An Introduction to the Modeling Language LPL

Tony Hürlimann
info@matmod.ch

June 7, 2022

Abstract

This paper gives a first impression of the LPL language. The paper is by no means complete or systematic. For learning LPL, I suggest to read through the tutorial [2] and consult the reference manual [5]. Surely, after digesting this paper, the reader would be able to read and understand with ease most LPL models and would be able to write some basic models.

After an introduction to the basics, the paper exposes highlights several unique features of LPL such as modularity, the drawing library, logical constraint, and goal programming.

It would be an advantage, if the reader were familiar with the basics of some programming language to follow smoothly all the small models in this text.

1 Introduction

There are virtually hundreds of software and computer tools to implement various kinds of models. There exist special software for particular domains, or there are general tools for symbolic and algebraic manipulations or for numerical solutions. There are also extensions of modern established programming languages to formulate and solve models. Finally, various algebraic languages exist. Several options and tools are presented in a separate paper (see [6]).
LPL is an advanced modeling language and seems to me the right tool to start learning mathematical modeling. Its syntax is close to the common mathematical notational and at least its basics are easy and quick to learn. With LPL one can formulated small and large linear and nonlinear models. It is linked to various free and commercial solvers. And one of its unique feature is to formulate discrete models using Boolean and logical operators. It is ideal in an educational environment. Aside from that, large linear and integer models are also implemented and used by large companies such as ABB (see paper) and SwissPort (see paper).

Furthermore, for testing and run your first models, there is no need to install any software on your computer, just use your favorite browser

to start modeling. Later on, when you need serious error handling and interactivity in model building, an academic version of the LPL software can be downloaded for free and is installed in no time. It is shown later on in this paper.
Learning the basics of LPL is no wasted time. The knowledge is also useful to switch to other, and for your application more appropriate software or tools. So let's start!

2 The Basics

Here is a small model to be implemented in LPL: A Simple Model

$$\begin{array}{ll} \max & 300x + 200y \\ \text{subject to} & 5x + 5y \leq 350 \\ & 6x + 2y \leq 300 \end{array}$$

The implementation in LPL of this model is straightforward:

```
model firstModel;
  variable  x; y;
  constraint C: 5*x + 5*y  <=  350;
    D: 6*x + 2*y  <=  300;
  maximize Obj: 300*x + 200*y;
  Writep(x,y);
end
```

1. Every model begin with the keyword **model** followed by an identifier and a semicolon and ends with keyword **end**.
2. The LPL model consists of five declarations and a statement.
3. Two variable declaration with the name x and y begin with keyword **variable**. If the same declaration repeats, the keyword can be dropped.
4. Two constraint declarations with name C and D followed by a colon and an expression. Note that constraints always have a name.
5. A maximizing declaration with name Obj also followed by a colon and an expression.
6. Finally, a statement, a function call *Writep* to print the two variables with their value.

Copy this model and paste it here: empty (A browser opens). Then click on the button *Run and Solve*. A moment later, the solution of this model is displayed: $x = 40, y = 30$.

To formulate larger models with thousands of variables and constraints, we use a notation in mathematics that is called *indexed notation*[1]. The following is a linear model with $m > 0$ constraints and $n > 0$ variables. Indexed Notation

[1] If the reader is not familiar with indexed notation, study the following paper [3].

$$
\begin{array}{lll}
\max & \sum_{j\in J} c_j x_j & \\
\text{subject to} & \sum_{j\in J} a_{i,j} x_j \le b_i & \text{forall } i \in I \\
& x_j \ge 0 & \text{forall } j \in J \\
\text{with} & I = \{1,\ldots,m\}, \quad J = \{1,\ldots,n\} & m,n \ge 0
\end{array}
$$

One of the main strength of LPL is to use the index notation. Data can be declared by parameters :

```
parameter m := 1000;
  n := 2000;
```

The keyword **parameter** starts a parameter declaration. Like variable a name follows and optional a assignment. It means that m gets a value of 1000, and n gets a value of 2000.
Sets also are a fundamental concept in larger models. LPL declares sets using the keyword **set** declaring the sets I and J in the same way as above in the mathematical notation :

```
set I := 1..m;
    J := 1..n;
```

