# Introduction to AMPL
# A Tutorial*

September 13, 2000

AMPL is a powerful language designed specifically for mathematical programming. AMPL has many features and options; however this tutorial covers a small subset of these[1]. Sections 1 through 5 provide an introduction to modeling Linear Programming (LP) problems with AMPL. Sections 6 and 7 introduce AMPL's modeling capabilities for Integer Programming (IP) and Nonlinear Programming (NLP), respectively.

## 1 A Simple LP Model

Consider the following example. Berkeley Paint Company makes two colors of paint, blue and gold. The blue paint sells for $10 per gallon, while the gold paint sells for $15 per gallon. The company has one factory, and can make only one color of paint at a time. However, blue paint is easier to make so the factory can make 40 gallons per hour of blue paint, but only 30 gallons per hour of gold paint. In addition, the marketing department tells manufacturing that they can sell at most 860 gallons of gold paint and 1000 gallons of blue paint. If a week is 40 hours and paint can't be stored until next week, we need to determine how many gallons of blue and gold paint to make; so that the total revenue is maximized.

*Let $PaintB$ represent the gallons of blue paint we will make, and $PaintG$ represent the gallons of gold paint we will make. Then the problem is formulated as the following LP:*

$$\begin{array}{ll} \max & 10PaintB + 15PaintG \\ s.t.: & (1/40)PaintB + (1/30)PaintG \leq 40 \\ & 0 \leq PaintB \leq 1000 \\ & 0 \leq PaintG \leq 860. \end{array}$$

AMPL is designed to allow mathematical programs to be entered in a syntax very close to the algebraic notation you see above. To use AMPL, we need to create a text file with the mathematical program code.

To create a text file, you can use any text editor you feel comfortable with. For example, you can use **Notepad**, which is found on every Windows 95 and Windows NT system. To create the following program file, start **Notepad**, and enter the following text.

*This tutorial was originally prepared by P. Kaminsky. The original version is later modified and expanded by D. Rajan.

[1]For more information, see the book

*AMPL: A Modeling Language for Mathematical Programming* by R. Fourer, D. M. Gay, and B. W. Kernighan (1993: Boyd & Fraser)

```
## Example One
var PaintB;  # amount of blue
var PaintG;  # amount of gold

maximize profit: 10*PaintB + 15*PaintG;
subject to time: (1/40)*PaintB + (1/30)*PaintG <=  40;
subject to blue_limit: 0 <= PaintB <= 1000;
subject to gold_limit: 0 <= PaintG <= 860;
```

Note the following things about the AMPL language:

- The `#` symbol indicates the start of a comment. Everything after that symbol is ignored.
- Variables must be declared using the **var** keyword.
- All lines of code **must** end with a semi-colon.
- The objective starts with **maximize** or **minimize**, a name, and a semi-colon (:). After that, the objective statement appears.
- Each constraint starts with **subject to**, a name, and a semi-colon. After that, the equation or inequality appears.
- Names must be unique. A variable and a constraint cannot have the same name.
- AMPL is case sensitive. Keywords must be in lower case.

After the file is created, it should be saved with the extension **.mod**. For the following examples, the file is saved as on the `z:` drive in the directory `myFiles` as `z:\myFiles\examp_1.mod`. Note that long file names cannot be used with AMPL. The filenames you select must be at most eight characters followed by a three character extension.

Also, sometimes **Notepad** appends the `.txt` extension to files automatically, so you may have to rename the file in order to get rid of the `.txt`. For example, some versions of **Notepad** will automatically save the file from the previous example as `z:\myfiles\examp_1.mod.txt`, so you may have to edit the filename to get ride of the extra extension.

Once the file is created and you've checked the file name, run the AMPL program by selecting it from the appropriate start menu. After some text appears, you should see the AMPL prompt, which is `ampl:`.

First, you have to select a solver, which is the program that AMPL calls to actually solve the math program. We will be using the solver called **cplex** for these examples, so type:

```
option solver cplex;
```

Don't forget to end every command with a semi-colon. Every time you start AMPL, you need to select a solver using this `option solver` command. For now, we will always use **cplex**, but we will later learn about other solvers.

