

# 18

---

---

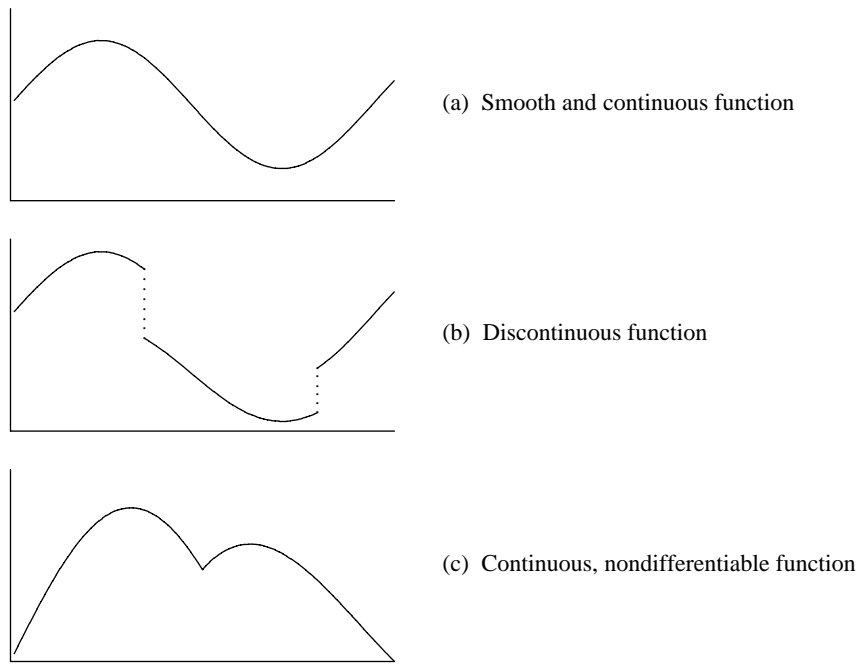
## Nonlinear Programs

Although any model that violates the linearity rules of Chapter 8 is ~~not linear~~, the term ~~nonlinear program~~ is traditionally used in a more narrow sense. For our purposes in this chapter, a nonlinear program, like a linear program, has continuous (rather than integer or discrete) variables; the expressions in its objective and constraints need not be linear, but they must represent ~~smooth~~ functions. Intuitively, a smooth function of one variable has a graph like that in Figure 18-1a, for which there is a well-defined slope at every point; there are no jumps as in Figure 18-1b, or kinks as in Figure 18-1c. Mathematically, a smooth function of any number of variables must be continuous and must have a well-defined gradient (vector of first derivatives) at every point; Figures 18-1b and 18-1c exhibit points at which a function is discontinuous and nondifferentiable, respectively.

Optimization problems in functions of this kind have been singled out for several reasons: because they are readily distinguished from other ~~not linear~~ problems, because they have many and varied applications, and because they are amenable to solution by well-established types of algorithms. Indeed, most solvers for nonlinear programming use methods that rely on the assumptions of continuity and differentiability. Even with these assumptions, nonlinear programs are typically a lot harder to formulate and solve than comparable linear ones.

This chapter begins with an introduction to sources of nonlinearity in mathematical programs. We do not try to cover the variety of nonlinear models systematically, but instead give a few examples to indicate why and how nonlinearities occur. Subsequent sections discuss the implications of nonlinearity for AMPL variables and expressions. Finally, we point out some of the difficulties that you are likely to confront in trying to solve nonlinear programs.

While the focus of this chapter is on nonlinear optimization, keep in mind that AMPL can also express systems of nonlinear equations or inequalities, even if there is no objective to optimize. There exist solvers specialized to this case, and many solvers for nonlinear optimization can also do a decent job of finding a feasible solution to an equation or inequality system.



**Figure 18-1:** Classes of nonlinear functions.

## 18.1 Sources of nonlinearity

We discuss here three ways that nonlinearities come to be included in optimization models: by dropping a linearity assumption, by constructing a nonlinear function to achieve a desired effect, and by modeling an inherently nonlinear physical process.

As an example, we describe some nonlinear variants of the linear network flow model `net1.mod` introduced in Chapter 15 (Figure 15-2a). This linear program's objective is to minimize total shipping cost,

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

where `cost[i,j]` and `Ship[i,j]` represent the cost per unit and total units shipped between cities `i` and `j`, with `LINKS` being the set of all city pairs between which shipment routes exist. The constraints are balance of flow at each city:

```
subject to Balance {k in CITIES}:
    supply[k] + sum {(i,k) in LINKS} Ship[i,k]
        = demand[k] + sum {(k,j) in LINKS} Ship[k,j];
```

with the nonnegative parameters `supply[i]` and `demand[i]` representing the units either available or required at city  $i$ .

### ***Dropping a linearity assumption***

The linear network flow model assumes that each unit shipped from city  $i$  to city  $j$  incurs the same shipping cost, `cost[i, j]`. Figure 18-2a shows a typical plot of shipping cost versus amount shipped in this case; the plot is a line with slope `cost[i, j]` (hence the term linear). The other plots in Figure 18-2 show a variety of other ways, none of them linear, in which shipping cost could depend on the shipment amount.

In Figure 18-2b the cost also tends to increase linearly with the amount shipped, but at certain critical amounts the cost per unit (that is, the slope of the line) makes an abrupt change. This kind of function is called piecewise-linear. It is not linear, strictly speaking, but it is also not smoothly nonlinear. The use of piecewise-linear objectives is the topic of Chapter 17.