In the parameter declaration, not only singleton data can be defined also vector data, or matrices, or higher dimensional data. The vectors b_i with $i \in I$, c_j with $j \in J$, and the matrix $a_{i,j}$ are declared (without assignment) :

```
parameter b{i in I};
  c{j in J};
  a{i in I, j in J};
```

These parameters can be assigned in the same way. The assignment can take place directly at the declaration or later on in a proper assignment as in :

```
a{i in I,j in J} := if(Rnd(0,1)<0.02 , Rnd(0,60));
c{j in J}        := if(Rnd(0,1)<0.87 , Rnd(0,9));
b{i in I}        := if(Rnd(0,1)<0.87 , Rnd(10,70000));
```

The assignment is done through an expression that generates random numbers. The function `if(cond , exp)` returns a value defined by `exp` if the Boolean condition `cond` is true else it returns zero. (It is the same as the ternary operator `c ? a : b` in same programming languages as C, Java, Python, and others.) The function `Rnd(a,b)` returns a random number uniformly distributed between `a` and `b`. Hence, the first assignment generates first a random number between 0 and 1, and if it is smaller than 0.02 then a second random number between 0 and 60 is generated and assigned to the matrix else 0 is assigned. This operation is repeated for all elements in I combined with all elements in J. The statement is similar to a double loop in a programming language like C (suppose `myrand()` returns a double between 0 and 1) :

```
double a[][];  int i,j;
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
```

```
      a[i,j] = myrand() < 0.02 ? 60*myrand() : 0;
    }
  }
```

As in the loops, the LPL statement

```
a{i in I, j in J} := ...
```

runs through the sets in lexicographical order, that is, right-most index first, left-most index last, and on each pass it assigns a random value or zero to the matrix entry.
In other words, the matrix `a` contains 2% of data that are different from zero. The large majority of its elements is zero. Of course, LPL only stored the non-zeroes.
A last remark about a simplified notation in LPL is notable: Although the previous notation as in `a{i in I, j in J}` is perfectly legal syntax in LPL, a shorter notation is preferable. In LPL set names are normally in lowercase letters and can be used as indexes too. There is generally no need to use a separate symbol for sets. So the previous declarations could also be written as:

```
set i := 1..m;
    j := 1..n;
parameter a{i,j};
    c{j};
    b{i};
```

In the same way as parameters, also variables (`x{i}`) and constraints `C{i}` can be indexed to specify a vector or a higher dimensional quantity of single objects. The same indexing mechanism can also be applied to several index operators, such as $\sum_i \ldots$ (in LPL `sum{i} ...`).
Now all elements are ready to formulate the general linear model with 1000 constraints and 2000 variables :

```
model largerModel;
  parameter m := 1000;  n := 2000;
  set i := 1..m;        j := 1..n;
  parameter
    a{i,j} := if(Rnd(0,1)<0.02 , Rnd(0,60));
    c{j}   := if(Rnd(0,1)<0.87 , Rnd(0,9));
    b{i}   := if(Rnd(0,1)<0.87 , Rnd(10,70000));
  variable x{j};
  constraint
    C{i}: sum{j} a[i,j]*x[j] <= b[i];
  maximize Obj: sum{j} c[j]*x[j];
  Write('Objective␣Value␣=␣%7.2f␣\n', Obj);
  Write{j|x}('␣␣␣x%-4s␣=␣%6.2f\n' , j,x);
end
```

Again, copy this model and paste it here: empty (A browser opens). Then click on the button *Run and Solve*. A moment later, the solution of this model is displayed:

```
Objective Value = 77599.07
 x22    =  34.64
 x31    =  14.46
 x69    = 110.61
 ... about 85 more values ...
```

About 90 variables out of the 2000 have a value different from zero. There are shown in the list.
Note that the function **Write** was used to write a formatted output of the solution. This function is a very powerful method to generate text output, sophisticated reports using the library of FastReport, as well as database tables or even Excel sheets. See the reference manual for more information [5].

