This is an old revision of the document!

---

start

AM121 AMPL User Guide

The Basics: Declaring Parameters and Variables in AMPL

In the model file you need to "declare" parameters and variables. You can also specify non-negativity and other basic constraints here:

Declare a parameter:

```
param foo >= 0;
```

Declare a vector of parameters over a set:

```
param foo{SET} >= 0;
```

Declare a matrix of parameters over two sets:

```
param foo{SET_A, SET_B} >= 0, <= 1;
```

Declare a variable:

```
var Bar >= 0;
```

Declare a vector of variables:

```
var Bar{SET} >= 0;
```

Declare a matrix of variables:

```
var Bar{SET_A, SET_B} >= 0;
```

*Note that we do not usually put constraints on our parameters, only on variables. The reason you might want to do this in your model file is to prevent typos in your data file. For example if you know that param cost{Products} has to be non-negative, but you accidentally type something negative, AMPL will tell you that you have made an error.

The Basics: Defining Values for Parameters and Variables in AMPL

In the data file you need to specify the values of parameters and variables that you declared in your model:

Define a single parameter:

```
param foo 0.376;
```

Define a vector of parameters:

```
param: foo:=
C1 5
C2 6
C3 7 ;
```

Define a matrix of parameters:

```
param foo: A1 A2 :=
    B1    1    2
    B2    3    4 ;
```

Here is an example that illustrates how the model file and the data file are related:

Model file

```
set A;
param L {A};
```

Data file

```
set A := a b c;
param L := a 4
           b 6
           c 4;
```

For parameters indexed over two sets, see the section on defining sets with pair and tuple elements. If you're really brave, see the section on defining three-dimensional parameters.

Script files, Output files, Logging

Have you found it tedious to get AMPL to log what you want? Have you felt the need to re-type stuff all the time? Have you read the updated hand-in directions in the problem set? If so, you are in the right place. The nice way to handle logging in AMPL is really to write a run file - which contains all the commands that you want ampl to execute (e.g. model light.mod; data light.dat; and so on). And then, you and call AMPL with this run file and it would just run all the commands you have typed and print the output to the console. For example, see the file light.run [http://www.courses.fas.harvard.edu/~apm121/sections/light.run] in the section notes [http://www.courses.fas.harvard.edu/~apm121/sections.html] section of the course website. It contains all the necessary commands we want. To run it, when running ampl, instead of typing ampl, type:

```
ampl light.run
```

And the output to your screen is something like:

```
CPLEX 10.1.0: optimal solution; objective 16.4516129
3 dual simplex iterations (0 in phase I)
Power
:=
1  0
2  6.00806
3  1.93548
```

```
4  8.50806
5  0
;
```

Now, you can easily just copy and paste the output to a file, and save the file as light.out, which contains the solution corresponding to light.mod and light.dat. But even simpler, you can 'pipe' the output from AMPL directly to an output file. So when running AMPL, type instead:

```
ampl light.run > light.out
```

This runs ampl with the run file, and 'pipes' the output into a file light.out. In the directory in which you are executing the command, you will now have a file light.out, with all the output that you saw before. Please see light.mod, light.dat, light.run, light.out on the sections section of the course site for the corresponding files. For submission then, for each AMPL exercise, you should turn in the model, data, the run script, and the output (the output is optional if you had already included it in your writeup).

Define number indexed sets

In defining number indexed sets, you may want to say something like set S := 1 .. 5 to get a set from 1 to 5, but you can't do this in the AMPL data file. See the following note and solution, from AMPL's website: (section 4.2 in http://www.ampl.com/FAQ/ [http://www.ampl.com/FAQ/])

4.2 Why does “set S := 50 .. 70” give me a set of only 3 members?

You have declared “set S” in your model and “set S := 50 .. 70” in your data. Set expressions are not recognized in AMPL's data mode, however. Instead AMPL tries to recognize 50 .. 70 as a space-delimited list of members of S, with the result that it has found three members: the numbers 50 and 70, and the string ”..”.

