Mathematical Programming in AMPL

Anthony Papavasiliou, National Technical University of Athens (NTUA)

Outline

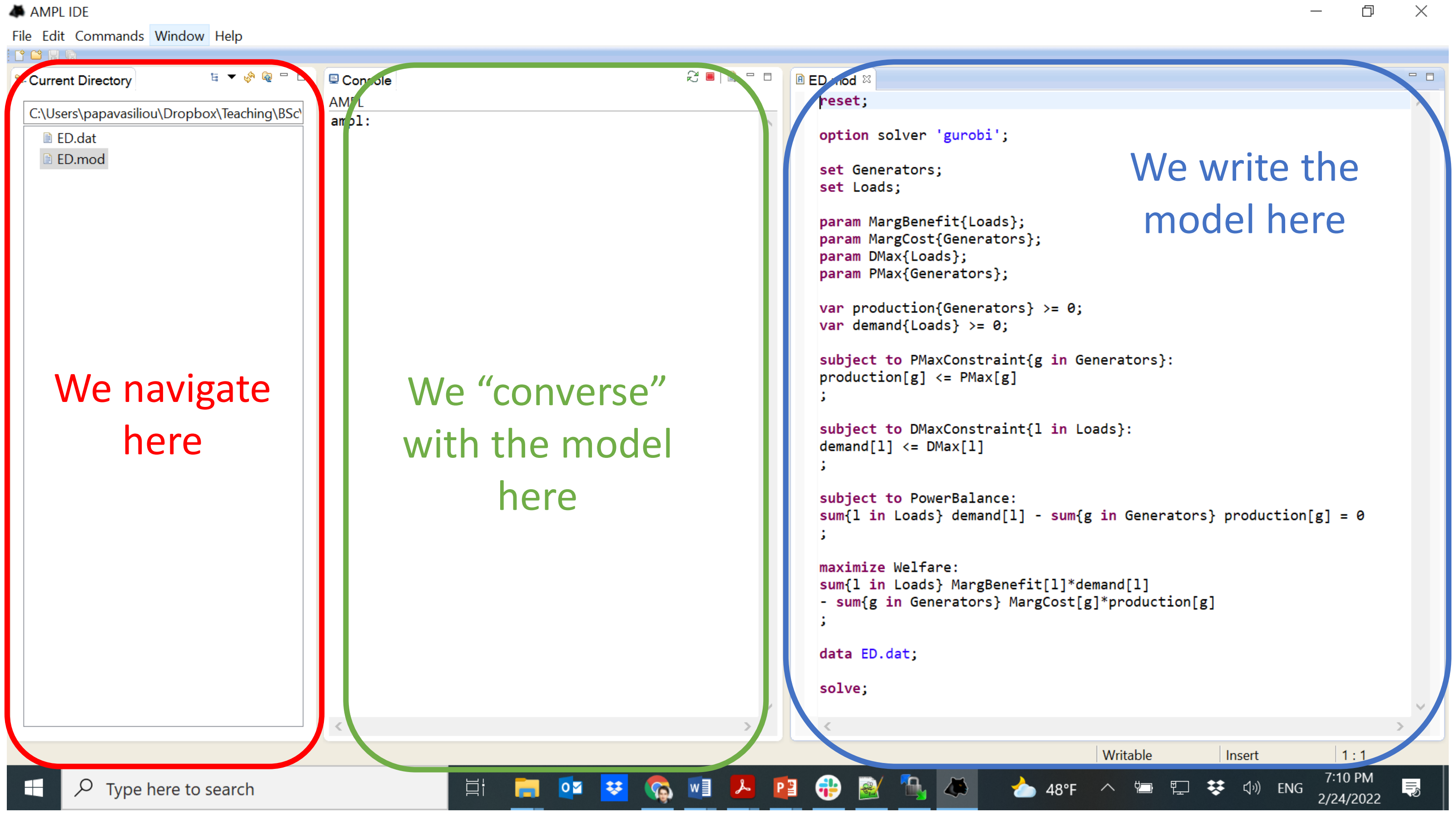
- Downloading and getting started with AMPL
- Modeling mathematical programs
    - Sets
    - Parameters
    - Decision variables
    - Constraints
    - Objective function
- Entering data
- Solving mathematical progams
- Resolving bugs
- The display and print commands
- Implementing algorithms
- Analyzing the solution

Downloading and getting started with AMPL

Download instructions

- Your instructor has the rights to a free academic version of AMPL
- The academic license gives access to the full functionalities of AMPL
- Contact your instructor for a link to the software

The AMPL IDE



We navigate here

```
AMPL
ampl:
```

We “converse” with the model here

```
reset;

option solver 'gurobi';

set Generators;
set Loads;

param MargBenefit{Loads};
param MargCost{Generators};
param DMax{Loads};
param PMax{Generators};

var production{Generators} >= 0;
var demand{Loads} >= 0;

subject to PMaxConstraint{g in Generators}:
production[g] <= PMax[g]
;

subject to DMaxConstraint{l in Loads}:
demand[l] <= DMax[l]
;

subject to PowerBalance:
sum{l in Loads} demand[l] - sum{g in Generators} production[g] = 0
;

maximize Welfare:
sum{l in Loads} MargBenefit[l]*demand[l]
- sum{g in Generators} MargCost[g]*production[g]
;

data ED.dat;

solve;
```

We write the model here

Types of AMPL files

- Model files: **.mod**
  - Here we describe the structure of the mathematical program (sets, parameters, decision variables, objective function and constraints)
- Data files: **.dat**
  - Here we describe the input data of the problem
- Complex program files: **.run**
  - Here we implement complex iterative algorithms

Modeling mathematical programs

Sets

Parameters

Decision variables

Constraints

Objective function

Definition of an optimization problem

- An **optimization problem** is defined as follows:

$$\max_x f(x)$$
$$x \in X$$

- In words:
  - We want to maximize an **objective function** $f(x): \mathbb{R}^n \to \mathbb{R}$
  - We can control the **decision variables** $x \in \mathbb{R}^n$
  - The decision variables obey the **constraints** $X \subseteq \mathbb{R}^n$