Now, type in the following command, which tells AMPL to read in the file you have created.

```
model  z:\myFiles\examp_1.mod;
```

If you typed in the model correctly, you will get no error messages. If you made a mistake, correct the model file. Before reloading the model, you must first reset AMPL by typing:

```
reset;
```

To reload the model, type in:

```
model  z:\myFiles\examp_1.mod;
```

Once the model file is successfully loaded, tell AMPL to run the model by typing:

```
solve;
```

Once AMPL has solved the model [2], it displays some information, followed by the objective value. Your solution should look something like:

```
CPLEX 6.5.3: optimal solution; objective 17433.33333
2 simplex iterations (0 in phase I)
```

Recall that what we are really interested in is how much blue and gold paint to make. Thus, we need to see the final values assigned to the decision variables $PaintB$ and $PaintG$. To see the value of $PaintB$, type:

```
display PaintB;
```

Do the same to see the value of $PaintG$.

If you want to direct the output of the `display` command to a file, use one of the following two formats. To direct output to a file on the `z:` drive in the directory `myFiles` called `z:\myfiles\examp_1.out`, enter:

```
display PaintB > z:\myFiles\examp_1.out;
```

However, this will **over-write** any other file called `z:\myfiles\examp_1.out`. If you instead want to add to an **already existing file** called `z:\myfiles\examp_1.out`, use the following syntax:

```
display PaintB >> z:\myFiles\examp_1.out;
```

Note that here we use the double "greater than" sign rather than the single.

# 2 A More General LP Model

Although it was easy to transform the previous LP into a format AMPL understands, it is clear that if the problem had more details, or changed frequently, it would be much harder.

For this reason, we typically use a more general algebraic way of stating linear programming models. Consider the following:

| | |
|---|---|
| Given: | $n$ = number of paint colors |
| | $t$ = total time available |
| | $p_i$ = profit per gallon of paint color $i$, $i = 1, \ldots, n$ |
| | $r_i$ = gallons per hour of paint color $i$, $i = 1, \ldots, n$ |
| | $m_i$ = maximum demand of paint color $i$, $i = 1, \ldots, n$ |
| Variable: | $x_i$ = gallons of paint color $i$ to be made, $i = 1, \ldots, n$ |
| Maximize: | $\sum_{i=1}^{n} p_i x_i$ |
| Subject to: | $\sum_{i=1}^{n} (1/r_i) x_i \leq t$ |
| | $0 \leq x_i \leq m_i$ for each $i, i = 1, \ldots, n$. |

Clearly, if $n = 2$, $t = 40$, $p_1 = 10$, $p_2 = 15$, $r_1 = 40$, $r_2 = 30$, $m_1 = 1000$, and $m_2 = 860$, this is exactly the same model as in the previous section. In fact, it took even longer to write! You can imagine, however, that as the number of paint colors increases, the algebraic formulation becomes more efficient. If we have

[2] AMPL actually doesn't solve the model itself. Instead, it calls a *solver* program to do that.

some way of defining the parameters $(n, t, p_i, r_i, m_i)$, the model will stay the same regardless of their values. Thus, we see that Example One is actually an instance of the Berkeley Paint Problem, rather than a model.

This is exactly what AMPL does. It gives us a way to express the algebraic representation of the model and these value for the parameters separately. It does this using two separate files, a model file and a data file. AMPL reads the model from the **.mod** file, data from the **.dat** file and puts them together into a format that the solver understands. Then, it hands over this problem instance to the solver, which in turn, solves the instance, and hands back the solution to AMPL. This is illustrated in Figure 1.

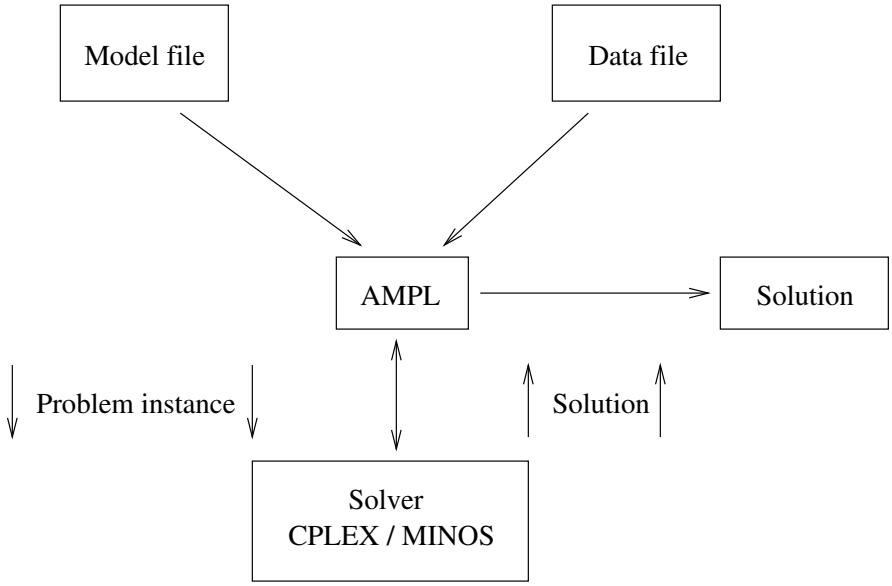


Figure 1.