To define S to equal 50 .. 70 by use of your data file, first declare S in your model by

```
param begin;
param end > begin;
set S := begin .. end;
```

Then state in the data file:

```
param begin := 50;
param end   := 70;
```

Alternatively, it's legal to give “set S := 50 .. 70” as your declaration of S. This is a less desirable approach, however, because it moves some of the specific data values into the model.

logical conditions and set indexing 'restriction' in AMPL

This syntax consists of a expression that provides a specifies a 'restriction' to a subset of a given set: subject to FOO: sum{a in A: a > 1} >= y Note the : and the additional condition in the syntax - only elements of A that are greater than 1 will be considered. This works for ordered sets (such as sets of numbers). With this : syntax it is possible to construct expressions that ensure that your indexing operations will not go beyond bounds.

multiple logical conditions in AMPL

You can use logical conditions for sets in indexing expressions. The optimal illumination problem in section had the following line:

```
sum{l in LIGHTS, c in CAST: i=l+c}
```

More generally, a set in an indexing expression may be followed by a colon and an arbitrarily complex logical expression (e.g. using “or”, “and”,…). Thus, you could for example write something like:

```
sum{c in CAST: 0 <= i+c and c != 0}
```

You can use this technique not only for sums but whenever you are using sets, for example when defining a new set:

```
set NUTREQ = {i in NUTR: i in MAXREQ or n_min[i] > 0}
```

or when defining a constraint:

```
subject to Required_Light {s in STREET: s != 3}
```

Here is a list of possible logic conditions:

```
=     equal (==)
<>    not equal (!=)
and   and (&&)
not   logical negation (!)
or    or (||)
```

Defining sets with pair/tuple elements

This comes from Chapter 6 and 9 of the AMPL book.

AMPL supports the notion of a Tuple, and in particular a Pair. This given by the syntax (“ATL”, “LGA”) for strings or (5, 6) for numbers. They can be grouped together into sets like {(“STL”, “LGA”), (“BOS”,”SFO”)}.

Suppose you have two sets:

```
set SRC := 1..3;
set SNK := 1..5;
```

What does {SRC,SNK} mean? It means the cross product of the two sets – or a set containing a the pairs of all combinations. You can assign this as:

```
set EDGES = {SRC, SNK};
```

These pair based sets can be restricted using the : operator, eg:

```
set EDGES = {s in SRC, k in SNK : s+k <= 6}
```

We can now use this set of pairs in our objective and constraints for example:

```
minimize Total_Cost:  sum{(i,j) in EDGES} cost[i,j] * Use[i,j];
```

This loops over all contents of the EDGES set when definiing the objective

A similar expression can be used to take a 'slice' through the two dimensional data. Suppose we want to create a constraint for each element of SRC, that constrains an expression over each SNK for the given SRC element:

```
subject to Exit_Capacity {s in SRC}: sum{(s, k) in EDGES} capacity(s,k) >= required_capacity(s);
```

Note that the s variable is defined for each constraint created – and then repeated in the summing expression. The re-use of this variable tells ampl that we are speaking about a “slice” through the 2d data, rather then the full contents of the EDGES set. Another way to write the same expression more explicitly is:

```
subject to Exit_Capacity {s in SRC}: sum{k in SNK: (s, k) in EDGES} capacity(s,k) >= required_capacity(s);
```

In addition to creating pair based sets via cross-product and restirction via the : operator, they can also be specified directly in the data file:

model file:

```
set EDGES within {SRC, SNK};
```

data file:

```
set EDGES:  1   2   3   4   5  :=
1           -   +   -   -   -
2           +   +   -   +   +
3           +   +   +   -   - ;
```

Combinations with the + will be in the set, those with - will be omitted.

Lastly, one can also define parameters over such sets:

model file:

```
param cheetos{EDGES};
```

data file:

```
param cheetos: 1   2   3   4   5  :=
1              .   3   .   .   .
2              2   4   .   3   1
3              0   3   1   .   . ;
```

The . is a placeholder that goes wherever the set is not defined.

the "expand" command

Sometimes, you want to be able to see the long, verbal form of an objective function or constraint within AMPL as a way to check whether you had actually encoded the program correctly. To do this, use the 'expand' command. After you load up your model and data, you can examine the long form of a objective or constraint by expanding it. For example:

```
model steel3.mod;
data steel3.dat;
expand Total_Profit;
```

Shows this:

```
maximize Total_Profit:
   25*Make['bands'] + 30*Make['coils'] + 29*Make['plate'];
```

Don't use AMPL's absolute value function

Some people try to use absolute values in their mathematical model and in their AMPL formulation for the current assignment. Note that taking the absolute value of something is NOT a linear function. Thus, never directly use “absoluate value” either in your mathematical formulation nor in AMPL!