Mathematical programming languages and algorithms

- The definition of the previous slide is quite abstract
- We can use mathematical programming languages that encode these problems in computers
  - AMPL
  - GAMS
  - Julia/JuMP
  - Python/Pyomo
- These languages send the problem to specialized algorithms that are extremely powerful
  - CPLEX (linear programs, mixed integer programs)
  - Gurobi (linear programs, mixed integer programs)
  - Knitro (non-linear programs)
- In order to encode a mathematical program we need to determine the following:
  - Sets
  - Decision variables
  - Parameters
  - Objective function
  - Constraints

Example: economic dispatch

Suppose that we would like to match the following offers in order to maximize welfare:

- Producer/seller 1: 30 MW at 12 $/MWh
- Producer/seller 2: 35 MW at 28 $/MWh
- Producer/seller 3: 25 MW at 80 $/MWh
- Consumer/buyer 1: 10 MW at 90 $/MWh
- Consumer/buyer 2: 40 MW at 40 $/MWh
- Consumer/buyer 3: 25 MW at 20 $/MWh

Building blocks of the linear program

- Sets:
  - Loads L
  - Generators G
- Decision variables:
  - Demand of load i, d_i
  - Production of generator i, p_i
- Parameters:
  - Marginal benefit of load i, MB_i
  - Marginal cost of generator i, MC_i
  - Maximum demand of load i, D_i^+
  - Maximum capacity of generator i, P_i^+

Economic dispatch as a linear program

$$\max_{p,d} \sum_{i \in L} MB_i \cdot d_i - \sum_{i \in G} MC_i \cdot p_i$$

Maximization of social welfare

$$d_i \leq D_i^+, i \in L$$
$$p_i \leq P_i^+, i \in G$$

Production/demand limits

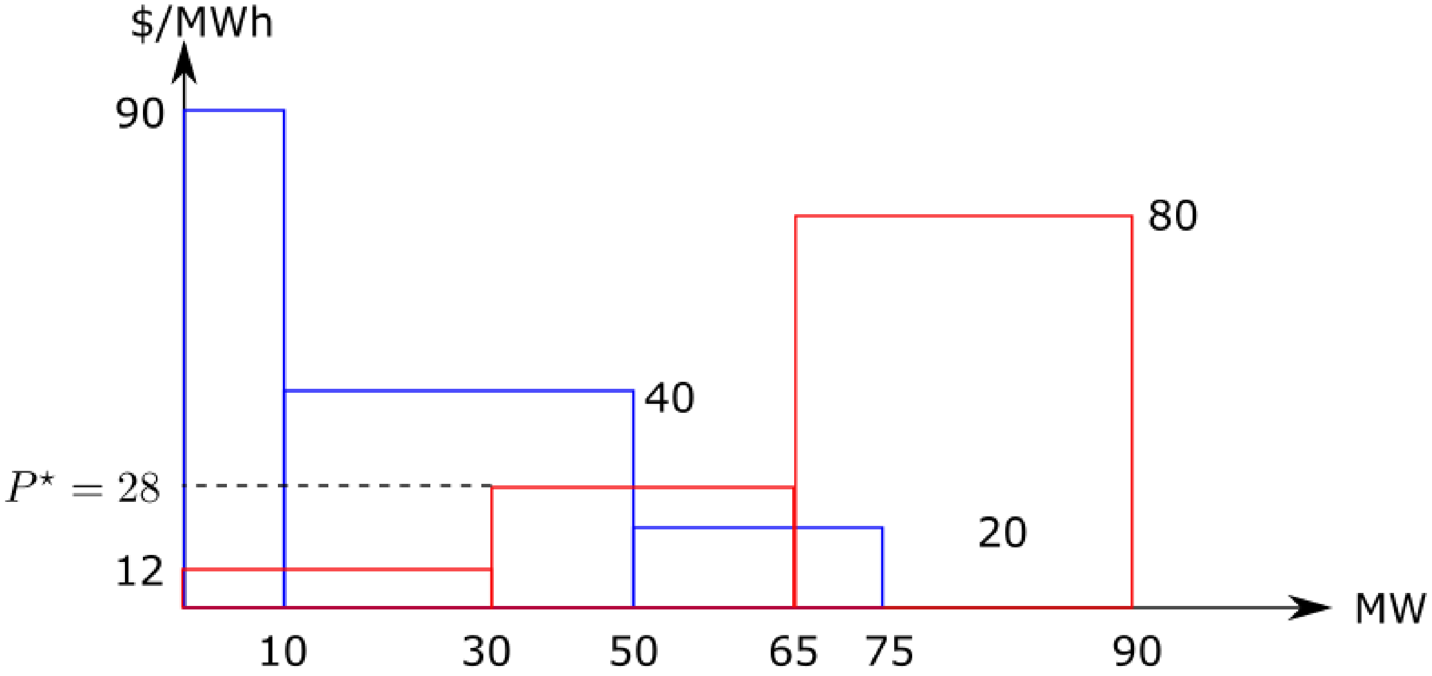
$$\sum_{i \in L} d_i - \sum_{i \in G} p_i = 0$$

Supply/demand equilibrium

$$d_i \geq 0, i \in L$$
$$p_i \geq 0, i \in G$$

Non-negative variables

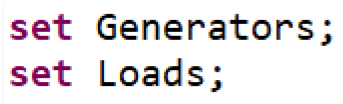
Graphical solution



The solution of the economic dispatch problem is at the intersection of the inverse supply and demand curves

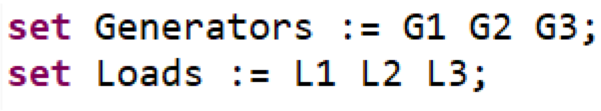
Sets

- Sets describe the entities over which indices run (indices of parameters, decisions and constraints)
- Usually we define sets of integers or sets of strings
- We declare sets in the .mod file



```
set Generators;
set Loads;
```

