Optimization Models
Draft of August 26, 2005

I. Formulating an Optimization Model: An Introductory Example

Robert Fourer

Department of Industrial Engineering and Management Sciences
Northwestern University
Evanston, Illinois 60208-3119, U.S.A.

(847) 491-3151

`4er@iems.northwestern.edu`
`http://www.iems.northwestern.edu/~4er/`

1. A Simple Model

To introduce the fundamentals of our subject, we begin with a simple example, the “diet problem”: choosing from a menu of available foods to produce a diet that meets daily nutritional requirements. Since there are many different combinations of foods that meet the requirements, our goal will be to identify a combination that does the job at the lowest possible cost. This is the characteristic goal of any ***optimization*** problem: to find a preferred arrangement among all that are acceptable.

This chapter works through the basic steps of generating and solving a diet problem. In the process, we show how to formulate a mathematical “model” that describes diet problems in a general way, and how to submit a model and data to a software package that computes minimum-cost diets and related information. Then in Chapter 2, your intuition for diets will enable you to see how our model must be refined to produce sensible results. You will also see how certain refinements can make the optimal diet harder to compute. These are characteristics that you will encounter repeatedly as you study different optimization problems and models for them.

Diet problems are only one example of the general idea of a minimum-cost input problem. Chapter 3 will take a tour through a variety of other applications of this idea, including blending, scheduling, and cutting. Then Chapter 4 will extend the idea even further to encompass output and input-output problems.

1.1 A small diet problem

When approaching any unfamiliar kind of optimization problem, it’s best to start with a version that’s small and simple. Thus let’s begin by imagining that your diet is limited to a selection of items from a well-known fast food restaurant. We’ll give each food a nickname to assist in referring to it:

`QP:`	Quarter Pounder	`FR:`	Fries, small
`MD:`	McLean Deluxe	`SM:`	Sausage McMuffin
`BM:`	Big Mac	`1M:`	1% Lowfat Milk
`FF:`	Filet-O-Fish	`OJ:`	Orange Juice
`MC:`	McGrilled Chicken		

Suppose also that you are interested in providing your diet with appropriate amounts of seven “nutrients”:

`Prot:`	Protein	`Iron:`	Iron
`VitA:`	Vitamin A	`Cals:`	Calories
`VitC:`	Vitamin C	`Carb:`	Carbohydrates
`Calc:`	Calcium		

Your problem is to find the lowest-cost combination of the foods that will provide a day’s requirements for the nutrients.

To solve this problem, you need to know how much of each nutrient is in one serving of each food, and the total of each nutrient that you require. You also

	QP	MD	BM	FF	MC	FR	SM	1M	OJ	
Cost	1.84	2.19	1.84	1.44	2.29	0.77	1.29	0.60	0.72	*Req'd*
Prot	28	24	25	14	31	3	15	9	1	55
VitA	15	15	6	2	8	0	4	10	2	100
VitC	6	10	2	0	15	15	0	4	120	100
Calc	30	20	25	15	15	0	20	30	2	100
Iron	20	20	20	10	8	2	15	0	2	100
Cals	510	370	500	370	400	220	345	110	80	2000
Carb	34	35	42	38	42	26	27	12	20	350

Figure 1–1. *Costs, requirements and nutritional values for the simple diet problem.* Costs are in dollars per serving, columns under the costs are in appropriate nutrient units per serving, and requirements are in nutrient units per day.

need the price per serving of each food. To save you the trouble of collecting this information, Figure 1–1 presents a table of all the relevant costs, requirements, and nutritional values.

What is the optimization problem involved here, and how can we describe it? There are three kinds of entities that we must identify in order to construct a suitable description, or ***formulation:***

- ◇ the decision variables,
- ◇ the objectives, and
- ◇ the constraints.

These entities are fundamental to many kinds of decision-making via optimization, and will appear repeatedly in subsequent chapters. We thus proceed to define each one in a general sense, with illustrations from the case of the small diet problem.

A ***decision variable*** represents a choice that an optimization problem requires. Often it is called simply a ***variable,*** when the context makes its purpose clear. Taken together, the *values* of the variables specify a decision that is to be made. Variables usually represent choices by taking numbers as values, though there are important exceptions.

In the simple diet problem, the decision is how much of each food to buy. Thus we define a decision variable corresponding to each food, representing the amount of that food to be purchased. Following the conventions of algebra, we use the letter x to refer to these (as yet) unknown quantities: we let x_{QP} represent the number of Quarter Pounders to be bought, x_{MD} the number of McLean Deluxes, and so forth for x_{BM}, x_{FF}, x_{MC}, x_{FR}, x_{SM}, x_{1M} and x_{OJ}. Any assignment of values to these 9 variables is a prospective decision in the context of the diet problem.

An ***objective*** is a quantity that you want to make as small or as large as possible. Equivalently, in the terminology of optimization, an objective is some characteristic of an optimization problem that you would like to ***minimize*** or ***maximize.*** To be useful in formulating optimization models, an objective must be computable from the values of the decision variables. In mathematical terms, objectives are *functions* of the variables, and in fact the term ***objective function*** is often used synonymously with objective.

For the diet problem as we have described it, there is one objective: the total cost of the foods purchased, which is of course to be minimized. It is easy to write a mathematical expression for this objective in terms of the decision variables previously defined. Consider first only the cost of the Quarter Pounders that you buy; this is equal to the cost per serving, \$1.84, times the number of servings to buy, as given by the decision variable x_{QP}. That is, your cost for Quarter Pounders comes to $1.84x_{\mathrm{QP}}$. By the same reasoning, your cost for McLean Deluxes is $2.19x_{\mathrm{MD}}$, your cost for Big Macs is $1.84x_{\mathrm{BM}}$, and so forth. The total cost of all your purchases is simply the sum of the individual costs for all of the foods:

$$\begin{aligned}&1.84x_{\mathrm{QP}} + 2.19x_{\mathrm{MD}} + 1.84x_{\mathrm{BM}} + 1.44x_{\mathrm{FF}} + \\ &2.29x_{\mathrm{MC}} + 0.77x_{\mathrm{FR}} + 1.29x_{\mathrm{SM}} + 0.60x_{\mathrm{1M}} + 0.72x_{\mathrm{OJ}}.\end{aligned}$$

This is your objective for the problem. Given any 9 values for the decision variables — what we call a ***solution*** — you can plug them into this expression to get the corresponding total cost. You can verify, for instance, that one serving of each food ($x_{\mathrm{QP}} = 1, \ldots, x_{\mathrm{OJ}} = 1$) costs \$12.98. A solution consisting of five servings of Quarter Pounders and Fries with three of milk and two of orange juice ($x_{\mathrm{QP}} = 5$, $x_{\mathrm{FR}} = 5$, $x_{\mathrm{1M}} = 3$, $x_{\mathrm{OJ}} = 2$, the rest 0) has a total cost of \$16.29.

The objective expression is said to be ***linear*** in this case, because it can be expressed as a sum of terms of the form *constant times variable*. Linearity implies that the cost for each food is proportional to the number of servings of that food purchased, and is independent of the amounts of other foods purchased. Your experience with food service can suggest that these are reasonable assumptions, at least for a simple case; we'll revisit them when we refine the diet model in Chapter 2.

A ***constraint*** is a restriction that any meaningful solution must satisfy. We can write constraints much like objectives, using expressions in terms of the decision variables. Specifically, a constraint may be specified as an equality (=) or inequality ($\le$ or $\ge$) between two expressions, one or both being a function of the variables.

In the case of the diet problem, we explicitly require that the purchased foods provide at least a certain total amount of each nutrient. Thus there is a constraint corresponding to each nutrient. Consider for example the protein requirement. The units of protein provided by all Quarter Pounders that you buy is equal to the units per serving times the number of servings, or $28x_{\mathrm{QP}}$. The units of protein provided by all McLean Deluxes is $24x_{\mathrm{MD}}$, and so forth. Reading across the `Prot` row of the data table, you can see that the total protein provided by all your purchases is

$$\begin{aligned}&28x_{\mathrm{QP}} + 24x_{\mathrm{MD}} + 25x_{\mathrm{BM}} + 14x_{\mathrm{FF}} + \\ &31x_{\mathrm{MC}} + 3x_{\mathrm{FR}} + 15x_{\mathrm{SM}} + 9x_{\mathrm{1M}} + 1x_{\mathrm{OJ}}.\end{aligned}$$

Since we require that the total protein be at least 55, this leads to the following inequality constraint:

$$\begin{aligned}&28x_{\mathrm{QP}} + 24x_{\mathrm{MD}} + 25x_{\mathrm{BM}} + 14x_{\mathrm{FF}} +\\ &31x_{\mathrm{MC}} + 3x_{\mathrm{FR}} + 15x_{\mathrm{SM}} + 9x_{\mathrm{1M}} + 1x_{\mathrm{OJ}} \ge 55.\end{aligned}$$

Similar reasoning applies to each of the other nutrients. Each line of the table in Figure 1–1 gives rise to a nutrient constraint, such as this one for vitamin A:

$$\begin{aligned}&15x_{\mathrm{QP}} + 15x_{\mathrm{MD}} + 6x_{\mathrm{BM}} + 2x_{\mathrm{FF}} +\\ &8x_{\mathrm{MC}} + 0x_{\mathrm{FR}} + 4x_{\mathrm{SM}} + 10x_{\mathrm{1M}} + 2x_{\mathrm{OJ}} \ge 100.\end{aligned}$$

Only a solution that satisfies *all* of these constraints — a so-called ***feasible*** solution — is acceptable.