First, let's look at the model file, which we save on the `z:` drive as `z:\myFiles\examp_2.mod`:

```
## Example Two - Model

param n;
param t;
param p{i in 1..n};
param r{i in 1..n};
param m{i in 1..n};

var paint{i in 1..n};

maximize profit: sum{i in 1..n} p[i]*paint[i];
subject to time: sum{i in 1..n} (1/r[i])*paint[i] <= t;
subject to capacity{i in 1..n}: 0 <= paint[i] <= m[i];
```

Compare this model to the one in the previous section. It is easy to see that this is a more general statement of the previous one. Note the following things:

- Each parameter declaration starts with the keyword **param**;

- Indexed parameters are declared using the syntax *namesome_variable* **in** *range*. For example, for the parameter which we declared algebraically as $p_i, i = 1..n$, we can use $i$ as the index variable, and $1..n$ as the range, and we thus get `p{i in 1..n}`.
- Indexed variables are declared in the same way, starting with the **var** keyword.
- Summations are written similarly. $\sum_{i=1}^{n}$ is written `sum{i in 1..n}`.
- Sometimes, we represent sets of constraints in a single line. In the formulation at the beginning of this section, we wrote:
  $0 \leq paint_i \leq m_i$ for each $i, i = 1, \ldots, n$
  for one of the constraints. Actually, this is a set of $n$ constraints, one for each paint color. In AMPL, we express this in the expression in braces following the name of the constraint. Thus, this constraint becomes:
  `subject to paint {i in 1..n}:  0 <= paint[i] <= m[i];`
- Variable and parameter names can be anything meaningful, made up of upper and lower case letters, digits, and underscores. We could have (and probably should have) used the names `rate` and `maxDemand` in place of `r` and `m`.

In addition to specifying the model, we also must specify the data. This is the data for the same problem we solved in the previous section, which we save on the `z:` drive as `z:\myFiles\examp_2.dat`.

```
## Example Two - Data

param n:= 2;
param t:= 40;

param p:= 1  10  2  15;
param r:= 1  40  2  30;
param m:= 1  1000  2   860;
```

As you can see, one way to specify parameter data is to use the **param** keyword, a colon and equal sign, and the value. If a parameter has more than one component, simply list the parameters index (in this case, `1` or `2`) followed by the value.

Note that because AMPL ignores carriage returns in files and instead looks for semi-colons to terminate lines, you could also write the last three parameters as:

```
param p:= 1  10
          2  15;
param r:= 1  40
          2  30;
param m:= 1  1000
          2   860;
```

Next, if the AMPL program is not running, run it the same way as before. If it is currently, running, first reset AMPL by typing:

```
reset;
```

Now, type in the following commands, which tells AMPL to read in the model and data files you have just created, and then solve the model. Notice that you must load in the model file before the data file. Also, if there is an error in the model or data file, don't forget to use `reset` before reloading them.

```
model  z:\myFiles\examp_2.mod;
data z:\myFiles\examp_2.dat;
solve;
```

If the model and data were entered correctly, you should get exactly the same output and objective value as before. To see the values assigned to `paint[1]` and `paint[2]`, type:

```
display paint;
```

AMPL will list both *paint* values.

By changing only the data file, you can now resolve this model with different cost parameters, and different numbers of possible paint colors.

# 3 A Different Way to Enter the Data

AMPL has many more ways to express data, some of which are more concise than the way we saw in the previous section. Although many of these only apply to more complex models, the data for the simple example displayed above could be written:

```
## Example Two - Another way to write the data

param n:= 2;
param t:= 40;

param:  p r m:=
1 10 40 1000
2 15 30 860;
```

# 4 Sets

In the previous examples, we had to remember that each parameter and variable number 1 was for blue paint, and 2 was for gold paint. Instead, AMPL lets us define a set with elements `blue` and `gold`, and use these names directly to index parameters and variables. Look at the following example:

```
## Example Two - Model with sets

set P;

param t;
param p{i in P};
param r{i in P};
param m{i in P};

var paint{i in P};

maximize profit: sum{i in P} p[i]*paint[i];
subject to time: sum{i in P} (1/r[i])*paint[i] <= t;
subject to capacity{i in P}: 0 <= paint[i] <= m[i];
```

If you compare this version of model two to the original version, you will see that they are very similar. However, there are several important differences:

- Rather than defining the parameter `n` to represent the *number of paint colors*, we have defined the **set** `P` which actually contains the paint colors. This will become clearer when we define the data below.
- Rather than indexing parameters over the indices `1..n`, we have indexed them over the members of the set `P`.