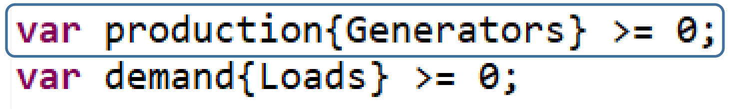
- And we populate the data of the sets in the .dat file



```
set Generators := G1 G2 G3;
set Loads := L1 L2 L3;
```

Decision variables

- Decision variables are the decisions that we want to reach
- Decision variables can be defined over sets (e.g. production of *each* generator in the market), in which case we use braces
- Decision variables are declared in the .mod file
- Decision variables can have a sign (≥ 0 or ≤ 0)



```
var production{Generators} >= 0;
var demand{Loads} >= 0;
```

Every producer generates a non-negative amount of energy

Parameters

- Parameters are used for determining the objective function $f(x)$ and the constraints X
- For instance, in a linear program:
  - The objective function is **linear**,

$$f(x) = \sum_{i=1}^{n} c_i \cdot x_i$$

And the parameters are the constants c_i

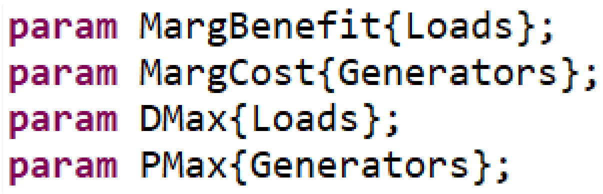
  - The set of constraints is a **polyhedron**

$$X = \left\{ x: \sum_{i=1}^{n} A_{ij} \cdot x_i \leq b_j, j = 1, \dots, m \right\}$$

And the parameters are the constants A_{ij} and b_j

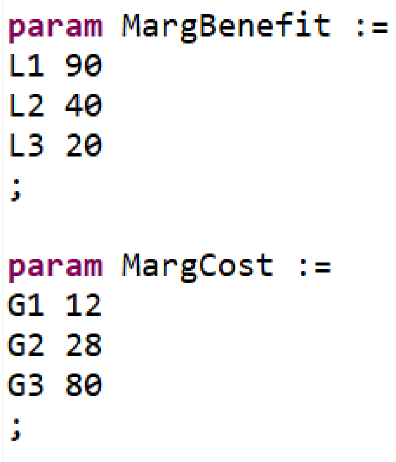
Parameters

- Parameters are numerical values
- We declare parameters in the .mod file

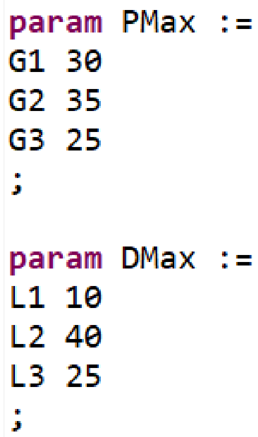


```
param MargBenefit{Loads};
param MargCost{Generators};
param DMax{Loads};
param PMax{Generators};
```

- We assign values to the parameters in the.dat file



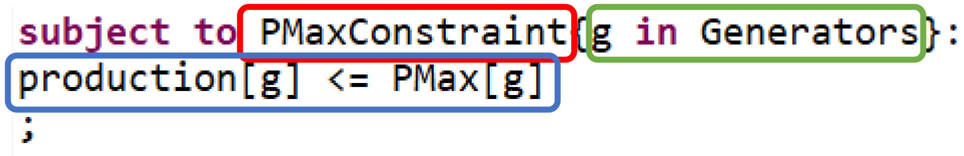
```
param MargBenefit :=
L1 90
L2 40
L3 20
;

param MargCost :=
G1 12
G2 28
G3 80
;
```



```
param PMax :=
G1 30
G2 35
G3 25
;

param DMax :=
L1 10
L2 40
L3 25
;
```

Constraints

- Constraints describe the set in which our decision must belong
- Constraints are declared in the.mod file
- The syntax for describing constraints reproduces how we would express these constraints mathematically on paper:
  - Each constraint has a name
  - Each constraint is defined over a set
  - Each constraint is described though a mathematical expression

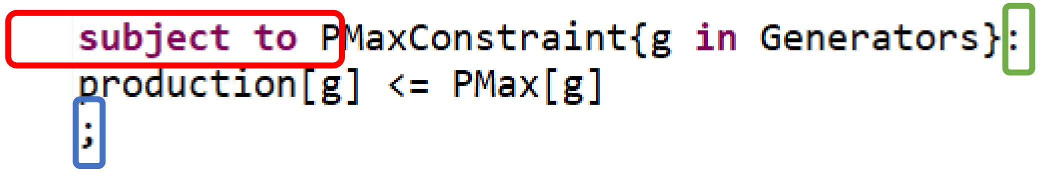


```
subject to PMaxConstraint{g in Generators}:
production[g] <= PMax[g]
;
```

$p_i \leq P_i^+, i \in G$

Constraint syntax

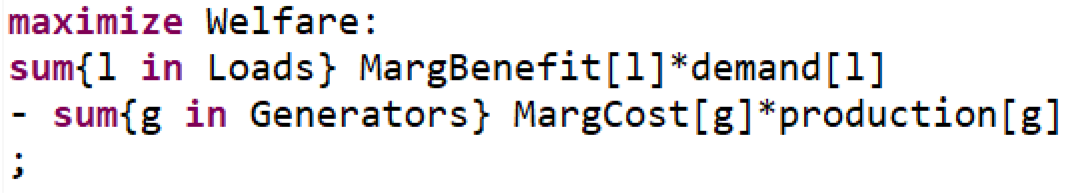
- The declaration of constraints starts with the expression subject to
- We use a colon symbol after the declaration of the set over which the constraint is defined
- We use a semi colon to complete the declaration of the constraint



```
subject to PMaxConstraint{g in Generators}:
production[g] <= PMax[g]
;
```