Given any particular solution, you can plug it into the constraint expressions to check its feasibility. Thus the \$16.29 solution given above is feasible for both protein and vitamin A; in fact you can verify that it satisfies the constraints for all of the nutrients. The \$12.98 solution (having one serving of each food) is feasible in the protein constraint, but does not satisfy the vitamin A constraint; it provides only 62 of the 100 required units of vitamin A. Hence although this solution costs less to buy, it is not acceptable for the given diet problem.

Each of the diet problem's nutrient constraints is linear, because the expression on the left of the inequality operator is a sum of terms of the form constant

$$\begin{array}{lrrrrrl}
\text{Minimize} & \multicolumn{6}{l}{1.84x_{\mathrm{QP}} + 2.19x_{\mathrm{MD}} + 1.84x_{\mathrm{BM}} + 1.44x_{\mathrm{FF}} +} \\
 & \multicolumn{6}{l}{2.29x_{\mathrm{MC}} + 0.77x_{\mathrm{FR}} + 1.29x_{\mathrm{SM}} + 0.60x_{\mathrm{1M}} + 0.72x_{\mathrm{OJ}}} \\
\text{Subject to} & 28x_{\mathrm{QP}} + & 24x_{\mathrm{MD}} + & 25x_{\mathrm{BM}} + & 14x_{\mathrm{FF}} + & & \\
 & 31x_{\mathrm{MC}} + & 3x_{\mathrm{FR}} + & 15x_{\mathrm{SM}} + & 9x_{\mathrm{1M}} + & x_{\mathrm{OJ}} & \ge 55 \\
 & 15x_{\mathrm{QP}} + & 15x_{\mathrm{MD}} + & 6x_{\mathrm{BM}} + & 2x_{\mathrm{FF}} + & & \\
 & 8x_{\mathrm{MC}} & + & 4x_{\mathrm{SM}} + & 10x_{\mathrm{1M}} + & 2x_{\mathrm{OJ}} & \ge 100 \\
 & 6x_{\mathrm{QP}} + & 10x_{\mathrm{MD}} + & 2x_{\mathrm{BM}} & & & \\
 & 15x_{\mathrm{MC}} + & 15x_{\mathrm{FR}} & + & 4x_{\mathrm{1M}} + & 120x_{\mathrm{OJ}} & \ge 100 \\
 & 30x_{\mathrm{QP}} + & 20x_{\mathrm{MD}} + & 25x_{\mathrm{BM}} + & 15x_{\mathrm{FF}} + & & \\
 & 15x_{\mathrm{MC}} & + & 20x_{\mathrm{SM}} + & 30x_{\mathrm{1M}} + & 2x_{\mathrm{OJ}} & \ge 100 \\
 & 20x_{\mathrm{QP}} + & 20x_{\mathrm{MD}} + & 20x_{\mathrm{BM}} + & 10x_{\mathrm{FF}} + & & \\
 & 8x_{\mathrm{MC}} + & 2x_{\mathrm{FR}} + & 15x_{\mathrm{SM}} & + & 2x_{\mathrm{OJ}} & \ge 100 \\
 & 510x_{\mathrm{QP}} + & 370x_{\mathrm{MD}} + & 500x_{\mathrm{BM}} + & 370x_{\mathrm{FF}} + & & \\
 & 400x_{\mathrm{MC}} + & 220x_{\mathrm{FR}} + & 345x_{\mathrm{SM}} + & 110x_{\mathrm{1M}} + & 80x_{\mathrm{OJ}} & \ge 2000 \\
 & 34x_{\mathrm{QP}} + & 35x_{\mathrm{MD}} + & 42x_{\mathrm{BM}} + & 38x_{\mathrm{FF}} + & & \\
 & 42x_{\mathrm{MC}} + & 26x_{\mathrm{FR}} + & 27x_{\mathrm{SM}} + & 12x_{\mathrm{1M}} + & 20x_{\mathrm{OJ}} & \ge 350 \\
 & \multicolumn{6}{r}{x_{\mathrm{QP}} \ge 0,\ x_{\mathrm{MD}} \ge 0,\ x_{\mathrm{BM}} \ge 0,\ x_{\mathrm{FF}} \ge 0} \\
 & \multicolumn{6}{r}{x_{\mathrm{MC}} \ge 0,\ x_{\mathrm{FR}} \ge 0,\ x_{\mathrm{SM}} \ge 0,\ x_{\mathrm{1M}} \ge 0,\ x_{\mathrm{OJ}} \ge 0}
\end{array}$$

Figure 1–2. *Objective and constraints of the simple diet problem.* Following familiar mathematical conventions, we write $1x_{\mathrm{OJ}}$ as x_{OJ} and leave out terms like $0x_{\mathrm{FR}}$.

times variable — like the linear objective — while the expression on the right is simply a constant. In general, a constraint is linear if each of its two expressions is linear, in the sense of being either a constant, a variable, a constant times a variable, or sums of such terms. In the diet context, linearity of the constraints follows from the reasonable assumptions that the nutrients each food provides are proportional to the amount consumed, and that different foods contribute their nutrients independently.

Of course we also require, for any solution to be meaningful, that no negative amounts be purchased: $x_{\mathtt{QP}} \geq 0$, $x_{\mathtt{MD}} \geq 0$, and so forth for all of the variables. Putting these trivially linear constraints and the nutrient constraints together with the objective, our small diet problem can be stated fully as shown in Figure 1–2. An optimization problems of this sort, where the objective and all constraints are linear in terms of the variables, is called a ***linear program,*** or an ***LP*** for short.

If you wanted the diet to supply *exactly* as much of some nutrients as required, you could change some of the $\geq$ signs to $=$ signs in the constraints above. (Your minimum-cost diet might be more expensive as a result.) There is no way to avoid the inequalities $x_{\mathtt{QP}} \geq 0, \ldots, x_{\mathtt{OJ}} \geq 0$, however. As a result, you can't solve linear programs by any of the "elimination" techniques that you may have learned for linear systems of equations. More complicated computational approaches must be applied, though fortunately they can still be quite efficient.

1.2 A simple diet model

To keep our first presentation of a linear program simple, we have arbitrarily limited our attention to 9 foods and 7 nutrients. In reality we were able to collect full data on 63 foods and 12 nutrients, for which the corresponding linear program has 63 variables and 12 constraints (plus nonnegativity). Imagine writing out this LP in Figure 1–2's format. Such a lengthy listing would be tedious to create, prone to error, and hard for others to read — and yet it would still be tiny by current-day standards. LPs involving thousands of constraints are common in practical applications, as some of our later chapters will show.

How can you manage linear programs of more than a few variables and constraints? Somehow you would like to be able to communicate, for example, that the full diet LP *has the same form* as the smaller one, only with more data. There is fortunately a way to do this, by specifying an abstract ***model*** that incorporates the essential features of all diet LPs, using symbols of some kind in place of specific numbers. There are several ways to do this, but for current purposes we'll adopt one approach, based on mathematical notation, that has the advantages of being both widely understood and broadly applicable.

To formulate a diet model, we begin by identifying its fundamental entities. Clearly they are of two kinds: nutrients and foods. We denote by the symbol $\mathcal{N}$ the collection, or ***set,*** of nutrients, and by $\mathcal{F}$ the set of foods. Each individual nutrient or food is a ***member*** of $\mathcal{N}$ or $\mathcal{F}$, respectively. In the case of the small diet LP, $\mathcal{N}$ has the 7 members `Prot`, `VitA`, `VitC`, `Calc`, `Iron`, `Cals`, and `Carb`, representing the 7 nutrients, and $\mathcal{F}$ has the 9 members `QP`, `MD`, `BM`, `FF`, `MC`,`FR`, SM, 1M, and OJ, representing the 9 foods. For the complete diet LP, there are 12 members of $\mathcal{N}$ and 63 members of $\mathcal{F}$.

We next define symbols for all of the numerical data, or ***parameters,*** of the model. In the case of the diet problem, a great many numbers may be involved in specifying the LP, but they are all of only three fundamental types: nutrient amounts, nutrient requirements, and costs. The mathematical convention is to denote each of these by a letter, so we'll say that a denotes the amounts, b the requirements, and c the costs. Individual parameter values of each kind are indicated by ***subscripting*** these letters with appropriate model entities. Thus c_{QP} stands for the cost per serving of Quarter Pounders, $b_{\mathtt{Calc}}$ for the requirement for units of calcium, and $a_{\mathtt{Iron},\mathrm{MD}}$ for the number of units of iron in one serving of McLean Deluxe; for our small diet LP they have the values 1.84, 100, and 20 from Table 1–1.

