
Production Models: Maximizing Profits

As we stated in the Introduction, mathematical programming is a technique for solving certain kinds of problems — notably maximizing profits and minimizing costs — subject to constraints on resources, capacities, supplies, demands, and the like. AMPL is a language for specifying such optimization problems. It provides an algebraic notation that is very close to the way that you would describe a problem mathematically, so that it is easy to convert from a familiar mathematical description to AMPL.

We will concentrate initially on linear programming, which is the best known and easiest case; other kinds of mathematical programming are taken up later in the book. This chapter addresses one of the most common applications of linear programming: maximizing the profit of some operation, subject to constraints that limit what can be produced. Chapters 2 and 3 are devoted to two other equally common kinds of linear programs, and Chapter 4 shows how linear programming models can be replicated and combined to produce truly large-scale problems. These chapters are written with the beginner in mind, but experienced practitioners of mathematical programming should find them useful as a quick introduction to AMPL.

We begin with a linear program (or LP for short) in only two decision variables, motivated by a mythical steelmaking operation. This will provide a quick review of linear programming to refresh your memory if you already have some experience, or to help you get started if you’re just learning. We’ll show how the same LP can be represented as a general algebraic model of production, together with specific data. Then we’ll show how to express several linear programming problems in AMPL and how to run AMPL and a solver to produce a solution.

The separation of model and data is the key to describing more complex linear programs in a concise and understandable fashion. The final example of the chapter illustrates this by presenting several enhancements to the model.

1.1 A two-variable linear program

An (extremely simplified) steel company must decide how to allocate next week's time on a rolling mill. The mill takes unfinished slabs of steel as input, and can produce either of two semi-finished products, which we will call bands and coils. (The terminology is not entirely standard; see the bibliography at the end of the chapter for some accounts of realistic LP applications in steelmaking.) The mill's two products come off the rolling line at different rates:

Tons per hour:	Bands	200
	Coils	140

and they also have different profitabilities:

Profit per ton:	Bands	\$25
	Coils	\$30

To further complicate matters, the following weekly production amounts are the most that can be justified in light of the currently booked orders:

Maximum tons:	Bands	6,000
	Coils	4,000

The question facing the company is as follows: If 40 hours of production time are available this week, how many tons of bands and how many tons of coils should be produced to bring in the greatest total profit?

While we are given numeric values for production rates and per-unit profits, the tons of bands and of coils to be produced are as yet unknown. These quantities are the decision *variables* whose values we must determine so as to maximize profits. The purpose of the linear program is to specify the profits and production limitations as explicit formulas involving the variables, so that the desired values of the variables can be determined systematically.

In an algebraic statement of a linear program, it is customary to use a mathematical shorthand for the variables. Thus we will write X_B for the number of tons of bands to be produced, and X_C for tons of coils. The total hours to produce all these tons is then given by

$$(\text{hours to make a ton of bands}) \cdot X_B + (\text{hours to make a ton of coils}) \cdot X_C$$

This number cannot exceed the 40 hours available. Since hours per ton is the reciprocal of the tons per hour given above, we have a *constraint* on the variables:

$$(1/200) X_B + (1/140) X_C \leq 40.$$

There are also production limits:

$$\begin{aligned} 0 &\leq X_B \leq 6000 \\ 0 &\leq X_C \leq 4000 \end{aligned}$$

In the statement of the problem above, the upper limits were specified, but the lower limits were assumed \hat{A} —it was obvious that a negative production of bands or coils would be meaningless. Dealing with a computer, however, it is necessary to be quite explicit.

By analogy with the formula for total hours, the total profit must be

$$(\text{profit per ton of bands}) \cdot X_B + (\text{profit per ton of coils}) \cdot X_C$$

That is, our objective is to maximize $25 X_B + 30 X_C$. Putting this all together, we have the following linear program:

$$\text{Maximize } 25 X_B + 30 X_C$$

$$\text{Subject to } (1/200) X_B + (1/140) X_C \leq 40$$

$$0 \leq X_B \leq 6000$$

$$0 \leq X_C \leq 4000$$

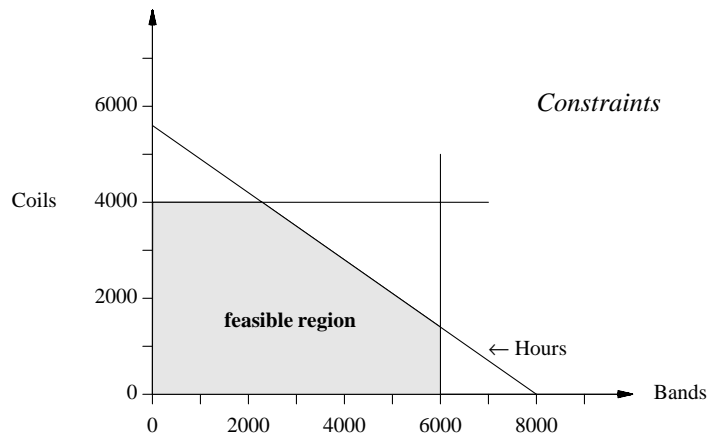
This is a very simple linear program, so we \hat{A} solve it by hand in a couple of ways, and then check the answer with AMPL.

First, by multiplying profit per ton times tons per hour, we can determine the profit per hour of mill time for each product:

Profit per hour:	Bands	\$5,000
	Coils	\$4,200

Bands are clearly a more profitable use of mill time, so to maximize profit we should produce as many bands as the production limit will allow \hat{A} 6,000 tons, which takes 30 hours. Then we should use the remaining 10 hours to make coils \hat{A} 1,400 tons in all. The profit is \$25 times 6,000 tons plus \$30 times 1,400 tons, for a total of \$192,000.

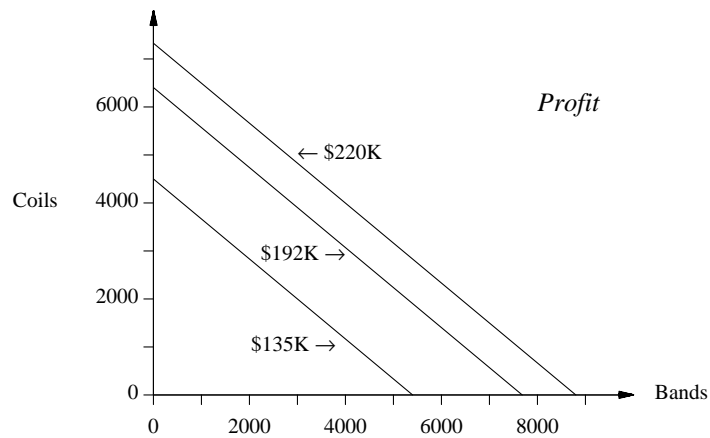
Alternatively, since there are only two variables, we can show the possibilities graphically. If X_B values are plotted along the horizontal axis, and X_C values along the vertical axis, each point represents a choice of values, or solution, for the decision variables:



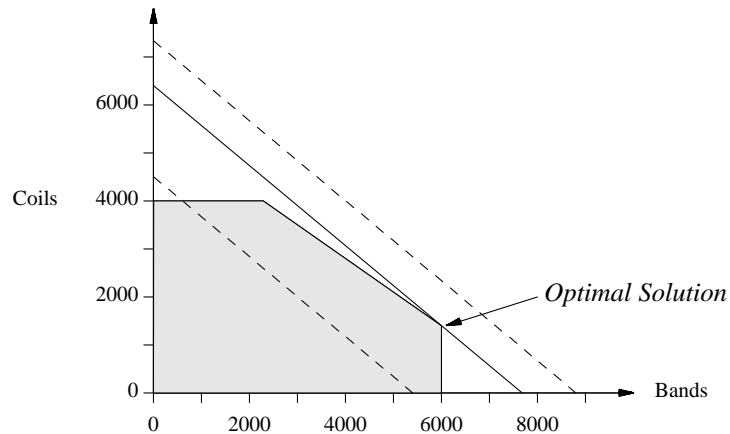
The horizontal line represents the production limit on coils, the vertical on bands. The diagonal line is the constraint on hours; each point on that line represents a combination of bands and coils that requires exactly 40 hours of production time, and any point downward and to the left requires less than 40 hours.

The shaded region bounded by the axes and these three lines corresponds exactly to the *feasible* solutions—those that satisfy all three constraints. Among all the feasible solutions represented in this region, we seek the one that maximizes the profit.

For this problem, a line of slope $-25/30$ represents combinations that produce the same profit; for example, in the figure below, the line from (0, 4500) to (5400, 0) represents combinations that yield \$135,000 profit. Different profits give different but parallel lines in the figure, with higher profits giving lines that are higher and further to the right.



If we combine these two plots, we can see the profit-maximizing, or *optimal*, feasible solution:



The line segment for profit equal to \$135,000 is partly within the feasible region; any point on this line and within the region corresponds to a solution that achieves a profit of \$135,000. On the other hand, the line for \$220,000 does not intersect the feasible region at all; this tells us that there is no way to achieve a profit as high as \$220,000. Viewed in this way, solving the linear program reduces to answering the following question: Among all profit lines that intersect the feasible region, which is highest and furthest to the right? The answer is the middle line, which just touches the region at one of the corners. This point corresponds to 6,000 tons of bands and 1,400 tons of coils, and a profit of \$192,000 — the same as we found before.

1.2 The two-variable linear program in AMPL

Solving this linear program with AMPL can be as simple as typing AMPL's description of the linear program,

```
var XB;
var XC;
maximize Profit: 25 * XB + 30 * XC;
subject to Time: (1/200) * XB + (1/140) * XC <= 40;
subject to B_limit: 0 <= XB <= 6000;
subject to C_limit: 0 <= XC <= 4000;
```

into a file — call it `prod0.mod` — and then typing a few AMPL commands:

```
ampl: model prod0.mod;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 192000
ampl: display XB, XC;
XB = 6000
XC = 1400
ampl: quit;
```

The invocation and appearance of an AMPL session will depend on your operating environment and interface, but you will always have the option of typing AMPL statements in response to the `ampl:` prompt, until you leave AMPL by typing **quit**. (Throughout the book, material you type is shown in *this slanted font*.)

The AMPL linear program that you type into the file parallels the algebraic form in every respect. It specifies the decision variables, defines the objective, and lists the constraints. It differs mainly in being somewhat more formal and regular, to facilitate computer processing. Each variable is named in a `var` statement, and each constraint by a statement that begins with `subject to` and a name like `Time` or `B_limit` for the constraint. Multiplication requires an explicit `*` operator, and the \leq relation is written `<=`.

The first command of your AMPL session, `model prod0.mod`, reads the file into AMPL, just as if you had typed it line-by-line at `ampl:` prompts. You then need only

type `solve` to have AMPL translate your linear program, send it to a linear program solver, and return the answer. A final command, `display`, is used to show the optimal values of the variables.

The message MINOS 5.5 directly following the `solve` command indicates that AMPL used version 5.5 of a solver called MINOS. We have used MINOS and several other solvers for the examples in this book. You may have a different collection of solvers available on your computer, but any solver should give you the same optimal objective value for a linear program. Often there is more than one solution that achieves the optimal objective, however, in which case different solvers may report different optimal values for the variables. (Commands for choosing and controlling solvers will be explained in Section 11.2.)

Procedures for running AMPL can vary from one computer and operating system to another. Details are provided in supplementary instructions that come with your version of the AMPL software, rather than in this book. For subsequent examples, we will assume that AMPL has been started up, and that you have received the first `ampl:` prompt. If you are using a graphical interface for AMPL, like one of those mentioned briefly in Section 1.7, many of the AMPL commands may have equivalent menu or dialog entries. You will still have the option of typing the commands as shown in this book, but you may have to open a ~~command window~~ of some kind to see the prompts.

1.3 A linear programming model

The simple approach employed so far in this chapter is helpful for understanding the fundamentals of linear programming, but you can see that if our problem were only slightly more realistic ~~— a few more products, a few more constraints —~~ it would be a nuisance to write down and impossible to illustrate with pictures. And if the problem were subject to frequent change, either in form or merely in the data values, it would be hard to update as well.

If we are to progress beyond the very tiniest linear programs, we must adopt a more general and concise way of expressing them. This is where mathematical notation comes to the rescue. We can write a compact description of the general form of the problem, which we call a *model*, using algebraic notation for the objective and the constraints. Figure 1-1 shows the production problem in algebraic notation.

Figure 1-1 is a symbolic linear programming model. Its components are fundamental to all models:

- **sets**, like the products
- **parameters**, like the production and profit rates
- **variables**, whose values the solver is to determine
- an **objective**, to be maximized or minimized
- **constraints** that the solution must satisfy.

Given: P , a set of products
 a_j = tons per hour of product j , for each $j \in P$
 b = hours available at the mill
 c_j = profit per ton of product j , for each $j \in P$
 u_j = maximum tons of product j , for each $j \in P$

Define variables: X_j = tons of product j to be made, for each $j \in P$

Maximize: $\sum_{j \in P} c_j X_j$

Subject to: $\sum_{j \in P} (1/a_j) X_j \leq b$
 $0 \leq X_j \leq u_j$, for each $j \in P$

Figure 1-1: Basic production model in algebraic form.

The model describes an infinite number of related optimization problems. If we provide specific values for data, however, the model becomes a specific problem, or *instance* of the model, that can be solved. Each different collection of data values defines a different instance; the example in the previous section was one such instance.