Objective function

- The objective function determines what it is we are trying to achieve with the optimization problem that we are solving
- The optimization problem is often either a maximization problem, $\max_x f(x)$, or a minimization problem, $\min_x f(x)$, (although both can be expressed equivalently: $\max_x f(x) \Leftrightarrow \min_x -f(x)$)
- In the .mod file we declare our objective function, if it is a maximization or minimization, and we describe it mathematically using the same syntax as constraints



```
maximize Welfare:
sum{l in Loads} MargBenefit[l]*demand[l]
- sum{g in Generators} MargCost[g]*production[g]
;
```

$$\max \sum_{i \in L} MB_i \cdot d_i - \sum_{i \in G} MC_i \cdot p_i$$

Entering data

The .dat file

- We call the.dat file from within the.mod file

```
data ED.dat;
```

- But we first need to declare any set or parameter to which we assign values
- The suffix .dat switches AMPL to “data input mode”, and the return from the.dat file switches it back to “model mode”
- If we want to assign values to parameters in the .mod file without re-entering a .dat file, an easy way is using the “let” command

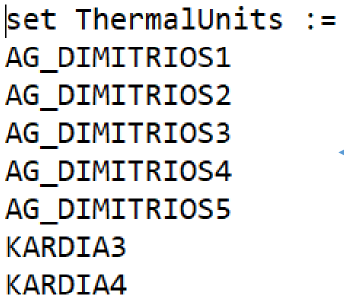
The .dat file

- In the .dat file we can enter numerical data for our problem
- We can enter data directly in the .dat file for small problems

```
set Generators := G1 G2 G3;
```

- Or we can refer the program to separate files for larger problems

```
data ThermalUnits.txt;
```



```
set ThermalUnits :=
AG_DIMITRIOS1
AG_DIMITRIOS2
AG_DIMITRIOS3
AG_DIMITRIOS4
AG_DIMITRIOS5
KARDIA3
KARDIA4
```

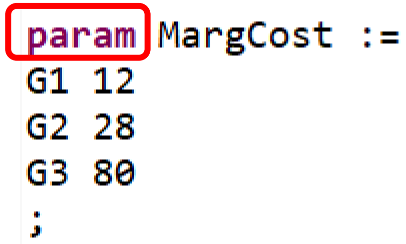
Part of the file ThermalUnits.txt

Syntax for assigning values to one-dimensional data

- For sets, we assign values using the word set, the assignment operator :=, we then enter the data (strings or numerical values), and we conclude with a semi-colon

```
set Generators := G1 G2 G3;
```

- For one-dimensional parameters, we use a similar syntax:
  - But replace “set” with param
  - And we enter the data as a two-column matrix

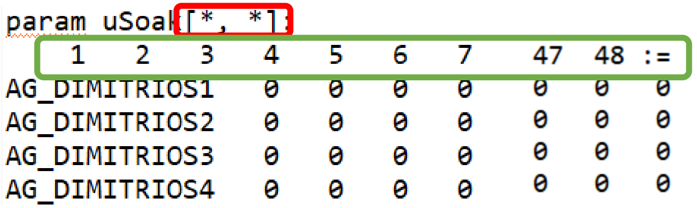


```
param MargCost :=
G1 12
G2 28
G3 80
;
```

Syntax for assigning values to multi-dimensional data

- For two-dimensional parameters, after we define the name of the parameter we use:
    - The sequence [*, *]:
    - We then enter the values of the columns followed by :=
    - We then enter the data where the name of each line is placed in the first column

```
param uSoak{AggregatedGenerators, Qs} default 0;
```



```
param uSoak[*, *]:
     1    2    3    4    5    6    7     47   48  :=
AG_DIMITRIOS1       0    0    0    0     0    0    0
AG_DIMITRIOS2       0    0    0    0     0    0    0
AG_DIMITRIOS3       0    0    0    0     0    0    0
AG_DIMITRIOS4       0    0    0    0     0    0    0
```

- This syntax makes it very easy to input data from excel or other databases
- We can go a long way with two dimensions, but if you want details about entering data in higher dimensions you can find them in [1]

Solving mathematical programs

Selecting an algorithm

- For linear programming, mixed integer linear programming, or convex quadratic programming problems you can use **Gurobi** and **CPLEX**
- For non-linear (non-convex) programming problems, a stable algorithm is **ipopt**
- You can access these algorithms, and many others, for free with an AMPL academic license
- In order to choose the algorithm that we want to use, we use the following syntax in the .mod file

```
option solver 'gurobi';
```

The solve command

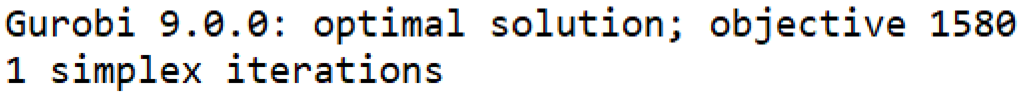
- Once we define our problem (declarations and data input), we can solve it using the solve command with the following syntax in the.mod file

```
solve;
```

- In order to run the program, we enter the model command on the AMPL terminal:

```
ampl: model ED.mod;
```

- The program will give us information about the size of the prlblem and its resolution in the IDE terminal
- The information that is printed on the terminal depends on the solver that we have selected



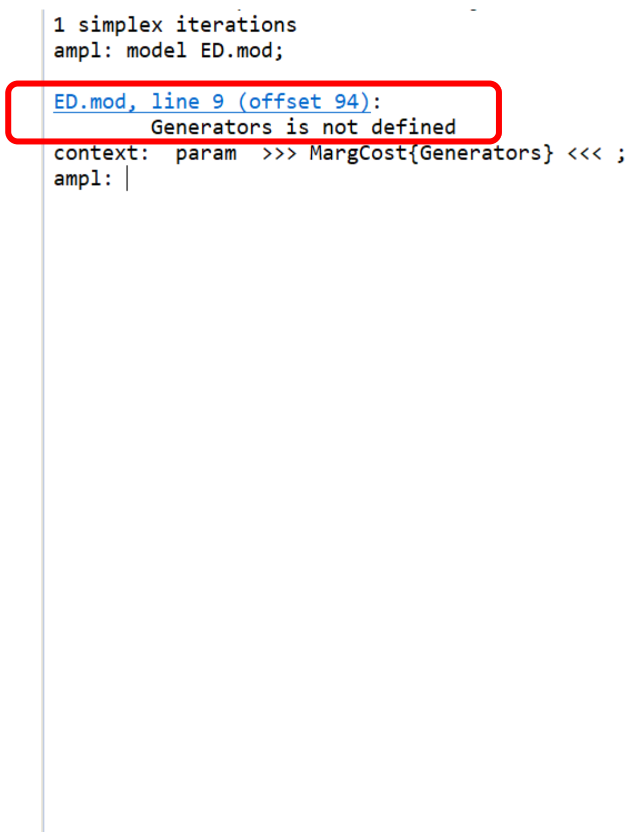
```
Gurobi 9.0.0: optimal solution; objective 1580
1 simplex iterations
```