You can see that certain parameters are associated with certain sets, and are subscripted only by the members of those sets. Specifically, b is subscripted only by members of $\mathcal{N}$, because our diet problem has requirements only for nutrient amounts; c is subscripted only by members of $\mathcal{F}$, because our problem involves costs only for foods. The parameter a is a little different, because there are nutrient amounts for each *combination* of a nutrient and a food. As a result a must have a pair of subscripts, one from $\mathcal{N}$ and one from $\mathcal{F}$. In summary, we can present the following symbolic description of the data required by any diet problem:

$a_{nf} \geq 0$	number of units of nutrient n in one serving of food f, for each $n \in \mathcal{N}$ and $f \in \mathcal{F}$
$b_n > 0$	number of units of nutrient n required, for each $n \in \mathcal{N}$
$c_f \geq 0$	cost per serving of food f, for each $f \in \mathcal{F}$

This description incorporates simple conditions that the data must satisfy, if the resulting linear program is to make any sense. We require $a_{nf} \geq 0$, for instance, because we can't imagine that any food could provide a negative amount of any nutrient. Conditions of this kind are distinct from the model's constraints, which are restrictions that involve the decision variables as well as the data.

The remainder of the model consists of symbolic counterparts to the components of an optimization problem introduced previously: the variables, objective, and constraints.

The ***variables*** are the same as parameters, except that their values are to be determined by optimization, rather than being given as part of the data. The diet example has only one type of variable, associated — like the costs — with the set of foods:

$x_f \geq 0$	number of servings of food f purchased, for each $f \in \mathcal{F}$

The small diet example thus has variables x_{QP} representing the number of Quarter Pounders to be bought, x_{MD} representing the number of McLean Deluxes to be bought, and so forth just as in Figure 1–2. The restrictions $x_f \geq 0$ are simple constraints on these variables, which we include for convenience in the variables' definition.

The ***objective*** of a symbolic model is an algebraic expression in parameters and variables. For the diet example, we can construct such an expression by first considering the costs attributable to individual foods. The amount in dollars that you spend on Quarter Pounders is given by the cost of Quarter Pounders in dollars per serving, times the number of servings you purchase, or $c_{\mathtt{QP}} x_{\mathtt{QP}}$. The amount you spend on McLean Deluxes is similarly $c_{\mathtt{MD}} x_{\mathtt{MD}}$, and so forth for each of the other foods. Thus the total cost is $c_{\mathtt{QP}} x_{\mathtt{QP}} + c_{\mathtt{MD}} x_{\mathtt{MD}} + \cdots + c_{\mathtt{OJ}} x_{\mathtt{OJ}}$.

While this is a more symbolic representation than the one in Figure 1–2, it is just as long and awkward. We want to say that the cost per serving times the number of servings purchased should be *summed over all foods,* but we want to say it without listing each food separately. Mathematical notation provides the $\sum$ operator to express exactly this kind of symbolic summation. To specify the sum of c_f times x_f, over all foods $f \in \mathcal{F}$, you write

$$\textstyle\sum_{f \in \mathcal{F}} c_f x_f.$$

This is a general formulation for the objective in the diet model. It is the same no matter which or how many foods you include in the set $\mathcal{F}$.

The same notation serves to concisely specify each ***constraint.*** Consider the small diet example's protein requirement. The amount of protein supplied to the diet through Quarter Pounders is given by the number of units of protein supplied in each serving, time the number of servings purchased, or $a_{\mathtt{Prot,QP}} x_{\mathtt{QP}}$. The protein supplied by McLean Deluxes is similarly $a_{\mathtt{Prot,MD}} x_{\mathtt{MD}}$, and so forth for each of the other foods. Thus the total protein supplied by purchases of all foods is $a_{\mathtt{Prot,QP}} x_{\mathtt{QP}} + a_{\mathtt{Prot,MD}} x_{\mathtt{MD}} + \cdots + a_{\mathtt{Prot,OJ}} x_{\mathtt{OJ}}$, or, using the concise $\sum$ notation for the sum,

$$\textstyle\sum_{f \in \mathcal{F}} a_{\mathtt{Prot},f} x_f.$$

Since an amount $b_{\mathtt{Prot}}$ of protein is required, the relevant constraint is

$$\textstyle\sum_{f \in \mathcal{F}} a_{\mathtt{Prot},f} x_f \geq b_{\mathtt{Prot}}.$$

The same reasoning yields an analogous constraint for each of the other nutrients:

$$\textstyle\sum_{f \in \mathcal{F}} a_{\mathtt{VitA},f} x_f \geq b_{\mathtt{VitA}}, \quad \sum_{f \in \mathcal{F}} a_{\mathtt{VitC},f} x_f \geq b_{\mathtt{VitC}}, \quad \ldots$$

This is a more concise statement of the constraints than in Figure 1–2, but still it requires that you write out a separate expression corresponding to each nutrient. These expressions are actually all the same except for the name of the nutrient. This suggests that we state the constraint for a generic nutrient n that runs over the set of nutrients $\mathcal{N}$:

$$\textstyle\sum_{f \in \mathcal{F}} a_{nf} x_f \geq b_n, \quad \text{for each } n \in \mathcal{N}.$$

This is the general constraint formulation for the diet model, applicable to any set of foods $\mathcal{F}$ and any set of nutrients $\mathcal{N}$.

Although you have probably seen notation like this in calculus or linear algebra, you may find that its use in specifying constraints will require some practice. The most common confusion has to do with the symbols f and n that stand for members of $\mathcal{F}$ and $\mathcal{N}$, respectively, in the general constraint. These ***index*** symbols are defined and used in distinctly different ways:

- ◇ The index n is defined by "for each $n \in \mathcal{N}$." This indicates that a separate constraint is to be generated by substituting each member of $\mathcal{N}$ for n in the constraint expression.
- ◇ The index f is defined by "$\sum_{f \in \mathcal{F}}$." This indicates that a separate term is to be generated by substituting each member of $\mathcal{F}$ for f in the summand $a_{nf}x_f$. All of these terms are then to be summed as part of the *same* constraint.

When you first substitute a particular member of $\mathcal{N}$ for n, and then expand the sum by substituting all members of $\mathcal{F}$ for f, the result is one of the explicit constraints shown in Figure 1–2. Since there are seven members of $\mathcal{N}$ in our example, there are seven different constraints. Each of these constraints contains a sum of terms for all nine foods (though a few are not explicit in the figure because they are zero).

A common mistake is to get the two kinds of index definitions mixed up, using $\sum$ to try to declare all constraints in one big summation ($\sum_{n \in \mathcal{N}} \sum_{f \in \mathcal{F}} a_{nf}x_f \geq \sum_{n \in \mathcal{N}} b_n$) or using "for each" to declare each summand in a separate constraint ($a_{nf}x_f \geq b_n$ for each $n \in \mathcal{N}$, $f \in \mathcal{F}$). You can avoid such errors by taking the approach we have used with the diet model: first describe the constraint in words, then build up the mathematical formulation so that it matches the word description.

A related error is to use an index like f or n in a way that is inconsistent with its area, or *scope,* of definition. Every appearance of an index in a constraint must lie in exactly one scope within that constraint. The scope of an index defined in a "for each" phrase is the entire constraint expression, but the scope of an index defined by a $\sum$ is only the next term: everything up to the next $+$, $-$, $\leq$, $=$, $\geq$, or right parenthesis, ignoring any parenthesized expressions. Thus all of the following constraint expressions are incorrect:

- ◇ $\sum_{f \in \mathcal{F}} a_{nf}x_f \geq b_f$, for each $n \in \mathcal{N}$: the scope of the index f defined by $\sum_{f \in \mathcal{F}}$ does not extend as far as b_f.
- ◇ $\sum_{n \in \mathcal{N}} \sum_{f \in \mathcal{F}} a_{nf}x_f \geq b_n$: the scope of the index n defined by $\sum_{n \in \mathcal{N}}$ does not extend as far as b_n.
- ◇ $\sum_{n \in \mathcal{N}} \sum_{f \in \mathcal{F}} a_{nf}x_f \geq b_n$, for each $n \in \mathcal{N}$: there are two overlapping scopes of index n, one defined by "for each $n \in \mathcal{N}$" and the other by $\sum_{n \in \mathcal{N}}$.

In our examples we have used f and n as the indices to run over $\mathcal{F}$ and $\mathcal{N}$, respectively. Adopting a convention like this will often help you to spot indexing scope errors, but it is not required, nor even possible in some more complex models to be introduced in later chapters. It would have been mathematically just as correct to use, say, c_j to denote the "cost per serving of food j, for each $j \in \mathcal{F}$." The general rule to remember is that each appearance of an index must be associated with members of some set, by appearing within exactly one defining scope.