It might seem that we have made things less rather than more concise, since our model is longer than the original statement of the linear program in Section 1.1. Consider what would happen, however, if the set P had 42 products rather than 2. The linear program would have 120 more data values (40 each for a_j , c_j , and u_j); there would be 40 more variables, with new lower and upper limits for each; and there would be 40 more terms in the objective and the hours constraint. Yet the abstract model, as shown above, would be no different. Without this ability of a short model to describe a long linear program, larger and more complex instances of linear programming would become impossible to deal with.

A mathematical model like this is thus usually the best compromise between brevity and comprehension; and fortunately, it is easy to convert into a language that a computer can process. From now on, we will assume models are given in the algebraic form. As always, reality is rarely so simple, so most models will have more sets, parameters and variables, and more complicated objectives and constraints. In fact, in any real situation, formulating a correct model and providing accurate data are by far the hardest tasks; solving a specific problem requires only a solver and enough computing power.

1.4 The linear programming model in AMPL

Now we can talk about AMPL. The AMPL language is intentionally as close to the mathematical form as it can get while still being easy to type on an ordinary keyboard and

```

set P;
param a {j in P};
param b;
param c {j in P};
param u {j in P};
var X {j in P};
maximize Total_Profit: sum {j in P} c[j] * X[j];
subject to Time: sum {j in P} (1/a[j]) * X[j] <= b;
subject to Limit {j in P}: 0 <= X[j] <= u[j];

```

Figure 1-2: Basic production model in AMPL (file `prod.mod`).

to process by a program. There are AMPL constructions for each of the basic components listed above—sets, parameters, variables, objectives, and constraints—and ways to write arithmetic expressions, sums over sets, and so on.

We first give an AMPL model that resembles our algebraic model as much as possible, and then present an improved version that takes better advantage of the language.

The basic model

For the basic production model of Figure 1-1, a direct transcription into AMPL would look like Figure 1-2.

The keyword `set` declares a set name, as in

```
set P;
```

The members of set `P` will be provided in separate data statements, which we will show in a moment.

The keyword `param` declares a parameter, which may be a single scalar value, as in

```
param b;
```

or a collection of values indexed by a set. Where algebraic notation says that there is an a_j for each j in P , one writes in AMPL

```
param a {j in P};
```

which means that `a` is a collection of parameter values, one for each member of the set `P`. Subscripts in algebraic notation are written with square brackets in AMPL, so an individual value like a_j is written `a[j]`.

The `var` declaration

```
var X {j in P};
```

names a collection of variables, one for each member of `P`, whose values the solver is to determine.

The objective is given by the declaration

```
maximize Total_Profit: sum {j in P} c[j] * X[j];
```

The name `Total_Profit` is arbitrary; a name is required by the syntax, but any name will do. The precedence of the `sum` operator is lower than that of `*`, so the expression is indeed a sum of products, as intended.

Finally, the constraints are given by

```
subject to Time: sum {j in P} (1/a[j]) * X[j] <= b;
subject to Limit {j in P}: 0 <= X[j] <= u[j];
```

The `Time` constraint says that a certain sum over the set P may not exceed the value of parameter b . The `Limit` constraint is actually a family of constraints, one for each member j of P : each $X[j]$ is bounded by zero and the corresponding $u[j]$.

The construct `{j in P}` is called an *indexing expression*. As you can see from our example, indexing expressions are used not only in declaring parameters and variables, but in any context where the algebraic model does something ~~for~~ *for* each j in P . Thus the `Limit` constraints are declared

```
subject to Limit {j in P}
```

because we want to impose a different restriction $0 \leq X[j] \leq u[j]$ for each different product j in the set P . In the same way, the summation in the objective is written

```
sum {j in P} c[j] * X[j]
```

to indicate that the different terms $c[j] * X[j]$, for each j in the set P , are to be added together in computing the profit.

The layout of an AMPL model is quite free. Sets, parameters, and variables must be declared before they are used but can otherwise appear in any order. Statements end with semicolons and can be spaced and split across lines to enhance readability. Upper and lower case letters are different, so `time`, `Time`, and `TIME` are three different names.

You have undoubtedly noticed several places where traditional mathematical notation has been adapted in AMPL to the limitations of normal keyboards and character sets. AMPL uses the word `sum` instead of Σ to express a summation, and `in` rather than \in for set membership. Set specifications are enclosed in braces, as in `{j in P}`. Where mathematical notation uses adjacency to signify multiplication in $c_j X_j$, AMPL uses the `*` operator of most programming languages, and subscripts are denoted by brackets, so $c_j X_j$ becomes `c[j]*X[j]`.

You will find that the rest of AMPL is similar ~~to~~ a few more arithmetic operators, a few more key words like `sum` and `in`, and many more ways to specify indexing expressions. Like any other computer language, AMPL has a precise grammar, but we won't stress the rules too much here; most will become clear as we go along, and full details are given in the reference manual, Appendix A.

Our original two-variable linear program is one of the many LPs that are instances of the Figure 1-2 model. To specify it or any other such instance, we need to supply the

```

set P := bands coils;

param:      a      c      u      :=
  bands    200    25    6000
  coils    140    30    4000 ;

param b := 40;

```

Figure 1-3: Production model data (file `prod.dat`).

membership of P and the values of the various parameters. There is no standard way to describe these data values in algebraic notation; usually some kind of informal tables are used, such as the ones we showed earlier. In AMPL, there is a specific syntax for data tables, which is sufficiently regular and unambiguous to be translated by a computer. Figure 1-3 gives data for the basic production model in that form. A `set` statement supplies the members (`bands` and `coils`) of set P , and a `param` table gives the corresponding values for a , c , and u . A simple `param` statement gives the value for b . These data statements, which are described in detail in Chapter 9, have a variety of options that let you list or tabulate parameters in convenient ways.

An improved model

We could go on immediately to solve the linear program defined by Figures 1-2 and 1-3. Once we have written the model in AMPL, however, we need not feel constrained by all the conventions of algebra, and we can instead consider changes that might make the model easier to work with. Figures 1-4a and 1-4b show a possible *improved* version. The short *mathematical* names for the sets, parameters and variables have been replaced by longer, more meaningful ones. The indexing expressions have become $\{p \text{ in PROD}\}$, or just $\{\text{PROD}\}$ in those declarations that do not use the index p . The bounds on variables have been placed within their `var` declaration, rather than in a separate constraint; analogous bounds have been placed on the parameters, to indicate the ones that must be positive or nonnegative in any meaningful linear program derived from the model.

Finally, comments have been added to help explain the model to a reader. Comments begin with `#` and end at the end of the line. As in any programming language, judicious use of meaningful names, comments and formatting helps to make AMPL models more readable and understandable.