Now, the data can be defined as follows:

```
## Example Two - Data with sets

set P:= blue gold;

param t:= 40;

param p:= blue  10
          gold  15;
param r:= blue  40
          gold  30;
param m:= blue  1000
          gold  860;
```

Note that after the set was defined, each parameter element was stated using the set element name, rather than the index number. As you can see, this is much easier to read and follow. In fact, we can combine the set notation with the more compact data notation introduced in the previous section, and we get:

```
## Example Two - Another way to write the data with sets

set P:= blue gold;
param t:= 40;

param:  p r m:=
blue 10 40 1000
gold 15 30 860;
```

# 5 Variables and Parameters With Two Dimensions

Often, a mathematical programming model has variables and parameters with two dimensions. We can write the parameter data in table form, and AMPL allows us to enter this information in table form in the data file.

Consider the following example. Berkeley Paint Company has expanded, and now has three warehouses, all filled with blue paint. In a particular week, paint has to be shipped to four customers. For each warehouse and customer, *per gallon* shipping costs are different. Shipping costs are displayed in the following table. The warehouses are in the first column, the customers are in the first row, and the warehouse-customer shipping cost can be read from the table.

| | Cust 1 | Cust. 2 | Cust. 3 | Cust. 4 |
|---|---|---|---|---|
| Warehouse 1 | 1 | 2 | 1 | 3 |
| Warehouse 2 | 3 | 5 | 1 | 4 |
| Warehouse 3 | 2 | 2 | 2 | 2 |

In addition, the supply of blue paint at each of the warehouses is:

Warehouse 1 250
Warehouse 2 800
Warehouse 3 760

The demand at each customer is:
Customer 1: 300
Customer 2: 320
Customer 3: 800
Customer 4: 390

Note that total supply equals total demand. The following AMPL model is set up to minimize total cost subject to meeting demand. We save the file on the `z:` drive as `z:\myFiles\examp_3.mod`.

```
## Example Three - Model

param warehouse;   # number of warehouses
param customer;   # number of customers
 #transportation cost from warehouse i
 #to customer j
param cost{i in 1..warehouse, j in 1..customer};
param supply{i in 1..warehouse};    #supply at warehouse i
param demand{i in 1..customer};    #demand at customer j

var amount{i in 1..warehouse, j in 1..customer};

minimize Cost:
    sum{i in 1..warehouse, j in 1..customer} cost[i,j]*amount[i,j];
subject to Supply {i in 1..warehouse}:
    sum{j in 1..customer} amount[i,j] = supply[i];
subject to Demand {j in 1..customer}:
    sum{i in 1..warehouse} amount[i,j] = demand[j];
subject to positive{i in 1..warehouse, j in 1..customer}:
    amount[i,j]>=0;
```

Note the following things about this model:

- Each parameter and variable has a longer and more descriptive name.
- Parameters and variables with double indexes (cost and amount) are indexed exactly as before, except that two indexes are given, separated by a comma.
- The objective `Cost` and the constraints `Supply` and `Demand` are capitalized so that AMPL doesn't confuse them with the parameters `supply`, `demand`, and `cost`.

To complete the model, we need to specify the data. The data file is saved on the `z:` drive as `z:\myfiles\examp_3.dat`.

```
## Example Three - Data

param warehouse:= 3;
param customer:= 4;
param cost:     1     2     3     4 :=
       1        1     2     1     3
```

```
       2        3     5     1     4
       3        2     2     2     2;

param supply:= 1 250   2 800   3 760;
param demand:= 1 300   2 320   3 800   4 390;
```

Pay particular attention to the way the `cost` parameter data is entered. Note the colon after the the parameter name, and the colon and equal sign after the final warehouse index. Also, note that the the first index (warehouses) is in the first column, while the second (customers) is in the first row, exactly like the table in which we first specified the data.