Putting all of our definitions together, the general ***algebraic*** statement of the diet ***model*** is as shown in Figure 1–3. Combined with appropriate ***data*** as in

Given	$\mathcal{F}$, a set of foods
	$\mathcal{N}$, a set of nutrients
and	$a_{nf} \geq 0$, the units of nutrient n in one serving of food f, for each $n \in \mathcal{N}$ and $f \in \mathcal{F}$
	$b_n > 0$, the units of nutrient n required, for each $n \in \mathcal{N}$
	$c_f \geq 0$, the cost per serving of food f, for each $f \in \mathcal{F}$
Define	$x_f \geq 0$, the number of servings of food f purchased, for each $f \in \mathcal{F}$
Minimize	$\sum_{f \in \mathcal{F}} c_f x_f$
Subject to	$\sum_{f \in \mathcal{F}} a_{nf} x_f \geq b_n$, for each $n \in \mathcal{N}$

Figure 1–3. *A general "algebraic" statement of the simple diet model.*

Figure 1–1, the diet model gives rise to a particular LP ***instance*** as in Figure 1–2. By changing the data, you generate other instances of the same model. In this way, the diet model provides a convenient way of thinking about the whole class of diet problems. The interrelationship of model and data will be a key theme through this and subsequent chapters.

1.3 Writing the simple diet model for a computer system

A diet problem would not be very interesting if you could only state a model and instantiate it with data, of course. You want to ***solve*** the resulting problem instance: to choose values of the variables that give the smallest possible value for the objective among all solutions that satisfy the constraints. You seek, in other words, an ***optimal*** solution to the problem.

Fortunately, the solving of linear programs has been studied for many decades, and is by now well understood — it is the first subject of the companion volume *Optimization Methods.* Solving LPs is not trivial, even for ones as small as our diet example. Of the several solution methods that are known to be effective, all are based on significant mathematical principles and require a computer to be carried out efficiently. These methods have been implemented in numerous software packages, which we'll refer to as LP ***solvers.*** Currently available solvers are sufficiently reliable to be applied to LPs from many different applications, and are sufficiently fast to solve in a few seconds or minutes LPs that have many thousands of variables and constraints.

For present purposes, therefore, we can assume that the technology for solving linear programs is available. That doesn't address all our difficulties, however, as we still need some way to communicate a linear program to a solver. This, too, is not a trivial matter, as you can see by considering just the problem of preparing input to represent our small diet problem.

One possibility would be to use an input format based on the problem representation exemplified by Figure 1–2. For convenience or efficiency, repeated

```
set NUTR;   # nutrients
set FOOD;   # foods

param nutrLo {NUTR} >= 0;    # lower bounds on nutrients in diet
param foodCost {FOOD} >= 0;  # costs of foods
param amt {NUTR,FOOD} >= 0;  # amounts of nutrient in each food

var Buy {FOOD} >= 0;         # amounts of foods to be purchased

minimize TotalCost: sum {f in FOOD} foodCost[f] * Buy[f];

subject to Need {i in NUTR}:
   sum {f in FOOD} amt[n,f] * Buy[f] >= nutrLo[n];
```

Figure 1–4. *An algebraic statement of the simple diet model of Figure 1-3, rewritten in the AMPL modeling language* (`diet1.mod`).

elements like the + and = signs might be omitted, subscripting might be replaced by a variable-naming convention more suitable to text files, and coefficients of the same variable might be rearranged so that they can be read at the same time. Most solvers do in fact accept one or more formats of this sort. As you can tell from even the diet example, however, these formulations tend to be large and cumbersome for all but the smallest and simplest LP problems. It is hard to imagine typing and maintaining even a few hundred variables and constraints without error. Thus, as a practical matter, input of this kind has to be created by use of a computer program, a so-called *matrix generator.* Each modeling project requires the creation of a new generator, and unfortunately these programs are difficult to write and maintain. They are particularly hard to debug, because their output is a lengthy list of variable names and coefficients that's hard to check for correctness.

A more appealing alternative is to use an input format based on the general algebraic model representation. The input then consists of a model as in Figure 1–3, together with appropriate data as in Figure 1–1. As we have previously observed, the size and complexity of the model depends only on the inherent complexity of the application, while the size of the LP to be solved is determined by the sizes of the sets and parameter tables that comprise the data. This approach has proved to be powerful and convenient for a broad range of linear programs and other optimization problems, and it is the approach adopted for much of this book. (Other input formats may be preferable for particular kinds of models, as we'll see eventually for the case of network optimization.)

Since the language of algebraic models is considerably richer than the language of explicit LP problems, the design and implementation of software to process models is considerably more challenging than the creation of individual matrix generators. Nevertheless, there are a number of well-established packages that incorporate ***algebraic modeling languages*** designed to serve the same purpose as the mathematical language of Figure 1–3.

Figure 1–4 shows our simple diet model expressed in one such language, AMPL. Comparing it to Figure 1–3, you can see many differences that reflect

```
param: FOOD:       foodCost :=
  "Quarter Pounder"    1.84     "Fries, small"        .77
  "McLean Deluxe"      2.19     "Sausage McMuffin"   1.29
  "Big Mac"            1.84     "1% Lowfat Milk"      .60
  "Filet-O-Fish"       1.44     "Orange Juice"        .72
  "McGrilled Chicken"  2.29 ;

param: NUTR: nutrLo :=
   Prot  55   VitA 100   VitC  100
   Calc 100   Iron 100   Cals 2000   Carb 350 ;

param amt (tr):         Cals  Carb  Prot  VitA  VitC  Calc  Iron :=
   "Quarter Pounder"     510    34    28    15     6    30    20
   "McLean Deluxe"       370    35    24    15    10    20    20
   "Big Mac"             500    42    25     6     2    25    20
   "Filet-O-Fish"        370    38    14     2     0    15    10
   "McGrilled Chicken"   400    42    31     8    15    15     8
   "Fries, small"        220    26     3     0    15     0     2
   "Sausage McMuffin"    345    27    15     4     0    20    15
   "1% Lowfat Milk"      110    12     9    10     4    30     0
   "Orange Juice"         80    20     1     2   120     2     2 ;
```

Figure 1–5. *Data for the simple diet model from Figure 1-1, rewritten in the AMPL data format* (`diet1.dat`).

the contrast between a human (though mathematical) language and a computer-readable language. Notable features of the AMPL representation include:

◇ longer, more mnemonic identifiers, such as `FOOD` for the set $\mathcal{F}$, `amt` for the parameter array a, and `Buy` for the variables x;

◇ brackets `[...]` enclosing and commas separating subscripts, so that a_{nf} and x_f become `amt[n,f]` and `Buy[f]`;

◇ multiplication represented by an explicit * operator;

◇ summation expressed by `sum` and a set in braces `{...}`, with $\sum_{f \in \mathcal{F}}$ becoming `sum {f in FOOD}`;

◇ statements ended by a semicolon, and explanatory comments separated from the formal language by the # symbol.

These differences stem mainly from two requirements, that AMPL use the standard (ASCII) character set, and that its syntax be sufficiently regular to be parsed by a computer.

Looking beyond the superficial syntactic differences, however, you can see that there is in fact a very close analogy between the mathematical model representation in Figure 1-3 and the AMPL representation in Figure 1-4. In particular, the statements of the objective and constraints are much the same. Total cost is represented by

$\sum_{f \in \mathcal{F}} c_f x_f \longleftrightarrow$ `sum {f in FOOD} cost[f] * Buy[f]`

and the requirement for nutrients is written

$\sum_{f \in \mathcal{F}} a_{nf} x_f \geq b_n \longleftrightarrow$

```
    sum {f in FOOD} amt[n,f] * Buy[f] >= nutrLo[n]
```

The specification of a constraint "for each $n \in \mathcal{N}$" is denoted by the AMPL phrase `subject to Need {n in NUTR}`. The identifier `Need` is merely a name that we have chosen for purposes of referring to these constraints in AMPL statements; the requirement of such a name is another consequence of the greater formality required in a computer language. The AMPL objective function is similarly named, by use of the phrase `minimize TotalCost`.

There's also an AMPL format for the data associated with a model. An AMPL representation of the small diet problem's data is shown in Figure 1-5. It closely resembles the data as we originally presented it in Figure 1-1, but like the model representation it has been standardized for computer processing. The representation of the complete data is entirely analogous, though the tables are longer to accommodate the full 63 foods and 12 nutrients.

1.4 Solving the diet problems

Suppose now that we have created the AMPL model and data representations for the small diet problem (Figures 1-4 and 1-5) and have saved them in files named, say, `diet1.mod` and `diet1.dat`. Then an AMPL session that finds a minimum-cost diet can be as simple as this:

```
ampl: model diet1.mod;
ampl: data diet1.dat;

ampl: solve;
MINOS 5.5: optimal solution found.
7 iterations, objective 14.8557377

ampl: display Buy;
Buy [*] :=
    '1% Lowfat Milk'  3.42213
           'Big Mac'  0
        Filet-O-Fish  0
      'Fries, small'  6.14754
'McGrilled Chicken'  0
     'McLean Deluxe'  0
      'Orange Juice'  0
   'Quarter Pounder'  4.38525
   'Sausage McMuffin'  0
```

Commands that you type are printed here in *`this slanted font`*, to distinguish them from AMPL's output. The *`model`* and *`data`* commands cause the appropriate files to be read and processed. Then the *`solve`* command constructs the linear program specified by the model and data, and sends it to a solver called MINOS. When AMPL receives the results, it displays a brief message from the solver, indicating that 7 steps or "iterations" of the solver algorithm determined the optimal objective value to be 14.8557377 dollars. Finally, the *`display`* command asks for a listing of the decision variables' optimal values, using their name, `Buy`, as defined in the model. You can see that to six digits of precision (the default for output of *`display`*) the minimum-cost acceptable diet consists of 4.38525 quarter pounders, 6.14754 servings of fries, and 3.42213 servings of milk.