There are always many ways to describe a particular model in AMPL. It is left to the modeler to pick the way that seems clearest or most convenient. Our earlier, mathematical approach is often preferred for working quickly with a familiar model. On the other hand, the second version is more attractive for a model that will be maintained and modified by several people over months or years.

```

set PROD; # products
param rate {PROD} > 0; # tons produced per hour
param avail >= 0; # hours available in week
param profit {PROD}; # profit per ton
param market {PROD} >= 0; # limit on tons sold in week
var Make {p in PROD} >= 0, <= market[p]; # tons produced
maximize Total_Profit: sum {p in PROD} profit[p] * Make[p];
# Objective: total profits from all products
subject to Time: sum {p in PROD} (1/rate[p]) * Make[p] <= avail;
# Constraint: total of hours used by all
# products may not exceed hours available

```

Figure 1-4a: Steel production model (steel.mod).

```

set PROD := bands coils;
param:    rate  profit  market :=
  bands   200    25     6000
  coils   140    30     4000 ;
param avail := 40;

```

Figure 1-4b: Data for steel production model (steel.dat).

If we put all of the model declarations into a file called `steel.mod`, and the data specification into a file `steel.dat`, then as before a solution can be found and displayed by typing just a few statements:

```

ampl: model steel.mod;
ampl: data steel.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 192000

ampl: display Make;
Make [*] :=
bands 6000
coils 1400
;

```

The model and data commands each specify a file to be read, in this case the model from `steel.mod`, and the data from `steel.dat`. The use of two file-reading commands encourages a clean separation of model from data.

Filenames can have any form recognized by your computer's operating system; AMPL doesn't check them for correctness. The filenames here and in the rest of the book refer to example files that are available from the AMPL web site and other AMPL distributions.

Once the model has been solved, we can show the optimal values of all of the variables `Make[p]`, by typing ***display Make***. The output from `display` uses the same formats as AMPL data input, so that there is only one set of formats to learn. (The `[*]` indicates a variable or parameter with a single subscript. It is not strictly necessary for input, since `Make` is one-dimensional, but `display` prints it as a reminder.)

Catching errors

You will inevitably make some mistakes as you develop a model. AMPL detects various kinds of incorrect statements, which are reported in error messages following the `model`, `data` or `solve` commands.

AMPL catches many errors as soon as the model is read. For example, if you use the wrong syntax for the bounds in the declaration of the variable `Make`, you will receive an error message like this, right after you enter the `model` command:

```
steel.mod, line 8 (offset 250):
    syntax error
context: var Make {p in PROD} >>> 0 <<<= Make[p] <= market[p];
```

If you inadvertently use `make` instead of `Make` in an expression like `profit[p] * make[p]`, you will receive this message:

```
steel.mod, line 11 (offset 339):
    make is not defined
context: maximize Total_Profit:
    sum {p in PROD} profit[p] * >>> make[p] <<< ;
```

In each case, the offending line is printed, with the approximate location of the error surrounded by `>>>` and `<<<`.

Other common sources of error messages include a model component used before it is declared, a missing semicolon at the end of a command, or a reserved word like `sum` or `in` used in the wrong context. (Section A.1 contains a list of reserved words.) Syntax errors in data statements are similarly reported right after you enter a `data` command.

Errors in the data values are caught after you type `solve`. If the number of hours were given as $\hat{A}40$, for instance, you would see:

```
ampl: model steel.mod;
ampl: data steel.dat;
ampl: solve;
Error executing "solve" command:
error processing param avail:
    failed check: param avail = -40
                  is not >= 0;
```

It is good practice to include as many validity checks as possible in the model, so that errors are caught at an early stage.

Despite your best efforts to formulate the model correctly and to include validity checks on the data, sometimes a model that generates no error messages and that elicits

an optimal solution. A report from the solver will nonetheless produce a clearly wrong or meaningless solution. All of the production levels might be zero, for example, or the product with a lower profit per hour may be produced at a higher volume. In cases like these, you may have to spend some time reviewing your formulation before you discover what is wrong.

The `expand` command can be helpful in your search for errors, by showing you how AMPL instantiated your symbolic model. To see what AMPL generated for the objective `Total_Profit`, for example, you could type:

```
ampl: expand Total_Profit;
maximize Total_Profit:
      25*Make[ÂbandsÂ] + 30*Make[ÂcoilsÂ];
```

This corresponds directly to our explicit formulation back in Section 1.1. Expanding the constraint works similarly:

```
ampl: expand Time;
subject to Time:
      0.005*Make[ÂbandsÂ] + 0.00714286*Make[ÂcoilsÂ] <= 40;
```

Expressions in the symbolic model, such as the coefficients `1/rate[p]` in this example, are evaluated before the expansion is displayed. You can expand the objective and all of the constraints at once by typing `expand` by itself.

The expressions above show that the symbolic model's `Make[j]` expands to the explicit variables `Make[ÂbandsÂ]` and `Make[ÂcoilsÂ]`. You can use expressions like these in AMPL commands, for example to expand a particular variable to see what coefficients it has in the objective and constraints:

```
ampl: expand Make[ÂcoilsÂ];
Coefficients of Make[ÂcoilsÂ]:
      Time          0.00714286
      Total_Profit  30
```

Either single quotes (Â) or double quotes (") may surround the subscript.

1.5 Adding lower bounds to the model

Once the model and data have been set up, it is a simple matter to change them and then re-solve. Indeed, we would not expect to find an LP application in which the model and data are prepared and solved just once, or even a few times. Most commonly, numerous refinements are introduced as the model is developed, and changes to the data continue for as long as the model is used.

Let's conclude this chapter with a few examples of changes and refinements. These examples also highlight some additional features of AMPL.

Suppose first that we add another product, steel plate. The model stays the same, but in the data we have to add `plate` to the list of members for the set `PROD`, and we have to add a line of parameter values for `plate`:

```
set PROD := bands coils plate;

param:    rate  profit  market :=
  bands    200    25     6000
  coils    140    30     4000
  plate    160    29     3500 ;

param avail := 40;
```

We put this version of the data in a file called `steel2.dat`, and use AMPL as before to get the solution:

```
ampl: model steel.mod; data steel2.dat; solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 196400

ampl: display Make;
Make [*] :=
bands 6000
coils  0
plate 1600
;
```

Profits have increased compared to the two-variable version, but now it is best to produce no coils at all! On closer examination, this result is not so surprising. Plate yields a profit of \$4640 per hour, which is less than for bands but more than for coils. Thus plate is produced to absorb the capacity not taken by bands; coils would be produced only if both bands and plate reached their market limits before the available hours were exhausted.

In reality, a whole product line cannot be shut down solely to increase weekly profits. The simplest way to reflect this in the model is to add lower bounds on the production amounts, as shown in Figures 1-5a and 1-5b. We have declared a new collection of parameters named `commit`, to represent the lower bounds on production that are imposed by sales commitments, and we have changed `>= 0` to `>= commit[p]` in the declaration of the variables `Make[p]`.