These model and data files are loaded and executed exactly as before. Note that if you enter `display amount;` after the model has run, AMPL displays all of the values of the **amount** variable. If the program settings are correct, **amount** will be displayed in tabular form.

Also, we could use the set notation introduced in the previous section to clarify the model and data. This is especially helpful if the customers and warehouses have names or location names. For example, a possible model and data follow:

```
## Example Three - Model using sets

set Warehouses;
set Customers;
 #transportation cost from warehouse i
 #to customer j
param cost{i in Warehouses, j in Customers};
param supply{i in Warehouses};    #supply at warehouse i
param demand{j in Customers};    #demand at customer j

var amount{i in Warehouses, j in Customers};

minimize Cost:
    sum{i in Warehouses, j in Customers} cost[i,j]*amount[i,j];
subject to Supply {i in Warehouses}:
    sum{j in Customers} amount[i,j] = supply[i];
subject to Demand {j in Customers}:
    sum{i in Warehouses} amount[i,j] = demand[j];
subject to positive{i in Warehouses, j in Customers}:
    amount[i,j]>=0;
```

The data file might now look like:

```
## Example Three - Data with sets

set Warehouses:= Oakland San_Jose Albany;
set Customers:= Home_Depot K_mart Wal_mart Ace;

param cost:     Home_Depot K_mart Wal_mart Ace:=
       Oakland        1      2      1       3
       San_Jose       3      5      1       4
       Albany         2      2      2       2;

param supply:= Oakland     250
               San_Jose    800
```

```
                Albany      760;
param demand:= Home_Depot  300
                K_mart      320
                Wal_mart    800
                Ace         390;
```

# 6 Integer Programming

Often, a mathematical programming model requires that some (or all) variables take on only integral values. Fortunately, AMPL allows us to incorporate this with only a small change in the model. By adding the keyword **integer** to the **var** declaration, we can restrict the declared variable to integral values. Furthermore, by adding the keyword **binary** to the **var** declaration, we can restrict the declared variable to the values **0** and **1**.

To illustrate, consider the following example. Berkeley Paint Company has now expanded the supply capacity at each of its warehouses. However, this expansion has come at a price. In addition to the shipping costs, now the company has to pay a fixed cost for opening a warehouse.

The supply capacity of paint at each of the warehouses is:
Warehouse 1 550
Warehouse 2 1100
Warehouse 3 1060

The cost of opening each of the warehouses is:
Warehouse 1 500
Warehouse 2 500
Warehouse 3 500

Note that available supply now exceeds total demand. The following AMPL model is set up to minimize total cost subject to meeting demand. In fact, this problem is called the Warehouse Location Problem. We save the file on the `z:` drive as `z:\myFiles\examp_4.mod`.

We use the set notation introduced in Section 4 to clarify the model and data. This is especially helpful if the customers and warehouses have names or location names.

```
## Example Four - Mixed-IP model file for the warehouse location problem

set Warehouses;
set Customers;
 #transportation cost from warehouse i
 #to customer j
param cost{i in Warehouses, j in Customers};
param supply{i in Warehouses};    #supply capacity at warehouse i
param demand{j in Customers};    #demand at customer j
param fixed_charge{i in Warehouses};    #cost of opening warehouse j

var amount{i in Warehouses, j in Customers};
var open{i in Warehouses} binary;   # = 1 if warehouse i is opened, 0 otherwise

minimize Cost:
    sum{i in Warehouses, j in Customers} cost[i,j]*amount[i,j]
          + sum{i in Warehouses} fixed_charge[i]*open[i];
subject to Supply {i in Warehouses}:
```

```
    sum{j in Customers} amount[i,j] <= supply[i]*open[i];
subject to Demand {j in Customers}:
    sum{i in Warehouses} amount[i,j] = demand[j];
subject to positive{i in Warehouses, j in Customers}:
    amount[i,j]>=0;
```

Note the following changes in this model:

- We introduced the variable **open** to indicate whether the warehouse is open, and defined it to be **0-1**.
- The parameter **fixed_charge** stores the cost for opening the warehouse.
- The total cost now includes both the shipping costs and the fixed charge for opening a warehouse.
- The constraint **Supply** is changed to ensure that a warehouse supplies paint only if it is open.