In response to *solve*, AMPL translates our model and data into what is essentially Figure 1–2's explicit linear program, but in a format more convenient for the solver than for human readers. In this small example, both AMPL's work of translation and the solver's work of finding optimal values are completed in a small fraction of a second. Thus for practical purposes you can think of *solve* as a "black box" that simply returns optimal values for the decision variables. Sometimes to check your work, however, you might want to use AMPL's *expand* command to display part of the explicit LP:

```
ampl: expand Need['Cals'];

subject to Need['Cals']:
        510*Buy['Quarter Pounder'] + 220*Buy['Fries, small'] +
        370*Buy['McLean Deluxe'] + 345*Buy['Sausage McMuffin'] +
        500*Buy['Big Mac'] + 110*Buy['1% Lowfat Milk'] +
        370*Buy['Filet-O-Fish'] + 80*Buy['Orange Juice'] +
        400*Buy['McGrilled Chicken'] >= 2000;
```

You can confirm that the constraint shown here, for the calories requirement, matches the calories constraint in Figure 1–2.

Once the solver has completed its work, it writes the resulting values back for AMPL to read. You can then use the *display* command to explore various aspects of the optimal solution. For the diet model, here's how you could look at some information about the constraints:

```
ampl: display Need.lb, Need.body, Need.slack;

:    Need.lb  Need.body     Need.slack  :=
Calc    100     234.221    134.221
Cals   2000    3965.37    1965.37
Carb    350     350          5.68434e-14
Iron    100     100          0
Prot     55     172.029    117.029
VitA    100     100         -2.84217e-14
VitC    100     132.213     32.2131
;
```

Each row corresponds to the Need constraint for one of the nutrients. The column headed Need.body shows the sum of the constraint's terms involving variables — the constraint *body* — which in the diet problem is the total amount of the nutrient provided by the diet. The Need.lb value is the lower bound on the constraint body, which is the lower limit on the nutrient provided; the Need.slack value is the amount by which the nutrient provided exceeds the lower bound, or the difference between the previous two values. In this example you can see that the minimum-cost diet meets the requirements for carbohydrates, iron and vitamin A exactly, while supplying more than enough of the other four nutrients. (A tiny number like 5.68434e-14, meaning 5.68434×10^{-14}, should be regarded as zero; its slight deviation from zero is an artifact of the computer's finite-precision representation of numbers.)

You may be surprised so many of the requirements are met exactly while using so few of the available foods, but in fact this is typical of optimal solutions to LPs. The theory of linear programming shows that if there is any optimal solution at all, there must be one in which the number of positive variables plus the number of positive slacks is no greater than the number of constraints (*Optimization Methods,* part II). Most solvers for linear programs can find a *basic* solution of this sort. In the present example, the total of three positive variables and four positive slacks exactly equals the number of constraints, there being one constraint for each of the seven nutrients.

To switch to the large diet problem, we can use the same model in conjunction with the larger data representation previously described. Supposing that the representation is stored in `diet1all.dat`, we can direct AMPL to switch to the new data and re-solve as follows:

```
ampl: reset data;
ampl: data diet1all.dat;

ampl: solve;
MINOS 5.5: optimal solution found.
16 iterations, objective -4.467016456e-14
```

The solving time remains negligible, even though there are now 63 foods and 12 nutrients. It is tempting to type *display Buy* at this point, but that will get us a listing of all 63 variables, one for each food. We can expect most of these variables to be zero, however; with only 12 constraints, one for each nutrient, there can be at most 12 positive variables as explained previously.

AMPL's *display* command is designed to deal with such a situation. Before issuing this command, we can set an AMPL *option* to direct that zeroes be suppressed:

```
ampl: option omit_zero_rows 1;
```

The *option* command is used to set options for a variety of purposes. An option setting remains in effect for all subsequent commands, until it is changed by another *option* command.

We can now see that only three of the variables are positive in the optimal solution:

```
ampl: display Buy;
Buy [*] :=
                 'Bacon Bits'  55
             'Barbeque Sauce'  50
          'Hot Mustard Sauce'  50
```

These are not three foods that we would expect to comprise a diet! At this point we have solved some diet problem, but evidently not yet the desired one. In Chapter 2 we develop some of the refinements that are necessary in a model that can be considered practical and realistic for analyses of the fast-food diet.

Addendum to Chapter 1: "Programming"

The term *programming* was used in the 1940's to describe the planning or scheduling of related activities within a large operation. Programmers found that they could represent the amount or level of each activity as a decision variable. Then they could mathematically describe the restrictions inherent in the planning or scheduling problem, as a series of equations or inequalities involving certain of the variables. A solution to all of these *constraints* would be considered an acceptable plan or schedule.

Experience soon showed that it was hard to model a complex operation simply by specifying constraints. If there were too few constraints, then many different solutions could satisfy them; if there were too many, then no solutions were possible. A key insight provided a way around this difficulty, however. One could specify, in addition to the constraints, an *objective:* a function of the variables, such as cost or profit, that could be used to decide whether one solution was better than another. Then it didn't matter that many different solutions satisfied the constraints — it sufficed to find one such solution that minimized or maximized the objective. The term "mathematical programming" came to be used to describe the minimization or maximization of an objective function of many variables, subject to constraints on the variables.

A special case of considerable interest occurs when the objective is a linear function, and the constraints are linear equations and linear inequalities. Then the problem is called a *linear program.*

There are several reasons why linear programming is particularly important. First, as these notes will show, a wide variety of problems can be modeled as linear programs. Furthermore, as you will later see, there are fast and reliable methods for solving linear programs even in hundreds or thousands of variables and constraints. The ideas of linear programming are also important for analyzing and solving programming models that are not linear.

All useful methods for solving linear programs require a computer. This is the reason why most of the study of linear programming has taken place since about 1950. Coincidentally, the development of computers gave rise to a new (and now familiar) meaning for the term "programming". There is no direct connection between the notions of linear programming and computer programming, although indirectly some computer programming must be done in order to solve a linear program.

2. A Realistic Model

Chances are that you found Chapter 1's "optimal" solutions to be more entertaining than enlightening. Can you seriously contemplate the small instance's daily diet of Quarter Pounders (ground beef), Fries (potatoes) and milk? Can you accept a calorie content that's almost twice the minimum? Do you expect to be able to purchase fractions of a serving, correctly measured to five decimal places? The solution to the large instance is entirely absurd. It finds a no-cost diet by selecting large quantities of three "foods" that cost zero dollars per unit.

These solutions fail the "laugh test" for the diet problem. They are mathematically correct for the given model and data, but clearly ridiculous from a practical standpoint. This state of affairs is not unusual at the outset of a modeling project, but it's particularly obvious in this case — at least to the extent that you're already quite familiar with fast food and diets. Most of the work of developing a new model lies in studying the situation to be modeled and refining the model accordingly until it is close enough to reality to be useful.

To address the small diet instance's most obvious shortcomings, we'll next investigate a series of additions and refinements: integer solutions, upper bounds on foods and on so-called nutrients, alternative objectives, and penalties for infeasibility. We'll conclude by looking at further, application-specific constraints that are necessary to bring some realism to the large diet instance's solution. By the end of this chapter we'll have a model capable of producing a diet that you might actually be able to follow, at least for a day.

2.1 Integer solutions

Fast-food establishments cannot be asked to serve arbitrarily fractional quantities of the foods on their menu. Indeed, they most often limit orders to whole numbers of servings, and this is the case in the situation we wish to model. We can think of this requirement as being an additional restriction on the decision variables, to the effect that they take only whole number, or *integer,* values. The objective and the nutrient constraints remain as given.

The desired integrality requirement is readily communicated through the AMPL version of the model, by adding the keyword `integer` to the variables' declaration:

```
var Buy {FOOD} integer >= 0;
```

If we store this modified model in file `diet1int.mod` and proceed as before, however, we get only the same result:

```
ampl: model diet1int.mod;
ampl: data diet1.dat;

ampl: solve;
MINOS 5.5: ignoring integrality of 9 variables
MINOS 5.5: optimal solution found.
7 iterations, objective 14.8557377
```

```
ampl: option omit_zero_rows 1;

ampl: display Buy;
Buy [*] :=
   '1% Lowfat Milk'  3.42213
     'Fries, small'  6.14754
  'Quarter Pounder'  4.38525
```

The message about `ignoring integrality` tells us that MINOS, the solver we have been using, is not designed to enforce integrality restrictions. We might try to get some use out of the solution anyway, by rounding it to the nearest integer values; AMPL provides an easy way of doing this:

```
ampl: let {f in FOOD} Buy[f] := round(Buy[f]);

ampl: display Buy;
Buy [*] :=
   '1% Lowfat Milk'  3
     'Fries, small'  6
  'Quarter Pounder'  4
```

AMPL's *`let`* command performs an *assignment* much as in other computer languages. For each `f` in the set `FOOD`, the value of variable `Buy[f]` is reassigned the value of expression `round(Buy[f])`, which is simply the integer closest to the fractional `Buy[f]` value.