After these changes are made, we can run AMPL again to get a more realistic solution:

```
ampl: model steel3.mod; data steel3.dat; solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 194828.5714

ampl: display commit, Make, market;
:      commit  Make  market  :=
bands   1000   6000    6000
coils    500    500    4000
plate    750   1028.57  3500
;
```

For comparison, we have displayed `commit` and `market` on either side of the actual production, `Make`. As expected, after the commitments are met, it is most profitable to

```

set PROD; # products
param rate {PROD} > 0; # produced tons per hour
param avail >= 0; # hours available in week
param profit {PROD}; # profit per ton
param commit {PROD} >= 0; # lower limit on tons sold in week
param market {PROD} >= 0; # upper limit on tons sold in week
var Make {p in PROD} >= commit[p], <= market[p]; # tons produced
maximize Total_Profit: sum {p in PROD} profit[p] * Make[p];
# Objective: total profits from all products
subject to Time: sum {p in PROD} (1/rate[p]) * Make[p] <= avail;
# Constraint: total of hours used by all
# products may not exceed hours available

```

Figure 1-5a: Lower bounds on production (steel3.mod).

```

set PROD := bands coils plate;
param:    rate  profit  commit  market :=
  bands   200    25     1000    6000
  coils   140    30      500    4000
  plate   160    29      750    3500 ;
param avail := 40;

```

Figure 1-5b: Data for lower bounds on production (steel3.dat).

produce bands up to the market limit, and then to produce plate with the remaining available time.

1.6 Adding resource constraints to the model

Processing of steel slabs is not a single operation, but a series of steps that may proceed at different rates. To motivate a more general model, imagine that we divide production into a reheat stage that can process the incoming slabs at 200 tons per hour, and a rolling stage that makes bands, coils or plate at the rates previously given. Further imagine that there are only 35 hours of reheat time, even though there are 40 hours of rolling time.

To cover this kind of situation, we can add a set `STAGE` of production stages to our model. The parameter and constraint declarations are modified accordingly, as shown in Figure 1-6a. Since there is a potentially different number of hours available in each stage, the parameter `avail` is now indexed over `STAGE`. Since there is a potentially different production rate for each product in each stage, the parameter `rate` is indexed over both `PROD` and `STAGE`. In the `Time` constraint, the production rate for product `p` in

```

set PROD;    # products
set STAGE;   # stages

param rate {PROD,STAGE} > 0; # tons per hour in each stage
param avail {STAGE} >= 0;    # hours available/week in each stage
param profit {PROD};         # profit per ton

param commit {PROD} >= 0;    # lower limit on tons sold in week
param market {PROD} >= 0;    # upper limit on tons sold in week

var Make {p in PROD} >= commit[p], <= market[p]; # tons produced

maximize Total_Profit: sum {p in PROD} profit[p] * Make[p];

                # Objective: total profits from all products

subject to Time {s in STAGE}:
    sum {p in PROD} (1/rate[p,s]) * Make[p] <= avail[s];

                # In each stage: total of hours used by all
                # products may not exceed hours available

```

Figure 1-6a: Additional resource constraints (steel4.mod).

stage s is referred to as $\text{rate}[p,s]$; this is AMPL's version of a doubly subscripted entity like a_{ps} in algebraic notation.

The only other change is to the constraint declaration, where we no longer have a single constraint, but a constraint for each stage, imposed by limited time available at that stage. In algebraic notation, this might have been written

$$\text{Subject to } \sum_{p \in P} (1/a_{ps}) X_p \leq b_s, \text{ for each } s \in S.$$

Compare the AMPL version:

```

subject to Time {s in STAGE}:
    sum {p in PROD} (1/rate[p,s]) * Make[p] <= avail[s];

```

As in the other examples, this is a straightforward analogue, adapted to the requirements of a computer language. In almost all models, most of the constraints are indexed collections like this one.

Since rate is now indexed over combinations of two indices, it requires a data table all to itself, as in Figure 1-6b. The data file must also include the membership for the new set STAGE , and values of avail for both reheat and roll .

After these changes are made, we use AMPL to get another revised solution:

```

ampl: reset;
ampl: model steel4.mod; data steel4.dat; solve;
MINOS 5.5: optimal solution found.
4 iterations, objective 190071.4286

```

```

set PROD := bands coils plate;
set STAGE := reheat roll;

param rate:  reheat  roll :=
    bands      200    200
    coils      200    140
    plate      200    160 ;

param:  profit  commit  market :=
    bands    25    1000   6000
    coils    30    500   4000
    plate    29    750   3500 ;

param avail :=  reheat 35   roll 40 ;

```

Figure 1-6b: Data for additional resource constraints (steel4.dat).

```

ampl: display Make.lb, Make, Make.ub, Make.rc;
:      Make.lb      Make      Make.ub      Make.rc      :=
bands    1000      3357.14      6000      5.32907e-15
coils     500        500      4000      -1.85714
plate     750      3142.86      3500      3.55271e-15
;

ampl: display Time;
Time [*] :=
reheat  1800
roll    3200
;

```

The `reset` command erases the previous model so a new one can be read in.

At the end of the example above we have displayed the $\hat{\pi}$ marginal values (also called $\hat{\pi}$ dual values or $\hat{\pi}$ shadow prices) associated with the `Time` constraints. The marginal value of a constraint measures how much the value of the objective would improve if the constraint were relaxed by a small amount. For example, here we would expect that up to some point, additional reheat time would produce another \$1800 of extra profit per hour, and additional rolling time would produce \$3200 per hour; decreasing these times would decrease the profit correspondingly. In output commands like `display`, AMPL interprets a constraint's name alone as referring to the associated marginal values.

We also display several quantities associated with the variables `Make`. First there are lower bounds `Make.lb` and upper bounds `Make.ub`, which in this case are the same as `commit` and `market`. We also show the \hat{c} reduced cost `Make.rc`, which has the same meaning with respect to the bounds that the marginal values have with respect to the constraints. Thus we see that, again up to some point, each increase of a ton in the lower bound (or commitment) for coil production should reduce profits by about \$1.86; each one-ton decrease in the lower bound should improve profits by about \$1.86. The production levels for bands and plates are between their bounds, so their reduced costs are essentially zero (recall that $e-15$ means $\bullet 10^{-15}$), and changing their levels will have no

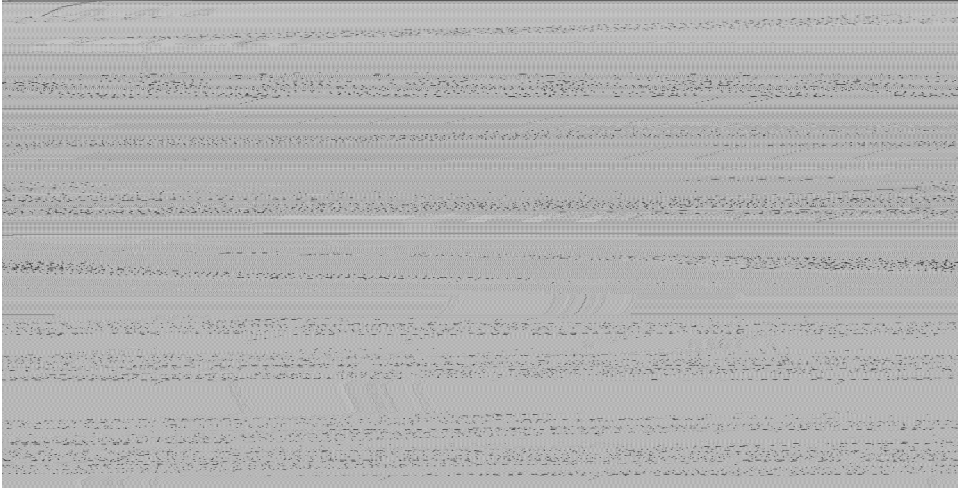


Figure 1-7a: A Java-based AMPL graphical user interface (Macintosh).

effect. Bounds, marginal (or dual) values, reduced costs and other quantities associated with variables and constraints are explored further in Section 12.5.