Resolving bugs

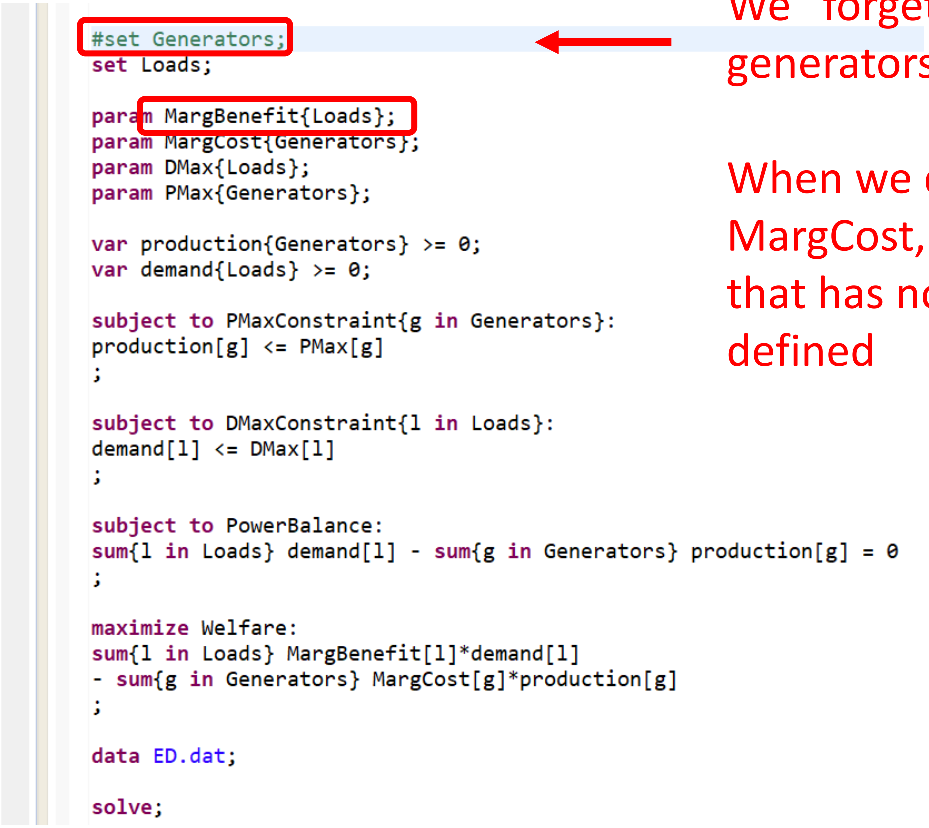
Resolving bugs

- There are simple syntax errors (easy to resolve) and more complex errors where the model does not behave “right” despite running (harder to resolve)
- The terminal gives us messages when something goes wrong in the first case, and we can iteratively correct our bugs

Resolving bugs: example



```
1 simplex iterations
ampl: model ED.mod;

ED.mod, line 9 (offset 94):
        Generators is not defined
context:  param  >>> MargCost{Generators} <<< ;
ampl: 
```



```
#set Generators;
set Loads;

param MargBenefit{Loads};
param MargCost{Generators};
param DMax{Loads};
param PMax{Generators};

var production{Generators} >= 0;
var demand{Loads} >= 0;

subject to PMaxConstraint{g in Generators}:
production[g] <= PMax[g]
;

subject to DMaxConstraint{l in Loads}:
demand[l] <= DMax[l]
;

subject to PowerBalance:
sum{l in Loads} demand[l] - sum{g in Generators} production[g] = 0
;

maximize Welfare:
sum{l in Loads} MargBenefit[l]*demand[l]
- sum{g in Generators} MargCost[g]*production[g]
;

data ED.dat;

solve;
```

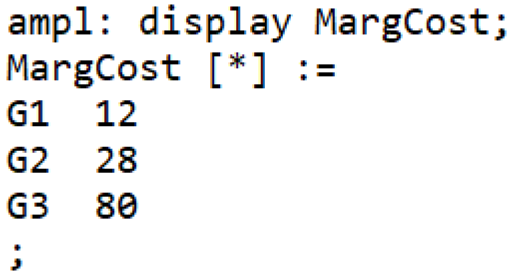
We “forget” to declare the set of generators

When we declare the parameter MargCost, we define it over a set that has not been previously defined

Display and print commands

The display command

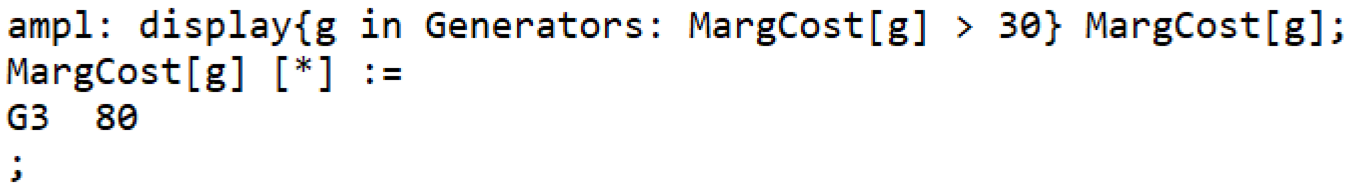
- An important advantage of AMPL is that we can “discuss” with the model after we solve it (or after we attempt to solve it and get a bug)
- The display command allows us to print parameters and variables on the terminal
- We can directly print a parameter (or variable) using the following syntax



```
ampl: display MargCost;
MargCost [*] :=
G1  12
G2  28
G3  80
;
```