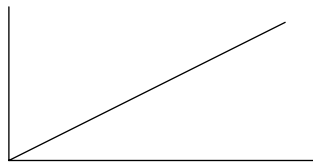
In Figure 18-2c the function itself jumps abruptly. When nothing is shipped, the shipping cost is zero; but when there is any shipment at all, the cost is linear starting from a value greater than zero. In this case there is a fixed cost for using the link from  $i$  to  $j$ , plus a variable cost per unit shipped. Again, this is not a function that can be handled by linear programming techniques, but it is also not a smooth nonlinear function. Fixed costs are most commonly handled by use of integer variables, which are the topic of Chapter 20.

The remaining plots illustrate the sorts of smooth nonlinear functions that we want to consider in this chapter. Figure 18-2d shows a kind of concave cost function. The incremental cost for each additional unit shipped (that is, the slope of the plot) is great at first, but becomes less as more units are shipped; after a certain point, the cost is nearly linear. This is a continuous alternative to the fixed cost function of Figure 18-2c. It could also be used to approximate the cost for a situation (resembling Figure 18-2b) in which volume discounts become available as the amount shipped increases.

Figure 18-2e shows a kind of convex cost function. The cost is more or less linear for smaller shipments, but rises steeply as shipment amounts approach some critical amount. This sort of function would be used to model a situation in which the lowest cost shippers are used first, while shipping becomes progressively more expensive as the number of units increases. The critical amount represents, in effect, an upper bound on the shipments.

These are some of the simplest functional forms. The functions that you consider will depend on the kind of situation that you are trying to model. Figure 18-2f shows a possibility that is neither concave nor convex, combining features of the previous two examples.

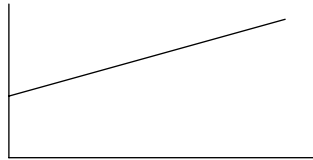
Whereas linear functions are essentially all the same except for the choice of coefficients (or slopes), nonlinear functions can be defined by an infinite variety of different formulas. Thus in building a nonlinear programming model, it is up to you to derive or specify nonlinear functions that properly represent the situation at hand. In the objective



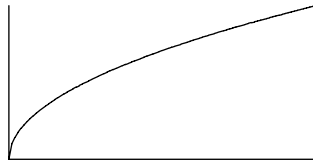
(a) Linear costs



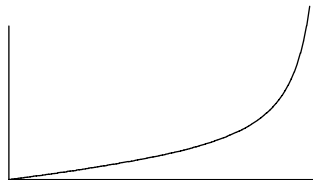
(b) Piecewise linear costs



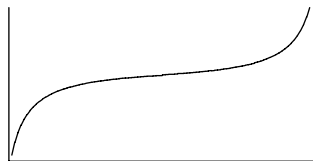
(c) Fixed + variable linear costs



(d) Concave nonlinear costs



(e) Convex nonlinear costs



(f) Combined nonlinear costs

**Figure 18-2:** Nonlinear cost functions.

of the transportation example, for instance, one possibility would be to replace the product  $\text{cost}[i, j] * \text{Ship}[i, j]$  by

$$(\text{cost1}[i, j] + \text{cost2}[i, j] * \text{Ship}[i, j]) / (1 + \text{Ship}[i, j]) * \text{Ship}[i, j]$$

This function grows quickly at small shipment levels but levels off to essentially linear at larger levels. Thus it represents one way to implement the curve shown in Figure 18-2d.

Another way to approach the specification of a nonlinear objective function is to focus on the *slopes* of the plots in Figure 18-2. In the linear case of Figure 18-2a, the slope of the plot is constant; that is why we can use a single parameter  $\text{cost}[i, j]$  to represent the cost per unit shipped. In the piecewise-linear case of Figure 18-2b, the slope is constant within each interval; we can express such piecewise-linear functions as explained in Chapter 17.

In the nonlinear case, however, the slope varies continuously with the amount shipped. This suggests that we go back to our original linear formulation of the network flow problem, and turn the parameter  $\text{cost}[i, j]$  into a variable  $\text{Cost}[i, j]$ :

```
var Cost {ORIG, DEST};          # shipment costs per unit
var Ship {ORIG, DEST} >= 0;     # units to ship

minimize Total_Cost:
    sum {i in ORIG, j in DEST} Cost[i, j] * Ship[i, j];
```

This is no longer a linear objective, because it multiplies a variable by another variable. We add some equations to specify how the cost relates to the amount shipped:

```
subject to Cost_Relation {i in ORIG, j in DEST}:
    Cost[i, j] =
        (cost1[i, j] + cost2[i, j]*Ship[i, j]) / (1 + Ship[i, j]);
```

These equations are also nonlinear, because they involve division by an expression that contains a variable. It is easy to see that  $\text{Cost}[i, j]$  is near  $\text{cost1}[i, j]$  where shipments are near zero, but levels off to  $\text{cost2}[i, j]$  at sufficiently high shipment levels. Thus the concave cost of Figure 18-2d is realized provided that the first cost is greater than the second.

Assumptions of nonlinearity can be found in constraints as well. The constraints of the network flow model embody only a weak linearity assumption, to the effect that the total shipped out of a city is the sum of the shipments to the other cities. But in the production model of Figure 1-6a, the constraint

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

embodies a strong assumption that the number of hours used in each stage  $s$  of making each product  $p$  grows linearly with the level of production.

### ***Achieving a nonlinear effect***

Sometimes nonlinearity arises from a need to model a situation in which a linear function could not possibly exhibit the desired behavior.