To complete the model, we need to specify the corresponding data. The data file is saved on the `z:` drive as `z:\myfiles\examp_4.dat`.

```
## Example Four - Data file for the warehouse location problem

set Warehouses:= Oakland San_Jose Albany;
set Customers:= Home_Depot K_mart Wal_mart Ace;

param cost:     Home_Depot K_mart Wal_mart Ace:=
       Oakland        1      2      1       3
       San_Jose       3      5      1       4
       Albany         2      2      2       2;

param supply:=       Oakland     550
                     San_Jose    1100
                     Albany      1060;
param demand:=       Home_Depot  300
                     K_mart      320
                     Wal_mart    800
                     Ace         390;
param fixed_charge:= Oakland     500
                     San_Jose    500
                     Albany      500;
```

# 7 Nonlinear programming

Many mathematical programs include nonlinear functions in the constraints and for the objective. Fortunately, AMPL allows us to model and solve nonlinear programs, as well.

First, you have to change the solver, since **cplex** does not solve nonlinear programming problems. For nonlinear problems, we shall use the solver **minos**. So type:

```
option solver minos;
```

To illustrate, let us consider the example of **Portfolio Optimization**. Suppose that we have a set of alternate investments **A**, and that we know the expected rate of return **R** for each of these investments for a set of years **T**. From this data, we can calculate the covariance matrix for the investments.

We would like to

- Maximize the expected return subject to a maximum risk threshold.
- Minimize the expected risk subject to a minimum expected return.

The following AMPL model is set up to maximize the expected return subject to a maximum risk threshold. We save the model file on the `z:` drive as `z:\myFiles\examp_5.mod`.

```
## Example 5 - Nonlinear Portfolio Optimization Model

set A;                  # asset categories
set T := {1984..1994};  # years

param s_max default 0.00305; # i.e., a 5.522 percent std. deviation on reward

param R {T,A};

param mean {j in A} := ( sum{i in T} R[i,j] )/card(T);

param Rtilde {i in T, j in A} := R[i,j] - mean[j];

var alloc{A} >=0;

minimize reward: - sum{j in A} mean[j]*alloc[j] ;

subject to risk_bound:
    sum{i in T} (sum{j in A} Rtilde[i,j]*alloc[j])^2 / card{T} <= s_max;

subject to tot_mass:
    sum{j in A} alloc[j] = 1;
```

Note the following:

- We introduced the variable **alloc** to indicate the fraction of our resources that we utilize for each investment
- The expected rate of return is given by $\sum_{j \in A} mean_j * alloc_j$.
- The expected risk is given by $\sum_{i \in T} \left(\sum_{j \in A} \tilde{R}_{ij} * alloc_j\right)^2 / card(T)$; where $\tilde{R}_{ij} = R_{ij} - mean_j$.
- The Portfolio Optimization model has a linear objective and a quadratic constraint.

To complete the model, we need to specify the corresponding data. The data file is saved on the `z:` drive as `z:\myfiles\examp_5.dat`.

```
## Example 5 - Data

set A :=
    US_3-MONTH_T-BILLS US_GOVN_LONG_BONDS SP_500 WILSHIRE_5000;

param R:
US_3-MONTH_T-BILLS US_GOVN_LONG_BONDS SP_500 WILSHIRE_5000 :=
1984   1.103   1.159   1.061   1.030
1985   1.080   1.366   1.316   1.326
1986   1.063   1.309   1.186   1.161
```

```
1987   1.061   0.925   1.052   1.023
1988   1.071   1.086   1.165   1.179
1989   1.087   1.212   1.316   1.292
1990   1.080   1.054   0.968   0.938
1991   1.057   1.193   1.304   1.342
1992   1.036   1.079   1.076   1.090
1993   1.031   1.217   1.100   1.113
1994   1.045   0.889   1.012   0.999
;
```

You should be aware of many of the pitfalls in non-linear optimization:

- Function range violations (some value goes to $\pm\infty$)
- Multiple local optima
- Stationary points

Most of these will be discussed during the lectures. The nonlinear optimization solver MINOS, which is used by AMPL, is only capable of finding a solution that satisfies "the first order condition" (will be discussed in class), which is not necessarily optimal. Very often, in order to get better solutions, we have no choice but to run the model again with a fresh starting point. This can be specified in the **var** declaration with an optional := phrase; for the above example, we can write

```
var x{A} >=0, :=0.25;
```

to set every x{A} to 0.25 initially. The solver will use these values as a starting guess.

# 8 Conclusion

In this tutorial, we have just scratched the surface of the things you can do with AMPL. The ultimate reference for AMPL questions is the book by *Fourer et al.*, which is on reserve in the Engineering Library.