries for a given column must be placed consecutively, although within a column the order of the entries (rows) is irrelevant. Rows not mentioned for a column are implied to have a coefficient of zero.

The RHS section allows one or more right-hand-side vectors to be defined; there is seldom more than one. In the above example, the name of the RHS vector is RHS1, and has non-zero values in all 3 of the constraint rows of the problem. Rows not mentioned in an RHS vector would be assumed to have a right-hand-side of zero.

The optional BOUNDS section specifies lower and upper bounds on individual variables, if they are not given by rows in the matrix. All the bounds that have a given name in column 5 are taken together as a set. Variables not mentioned in a given BOUNDS set are taken to be non-negative (lower bound zero, no upper bound). A bound of type UP means an upper bound is applied to the variable. A bound of type LO means a lower bound is applied. A bound type of FX ("fixed") means that the variable has upper and lower bounds equal to a single value. A bound type of FR ("free") means the variable has neither lower nor upper bounds and so can take on negative values. A variation on that is MI for free negative, giving an upper bound of 0 but no lower bound. Bound type PL is for a free positive for zero to plus infinity, but as this is the normal default, it is seldom used. There are also bound types for use in MIP models - BV for binary, being 0 or 1. UI for upper integer and LI for lower integer. SC stands for semi-continuous and indicates that the variable may be zero, but if not must be equal to at least the value given.

Another optional section called RANGES specifies double-inequalities, in a somewhat counterintuitive way not described here. Ways to mark integer variables are also beyond the scope of this article(keyword MARKER and possibly SOS are involved). The final card must be ENDATA (notice the odd spelling).

A few special cases of the MPS standard are not consistently handled by implementations. In the BOUNDS section, if a variable is given a nonpositive upper bound but no lower bound, its lower bound may default to zero or to minus infinity (also, if the upper bound is given as zero, the lower bound might be zero or negative infinity).[3] If an integer variable has no upper bound specified, its upper bound may default to one rather than to plus infinity.

3 Read MPS in Matlab

There are different ways of reading MPS files in Matlab. We first consider the package `readmps`, published by Dr. Brian Borchers in http://infohost.nmt.edu/~borchers/ and providing MATLAB routines to read data files in the MPS format. The code can handle most common variations on the MPS format, including linear and integer programming problems and problems with a quadratic objective function. The code is available in http://infohost.nmt.edu/~borchers/readmps.html.

```
>> problem = readmps('knapsack.mps')

problem =

           name: 'knapsack'
       objsense: 'MINIMIZE'
        objname: ''
         refrow: ''
       rownames: {'R0001'  'R0002'}
       rowtypes: {'G'  'N'}
    columnnames: {'C0001'  'C0002'  'C0003'  'C0004'  'C0005'  'C0006'  'C0007'  'C0008'}
     boundnames: {'BOUND'}
       rhsnames: {'B'}
     rangenames: {}
          lbnds: [0 0 0 0 0 0 0 0]
          ubnds: [1 1 1 1 1 1 1 1]
            rhs: [2x1 double]
         ranges: []
      bintflags: [0 0 0 0 0 0 0 0]
       intflags: [1 1 1 1 1 1 1 1]
      sos1flags: []
      sos2flags: []
      sos3flags: []
              Q: []
              A: [2x8 double]
       rowtable: [1x1 struct]
       coltable: [1x1 struct]
     boundtable: [1x1 struct]
       rhstable: [1x1 struct]
     rangetable: [1x1 struct]

>>
```