The display command

- Μπορούμε να εισάγουμε συνθήκες για να στοχεύσουμε την απεικόνιση των παραμέτρων ή μεταβλητών του προβλήματος

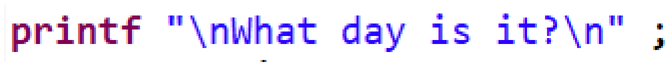


```
ampl: display{g in Generators: MargCost[g] > 30} MargCost[g];
MargCost[g] [*] :=
G3  80
;
```

- Αυτό είναι πολύ χρήσιμο για τη «συνομιλία» με το μοντέλο όταν προσπαθούμε να καταλάβουμε τη συμπεριφορά της λύσης ή όταν πιστεύουμε ότι υπάρχει λάθος στον κώδικα και ας είναι σωστό το συντακτικό

The printf command

- When we want to print with a format that we can control, we use the printf command

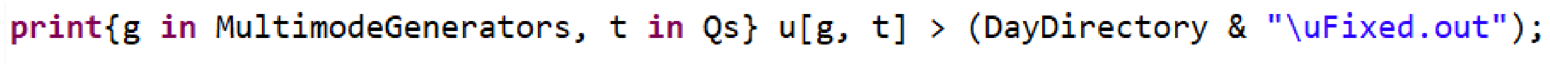


```
printf "\nWhat day is it?\n" ;
```

- Useful when we want to ask the user for input

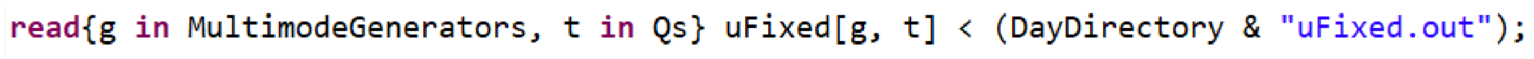
Printing in output files: the print and read commands

- The print command is not very human-readable, but is useful for passing output from one model as input to another
- For instance, a model that runs the Greek balancing market determines how units are activated (RTBM.mod), and then another model (RTBMPricing.mod) which determines prices reads the setpoints of units, based on the solution of the first model



```
print{g in MultimodeGenerators, t in Qs} u[g, t] > (DayDirectory & "\uFixed.out");
```

Command at the end of the RTBM.mod file



```
read{g in MultimodeGenerators, t in Qs} uFixed[g, t] < (DayDirectory & "uFixed.out");
```

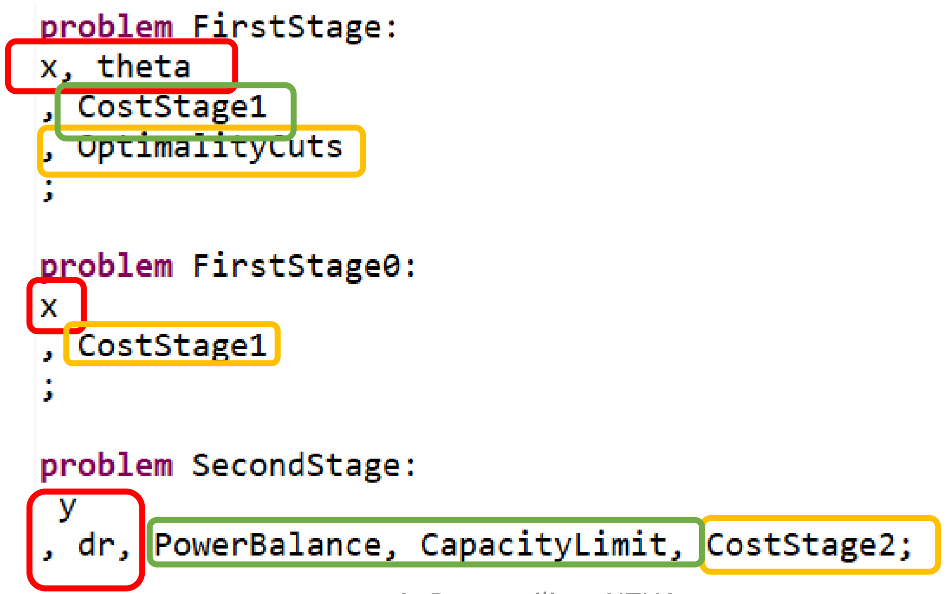
We run this after the RTBM.mod has executed

- Careful about reading data in the same order in both the input and output, because the print command only prints numerical values without labels

Implementing algorithms

The problem command

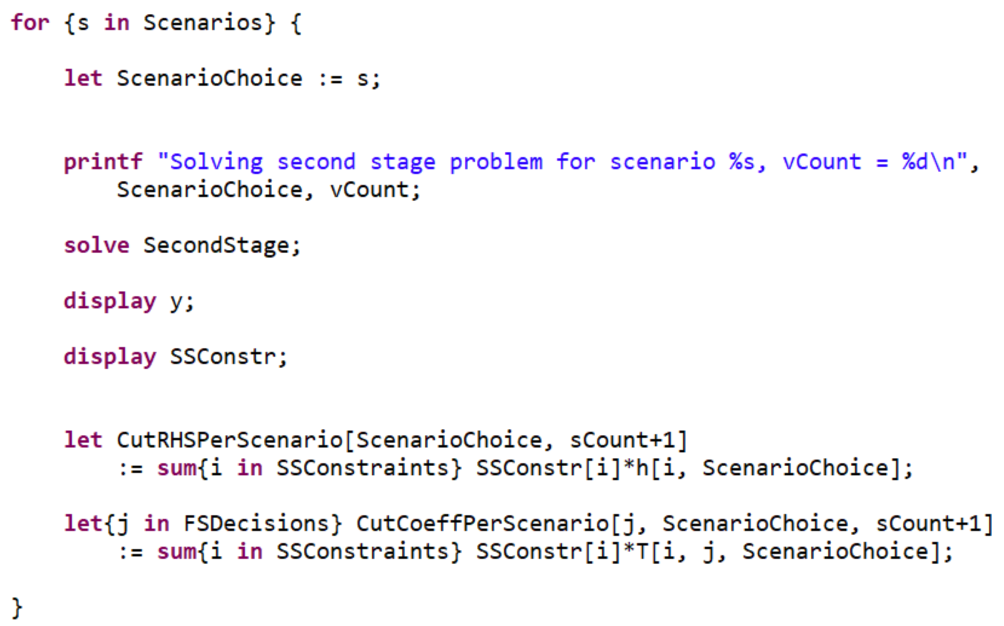
- When we implement optimization algorithms, we often use a subset of decision variables, constraints, and objective functions for defining different problems



```
problem FirstStage:
x, theta
, CostStage1
, OptimalityCuts
;

problem FirstStage0:
x
, CostStage1
;

problem SecondStage:
 y
, dr, PowerBalance, CapacityLimit, CostStage2;
```