1) Operators When defining sets, you can use conditionals with the colon syntax (see link to post below). When you want to specify multiple conditions that must be satisfied, you can use the keyword 'and'. But there are other keywords as well, some of which you may find useful. Here is a table of them all:

```
=      equal (==)
<>     not equal (!=)
and    and (&&)
not    logical negation (!)
or     or (||)
```

Conditional for printing

In AMPL scripts, you can use if statements. This is particularly useful for printing output in the way you want it to appear. For example:

```
print {i in ROWS}: {j in COLUMNS} if (i,j) in MATRIX then Mat[i,j] else 0;
```

This command prints everything from a matrix Mat, such that if the location is in the set MATRIX then it prints the element in Mat, but otherwise, it prints a 0.

Printing to a file

If you want to print a matrix to its own specific file, then you can use ”>” to specify a file:

```
print {i in ROWS}: {j in COLS} Mat[i,j] > matrix.out;
print x > x.out
```

This prints the contents of the matrix Mat to the file matrix.out and the value of x to the file x.out.

Modulo operator in AMPL

A modulo operator can be particularly useful for the Post Office problem we discussed in class last Thursday. Here is a way to express all 7 constraints (one for each day) in just one constraint using the 'mod' operator: 1. define the set of days to be from 0..6 2. define the parameter empReq as avector over the set of days and then use the following constraint: subject to Union_Rule{d in DAYS}:

```
   sum{i in DAYS: i!= (d+1) mod 7 and i!= (d+2) mod 7} StartWork[i] >= empReq[d];
```

Defining three-dimensional parameters

Suppose you had the following lines of code in your model file:

```
set ORIG;   # origins
set DEST;   # destinations
set PROD;   # products
param cost {ORIG,DEST,PROD} >= 0;  # shipment costs per unit
```

This is part of a larger model that seeks to minimize transportation costs. Each product in the model has a different transportation cost depending on the origin and destination. The cost parameter encodes this. The data file corresponding to these parts of the model looks something like this:

```
set ORIG := GARY CLEV PITT ;
set DEST := FRA DET LAN WIN STL FRE LAF ;
set PROD := bands coils plate ;
param cost :=

 [*,*,bands]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
       GARY     30   10    8   10   11   71    6
       CLEV     22    7   10    7   21   82   13
       PITT     19   11   12   10   25   83   15

 [*,*,coils]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
       GARY     39   14   11   14   16   82    8
       CLEV     27    9   12    9   26   95   17
       PITT     24   14   17   13   28   99   20

 [*,*,plate]:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
       GARY     41   15   12   16   17   86    8
       CLEV     29    9   13    9   28   99   18
       PITT     26   14   17   13   31  104   20 ;
```

As you can see, each product has its own cost matrix associated with it. Each one of these matrices has rows corresponding to the origins from the set ORIGINS and columns corresponding to the destinations in the set DESTINATIONS. Filling in the matrix for each product is very similar to filling in two-dimensional parameters. The only difficulty is in the syntax for specifying which matrix you are defining.