In a network model of traffic flows, as one example, it may be necessary to take congestion into account. The total time to traverse a shipment link should be essentially a constant for small shipment amounts, but should increase rapidly towards infinity as the capacity of the link is approached. No linear function has this property, so we are forced to make travel time a nonlinear function of shipment load in order to get the desired effect.

One possibility for expressing the travel time is given by the function

$$\text{time}[i,j] + (\text{sens}[i,j] * \text{Ship}[i,j]) / (1 - \text{Ship}[i,j] / \text{cap}[i,j])$$

This function is approximately  $\text{time}[i,j]$  for small values of  $\text{Ship}[i,j]$ , but goes to infinity as  $\text{Ship}[i,j]$  approaches  $\text{cap}[i,j]$ ; a third parameter  $\text{sens}[i,j]$  governs the shape of the function between the two extremes. This function is always convex, and so has a graph resembling Figure 18-2e. (Exercise 18-4 suggests how this travel time function can be incorporated into a network model of traffic flows.)

As another example, we may want to allow demand to be satisfied only approximately. We can model this possibility by introducing a variable  $\text{Discrepancy}[k]$ , to represent the deviation of the amount delivered from the amount demanded. This variable, which can be either positive or negative, is added to the right-hand side of the balance constraint:

$$\begin{aligned} \text{subject to Balance } \{k \text{ in CITIES}\}: \\ \text{supply}[k] + \sum \{(i,k) \text{ in LINKS}\} \text{Ship}[i,k] \\ = \text{demand}[k] + \text{Discrepancy}[k] + \\ \sum \{(k,j) \text{ in LINKS}\} \text{Ship}[k,j]; \end{aligned}$$

One established approach for keeping the discrepancy from becoming too large is to add a penalty cost to the objective. If this penalty is proportional to the amount of the discrepancy, then we have a convex piecewise-linear penalty term,

$$\begin{aligned} \text{minimize Total\_Cost}: \\ \sum \{(i,j) \text{ in LINKS}\} \text{cost}[i,j] * \text{Ship}[i,j] + \\ \sum \{k \text{ in CITIES}\} \text{pen} * \langle\langle -1,1; 0 \rangle\rangle \text{Discrepancy}[k]; \end{aligned}$$

where  $\text{pen}$  is a positive parameter. AMPL readily transforms this objective to a linear one.

This form of penalty may not achieve the effect that we want, however, because it penalizes each unit of the discrepancy equally. To discourage large discrepancies, we would want the penalty to become steadily larger per unit as the discrepancy becomes worse, but this is not a property that can be achieved by linear penalty functions (or piecewise-linear ones that have a finite number of pieces). Instead a more appropriate penalty function would be quadratic:

```

minimize Total_Cost:
    sum {(i,j) in LINKS} cost[i,j] * Ship[i,j] +
    sum {k in CITIES} pen * Discrepancy[k] %2;

```

Nonlinear objectives that are a sum of squares of some quantities are common in optimization problems that involve approximation or data fitting.

### ***Modeling an inherently nonlinear process***

There are many sources of nonlinearity in models of physical activities like oil refining, power transmission, and structural design. More often than not, the nonlinearities in these models cannot be traced to the relaxation of any linearity assumptions, but are a consequence of inherently nonlinear relationships that govern forces, volumes, currents and so forth. The forms of the nonlinear functions in physical models may be easier to determine, because they arise from empirical measurements and the underlying laws of physics (rather than economics). On the other hand, the nonlinearities in physical models tend to involve more complicated functional forms and interactions among the variables.

As a simple example, a model of a natural gas pipeline network must incorporate not only the shipments between cities but also the pressures at individual cities, which are subject to certain bounds. Thus in addition to the flow variables `Ship[i,j]` the model must define a variable `Press[k]` to represent the pressure at each city `k`. If the pressure is greater at city `i` than at city `j`, then the flow is from `i` to `j` and is related to the pressure by

$$\text{Flow}[i,j] = c[i,j] * (\text{Press}[i] - \text{Press}[j])$$

where `c[i,j]` is a constant determined by the length, diameter, and efficiency of the pipe and the properties of the gas. Compressors and valves along the pipeline give rise to different nonlinear flow relationships. Other kinds of networks, notably for transmission of electricity, have their own nonlinear flow relationships that are dictated by the physics of the situation.

If you know the algebraic form of a nonlinear expression that you want to include in your model, you can probably see a way to write it in AMPL. The next two sections of this chapter consider some of the specific issues and features relevant to declaring variables for nonlinear programs and to writing nonlinear expressions. Lest you get carried away by the ease of writing nonlinear expressions, however, the last section offers some cautionary advice on solving nonlinear programs.

## **18.2 Nonlinear variables**

Although AMPL variables are declared for nonlinear programs in the same way as for linear programs, two features of variables—initial values and automatic substitution—are particularly useful in working with nonlinear models.

### **Initial values of variables**

You may specify values for AMPL variables. Prior to optimization, these ~~initial~~ values can be displayed and manipulated by AMPL commands. When you type `solve`, they are passed to the solver, which may use them as a starting guess at the solution. After the solver has finished its work, the initial values are replaced by the computed optimal ones.