The for loop

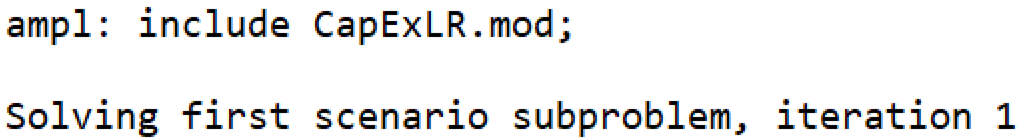
The for loop allows us to implement iterative algorithms, such as Lagrange relaxation, Benders decomposition, or Monte Carlo simulations



```
for {s in Scenarios} {

    let ScenarioChoice := s;


    printf "Solving second stage problem for scenario %s, vCount = %d\n",
        ScenarioChoice, vCount;

    solve SecondStage;

    display y;

    display SSConstr;


    let CutRHSPerScenario[ScenarioChoice, sCount+1]
        := sum{i in SSConstraints} SSConstr[i]*h[i, ScenarioChoice];

    let{j in FSDecisions} CutCoeffPerScenario[j, ScenarioChoice, sCount+1]
        := sum{i in SSConstraints} SSConstr[i]*T[i, j, ScenarioChoice];

}
```

The include command

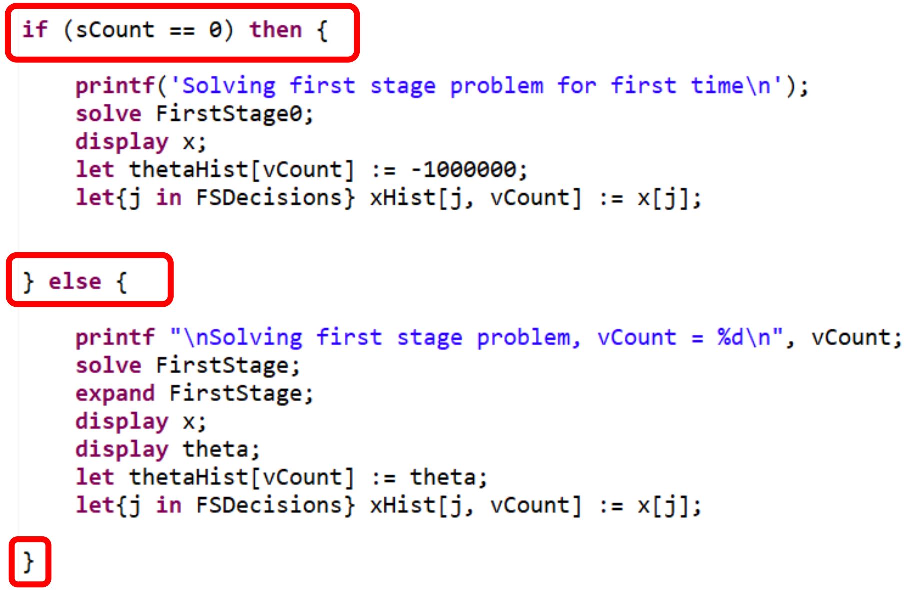
- The include command allows us to run iterative algorithms on the terminal



```
ampl: include CapExLR.mod;

Solving first scenario subproblem, iteration 1
```

The if-then-else logical checks

- The implementation of iterative algorithms often requires checking a logical condition
- The syntax follows the structure of the example



```
if (sCount == 0) then {

    printf('Solving first stage problem for first time\n');
    solve FirstStage0;
    display x;
    let thetaHist[vCount] := -1000000;
    let{j in FSDecisions} xHist[j, vCount] := x[j];

} else {

    printf "\nSolving first stage problem, vCount = %d\n", vCount;
    solve FirstStage;
    expand FirstStage;
    display x;
    display theta;
    let thetaHist[vCount] := theta;
    let{j in FSDecisions} xHist[j, vCount] := x[j];

}
```

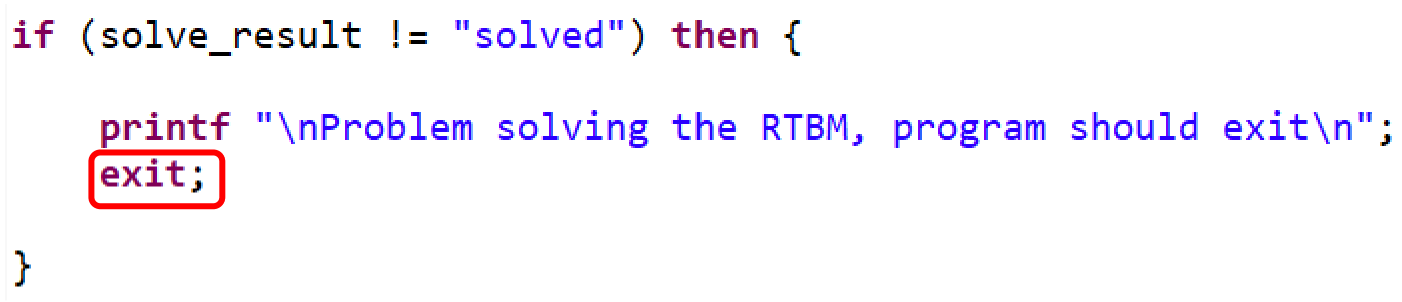
Receiving data interactively from the user