Importing AMPL Files into Latex

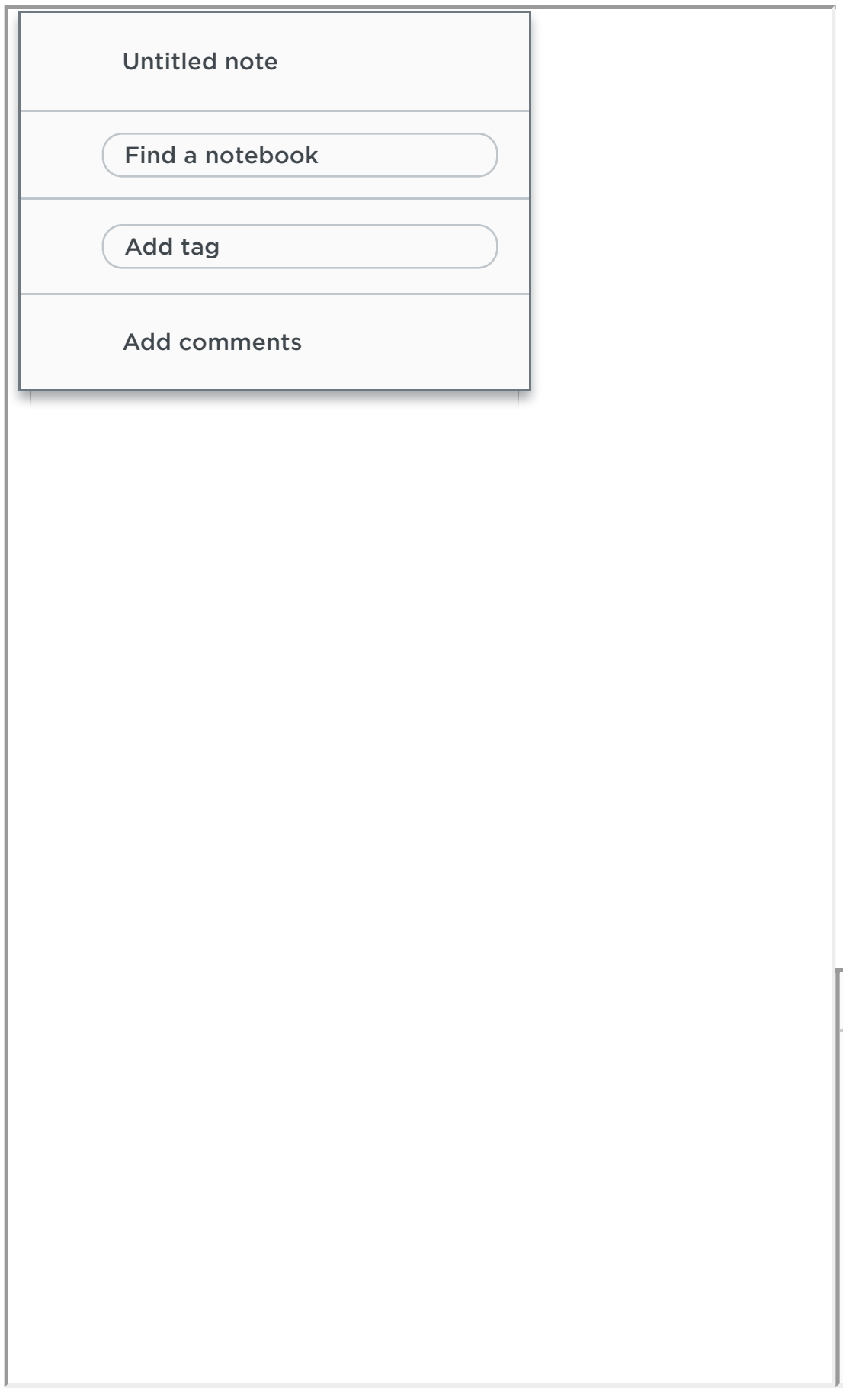
If you're at all particular about how you're typesetting your problem sets, you've probably been frustrated with having to print off your PDF of your problem set, as well as the model (.mod), data (.dat), and command logs, and output (usually .out). In order to important all of these directly into

your LaTeX, you can use a very useful package.

In your preamble, import the listings package with the command \usepackage{listings}. Then, any time in your file, you can directly import a text file (and a lot of other file types, such as source code!) by using the command \lstinputlisting{your_file_name}. Of course, this requires that you keep your .tex file in the same directory as the files that you are attempting to import.

CPLEX AMPL Guide Link

http://www.courses.fas.harvard.edu/~apm121/amplcplex110userguide.pdf
[http://www.courses.fas.harvard.edu/~apm121/amplcplex110userguide.pdf]



Saving clip...

To:

Clip

Article

Simplified Article

Share

Selection

Save

Email

Bookmark

Screenshot