Comparing this session with our previous one, we see that the additional reheat time restriction reduces profits by about \$4750, and forces a substantial change in the optimal solution: much higher production of plate and lower production of bands. Moreover, the logic underlying the optimum is no longer so obvious. It is the difficulty of solving LPs by logical reasoning alone that necessitates computer-based systems such as AMPL.

1.7 AMPL interfaces

The examples that we have presented so far all use AMPL's command interface: the user types textual commands and the system responds with textual results. This is what we will use throughout the book to illustrate AMPL's capabilities. It permits access to all of AMPL's rich collection of features, and it will be the same in all environments. A text-based interface is most natural for creating scripts of frequently used commands and for writing programs that use AMPL's programming constructs (the topics of Chapter 13). And text commands are used in applications where AMPL is a hidden or behind-the-scenes part of some larger process.

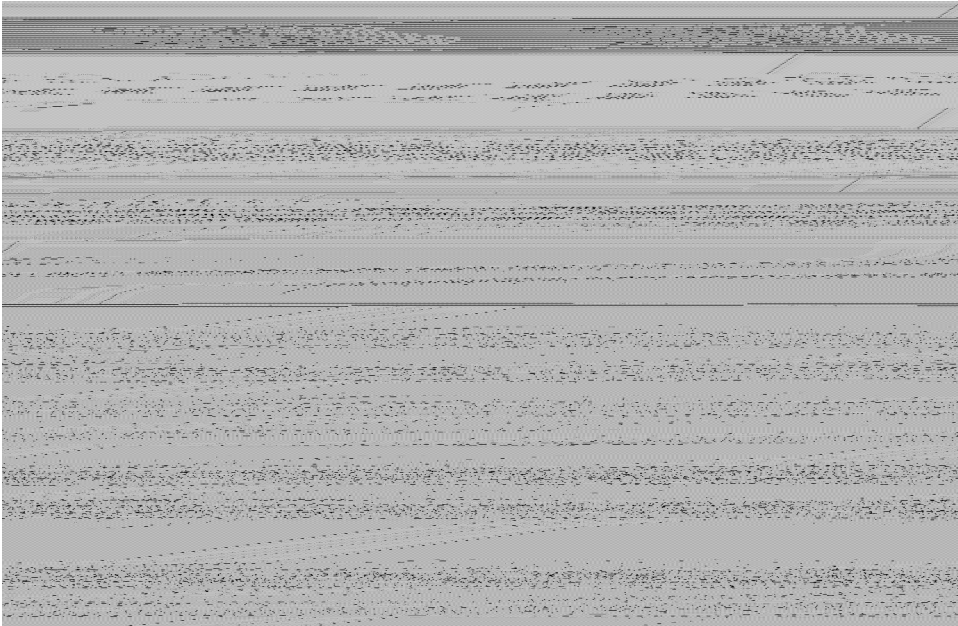


Figure 1-7b: A Tcl/Tk-based AMPL graphical user interface (Unix).

All that said, however, there are plenty of times where a graphical user interface can make a program easier to use, helping novices to get started and casual or infrequent users to recall details. AMPL is no exception. Thus there are a variety of graphical interfaces for AMPL, loosely analogous to the ~~integrated~~ development environments~~for~~ for conventional programming languages, though AMPL~~s~~ environments are much less elaborate. An AMPL graphical interface typically provides a way to easily execute standard commands, set options, invoke solvers, and display the results, often by pushing buttons and selecting menu items instead of by typing commands.

Interfaces exist for standard operating system platforms. For example, Figure 1-7a shows a simple interface based on Java that runs on Unix and Linux, Windows, and Macintosh, presenting much the same appearance on each. (The Mac interface is shown.) Figure 1-7b shows a similar interface based on Tcl/Tk, shown running on Unix but also portable to Windows and Macintosh. Figure 1-7c shows another interface, created with Visual Basic and running on Windows.

There are also web-based interfaces that provide client-server access to AMPL or solvers over network connections, and a number of application program interfaces (API~~s~~) for calling AMPL from other programs. The AMPL web site, www.ampl.com, provides up to date information on all types of available interfaces.

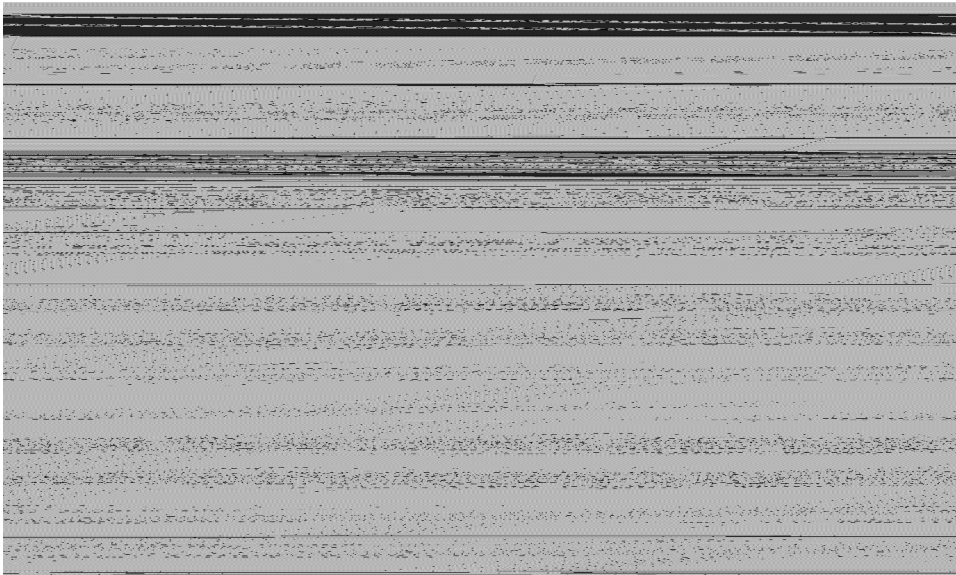


Figure 1-7c: A Visual Basic AMPL graphical user interface (Windows).