All of the AMPL features for assigning values to parameters are also available for variables. A `var` declaration may also specify initial values in an optional `:=` phrase; for the transportation example, you can write

```
var Ship {LINKS} >= 0, := 1;
```

to set every `Ship[i, j]` initially to 1, or

```
var Ship {(i,j) in LINKS} >= 0, := cap[i,j] - 1;
```

to initialize each `Ship[i, j]` to 1 less than `cap[i, j]`. Alternatively, initial values may be given in a data statement along with the other data for a model:

```
var Ship:      FRA  DET  LAN  WIN  STL  FRE  LAF :=
    GARY      800  400  400  200  400  200  200
    CLEV      800  800  800  600  600  500  600
    PITT      800  800  800  200  300  800  500 ;
```

Any of the data statements for parameters can also be used for variables, as explained in Section 9.4.

All of these features for assigning values to the regular ~~(primal)~~ variables also apply to the dual variables associated with constraints (Section 12.5). AMPL interprets an assignment to a constraint name as an assignment to the associated dual variable or (in the terminology more common in nonlinear programming) to the associated Lagrange multiplier. A few solvers, such as MINOS, can make use of initial values for these multipliers.

You can often speed up the work of the solver by suggesting good initial values. This can be so even for linear programs, but the effect is much stronger in the nonlinear case. The choice of an initial guess may determine what value of the objective is found to be ~~optimal~~ by the solver, or even whether the solver finds any optimal solution at all. These possibilities are discussed further in the last section of this chapter.

If you don't give any initial value for a variable, then AMPL will tentatively set it to zero. If the solver incorporates a routine for determining initial values, then it may re-set the values of any uninitialized variables, while making use of the values of variables that have been initialized. Otherwise, uninitialized variables will be left at zero. Although zero is an obvious starting point, it has no special significance; for some of the examples that we will give in Section 18.4, the solver cannot optimize successfully unless the initial values are reset away from zero.



### Automatic substitution of variables

The issue of substituting variables has already arisen in an example of the previous section, where we declared variables to represent the shipping costs, and then defined them in terms of other variables by use of a constraint:

```
subject to Cost_Relation {(i,j) in LINKS}:
    Cost[i,j] =
        (cost1[i,j] + cost2[i,j]*Ship[i,j]) / (1 + Ship[i,j]);
```

If the expression to the right of the = sign is substituted for every appearance of `Cost[i,j]`, the `Cost` variables can be eliminated from the model, and these constraints need not be passed to the solver. There are two ways in which you can tell AMPL to make such substitutions automatically.

First, by changing option `substout` from its default value of zero to one, you can tell AMPL to look for all ~~defining~~ constraints that have the form shown above: a single variable to the left of an = sign. When this alternative is employed, AMPL tries to use as many of these constraints as possible to substitute variables out of the model. After you have typed `solve` and a nonlinear program has been generated from a model and data, the constraints are scanned in the order that they appeared in the model. A constraint is identified as ~~defining~~ provided that

- it has just one variable to the left of an = sign;
- the left-hand variable's declaration did not specify any restrictions, such as integrality or bounds; and
- the left-hand variable has not already appeared in a constraint identified as defining.

The expression to the right of the = sign is then substituted for every appearance of the left-hand variable in the other constraints, and the defining constraint is dropped. These rules give AMPL an easy way to avoid circular substitutions, but they do imply that the nature and number of substitutions may depend on the ordering of the constraints.

As an alternative, if you want to specify explicitly that a certain collection of variables is to be substituted out, use an = phrase in the declarations of the variables. For the preceding example, you could write:

```
var Cost {(i,j) in LINKS}
    = (cost1[i,j] + cost2[i,j]*Ship[i,j]) / (1 + Ship[i,j]);
```

Then the variables `Cost[i,j]` would be replaced everywhere by the expression following the = sign. Declarations of this kind can appear in any order, subject to the usual requirement that every variable appearing in an = phrase must be previously defined.

Variables that can be substituted out are not mathematically necessary to the optimization problem. Nevertheless, they often serve an essential descriptive purpose; by associating names with nonlinear expressions, they permit more complicated expressions to be written understandably. Moreover, even though these variables have been removed from the problem sent to the solver, their names remain available for use in browsing through the results of optimization.

When the same nonlinear expression appears more than once in the objective and constraints, introducing a defined variable to represent it may make the model more concise as well as more readable. AMPL also processes such a substitution efficiently. In generating a representation of the nonlinear program for the solver, AMPL does not substitute a copy of the whole defining expression for each occurrence of a defined variable. Instead it breaks the expression into a linear and a nonlinear part, and saves one copy of the nonlinear part together with a list of the locations where its value is to be substituted; only the terms of the linear part are substituted explicitly in multiple locations. This separate treatment of linear terms is advantageous for solvers (such as MINOS) that handle the linear terms specially, but it may be turned off by setting option `linelim` to zero.

From the solver's standpoint, substitutions reduce the number of constraints and variables, but tend to make the constraint and objective expressions more complex. As a result, there are circumstances in which a solver will perform better if defined variables are not substituted out. When developing a new model, you may have to experiment to determine which substitutions give the best results.

### 18.3 Nonlinear expressions

Any of AMPL's arithmetic operators (Table 7-1) and arithmetic functions (Table 7-2) may be applied to variables as well as parameters. If any resulting objective or constraint does not satisfy the rules for linearity (Chapter 8) or piecewise-linearity (Chapter 17), AMPL treats it as *not linear*. When you type `solve`, AMPL passes along instructions that are sufficient for your solver to evaluate every expression in every objective and constraint, together with derivatives if appropriate.