The `display` command introduced by the previous chapter can now be applied to the rounded solution:

```
ampl: display Need.lb, Need.body, Need.slack;

:    Need.lb Need.body Need.slack  :=
Calc    100       210        110
Cals   2000      3690       1690
Carb    350       328        -22
Iron    100        92         -8
Prot     55       157        102
VitA    100        90        -10
VitC    100       126         26
```

We see that the rounded diet provides insufficient amounts of carbohydrates, iron, and vitamin A. As it happens, rounding in this case reduces the amounts of all three foods in the diet. Because all of the constraints in this problem are of "≥" type, we could guarantee a feasible solution by instead *rounding up* the amounts. But then the total cost would also necessarily increase. We can investigate this situation by re-solving, rounding all variables up — using AMPL's function `ceil` rather than `round` in the assignment — and displaying the value of objective `TotalCost` that results:

```
ampl: solve;
MINOS 5.5: ignoring integrality of 9 variables
MINOS 5.5: optimal solution found.
7 iterations, objective 14.8557377

ampl: let {f in FOOD} Buy[f] := ceil(Buy[f]);

ampl: display TotalCost;
TotalCost = 16.99
```

This feasible diet is almost 15% more expensive. Perhaps a cheaper feasible diet could be found by rounding some amounts up and some down, but even if you found such a diet you could not be sure that it was the lowest in cost.

To find the least-cost integer solution, a different kind of solver is needed. Here we show the results of switching to such a solver, CPLEX:

```
ampl: option solver cplex;

ampl: solve;
CPLEX 8.1.0: optimal integer solution; objective 15.05
27 MIP simplex iterations
15 branch-and-bound nodes

ampl: display Buy;
Buy [*] :=
   '1% Lowfat Milk'  4
       Filet-O-Fish  1
     'Fries, small'  5
  'Quarter Pounder'  4
```

The solver requires 27 "iterations" compared to 7 before, and constructs 15 "nodes" of a kind of search tree; this all reflects the fact that the problem is substantially more difficult when the variables are required to be integer. The lowest-cost integer diet, which is only about 1.3% more expensive than the fractional one, is not a rounding of any kind. The orders of small fries go from 6.14754 down to 5, and one order of a fourth food, "filet-o-fish," is added.

2.2 Upper bounds

Another unrealistic aspect of our solutions has been their lack of variety. A large proportion of variables at zero is characteristic of optimal solutions to linear programs, as we have previously remarked. Thus if we want the diet to be more varied, we must add constraints to that effect. There is no one way to do so, but as an example we consider the straightforward approach of limiting the number of servings of any one food that may appear in the diet.

Limits on the number of servings are new parameters of our diet model. There is one of these parameters corresponding to each food. Thus, continuing in the AMPL modeling language, we can call these parameters `foodLim` and index them over the set of foods:

```
param foodLim {FOOD} >= 0;
```

Then a collection of constraints, also indexed over foods, may state that the amount of each food to be purchased must be less than or equal to the limit for that food:

```
subject to Limit {f in FOOD}: Buy[f] <= foodLim[f];
```

Simple bound constraints of this sort are usually viewed more naturally as properties of the variables, however. Indeed, we have already seen how the variables' lower bounds (`>= 0`) are conveniently included in the AMPL `var Buy` declaration. Analogous upper bounds are readily added:

```
var Buy {f in FOOD} integer >= 0, <= foodLim[f];
```

(Notice that now the index `f` has to be defined in this declaration, so that a separate limit `foodLim[f]` can be specified for each food `f`.) Some solvers can take advantage of constraints, like `Limit`, that express simple upper bounds. But this is not something that you need be concerned with; AMPL automatically recognizes simple bound constraints and flags them for the solver. You can use whichever of the above alternatives you prefer to write.

Because it is possible to have too much of "nutrients" like calories and carbohydrates in the diet, it also makes sense to add some limits to the constraints. In AMPL we can define both lower and upper limits on nutrients like this:

```
param nutrLo {NUTR} >= 0;
param nutrHi {n in NUTR} >= nutrLo[n];
```

We specify that each upper limit has to be greater than or equal to the corresponding lower limit; when it comes time to solve an instance of the model, AMPL will report an error if any limit data violate this restriction. The constraint on the diet now becomes:

```
subject to Need {n in NUTR}:
   nutrLo[n] <= sum {f in FOOD} amt[n,f] * Buy[f] <= nutrHi[n];
```

This could be written as two separate constraints, but AMPL allows such constraints to be combined so long as the lower and upper limits on the constraint body involve no variables.

To give a specific example, we add parameters `foodLim` and `nutrHi` and modify the `Buy` and `Need` declarations as above, and store the revised AMPL model in file `diet1lim.mod`. We add to the data listing to try an upper limit of 3 on servings of each food,

```
param: FOOD:       foodCost foodLim :=
  'Quarter Pounder'    1.84  3     'Fries, small'        .77  3
  'McLean Deluxe'      2.19  3     'Sausage McMuffin'   1.29  3
  'Big Mac'            1.84  3     '1% Lowfat Milk'      .60  3
  'Filet-O-Fish'       1.44  3     'Orange Juice'        .72  3
  'McGrilled Chicken'  2.29  3 ;
```

and to place limits on calories and carbohydrates:

```
param: NUTR: nutrLo nutrHi :=
   Prot  55 Infinity   Calc 100 Infinity   Cals 2000 3450
   VitA 100 Infinity   Iron 100 Infinity   Carb 350   700
   VitC 100 Infinity ;
```

We store the result in `diet1lim.dat`. Then here's the result we get from the solver that does not enforce integrality:

```
ampl: model diet1lim.mod;
ampl: data diet1lim.dat;

ampl: option solver minos;
ampl: solve;

MINOS 5.5: ignoring integrality of 9 variables
MINOS 5.5: optimal solution found.
16 iterations, objective 15.72082743

ampl: option omit_zero_rows 1;

ampl: display Buy;

Buy [*] :=
   '1% Lowfat Milk'  3
       Filet-O-Fish  0.877155
     'Fries, small'  3
    'McLean Deluxe'  1.57786
     'Orange Juice'  3
  'Quarter Pounder'  2.57185
```

The diet now contains twice as many foods, though at a total cost of almost a dollar more — adding constraints can only force the minimum higher!

Again we see the tendency of optimal LP solution toward extremes, with three of the foods at their lower limit of 0 and three at their upper limit of 3, leaving only the remaining three at fractional values in between. We can no longer simply round up to a feasible solution, however, because there are now upper as well as lower bounds (`Need.ub` as well as `Need.lb`) for the constraints to satisfy:

```
ampl: let {f in FOOD} Buy[f] := ceil(Buy[f]);

ampl: display Need.lb, Need.body, Need.ub;

:     Need.lb Need.body   Need.ub  :=
Calc    100       241     Infinity
Cals   2000      3870         3450
Carb    350       384          700
Iron    100       122     Infinity
Prot     55       185     Infinity
VitA    100       113     Infinity
VitC    100       455     Infinity
```

We see that rounding up causes the upper bound on calories to be exceeded. In fact there is no combination of roundings of this solution that leads to a feasible solution, though some come close.

The solution in whole number amounts retains 3 servings of those foods that were at upper limit in the fractional solution, but uses notably different amounts of the other foods:

```
ampl: option solver cplex;
ampl: solve;

CPLEX 8.1.0: optimal integer solution; objective 16.52
83 MIP simplex iterations
29 branch-and-bound nodes

ampl: display Buy;

Buy [*] :=
   '1% Lowfat Milk'  3
          'Big Mac'  1
     'Fries, small'  3
    'McLean Deluxe'  3
     'Orange Juice'  3
  'Quarter Pounder'  1
```

Because adding the integrality restriction imposes in effect another kind of constraint, the minimum cost increases yet again.

2.3 Alternative objectives

Cost is not the only function of our diet that we might want to minimize. We might want to look into a diet that is as low as possible in a particular nutrient, for example. We already have an expression for the amount of any nutrient n in the diet: $\sum_{f \in \mathcal{F}} a_{nf} x_f$. In AMPL this is `sum {f in FOOD} amt[n,f] * Buy[f]`, which is readily incorporated into a `minimize` statement:

```
minimize TotalNutr {n in NUTR}:
   sum {f in FOOD} amt[n,f] * Buy[f];
```

Adding this statement to the model (file `diet1obj.mod`) defines a collection of alternative objectives, one for the total amount of each nutrient. The total-calories objective is `TotalNutr['Cals']`, for example.