Bibliography

Julius S. Aronofsky, John M. Dutton and Michael T. Tayyabkhan, *Managerial Planning with Linear Programming: In Process Industry Operations*. John Wiley & Sons (New York, NY, 1978). A detailed account of a variety of profit-maximizing applications, with emphasis on the petroleum and petrochemical industries.

Vašek Chvátal, *Linear Programming*, W. H. Freeman (New York, NY, 1983). A concise and economical introduction to theoretical and algorithmic topics in linear programming.

Tibor Fabian, "A Linear Programming Model of Integrated Iron and Steel Production," *Management Science* **4** (1958) pp. 415-449. An application to all stages of steelmaking - from coal and ore through finished products - from the early days of linear programming.

Robert Fourer and Goutam Dutta, "A Survey of Mathematical Programming Applications in Integrated Steel Plants," *Manufacturing & Service Operations Management* **4** (2001) pp. 387-400.

David A. Kendrick, Alexander Meeraus and Jaime Alatorre, *The Planning of Investment Programs in the Steel Industry*. The Johns Hopkins University Press (Baltimore, MD, 1984). Several detailed mathematical programming models, using the Mexican steel industry as an example.

Robert J. Vanderbei, *Linear Programming: Foundations and Extensions* (2nd edition). Kluwer Academic Publishers (Dordrecht, The Netherlands, 2001). An updated survey of linear programming theory and methods.

Exercises

1-1. This exercise starts with a two-variable linear program similar in structure to the one of Sections 1.1 and 1.2, but with a quite different story behind it.

(a) You are in charge of an advertising campaign for a new product, with a budget of \$1 million. You can advertise on TV or in magazines. One minute of TV time costs \$20,000 and reaches 1.8 million potential customers; a magazine page costs \$10,000 and reaches 1 million. You must sign up for at least 10 minutes of TV time. How should you spend your budget to maximize your audience? Formulate the problem in AMPL and solve it. Check the solution by hand using at least one of the approaches described in Section 1.1.

(b) It takes creative talent to create effective advertising; in your organization, it takes three person-weeks to create a magazine page, and one person-week to create a TV minute. You have only 100 person-weeks available. Add this constraint to the model and determine how you should now spend your budget.

(c) Radio advertising reaches a quarter million people per minute, costs \$2,000 per minute, and requires only 1 person-day of time. How does this medium affect your solutions?

(d) How does the solution change if you have to sign up for at least two magazine pages? A maximum of 120 minutes of radio?

1-2. The steel model of this chapter can be further modified to reflect various changes in production requirements. For each part below, explain the modifications to Figures 1-6a and 1-6b that would be required to achieve the desired changes. (Make each change separately, rather than accumulating the changes from one part to the next.)

(a) How would you change the constraints so that total hours used by all products must *equal* the total hours available for each stage? Solve the linear program with this change, and verify that you get the same results. Explain why, in this case, there is no difference in the solution.

(b) How would you add to the model to restrict the total weight of all products to be less than a new parameter, `max_weight`? Solve the linear program for a weight limit of 6500 tons, and explain how this extra restriction changes the results.

(c) The incentive system for mill managers may tend to encourage them to produce as many tons as possible. How would you change the objective function to maximize total tons? For the data of our example, does this make a difference to the optimal solution?

(d) Suppose that instead of the lower bounds represented by `commit[p]` in our model, we want to require that each product represent a certain share of the total tons produced. In the algebraic notation of Figure 1-1, this new constraint might be represented as

$$X_j \geq s_j \sum_{k \in P} X_k, \text{ for each } j \in P$$

where s_j is the minimum share associated with project j . How would you change the AMPL model to use this constraint in place of the lower bounds `commit[p]`? If the minimum shares are 0.4 for bands and plate, and 0.1 for coils, what is the solution?

Verify that if you change the minimum shares to 0.5 for bands and plate, and 0.1 for coils, the linear program gives an optimal solution that produces nothing, at zero profit. Explain why this makes sense.

(e) Suppose there is an additional finishing stage for plates only, with a capacity of 20 hours and a rate of 150 tons per hour. Explain how you could modify the data, without changing the model, to incorporate this new stage.

1-3. This exercise deals with some issues of sensitivity in the steel models.

(a) For the linear program of Figures 1-5a and 1-5b, display `Time` and `Make.rc`. What do these values tell you about the solution? (You may wish to review the explanation of marginal values and reduced costs in Section 1.6.)

(b) Explain why the reheat time constraints added in Figure 1-6a result in a higher production of plate and a lower production of bands.

(c) Use AMPL to verify the following statements: If the available reheat time is increased from 35 to 36 in the data of Figure 1-6b, then the profit goes up by \$1800 as predicted in Section 1.6. If the reheat time is further increased to 37, the profit goes up by another \$1800. However, if the reheat time is increased to 38, there is a smaller increase in the profit, and further increases past 38 have no effect on the optimal profit at all. To change the reheat time to, say, 26 without changing and reading the data file over again, type the command

```
let avail["reheat"] := 36;
```

By trying some other values of the reheat time, confirm that the profit increases by \$1800 per extra hour for any number of hours between 35 and $37 \frac{9}{14}$, but that any increase in the reheat time beyond $37 \frac{9}{14}$ hours doesn't give any further profit.

Draw a plot of the profit versus the number of reheat hours available, for hours ≥ 35 .

(d) To find the slope of the plot from (c) — profit versus reheat time available — at any particular reheat time value, you need only look at the marginal value of `Time["reheat"]`. Using this observation as an aid, extend your plot from (c) down to 25 hours of reheat time. Verify that the slope of the plot remains at \$6000 per hour from 25 hours down to less than 12 hours of reheat time. Explain what happens when the available reheat time drops to 11 hours.

1-4. Here is a similar profit-maximizing model, but in a different context. An automobile manufacturer produces several kinds of cars. Each kind requires a certain amount of factory time per car to produce, and yields a certain profit per car. A certain amount of factory time has been scheduled for the next week, and it is desired to use all this time; but at least a certain number of each kind of car must be manufactured to meet dealer requirements.

(a) What are the data values that define this problem? How would you declare the sets and parameter values for this problem in AMPL? What are the decision variables, and how would you declare them in AMPL?

(b) Assuming that the objective is to maximize total profit, how would you declare an objective in AMPL for this problem? How would you declare the constraints?

(c) For purposes of experiment, suppose that there are three kinds of cars, known at the factory as *T*, *C* and *L*, that 120 hours are available, and that the time per car, profit per car and dealer orders for each kind of car are as follows:

Car	time	profit	orders
T	1	200	10
C	2	500	20
L	3	700	15

How much of each car should be produced, and what is the maximum profit? You should find that your solution specifies a fractional amount of one of the cars. As a practical matter, how could you make use of this solution?