- The read command allows the user to assign values to parameters that affect the execution of the program (e.g. which electricit market we are clearing)
- The syntax uses the symbols < - in order to request input from the user, which are assigned as values to a parameter

```
printf "\nWhat day is it?\n" ;
read DayChoice <- ;
```

The exit command

- If something goes wrong during the execution of an algorithm, we can exit a loop and the entire program using the exit command

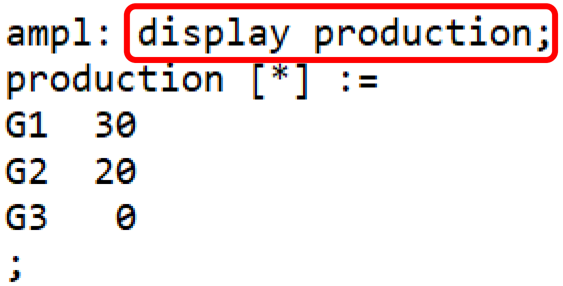


```
if (solve_result != "solved") then {

    printf "\nProblem solving the RTBM, program should exit\n";
    exit;

}
```

Analyzing the solution

Displaying decision variables

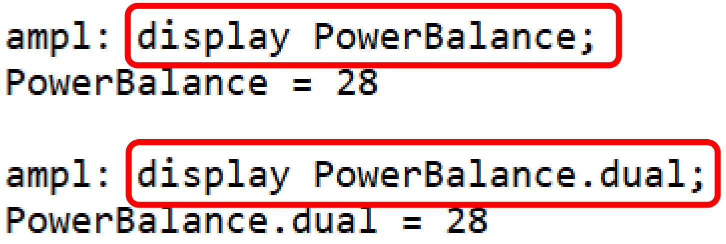
- After we solve our problem, we can use the display command to present the optimal value of decision variables
- The syntax is display x, where x is the name of the decision variable
- For the auction that we saw in the beginning of the presentation, the optimal production of electricity is displayed as follows:



```
ampl: display production;
production [*] :=
G1  30
G2  20
G3   0
;
```

Displaying dual values

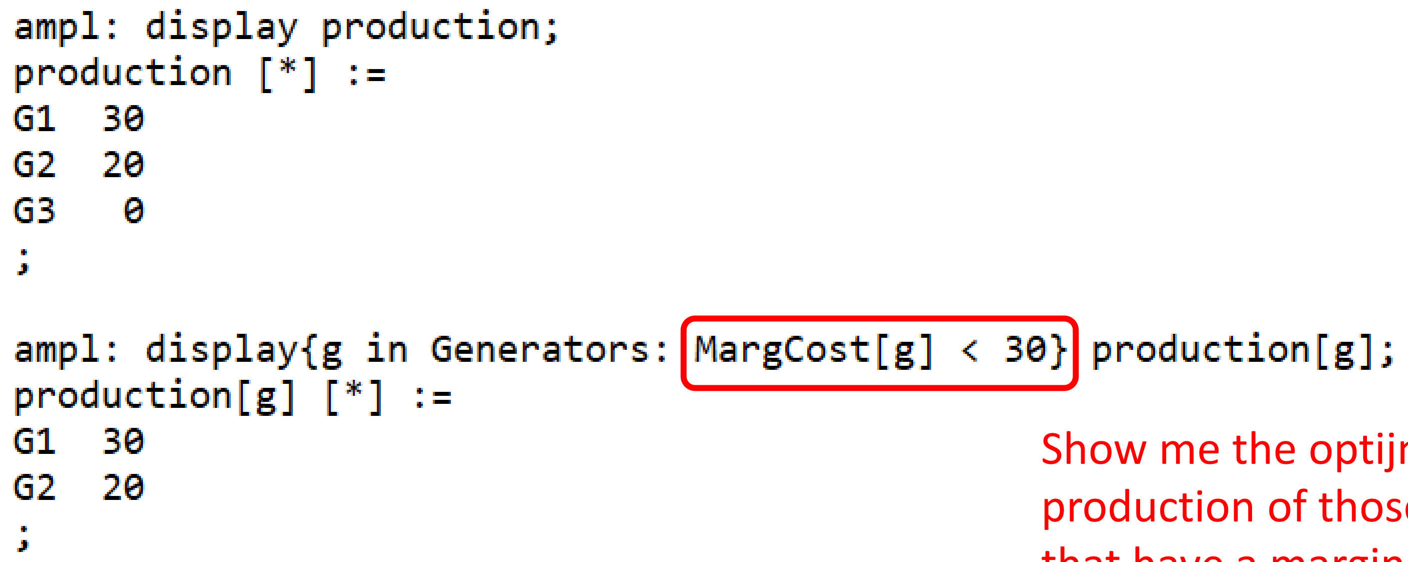
- After we solve our problem, we can also use the display command in order to present the optimal value of dual variables, which admit an economic interpretation
- The syntax is display constraint, or display constaint.dual, where constraint is the name of the relevant constraint
- For instance, the market clearing price of the auction is given as:



```
ampl: display PowerBalance;
PowerBalance = 28

ampl: display PowerBalance.dual;
PowerBalance.dual = 28
```

Displaying results under conditions

We can use logical conditions to filter the kind of information that is displayed



```
ampl: display production;
production [*] :=
G1  30
G2  20
G3   0
;

ampl: display{g in Generators: MargCost[g] < 30} production[g];
production[g] [*] :=
G1  30
G2  20
;
```

Show me the optijmal production of those generators that have a marginal cost lower than 30 $/MWh

References

[1] Robert Fourer, David M. Gay, and Brian W. Kernighan, “AMPL: A Modeling Language for Mathematical Programming” https://ampl.com/resources/the-ampl-book/