By including both the `TotalCost` and `TotalCalories` objectives in the model, we can investigate the tradeoff between them. We can minimize `TotalCost` and display `TotalNutr['Cals']`, for instance, then minimize `TotalNutr['Cals']` and display `TotalCost`. AMPL's `objective` statement lets us switch between one objective and the other:

```
ampl: model diet1obj.mod;
ampl: data diet1lim.dat;

ampl: objective TotalCost;
ampl: solve;

CPLEX 8.1.0: optimal integer solution; objective 16.52
83 MIP simplex iterations
29 branch-and-bound nodes

ampl: display TotalNutr['Cals'];
TotalNutr['Cals'] = 3350
```

```
ampl: objective TotalNutr['Cals'];
ampl: solve;

CPLEX 8.1.0: optimal integer solution; objective 3195
40 MIP simplex iterations
9 branch-and-bound nodes

ampl: display TotalCost;
TotalCost = 17.84
```

The $16.52 minimum-cost diet, seen already in the preceding section, has a total of 3350 calories. The minimum-calorie diet has only 3195 calories, but a total cost of $17.84.

We should not be surprised that the low-calorie diet costs more, since it is optimized for calories rather than cost. Is there perhaps some other diet that achieves the same calorie minimum at some lower cost? One way to find out is to fix calories at 3195 and then minimize cost again. AMPL's `let` command does the job here, by setting the lower limit (`nutrLo['Cals']`) and the upper limit (`nutrHi['Cals']`) to the same value:

```
ampl: let nutrLo['Cals'] := TotalNutr['Cals'];
ampl: let nutrHi['Cals'] := TotalNutr['Cals'];

ampl: objective TotalCost;
ampl: solve;

CPLEX 8.1.0: optimal integer solution; objective 17.84
15 MIP simplex iterations
0 branch-and-bound nodes
```

There is indeed no cheaper minimum-calorie solution.

We can infer from these results that no solution minimizes both cost and calories. There may be other tradeoffs, however. If we relax the upper limit on calories to 3250, for instance, then we get another solution:

```
ampl: let nutrHi['Cals'] := 3250;
ampl: solve;

CPLEX 8.1.0: optimal integer solution; objective 16.97
31 MIP simplex iterations
2 branch-and-bound nodes
```

The cost is driven to a value less than $17.14, but not as low as the minimum possible value of $16.52.

As this example suggests, there may be more to the diet problem than the minimization of a single objective function. An optimization model is often best viewed not as a device for finding *the* optimum, but as a powerful tool for generating a range of attractive solutions.

2.4 Penalties for infeasibility

To illustrate another common situation, let’s try making the diet still more varied, and perhaps a bit healthier. We set up the model as in section 2.2, but then use AMPL’s `let` command to reduce the limit on each food to 2 servings, and the limit on Quarter Pounders to 1 serving. The MINOS solver can be applied as before:

```
ampl: model diet1lim.mod;
ampl: data diet1lim.dat;

ampl: let {j in FOOD} foodLim[j] := 2;
ampl: let foodLim['Quarter Pounder'] := 1;

ampl: solve;
MINOS 5.5: ignoring integrality of 6 variables
MINOS 5.5: infeasible problem. 3 iterations
```

MINOS performs several “iterations” as previously, but wait — the result message now says `infeasible problem` rather than `optimal solution found`. A solution is returned, but it specifies an unacceptable negative amount for one of the variables:

```
ampl: display Buy;

Buy [*] :=
   '1% Lowfat Milk'   2
          'Big Mac'   2
       Filet-O-Fish   0
     'Fries, small'  -0.240385
'McGrilled Chicken'   2
    'McLean Deluxe'   2
     'Orange Juice'   2
  'Quarter Pounder'   1
 'Sausage McMuffin'   0.75
```

As its message has suggested, MINOS has determined that there does not exist any solution — even a fractional solution — that satisfies all of the constraints in this case. It has only returned the last infeasible solution that it found in its search for a feasible one.

A no-feasible-solution situation like this is usually a sign of inadequacy in the model. You are not going to stop eating, after all. If not all constraints may be met, the model ought to allow for appropriate ones to be relaxed. The nonnegativity constraints are clearly not candidates for relaxation, but any of the others may be.

As an example, suppose that we’re willing to relax the nutrient minimums somewhat. Allowing such a relaxation is an additional decision, so it requires an additional decision variable in the model. Let’s call this variable `Frac`, with the idea that it will represent the fraction of the nutrient lower limits that we are able to satisfy. Ideally it should be 1, but if necessary it should be some smaller nonnegative value:

```
var Frac >= 0, <= 1;
```

We now change the nutrient lower limits from `nutrLo[n]` to `Frac * nutrLo[n]`. In AMPL our constraints become:

```
subject to NeedLo {n in NUTR}:
   sum {f in FOOD} amt[n,f] * Buy[f] >= Frac * nutrLo[n];

subject to NeedHi {n in NUTR}:
   sum {f in FOOD} amt[n,f] * Buy[f] <= nutrHi[n];
```

(We now have to separate the lower and upper limit constraints, because the lower limit is expressed in terms of a variable.)

To complete the revised model, we must modify the formulation to force `Frac` to be as close to 1 as possible. This can be done through the objective, by means of what is known as a *penalty term.* We choose a suitably large parameter `penalty`, and add the term `penalty * (1-Frac)` to the objective function:

```
param penalty = 1000;

minimize PenalizedTotalCost:
   sum {f in FOOD} foodCost[f] * Buy[f] + penalty * (1-Frac);
```

It can be shown that, so long as `penalty` is not too small and the nutrient requirements can be satisfied, the optimal solution will have `Frac` equal to 1 and hence the penalty term equal to 0; the solution will be the same as if the penalty had not been introduced. On the other hand, when it is impossible to satisfy the nutrient requirements, then `Frac` will have to be less than 1, but the minimization of the sum of the total cost plus the penalty term will tend to force `Frac` as close to 1 as possible. (Some experimentation may be necessary to determine how large a parameter like `penalty` ought to be, but a good start is to choose it so that the penalty part of the objective will be several times greater than the regular cost part when a significant infeasibility is encountered.)

The revised model is shown in Figure 2–1. Although the variables `Buy[j]` are declared `integer`, the variable `Frac` is not; this kind of model is known as a ***mixed-integer program*** or ***MIP.*** Repeating our previous test, we find that a feasible solution is now reported even when the additional restrictions of integrality are imposed. `Frac` is indeed less than 1 in the optimal solution:

```
ampl: model diet1relax.mod;
ampl: data diet1lim.dat;

ampl: let {f in FOOD} foodLim[f] := 2;
ampl: let foodLim['Quarter Pounder'] := 1;

ampl: option solver cplex;
ampl: solve;

CPLEX 8.1.0: optimal integer solution; objective 57.12
22 MIP simplex iterations
0 branch-and-bound nodes

ampl: display Frac;
Frac = 0.96
```

Although `PenalizedTotalCost` is the objective being minimized, the values of the other objectives `TotalNutr[n]` can be displayed to see how they compare

```
set NUTR;   # nutrients
set FOOD;   # foods

param nutrLo {NUTR} >= 0;
param nutrHi {n in NUTR} >= nutrLo[i];
                       # requirements for nutrients
param penalty = 1000;  # penalty for falling short of requirements

param foodLim {FOOD} >= 0;   # limits on food amounts
param foodCost {FOOD} >= 0;  # costs of foods
param amt {NUTR,FOOD} >= 0;  # amounts of nutrient in each food

var Buy {f in FOOD} integer >= 0, <= foodLim[f];
                       # amounts of foods to be purchased
var Frac >= 0, <= 1;   # fraction of nutrient requirement met

minimize PenalizedTotalCost:
   sum {f in FOOD} foodCost[f] * Buy[f] + penalty * (1-Frac);

minimize TotalNutr {n in NUTR}:
   sum {f in FOOD} amt[n,f] * Buy[f];

subject to NeedLo {n in NUTR}:
   sum {f in FOOD} amt[n,f] * Buy[f] >= Frac * nutrLo[n];

subject to NeedHi {n in NUTR}:
   sum {f in FOOD} amt[n,f] * Buy[f] <= nutrHi[n];
```

Figure 2–1. *A revised diet model that allows for relaxation of the nutrient requirements if necessary to achieve a feasible solution* (`diet1relax.mod`).

with the nutrient minimums at the current solution:

```
ampl: display TotalNutr, nutrLo;

:    TotalNutr nutrLo  :=
Calc     214       100
Cals    3430      2000
Carb     336       350
Iron     120       100
Prot     208        55
VitA      97       100
VitC     308       100
```

We see that a reduced lower limit of only 4% in carbohydrates and 3% in vitamin A would be sufficient to accommodate the reductions we have made to the food limits.