If you are using a typical nonlinear solver, it is up to you to define your objective and constraints in terms of the *smooth* functions that the solver requires. The generality of AMPL's expression syntax can be misleading in this regard. For example, if you are trying to use variables `Flow[i, j]` representing flow between points `i` and `j`, it is tempting to write expressions like

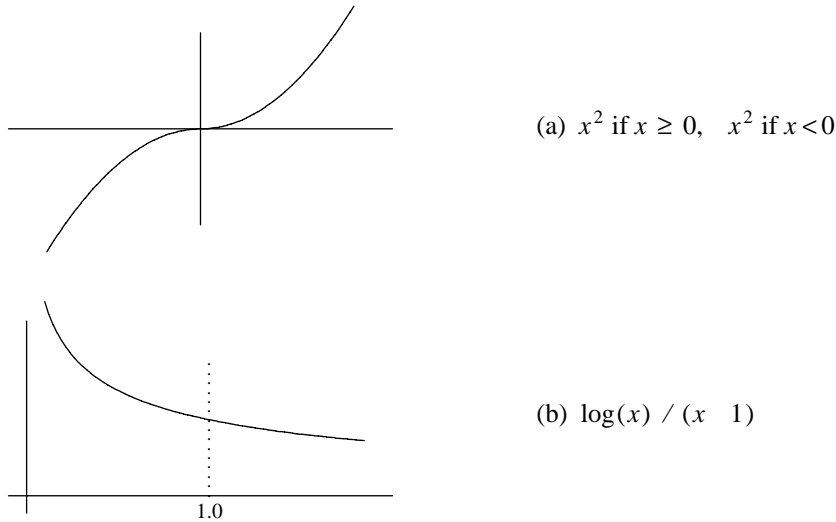
```
cost[i, j] * abs(Flow[i, j])
```

or

```
if Flow[i, j] = 0 then 0 else base[i, j] + cost[i, j]*Flow[i, j]
```

These are certainly not linear, but the first is not smooth (its slope changes abruptly at zero) and the second is not even continuous (its value jumps suddenly at zero). If you try to use such expressions, AMPL will not complain, and your solver may even return what it claims to be an optimal solution *but* the results could be wrong.

Expressions that apply nonsmooth functions (such as `abs`, `min`, and `max`) to variables generally produce nonsmooth results; the same is true of *if-then-else* expressions in which a condition involving variables follows *if*. Nevertheless, there are useful exceptions where a carefully written expression can preserve smoothness. As an exam-



**Figure 18-3:** Smooth nonlinear functions.

ple, consider again the flow-pressure relationship from Section 18.1. If the pressure is greater at city  $i$  than at city  $j$ , then the flow is from  $i$  to  $j$  and is related to the pressure by

$$\text{Flow}[i, j] = c[i, j] * (\text{Press}[i] - \text{Press}[j])$$

If instead the pressure is greater at city  $j$  than at city  $i$ , a similar equation can be written:

$$\text{Flow}[j, i] = c[j, i] * (\text{Press}[j] - \text{Press}[i])$$

But since the constants  $c[i, j]$  and  $c[j, i]$  refer to the same pipe, they are equal. Thus instead of defining a separate variable for flow in each direction, we can let  $\text{Flow}[i, j]$  be unrestricted in sign, with a positive value indicating flow from  $i$  to  $j$  and a negative value indicating the opposite. Using this variable, the previous pair of flow-pressure constraints can be replaced by one:

$$\begin{aligned} &(\text{if } \text{Flow}[i, j] \geq 0 \text{ then } \text{Flow}[i, j] \text{ else } -\text{Flow}[i, j]) \\ &= c[i, j] * (\text{Press}[i] - \text{Press}[j]) \end{aligned}$$

Normally the use of an if expression would give rise to a nonsmooth constraint, but in this case it gives a function whose two quadratic halves meet smoothly where  $\text{Flow}[i, j]$  is zero, as seen in Figure 18-3a.

As another example, the convex function in Figure 18-3b is most easily written  $\log(\text{Flow}[i, j]) / (\text{Flow}[i, j] - 1)$ , but unfortunately if  $\text{Flow}[i, j]$  is 1 this simplifies to  $0/0$ , which would be reported as an error. In fact, this expression does not evaluate accurately if  $\text{Flow}[i, j]$  is merely very close to zero. If instead we write

```

if abs(Flow[i,j]-1) > 0.00001 then
    log(Flow[i,j])/(Flow[i,j]-1)
else
    1.5 - Flow[i,j]/2

```

a highly accurate linear approximation is substituted at small magnitudes of  $\text{Flow}[i, j]$ . This alternative is not smooth in a literal mathematical sense, but it is numerically close enough to being smooth to suffice for use with some solvers.

In the problem instance that it sends to a solver, AMPL distinguishes linear from nonlinear constraints and objectives, and breaks each constraint and objective expression into a sum of linear terms plus a not-linear part. Additional terms that become linear due to the fixing of some variables are recognized as linear. For example, in our example from Section 18.1,

```

minimize Total_Cost:
    sum {i in ORIG, j in DEST} Cost[i,j] * Ship[i,j];

```

each fixing of a  $\text{Cost}[i, j]$  variable gives rise to a linear term; if all the  $\text{Cost}[i, j]$  variables are fixed, then the objective is represented to the solver as linear. Variables may be fixed by a `fix` command (Section 11.4) or through the actions of the presolve phase (Section 14.1); although the presolving algorithm ignores nonlinear constraints, it works with any linear constraints that are available, as well as any constraints that become linear as variables are fixed.