Thus, when using `readmps` the fields of the problem output are

`name`	`Problem name.`
`objsesense`	`'MINIMIZE', 'MIN', 'MAX', or 'MAXIMIZE'.`
`objname`	`Name of the objective function row.`
`problem.refrow`	`Name of the reference row for SOS's.`
`rownames`	`Cell array of row names.`
`rowtypes`	`Cell array of row types ('L','G','N','E').`
`columnnames`	`Cell array of column names.`
`boundnames`	`Cell array of names of bounds.`
`rhsnames`	`Cell array of names of right hand sides.`
`rangenames`	`Cell array of names of ranges.`
`lbnds`	`Sparse array of lower bounds.`
`ubnds`	`Sparse array of upper bounds.`
`rhs`	`Sparse array of right hand sides.`
`ranges`	`Sparse array of ranges.`
`bintflags`	`Sparse array of flags. bintflags(i,j)=1 if column j in bound set i is an integer column in bound set i (different bound sets might have different integer columns.).`
`intflags`	`intflags(j)=1 if column j is an integer column.`
`sos1flags`	`sos1flags(j)=1 if column j is in an SOS1.`
`sos2flags`	`sos1flags(j)=1 if column j is in an SOS2.`
`sos3flags`	`sos1flags(j)=1 if column j is in an SOS3.`
`Q`	`Sparse array of quadratic objective function coefficients.`
`rowtable`	`hash table for row names.`
`coltable`	`hash table for column names.`
`boundtable`	`hash table for bound names.`
`rhstable`	`hash table for right hand side names.`
`rangetable`	`hash table for range names.`

Another way of reading MPS files from Matlab is provided by the Cplex Class AP for Matlab. The Cplex class stores the model and provides methods for the solution, analysis, manipulation and reading/writing of the model file. The filename must end in one of these suffixes: `.lp`, `.mps`, `.sav` and `.gz`. All of the data associated with the problem is stored in the properties of a Cplex object. These class properties are standard Matlab data structures and can be manipulated directly within Matlab.

The properties of the Cplex class include:

`Cplex.Model`	`stores the data of the model`
`Cplex.Solution`	`stores the solution of the model`
`Cplex.Param`	`stores the parameters (options) of the model`
`Cplex.Start`	`stores the start of the LP model`
`Cplex.MipStart`	`stores the start of the MIP model`
`Cplex.InfoCallback`	`pointer to an informational callback`
`Cplex.Conflict`	`stores the conflict information of a conflicted model`
`Cplex.DisplayFunc`	`pointer to a function which provides control of display of output`

The following informative methods are provided:

`Cplex.getVersion`	`returns the CPLEX version`
`Cplex.getProbType`	`returns the problem type of the model`

The following methods are provided for reading from and writing to files:

```
Cplex.readModel
Cplex.writeModel
Cplex.readBasis
Cplex.writeBasis
Cplex.readMipStart
Cplex.writeMipStart
Cplex.readParam
Cplex.writeParam
Cplex.writeConflict
```

The following methods are provided to solve and analyze the model, solution and mipstart:

```
Cplex.solve
Cplex.populate
Cplex.feasOpt
Cplex.refineConflict
Cplex.refineMipStartConflict
Cplex.terminate
```

The following methods are provided to solve, set and query parameters:

```
Cplex.tuneParam
Cplex.setDefault
Cplex.getChgParam
```

Although a model can be modified by manipulating the MATLAB data structures directly, the following functions are provided to make modifications easier:

```
Cplex.addCols
Cplex.addRows
Cplex.delCols
Cplex.delRows
Cplex.addSOSs
Cplex.addQCs
Cplex.addIndicators
```

We consider the function `cplex.readModel(filename)`, which provides a method of the Cplex class that reads a model from a file and copies it into the problem object. The Cplex MPS file reader is highly compatible with files created by other modeling systems that respect the MPS format. There is generally no need to modify existing problem files to use them with CPLEX.

