

ILOG OPL Studio Introduction

(This tutorial is adapted (significantly) from the ILOG help module. A more detailed version can be found in Help->Language->Short Tour of OPL.)

An optimization problem is typically specified by an objective function and a set of constraints over some variables. A solution to the problem is an assignment of values to the variables that satisfies the constraints and optimizes the value of the objective function. The purpose of an OPL statement is thus to express these two components for the application at hand. We will introduce the specific details of the OPL language via a series of examples

Basic Linear Programming Example

Consider a Belgian company Volsay, which specializes in producing ammoniac gas (NH_3) and ammonium chloride (NH_4Cl). Volsay has at its disposal 50 units of nitrogen (N), 180 units of hydrogen (H), and 40 units of chlorine (Cl). The company makes a profit of 40 Euro for each sale of an ammoniac gas unit and Euro for each sale of an ammonium chloride unit. Volsay would like a production plan maximizing its profits given its available stocks.

We will model this problem using *Linear Programming*. Linear programming is the class of problems that can be expressed as the optimization of a linear objective function subject to a set of linear constraints (i.e., linear equations and inequalities) over real numbers, i.e. continuous variables. Linear programming models can be solved for large numbers of variables and constraints and are, from a computational standpoint, the simplest optimization problems.

The following OPL program formalizes this linear program.

```

dvar float+ Gas;
dvar float+ Chloride;
maximize
    40 * Gas + 50 * Chloride;
subject to {
    Gas + Chloride <= 50;
    3 * Gas + 4 * Chloride <= 180;
    Chloride <= 40;
}

```

This model is saved as *gasmodelA.mod* in the course website. At this point open OPL Studio and open this model by using the command *File->Open model*. Create a new project, and then, by right clicking on the project name in the project window, add *gasmodelA.mod* to the project. Then right click on *gasmodelA.mod* to add it to the run configuration.

The file declares two real variables `gas` and `chloride`, representing the production of ammoniac gas and ammonium chloride. The notation “dvar” means that these are decision variables. These variables are of type `float+`, which means that they are required to be nonnegative. The objective function `maximize 40 * gas + 50 * chloride`

states that the profit must be maximized. The constraints ensure that the production plan does not exceed the available stocks of nitrogen, hydrogen, and chlorine, respectively. The constraint `gas + chloride <= 50` represents the capacity constraint for nitrogen, since each unit of ammoniac gas and of ammonium chloride uses one unit of nitrogen. The next two constraints, for hydrogen and chlorine respectively, are similar in nature.

Note that every OPL statement ends with a semicolon (;).

As mentioned previously, a solution to an optimization problem is typically an assignment of values to the variables that satisfies the constraints and optimizes the objective function. At this point execute the model using the *Execution->Run or F5*. OPL Studio will produce the following solution for the Volsay production-planning problem.

```
Final Solution with Objective
= 2300.0000
    gas = 20.0000
    chloride = 30.0000
```

One can get more information by clicking on the “Cplex Log” or “CPLEX Statistics” tabs at the bottom of the page. Later, we will also see how to get even more information about the solution.

Sets and Arrays

The above statement is very specific to the application at hand. In general, it is desirable to write generic models that can be extended, modified easily, and applied in different contexts. The next two sections describe a number of OPL concepts to simplify the process of creating such models. A first step towards more generality is the use of arrays, which makes it easier, for instance, to accommodate new products in the future.

The Volsay production-planning model can be rewritten using arrays as follows.

```

{string} Products = { "gas", "chloride" };
{string} Components = { "nitrogen",
"hydrogen", "chlorine" };

float demand[Products,Components] = [[1,
3, 0], [1, 4, 1]];
float profit[Products] = [40, 50];
float stock[Components] = [50, 180, 40];

dvar float+ production[Products];

maximize
    sum(p in Products) profit[p] *
production[p] ;

subject to {
    forall(c in Components)
        sum(p in Products) demand[p,c] *
production[p] <= stock[c] ;
};

```

First, the instruction

```
{string} Products = { "gas", "chloride" };
```

declares a variable `Products` that represents the set of products of the company.

A second fundamental step towards a more general model amounts to representing the problem data explicitly.

First, the instruction

```
{string} Products = { "gas", "chloride" };
```

declares a variable `Products` that represents the set of products of the company.

In addition to the products, the problem data obviously consists of the components (i.e., nitrogen, hydrogen, and chlorine), the demand of each product for each component, the profit of each product, and the stock available for each component.

The declaration

```
float profit[Products] = [40, 50];
```

declares an array called `profit` which is indexed by `Product` and takes on two possible values 30 and 40. In other words, you can say that

```
profit[gas] = 40
```

```
profit[chloride] = 50
```

These variables are used in the rest of the statement, which remains essentially the same as before. As will become clear subsequently, one of the novel features of OPL is the generality of its arrays: OPL arrays can have an arbitrary number of dimensions and their index sets can be arbitrary finite sets, possibly involving complex data structures.