AMPL's built-in functions are only some of the ones most commonly used in model formulations. Libraries of other useful functions can be introduced when needed. To use cumulative normal and inverse cumulative normal functions from a library called `statlib`, for example, you would first load the library with a statement such as

```
load statlib.dll;
```

and declare the functions by use of AMPL `function` statements:

```

function cumnormal;
function invcumnormal;

```

Your model could then make use of these functions to form expressions such as `cumnormal(mean[i], sdev[i], Inv[i, t])` and `invcumnormal(6)`. If these functions are applied to variables, AMPL also arranges for function evaluations to be carried out during the solution process.

A function declaration specifies a library function's name and (optionally) its required arguments. There may be any number of arguments, and even iterated collections of arguments. Each function's declaration must appear before its use. For your convenience, a script containing the function declarations may be supplied along with the library, so that a statement such as `include statlib` is sufficient to provide access to all of the library's functions. Documentation for the library will indicate the functions available and the numbers and meanings of their arguments.

Determining the correct `load` command may involve a number of details that depend on the type of system you're using and even its specific configuration. See Section A.22 for further discussion of the possibilities and the related `AMPLFUNC` option.

If you are ambitious, you can write and compile your own function libraries. Instructions and examples are available from the AMPL web site.

## 18.4 Pitfalls of nonlinear programming

While AMPL gives you the power to formulate diverse nonlinear optimization models, no solver can guarantee an acceptable solution every time you type `solve`. The algorithms used by solvers are susceptible to a variety of difficulties inherent in the complexities of nonlinear functions. This is in unhappy contrast to the linear case, where a well-designed solver can be relied upon to solve almost any linear program.

This section offers a brief introduction to the pitfalls of nonlinear programming. We focus on two common kinds of difficulties, function range violations and multiple local optima, and then mention several other traps more briefly.

For illustration we begin with the nonlinear transportation model shown in Figure 18-4. It is identical to our earlier transportation example (Figure 3-1a) except that the terms `cost[i, j] * Trans[i, j]` are replaced by nonlinear terms in the objective:

```
minimize Total_Cost:
    sum {i in ORIG, j in DEST}
        rate[i, j] * Trans[i, j] / (1 - Trans[i, j]/limit[i, j]);
```

Each term is a convex increasing function of `Trans[i, j]` like that depicted in Figure 18-2e; it is approximately linear with slope `rate[i, j]` at relatively small values of `Trans[i, j]`, but goes to infinity as `Trans[i, j]` approaches `limit[i, j]`. Associated data values, also similar to those used for the linear transportation example in Chapter 3, are given in Figure 18-5.

### Function range violations

An attempt to solve using the model and data as given proves unsuccessful:

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

ampl: solve;
MINOS 5.5 Error evaluating objective Total_Cost
can't compute 8000/0
MINOS 5.5: solution aborted.
8 iterations, objective 0
```

The solver's message is cryptic, but strongly suggests a division by zero while evaluating the objective. That could only happen if the expression

```
1 - Trans[i, j]/limit[i, j]
```

---

```

set ORIG;    # origins
set DEST;    # destinations

param supply {ORIG} >= 0;    # amounts available at origins
param demand {DEST} >= 0;    # amounts required at destinations

    check: sum {i in ORIG} supply[i] = sum {j in DEST} demand[j];

param rate {ORIG,DEST} >= 0;    # base shipment costs per unit
param limit {ORIG,DEST} > 0;    # limit on units shipped

var Trans {i in ORIG, j in DEST} >= 0; # units to ship

minimize Total_Cost:
    sum {i in ORIG, j in DEST}
        rate[i,j] * Trans[i,j] / (1 - Trans[i,j]/limit[i,j]);

subject to Supply {i in ORIG}:
    sum {j in DEST} Trans[i,j] = supply[i];

subject to Demand {j in DEST}:
    sum {i in ORIG} Trans[i,j] = demand[j];

```

**Figure 18-4:** Nonlinear transportation model (nltrans.mod).

```

param: ORIG:  supply :=
        GARY   1400    CLEV   2600    PITT   2900 ;

param: DEST:  demand :=
        FRA    900    DET    1200    LAN    600
        WIN    400    STL    1700    FRE    1100
        LAF    1000 ;

param rate :  FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY  39   14   11   14   16   82   8
        CLEV  27   9    12   9    26   95   17
        PITT  24   14   17   13   28   99   20 ;

param limit :  FRA  DET  LAN  WIN  STL  FRE  LAF :=
        GARY  500 1000 1000 1000 800  500 1000
        CLEV  500 800  800  800  500  500 1000
        PITT  800 600  600  600  500  500 900 ;

```

**Figure 18-5:** Data for nonlinear transportation model (nltrans.dat).

---

is zero at some point. If we use `display` to print the pairs where `Trans[i,j]` equals `limit[i,j]`:

```

ampl: display {i in ORIG, j in DEST: Trans[i,j] = limit[i,j]};
set {i in ORIG, j in DEST: Trans[i,j] == limit[i,j]}
    := (GARY,LAF) (PITT,LAN);

ampl: display Trans[GARY,LAF], limit[GARY,LAF];
Trans[GARY,LAF] = 1000
limit[GARY,LAF] = 1000

```

we can see the problem. The solver has allowed `Trans[GARY,LAF]` to have the value 1000, which equals `limit[GARY,LAF]`. As a result, the objective function term