For the aforementioned case of the knapsack problem the the function `cplex.readModel(filename)` can be used as follows:

```
>> problem = Cplex('knapsack.mps')
>> problem = readModel('knapsack.mps')
   Selected objective sense:  MINIMIZE
   Selected objective  name:  R0076
   Selected RHS        name:  B
   Selected bound      name:  BOUND
>>
```

4 Integer Programming Problems in Matlab

A commonly used function to solve binary programming problems, such as the knapsack problem, is `bintprog()`. It implements an LP-based branch-and-bound method and represent an efficient and gentile alternative to dealing with this kind of problems.

```
>> problem = readmps('knapsack.mps');
>> A = problem.A(1,:);
>> c = problem.A(2,:)';
>> b = problem.rhs(1,1);
>> lb = problem.lbnds;
>> ub = problem.ubnds;
>> [x1 fval, exitflag,output] = bintprog(c,[],[],A,b);
>> output

output =

          iterations: 30
               nodes: 35
                time: 0.3700
           algorithm: 'LP-based branch-and-bound'
      branchStrategy: 'maximum integer infeasibility'
    nodeSrchStrategy: 'best node search'
             message: 'Optimization terminated.'
```

There are two ways to use CPLEX in MATLAB: 1) a toolbox of functions and 2) a class API. The toolbox contains functions for solving optimization problems, where the input matrices are provided to the function and results returned.

```
>> problem = readmps('knapsack.mps');
>> A = problem.A(1,:);
>> c = problem.A(2,:)';
>> b = problem.rhs(1,1);
>> lb = problem.lbnds;
>> ub = problem.ubnds;
>> [x1 fval, exitflag,output] = cplexbilp(c, [], [], A, b);
>> output

output =

          cplexstatus: 101
    cplexstatusstring: 'integer optimal solution'
           iterations: 4
            algorithm: 12
                 time: 0.0181
              message: 'Function converged to a solution x.'
```

With the class API, objects can be created, and those objects carry a state. The benefits of using the Cplex class API include the ability to:

- build up a model by manipulating a Cplex object.
- use computation methods such as Cplex.solve() and Cplex.refineConflict() that modify the object so results can be queried as needed.
- perform restarts after manipulation.
- attach an output parser, a GUI with stop buttons, and other controls.

```
>> p = Cplex()

ans=

   Cplex handle
   Properties:
      Model: [1x1 struct]
      Param: [1x1 struct]
      DisplayFunc: @disp

   Methods, Events, Superclasses
>> p.Model

ans =

    sense: 'minimize'
      obj: []
       lb: []
       ub: []
        A: []
      lhs: []
      rhs: []
     name: 'CPLEX'

>>

p = Cplex()

ans=

   Cplex handle

   Properties:
      Model: [1x1 struct]
      Param: [1x1 struct]
      DisplayFunc: @disp

   Methods, Events, Superclasses
>> p.Model

ans =

    sense: 'minimize'
      obj: []
       lb: []
       ub: []
        A: []
      lhs: []
      rhs: []
     name: 'CPLEX'

>> p.Model.A = problem.A(1,:);
>> p.Model.obj = problem.A(2,:)';
>> p.Model.rhs = problem.rhs(1,1);
>> p.Model.lhs = problem.rhs(1,1);
>> p.Model.lb = problem.lbnds;
>> p.Model.ub = problem.ubnds;

p.Model

ans =

    sense: 'minimize'
      obj: [8x1 double]
       lb: [0 0 0 0 0 0 0 0]
       ub: [1 1 1 1 1 1 1 1]
        A: [30 24 11 35 29 8 31 18]
      lhs: 60
      rhs: 60
     name: 'CPLEX'
    ctype: 'B'

>> p.solve;
Tried aggregator 1 time.
Probing time =    0.00 sec.
Tried aggregator 1 time.
Presolve time =    0.00 sec.
Probing time =    0.00 sec.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
```

```
Parallel mode: deterministic, using up to 24 threads.
Root relaxation solution time =    0.00 sec.

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

*     0     0      integral     0        4.7419        4.7419        1    0.00%
Elapsed real time =   0.02 sec. (tree size =  0.00 MB, solutions = 1)

Root node processing (before b&c):
  Real time             =    0.01
Parallel b&c, 24 threads:
  Real time             =    0.00
  Sync time (average)   =    0.00
  Wait time (average)   =    0.00
                          -------
Total (root+branch&cut) =    0.01 sec.
```