inline data

Data can also be directly added to the model. As an exercise, the simple example above is written in an indexed notation:

```
model secondModel;
  set i := [1 2];
      j := [a b];
  parameter b{i} := [350 300];
    c{j} := [300 200];
    a{i,j} := [5 5 , 6 2];
  variable  x{j};
  constraint C{i}: sum{j} a[i,j]*x[j] <= b[i];
  maximize Obj: sum{j} c[j]*x[j];
  Writep(x);
end
```

(As before, copy this model, paste it here: empty and run it.) Inline data tables are enclosed in [...] and are evaluated and assigned before any other assignment. The elements are listed in lexicographical order. Note that set elements in this case are strings.

Loops and Ifs

Another feature in LPL is repeated execution and branching in its execution. The two statements for looping begin with **for** and **while** and for branching it begins with **if**. The syntax is:

```
for{setName} do
  ...statement list...
end;

while expr do
  ...statement list...
end

if expr then
  ...statement list...
else          //optional part
...statement list...
end
```

The following program implements the greatest common divider of two numbers (the algorithm is quite inefficient – there are much better methods[2], but it shows a loop and a branching statement):

```
model gcd;
  integer a := 1943;  b := 2813;
    c := if(a<b, a, b);
```

[2]This method loops through ALL integers i from 1 to the smaller number (a or b) and checks whether a and b is divisible by i, and if it is it retains the last found number. A better way is the Euclid algorithm.

```
    d := 1;
  for{i in 1..c} do
    if a%i = 0 and  b%i = 0 then  d:=i; end
  end
  Write('The gcd of %d and %d is %d\n', a, b, d);
end
```

(Copy this model, paste it here: empty and run it.) Note first, that the assignment operator is := (not = like in C, Python, Java etc.) and the Boolean equal operator is = (not ==). % is the modulo operator. The keyword **integer** start a parameter declaration that is integer. It is a shortcut for **integer parameter**. **if** has two different uses: the first was already explained above and is a function. The second starts a branching statement. Finally, `{i in 1..c}` declare an anonymous set with a local index i.
The example shows that LPL is a complete programming language and is Turing complete, in order to be able to pre- and postprocess data before and after solving a model. However, it is far from be a modern programming language – in its actual version[3]. It lacks object oriented concept; its memory is not stack-based, that is, although one can define user defined functions (see below), the function cannot be called recursively; it is interpreted and a just-in-time compiler is missing; so it is not very efficient for more many algorithmic tasks. Nevertheless, LPL does something very well: large models can be efficient and compactly formulated and ran. It links well to several commercial and free solver libraries.

3 Sub-models

Models can be organized into sub-models each with its own name space, and they can be distributed into several file. Let's give a small simple example that shows also how the data can be separated from a model structure by using the two linear models above (open subModels to run it, modify it by assigning 2 to the parameter `selectData`) :

```
model subModels;
  parameter selectData:=1;
  if selectData=1 then data1; else data2; end;
  set i; j;
  parameter a{i,j}; c{j}; b{i};
  variable x{j};
  constraint C{i}: sum{j} a[i,j]*x[j] <= b[i];
  maximize Obj: sum{j} c[j]*x[j];
  Write('Objective Value = %7.2f \n', Obj);
  Write{j|x}('   x%-4s = %6.2f\n' , j,x);

  model data1;
    i := [1 2];
    j := [a b];
    b{i} := [350 300];
```

[3]My vision of a fully-fledged modeling language are exposed in another paper, see [7].

```
    c{j} := [300 200];
    a{i,j} := [5 5 , 6 2];
  end;

  model data2;
    parameter m := 1000;  n := 2000;;
    i := 1..m;  j := 1..n;
    a{i,j} := if(Rnd(0,1)<0.02 , Rnd(0,60));
    c{j}   := if(Rnd(0,1)<0.87 , Rnd(0,9));
    b{i}   := if(Rnd(0,1)<0.87 , Rnd(10,70000));
  end;
end
```

The code contains the main model `subModels` and two sub-models `data1` and `data2`. Note that the main model does not contain data, only the declarations. One of the sub-model is called and ran in the from the main model, depending on the value of the parameter `selectData`. Note also the double semicolon in the sub-model `data2`. `m` and `n` are two local parameters and after their declaration follows a double semicolon. This is important in this context because it terminates the declaration with an empty statement (a semicolon, because the next `i := ...` is not local it is an assignment to the global set.