2.5 Application-specific constraints

Let us now return to the large diet problem instance, with its 63 foods and 12 nutrients. We store as `diet2orig.mod` the AMPL diet model incorporating the integrality and upper bounds introduced in previous sections of this chapter, and as `diet2orig.dat` the corresponding data with `Infinity` for upper limits on the food and nutrient amounts. (We'll leave for an exercise the application of the additional objectives and the infeasibility penalty in the large instance.) As you should recall, a bizarre solution is returned:

```
ampl: model diet2orig.mod;
ampl: data diet2orig.dat;
ampl: solve;

MINOS 5.5: ignoring integrality of 63 variables
MINOS 5.5: optimal solution found.
16 iterations, objective -4.467016456e-14

ampl: option omit_zero_rows 1;

ampl: display Buy;
Buy [*] :=
                 'Bacon Bits'  55
             'Barbeque Sauce'  50
          'Hot Mustard Sauce'  50
```

The supposedly optimal diet consists of very large numbers of three zero-cost foods.

We could try to improve the solution, as in the case of the small diet instance, by putting an upper limit on the amount of each food allowed. But this would overlook the true cause of the difficulty. The zero-cost foods are *condiments,* which can be obtained only in conjunction with purchases of other foods. In fact there are three kinds of condiments:

- ▷ The four "sauce" condiments (including `Honey`) come only with what McDonald's calls Chicken McNuggets. You can get one sauce package with a 6-piece serving of McNuggets, up to two with a 9-piece serving, or up to four with a 20-piece serving.
- ▷ The five "dressing" condiments come only with salads. You can get one package with each salad serving.
- ▷ Two topping condiments — `Croutons` and `Bacon Bits` — also come only with salads. You can get one package of these with each salad serving, too.

Each of these restrictions is readily incorporated into the model, as we now show.

For the Chicken McNuggets constraints, we require sets of the relevant foods and sauces:

```
set F_NUG within FOOD;       # Chicken McNuggets foods
set F_NUG_SCE within FOOD;   # Chicken McNuggets sauces

param amt_nug_sce {F_NUG} > 0;
                  # Limit on sauces per McNuggets serving
```

For each food fnug in F_NUG, we are allowed up to a total of amt_nug_sce[fnug] sauce packages. Hence the total allowance for sauce packages is sum {fnug in F_NUG} amt_nug_sce[fnug] * Buy[fnug]. On the other hand, the total sauce packages "bought" is given by the expression sum {fsce in F_NUG_SCE} Buy[fsce]. (An index in AMPL can be, like fnug or fsce, more than one letter.) The model constrains the total bought to be no more than the allowance:

```
subject to NuggetSauceLimit:
   sum {fsce in F_NUG_SCE} Buy[fsce]
      <= sum {fnug in F_NUG} amt_nug_sce[fnug] * Buy[fnug];
```

It remains only to add the relevant values to the data file:

```
param: F_NUG: amt_nug_sce :=
  "Chicken McNuggets (6 pcs)"   1
  "Chicken McNuggets (9 pcs)"   2
  "Chicken McNuggets (20 pcs)"  4 ;

set F_NUG_SCE :=
  "Hot Mustard Sauce" "Barbeque Sauce"
  "Sweet 'N Sour Sauce" "Honey" ;
```

The situation for the salads can be handled in an analogous fashion, as shown in the completed model, Figure 2–2.

To finish off the model we add two further *ad hoc* constraints that are relevant to our notion of a good diet. We allow for the specification of a certain number of drinks, by defining the subset of FOODS that we consider to be drinks, and a parameter indicating the desired number:

```
set DRINKS within FOOD;    # Drinks
param drinkNum > 0;        # Number of drinks required in diet
```

The constraint says simply that the total purchases of all drinks must equal the specified number:

```
subject to DrinkLimit:
   sum {fd in DRINKS} Buy[fd] = drinkNum;
```

For a one-day diet, we can reasonably think of dividing the foods among three meals, with one drink at each. Thus we add param drinkNum := 3 to the data.

Finally, in a good diet, not more than a certain fraction of calories are from fat. This requires one more parameter, the fraction:

```
param fracCalFat >= 0, <= 1;
```

The constraint says that the sum of calories from fat in all foods purchased must not exceed this number times the sum of calories in all foods purchased:

```
subject to CalFatLimit:
   sum {f in FOOD} amt['CalFat',f] * Buy[f]
      <= fracCalFat * sum {f in FOOD} amt['Cal',f] * Buy[f];
```

This is the one place in the model where a constraint is so special as to merit references to particular set members. Elsewhere we have stated the constraints more generally in terms of indexing over sets, so that the general form of each constraint is clear and changes can be made through updates to the data file. We also avoid placing particular numerical values in the model, in favor of symbolic parameters like `drinkNum` and `fracCalFat` whose values can be read from the data. Specific set members or data values belong in the model only when they are a fundamental aspect of the model that is not subject to change.

Figure 2–2 shows the completed model. It has become quite a bit longer and more complex than our original diet model of Figure 1–4, but that's generally the price to be paid for making a model realistic.

It remains only to try solving the enhanced model, with the complete data. As an example, here are the results we obtain with an upper limit of 2 on servings of all foods except condiments, and with at most 30% of calories from fat:

```
ampl: model diet2.mod
ampl: data diet2.dat

ampl: solve;

CPLEX 8.1.0: optimal integer solution; objective 9.06
866 MIP simplex iterations
507 branch-and-bound nodes

ampl: option omit_zero_rows 1;

ampl: display Buy;
Buy [*] :=
                     Cheerios  1
                 Cheeseburger  1
             'Chocolate Shake'  1
      'Cinnamon Raisin Danish'  1
                     Croutons  1
             'English Muffin'  1
    'H-C Orange Drink (large)'  1
                    Hamburger  2
               'Orange Juice'  1
                 'Side Salad'  1
```

The reported number of "iterations" has gone up by about a factor of 10 compared to our CPLEX run in section 2.2, reflecting the greater difficulty of solving this larger problem. It is not unusual to find that the work of solving an integer program increases faster than the size of the data.

At this point our solution is beginning to look like three meals. One possibility would be:

Orange Juice	H-C Orange Drink (lg)	Chocolate Shake
Cheerios	2 Hamburgers	Cheeseburger
Cinn. Raisin Danish	English Muffin	Side Salad w/ Croutons

Examination of this result might suggest further *ad hoc* constraints — perhaps on the number of "breakfast foods."

The additional constraints developed in this chapter may seem obvious at this point. That is mainly a reflection of your knowledge of diets and fast-food restaurants, however. If you did not know, say, how the condiments were meant to be used, then you would have no way of knowing how to formulate

```
set NUTR;   # nutrients
set FOOD;   # foods

set F_NUG within FOOD;       # Chicken McNuggets foods
set F_NUG_SCE within FOOD;   # Chicken McNuggets sauces

param amt_nug_sce {F_NUG} > 0;
                  # Limit on sauces per serving

set F_SAL within FOOD;       # Salads
set F_SAL_DRE within FOOD;   # Salad dressings
set F_SAL_TOP within FOOD;   # Salad toppings

param amt_sal_dre {F_SAL} > 0;
param amt_sal_top {F_SAL} > 0;
                  # Limits on dressings & toppings per serving

set DRINKS within FOOD;      # Drinks
param drinkNum > 0;          # Number of drinks required in diet

param fracCalFat >= 0, <= 1;
                  # Fraction of calories that may be from fat

param nutrLo {NUTR} >= 0;
param nutrHi {n in NUTR} >= nutrLo[n];
                             # requirements for nutrients

param foodLim {FOOD} >= 0;   # limits on food amounts
param foodCost {FOOD} >= 0;  # costs of foods
param amt {NUTR,FOOD} >= 0;  # amounts of nutrient in each food

var Buy {f in FOOD} integer >= 0, <= foodLim[f];
                             # amounts of foods to be purchased

minimize TotalCost: sum {f in FOOD} foodCost[f] * Buy[f];

subject to Need {n in NUTR}:
   nutrLo[n] <= sum {f in FOOD} amt[n,f] * Buy[f] <= nutrHi[n];

subject to NuggetSauceLimit:
   sum {fsce in F_NUG_SCE} Buy[fsce]
      <= sum {fnug in F_NUG} amt_nug_sce[fnug] * Buy[fnug];

subject to SaladDressingLimit:
   sum {fdre in F_SAL_DRE} Buy[fdre]
      <= sum {fsal in F_SAL} amt_sal_dre[fsal] * Buy[fsal];

subject to SaladToppingLimit:
   sum {ftop in F_SAL_TOP} Buy[ftop]
      <= sum {fsal in F_SAL} amt_sal_top[fsal] * Buy[fsal];

subject to DrinkLimit:
   sum {fd in DRINKS} Buy[fd] = drinkNum;

subject to CalFatLimit:
   sum {f in FOOD} amt['CalFat',f] * Buy[f]
      <= fracCalFat * sum {f in FOOD} amt['Cal',f] * Buy[f];
```

Figure 2–2. *The AMPL statement of the refined diet model, incorporating the improvements introduced in Sections 2.1, 2.2, and 2.5 of this chapter* (`diet2.mod`).

a constraint like `NuggetSauceLimit`. Every realistic modeling project involves learning the situation to be modeled as well as constructing the mathematical formulation; in fact learning the situation is often the harder part of the exercise.