(d) If you maximize the total number of cars produced instead of the total profit, how many more cars do you make? How much less profit?

(e) Each kind of car achieves a certain fuel efficiency, and the manufacturer is required by law to maintain a certain fleet average efficiency. The fleet average is computed by multiplying the efficiency of each kind of car times the number of that kind produced, summing all of the resulting products, and dividing by the total of all cars produced. Extend your AMPL model to contain a minimum fleet average efficiency constraint. Rearrange the constraint as necessary to make it linear—no variables divided into other variables.

(f) Find the optimal solution for the case where cars *T*, *C* and *L* achieve fuel efficiencies of 50, 30 and 20 miles/gallon, and the fleet average efficiency must be at least 35 miles/gallon. Explain how this changes the production amounts and the total profit. Dealing with the fractional amounts in the solution is not so easy in this case. What might you do?

If you had 10 more hours of production time, you could make more profit. Does the addition of the fleet average efficiency constraint make the extra 10 hours more or less valuable?

(g) Explain how you could further refine this model to account for different production stages that have different numbers of hours available per stage, much as in the steel model of Section 1.6.

1-5. A group of young entrepreneurs earns a (temporarily) steady living by acquiring inadequately supervised items from electronics stores and re-selling them. Each item has a street value, a weight, and a volume; there are limits on the numbers of available items, and on the total weight and volume that can be managed at one time.

(a) Formulate an AMPL model that will help to determine how much of each item to pick up, to maximize one day's profit.

(b) Find a solution for the case given by the following table,

	Value	Weight	Volume	Available
TV	50	35	8	20
radio	15	5	1	50
camera	85	4	2	20
CD player	40	3	1	30
VCR	50	15	5	30
camcorder	120	20	4	15

and by limits of 500 pounds and 300 cubic feet.

(c) Suppose that it is desirable to acquire some of each item, so as to always have stock available for re-sale. Suppose in addition that there are upper bounds on how many of each item you can reasonably expect to sell. How would you add these conditions to the model?

(d) How could the group use the dual variables on the maximum-weight and maximum-volume constraints to evaluate potential new partners for their activities?

(e) Through adverse circumstances the group has been reduced to only one member, who can carry a mere 75 pounds and five cubic feet. What is the optimum strategy now? Given that this requires a non-integral number of acquisitions, what is the best all-integer solution? (The integrality constraint converts this from a standard linear programming problem into a much harder problem called a Knapsack Problem. See Chapter 20.)

1-6. Profit-maximizing models of oil refining were one of the first applications of linear programming. This exercise asks you to model a simplified version of the final stage of the refining process.

A refinery breaks crude oil into some collection of intermediate materials, then blends these materials back together into finished products. Given the volumes of intermediates that will be available, we want to determine how to blend the intermediates so that the resulting products are most profitable. The decision is made more complicated, however, by the existence of upper limits on certain attributes of the products, which must be respected in any feasible solution.

To formulate an algebraic linear programming model for this problem, we can start by defining sets I of intermediates, J of final products, and K of attributes. The relevant technological data may be represented by

- a_i barrels of intermediate i available, for each $i \in I$
- r_{ik} units of attribute k contributed per barrel of intermediate i , for each $i \in I$ and $k \in K$
- u_{jk} maximum allowed units of attribute k per barrel of final product j ,
for each $j \in J$ and $k \in K$
- δ_{ij} 1 if intermediate i is allowed in the blend for product j , or 0 otherwise,
for each $i \in I$ and $j \in J$

and the economic data can be given by

- c_j revenue per barrel of product j , for each $j \in J$

There are two collections of decision variables:

- X_{ij} barrels of intermediate i used to make product j , for each $i \in I$ and $j \in J$
- Y_j barrels of product j made, for each $j \in J$

The objective is to

$$\text{maximize } \sum_{j \in J} c_j Y_j,$$

which is the sum of the revenues from the various products.

It remains to specify the constraints. The amount of each intermediate used to make products must equal the amount available:

$$\sum_{j \in J} X_{ij} = a_i, \text{ for each } i \in I.$$

The amount of a product made must equal the sum of amounts of the components blended into it:

$$\sum_{i \in I} X_{ij} = Y_j, \text{ for each } j \in J.$$

For each product, the total attributes contributed by all intermediates must not exceed the total allowed:

$$\sum_{i \in I} r_{ik} X_{ij} \leq u_{jk} Y_j, \text{ for each } j \in J \text{ and } k \in K.$$

Finally, we bound the variables as follows:

$$\begin{aligned} 0 &\leq X_{ij} \leq \delta_{ij} a_i, \text{ for each } i \in I, j \in J, \\ 0 &\leq Y_j, \text{ for each } j \in J. \end{aligned}$$

The upper bound on X_{ij} assures that only the appropriate intermediates will be used in blending. If intermediate i is not allowed in the blend for product j , as indicated by δ_{ij} being zero, then the upper bound on X_{ij} is zero; this ensures that X_{ij} cannot be positive in any solution. Otherwise, the upper bound on X_{ij} is just a_i , which has no effect since there are only a_i barrels of intermediate i available for blending in any case.

(a) Transcribe this model to AMPL, using the same names as in the algebraic form for the sets, parameters and variables as much as possible.

(b) Re-write the AMPL model using meaningful names and comments, in the style of Figure 1-4a.

(c) In a representative small-scale instance of this model, the intermediates are SRG (straight run gasoline), N (naphtha), RF (reformate), CG (cracked gasoline), B (butane), DI (distillate intermediate), GO (gas oil), and RS (residuum). The final products are PG (premium gasoline), RG (regular gasoline), D (distillate), and HF (heavy fuel oil). Finally, the attributes are vap (vapor pressure), oct (research octane), den (density), and sul (sulfur).

The following amounts of the intermediates are scheduled to be available:

SRG	N	RF	CG	B	DI	GO	RS
21170	500	16140	4610	370	250	11600	25210

The intermediates that can be blended into each product, and the amounts of the attributes that they possess, are as follows (with blank entries representing zeros):

	Premium & regular gasoline		Distillate		Heavy fuel oil	
	vap	oct	den	sul	den	sul
SRG	18.4	85.5				
N	6.54	65.0	272	.283		
RF	2.57	104.0				
CG	6.90	93.7				
B	199.2	91.8				
DI			292	.526		
GO			295	.353	295	.353
RS					343	4.70

The attribute limits and revenues/barrel for the products are:

	vap	oct	den	sul	revenue
PG	12.2	90			10.50
RG	12.7	86			9.10
D			306	0.5	7.70
HF			352	3.5	6.65

Limits left blank, such as density for gasoline, are irrelevant and may be set to some relatively large number.

Create a data file for your AMPL model and determine the optimal blend and production amounts.

(d) It looks a little strange that the attribute amounts for research octane are negative. What is the limit constraint for this attribute really saying?