There are also two dimensional arrays,

```
float demand[Products,Components] = [[1, 3, 0], [1, 4, 1]]
```

declares a two dimensional array, where the first dimension is product and the second component. Note that the two dimensional array is actually a one dimensional array in which each component is itself an array. So the demand for nitrogen in gas is 1, and the demand for hydrogen in chloride is 4.

Note that the multidimensional arrays are initialized one row at a time.

The rest of the statement can be obtained easily by replacing the numbers by the relevant data items. For instance, the objective function is simply written as

```
maximize profit[gas]* production[gas] +  
profit[chloride] * production[chloride]
```

It should be clear, however, that the statement above contains much redundancy. All constraints, and all arithmetic terms in these constraints and in the objective function, are similar: they differ only in their indices.

OPL has two features to factorize these commonalities, aggregate operators and quantifiers, which are used in the new model. The objective function

```
maximize sum(p in Products)  
profit[p] * production[p]
```

illustrates the use of the aggregate operator `sum` to take the summation of the individual profits. A variety of aggregate operators are available in OPL, including `sum`, `prod`, `min`, and `max`. The instruction

```
forall(c in Components)  
    sum(p in Products) demand[p,c] *  
production[p] <= stock[c]
```

shows how the universal quantifier `forall` can be used to state closely related constraints. It generates one constraint for each chemical component, each constraint stating that the total demand for the component cannot exceed its available stock. This code snippet is in the file [gasmodelB.mod](#).

Using Projects to isolate the code and the data

Another fundamental step in making models reusable is to separate the model and the instance data. OPL Studio supports this clean separation through the notions of *projects*. A project is the association of a model (a file whose suffix is `.mod`) and a set of data files (files whose suffixes are `.dat`). The model declares the data but does not initialize them. The data files contain the initializations of each declared data item. The model part of the Volsay production-planning problem is as follows. It is contained in the file [*gasmodelC.mod*](#).

```
{string} Products = ...;
{string} Components = ...;

float demand[Products,Components] = ...;
float profit[Products] = ...;
float stock[Components] = ...;

dvar float+ production[Products];

maximize
    sum(p in Products) profit[p] *
production[p] ;

subject to {
    forall(c in Components)
        sum(p in Products) demand[p,c] *
production[p] <= stock[c] ;
};
```

The statement

```
float  
profit[Products] =  
...;
```

declares the array `profit` and specifies that its initialization is given in a data file. The data file simply associates an initialization with each non-initialized piece of data.

OPL offers a variety of ways to initialize data. One particularly useful feature is the possibility of associating indices with values to avoid various kinds of errors.

```
Products = {gas, chloride};  
Components = {nitrogen,  
hydrogen, chlorine};  
demand = [[1, 3, 0], [1, 4,  
1]];  
profit = [40, 50];  
stock = [50, 180, 40];
```

This initialization is contained in the file `gas.dat`.

```
Products = {gas chloride};  
Components = {nitrogen hydrogen  
chlorine};  
  
profit = #[gas:30 chloride:40]#;  
stock = #[nitrogen:50  
hydrogen:180 chlorine:40]#;  
demand = #[  
    gas: #[nitrogen:1 hydrogen:3  
chlorine:0]#  
    chloride: #[nitrogen:1  
hydrogen:4 chlorine:1]#  
]#;
```

This initialization is contained in the file `gasn.dat`.

The initialization

```
profit = #[gas:30  
chloride:40]#;
```

describes the initialization of array `profit` by associating the value 30 with index `gas` and the value 40 with index `chloride`. (Of course, the order of the pairs has no importance in these initializations.) When using `index:value` pairs, the delimiters `#[` and `]#` must be used instead of `[` and `]`. Note also that, in data files, the items can be initialized in any order and the commas can be omitted freely.

So, now we have two files. One stores the optimization model and the other stores the problem data. Next we need to stitch them together so the model knows which data files to use. This is done using *projects*. Open a new project by using the command *File->New->Project*. You should see the project *noname.prj* in the top left pane of the OPL Studio window. If you don't see it, click on the Projects Tab. Right-click on the project and use the option *Add Existing Document ...* to insert the file *gasmodelC.mod*. Next, right-click and use the option *Add Existing Document ...* to insert one of the two data files. Then right click on each file name to add them to the run configuration. Next, run the project.

Labeling Constraints

It is useful, for reasons we will see, to label constraints.

Consider the following file, which is stored as

gasmodelD.mod.

```

{string} Products = ...;
{string} Components = ...;

constraint ct[Components];

float demand[Products,Components] = ...;
float profit[Products] = ...;
float stock[Components] = ...;

dvar float+ production[Products];

maximize
    sum(p in Products) profit[p] * production[p] ;
subject to {
    forall(c in Components) {
        ct[c]:
            sum(p in Products) demand[p,c] * production[p] <=
stock[c] ;
    };
};

```

The new wrinkle is the “constraint” command, and then the constraints are labeled. Add this to your project and run it. Then look at [View->Show/Hide Problem Browser](#).

First, click on the + sign by constraints, and then right click, to enable you to view the slacks or dual variables. You can also right click to view variable values, which will be more useful later on.