4 Drawings

Specially for the presentation of the results, figure and drawings are very helpful to understand the solution of a model. Therefore, LPL contains a small drawing library to generate an vector graph in svg format. The following model show the use of this library (open qchess5b to run the model and to see the graphic) :

```
model chess;
  set i,j := [1..8];
  integer variable x{i} [1..#i];
  constraint
    D: Alldiff({i} x[i]);
    S: Alldiff({i} (x[i]+i));
    T: Alldiff({i} (x[i]-i));
  solve;
  parameter X{i,j} := if(j=x[i],1);
  Draw.Scale(50,50);
  {i,j} Draw.Rect(i,j,1,1,if((i+j)%2,0,1),0);
  {i,j|X} Draw.Circle(j+.5,i+.5,.3,5);
end
```

This model implements a placement of 8 queens on a chessboard which cannot attack each other. The model shows some more interesting features:

1. The same set definition has *two* names: `i` and `j`. This is handy because they are used for the rows and the columns on the chessboard.

2. The variable are declared as integer variable in the range $[1, 8]$. (`#i` is a notation for the cardinality of the set `i`. For example, $x_2 = 3$

means that the queen on the second row should be placed in column 3.

3. The constraints implements a global constraint called `Alldiff` meaning that all the variables must be different from each other.

4. The `solve` statement says to solve the constraints and return (any) possible solution.

After the solution has been found, a graphic is generated using the library `Draw`. This library contains a dozens of function such as `Scale` (for the $x-y$ sizing), `Rect` (to draw a rectangle), `Circle` (to draw an circle), etc. (Note that `{i,j}` is a shortcut for `for{i,j} do ... end;`). Hence, 8×8 rectangles are drawn at position (i, j) with size $(1, 1)$ (multiplied by 50 pixels), alternating with color black (0) and white (1) forming the chessboard grid. Then a circle is drawn on each field with color blue (5) if $X_{i,j}$ is 1 (8 times).

5 Logical and Integer Modeling

One of the unique feature of LPL is its capability to use logical operators to model constraints. A small example is given as follows (see also logic4 for further explanation) :

```
model Logic4;
  variable x [0..10]; y [0..20];
  constraint Grey: (x<=5 and y<=2) or (y<=x and y<=6-x);
  maximize obj: x+y;
  Write('Optimum is: (%d,%d)\n' ,x,y);
end
```

Two variables x and y with lower bounds 0 and upper bound 10 and 20 are declared. The constraint contains arithmetic and Boolean operators defining a concave set. Besides *and* and *or*, LPL also defines additions Boolean and logical indexed operators, such as `not`, `nand`, implication, and indexed `and`, or `atleast`. For more information, see the paper [4], and the reference manual [5].

6 Conclusion

References

[1] MatMod. Homepage for Learning Mathematical Modeling : https://matmod.ch .

[2] Hürlimann T. Dynamic Programming : A Case Study. https://matmod.ch/lpl/doc/tutor.pdf.

[3] Hürlimann T. Index Notation in Mathematics and Modelling Language LPL: Theory and Exercises. https://matmod.ch/lpl/doc/indexing.pdf.

[4] Hürlimann T. Logical modeling. https://matmod.ch/lpl/doc/logical.pdf.

[5] Hürlimann T. Reference Manual for the LPL Modelling Language, most recent version. https://matmod.ch/lpl/doc/manual.pdf.

[6] Hürlimann T. Various Modeling Toolls. https://matmod.ch/lpl/doc/modeling4.pdf.

[7] Hürlimann T. Various Modeling Toolls. https://matmod.ch/lpl/doc/modeling5.pdf.