```
rate[GARY,LAF] * Trans[GARY,LAF]
/ (1 - Trans[GARY,LAF]/limit[GARY,LAF])
```

evaluates to 8000/0. Since the solver is unable to evaluate the objective function, it gives up without finding an optimal solution.

Because the behavior of a nonlinear optimization algorithm can be sensitive to the choice of starting guess, we might hope that the solver will have greater success from a different start. To ensure that the comparison is meaningful, we first set

```
ampl: option send_statuses 0;
```

so that status information about variables that was returned by the previous solve will not be sent back to help determine a starting point for the next solve. Then AMPL's `let` command may be used to suggest, for example, a new initial value for each `Trans[i,j]` that is half of `limit[i,j]`:

```
ampl: let {i in ORIG, j in DEST} Trans[i,j] := limit[i,j]/2;
ampl: solve;
MINOS 5.5: the current point cannot be improved.
32 iterations, objective -7.385903389e+18
```

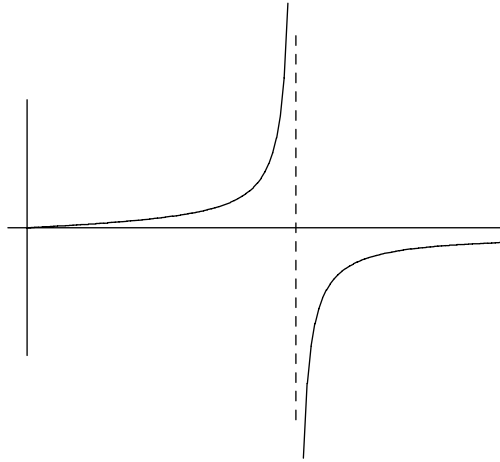
This time the solver runs to completion, but there is still something wrong. The objective is less than  $10^{18}$ , or  $\infty$  for all practical purposes, and the solution is described as `cannot be improved` rather than optimal.

Examining the values of `Trans[i,j]/limit[i,j]` in the solution that the solver has returned gives a clue to the difficulty:

```
ampl: display {i in ORIG, j in DEST} Trans[i,j]/limit[i,j];
Trans[i,j]/limit[i,j] [*,*] (tr)
:      CLEV      GARY      PITT      :=
DET    -6.125e-14    0          2
FRA      0          1.5        0.1875
FRE      0.7         1          0.5
LAF      0.4         0.15       0.5
LAN      0.375       7.03288e-15 0.5
STL      2.9         0          0.5
WIN      0.125       0          0.5
;
```

These ratios show that the shipments for several pairs, such as `Trans[CLEV,STL]`, significantly exceed their limits. More seriously, `Trans[GARY,FRE]` seems to be right at `limit[GARY,FRE]`, since their ratio is given as 1. If we display them to full precision, however, we see:

```
ampl: option display_precision 0;
ampl: display Trans[GARY,FRE], limit[GARY,FRE];
Trans[GARY,FRE] = 500.000000000000028
limit[GARY,FRE] = 500
```



**Figure 18-6:** Singularity in cost function  $y = x/(1 - x/c)$ .

The variable is just slightly larger than the limit, so the cost term has a huge negative value. If we graph the entire cost function, as in Figure 18-6, we see that indeed the cost function goes off to  $-\infty$  to the right of the singularity at `limit[GARY,FRE]`.

The source of error in both runs above is our assumption that, since the objective goes to  $+\infty$  as `Trans[i,j]` approaches `limit[i,j]` from below, the solver will keep `Trans[i,j]` between 0 and `limit[i,j]`. At least for this solver, we must enforce such an assumption by giving each `Trans[i,j]` an explicit upper bound that is slightly less than `limit[i,j]`, but close enough not to affect the eventual optimal solution:

```
var Trans {i in ORIG, j in DEST} >= 0, <= .9999 * limit[i,j];
```

With this modification, the solver readily finds an optimum:

```
ampl: option display_precision 6;
ampl: model nltransb.mod; data nltrans.dat; solve;
MINOS 5.5: optimal solution found.
81 iterations, objective 1212117
ampl: display Trans;
Trans [*,*] (tr)
:      CLEV      GARY      PITT      :=
DET    586.372    187.385    426.242
FRA    294.993     81.2205    523.787
FRE    365.5      369.722    364.778
LAF    490.537      0        509.463
LAN    294.148      0        305.852
STL    469.691    761.672    468.637
WIN     98.7595      0        301.241
;
```



These values of the variables are well away from any singularity, with  $\text{Trans}[i,j]/\text{limit}[i,j]$  being less than 0.96 in every case. (If you change the starting guess to be  $\text{limit}[i,j]/2$  as before, you should find that the solution is the same but the solver needs only about half as many iterations to find it.)

The immediate lesson here is that nonlinear functions can behave quite badly outside the intended range of the variables. The incomplete graph in Figure 18-2e made this cost function look misleadingly well-behaved, whereas Figure 18-6 shows the need for a bound to keep the variable away from the singularity.

A more general lesson is that difficulties posed by a nonlinear function may lead the solver to fail in a variety of ways. When developing a nonlinear model, you need to be alert to bizarre results from the solver, and you may have to do some detective work to trace the problem back to a flaw in the model.

### **Multiple local optima**

To illustrate a different kind of difficulty in nonlinear optimization, we consider a slightly modified objective function that has the following formula:

```
minimize Total_Cost:
    sum {i in ORIG, j in DEST}
        rate[i,j] * Trans[i,j]0.8 / (1 - Trans[i,j]/limit[i,j]);
```

By raising the amount shipped to the power 0.8, we cause the cost function to be concave at lower shipment amounts and convex at higher amounts, in the manner of Figure 18-2f. Attempting to solve this new model, we again initially run into technical difficulties:

```
ampl: model nltransc.mod; data nltrans.dat; solve;
MINOS 5.5: Error evaluating objective Total_Cost:
           can't evaluate pow(0,0.8)
MINOS 5.5: solution aborted.
8 iterations, objective 0
```

This time our suspicion naturally centers upon  $\text{Trans}[i,j]^{0.8}$ , the only expression that we have changed in the model. A further clue is provided by the error message (reference to  $\text{pow}(0,0.8)$ ), which denotes the derivative of the exponential (power) function at zero. When  $\text{Trans}[i,j]$  is zero, this function has a well-defined value, but its derivative with respect to the variable is infinite. The slope of the graph in Figure 18-2f is infinite. As a result, the partial derivative of the total cost with respect to any variable at zero cannot be returned to the solver; since the solver requires all the partial derivatives for its optimization algorithm, it gives up.

This is another variation on the range violation problem, and again it can be remedied by imposing some bounds to keep the solution away from troublesome points. In this case, we move the lower bound from zero to a very small positive number:

```
var Trans {i in ORIG, j in DEST}
    >= 1e-10, <= .9999 * limit[i,j], := 0;
```

We might also move the starting guess away from zero, but in this example the solver takes care of that automatically, since the initial values only suggest a starting point.

With the bounds adjusted, the solver runs normally and reports a solution:

```

ampl: model nltransd.mod; data nltrans.dat; solve;
MINOS 5.5: optimal solution found.
65 iterations, objective 427568.1225
ampl: display Trans;
Trans [*,*] (tr)
:      CLEV      GARY      PITT      :=
DET      689.091      1e-10      510.909
FRA      1e-10      199.005      700.995
FRE      385.326      326.135      388.54
LAF      885.965      114.035      1e-10
LAN      169.662      1e-10      430.338
STL      469.956      760.826      469.218
WIN      1e-10      1e-10      400
;

```

We can regard each 1e-10 as a zero, since such a small value is negligible in comparison with the rest of the solution.

Next we again try a starting guess at  $\text{limit}[i, j]/2$ , in the hope of speeding things up. This is the result:

```

ampl: let {i in ORIG, j in DEST} Trans[i,j] := limit[i,j]/2;
ampl: solve;
MINOS 5.5: optimal solution found.
40 iterations, objective 355438.2006

ampl: display Trans;
Trans [*,*] (tr)
:      CLEV      GARY      PITT      :=
DET      540.601      265.509      393.89
FRA      328.599      1e-10      571.401
FRE      364.639      371.628      363.732
LAF      491.262      1e-10      508.738
LAN      301.741      1e-10      298.259
STL      469.108      762.863      468.029
WIN      104.049      1e-10      295.951
;

```

Not only is the solution completely different, but the optimal value is 17% lower! The first solution could not truly have minimized the objective over all solutions that are feasible in the constraints.

Actually both solutions can be considered correct, in the sense that each is *locally* optimal. That is, each solution is less costly than any other nearby solutions. All of the classical methods of nonlinear optimization, which are the methods considered in this chapter, are designed to seek a local optimum. Started from one specified initial guess, these methods are not guaranteed to find a solution that is *globally* optimal, in the sense of giving the best objective value among all solutions that satisfy the constraints. In general, finding a global optimum is much harder than finding a local one.

Fortunately, there are many cases in which a local optimum is entirely satisfactory. When the objective and constraints satisfy certain properties, any local optimum is also global; the model considered at the beginning of this section is one example, where the convexity of the objective, together with the linearity of the constraints, guarantees that the solver will find a global optimum. (Linear programming is an even more special case with this property; that's why in previous chapters we never encountered local optima that were not global.)

Even when there is more than one local optimum, a knowledge of the situation being modeled may help you to identify the global one. Perhaps you can choose an initial solution near to the global optimum, or you can add some constraints that rule out regions known to contain local optima.

Finally, you may be content to find a very good local optimum, even if you don't have a guarantee that it is global. One straightforward approach is to try a series of starting points systematically, and take the best among the solutions. As a simple illustration, suppose that we declare the variables in our example as follows:

```
param alpha >= 0, <= 1;
var Trans {i in ORIG, j in DEST}
    >= 1e-10, <= .9999 * limit[i,j], := alpha * limit[i,j];
```

For each choice of  $\alpha$  we get a different starting guess, and potentially a different solution. Here are some resulting objective values for  $\alpha$  ranging from 0 to 1:

$\alpha$	Total_Cost
0.0	427568.1
0.1	366791.2
0.2	366791.2
0.3	366791.2
0.4	366791.2
0.5	355438.2
0.6	356531.5
0.7	376043.3
0.8	367014.4
0.9	402795.3
1.0	365827.2

The solution that we previously found for an  $\alpha$  of 0.5 is still the best, but in light of these results we are now more inclined to believe that it is a very good solution. We might also observe that, although the reported objective value varies somewhat erratically with the choice of starting point, a feature of nonlinear programs generally, the second-best value of Total\_Cost was found by setting  $\alpha$  to 0.6. This suggests that a closer search of  $\alpha$  values around 0.5 might be worthwhile.

Some of the more sophisticated methods for global optimization attempt to search through starting points in this way, but with a more elaborate and systematic procedure for deciding which starting points to try next. Others treat global optimization as more of a combinatorial problem, and apply solution methods motivated by those for integer pro-

gramming (Chapter 20). Global optimization methods are still at a relatively early stage of development, and are likely to improve as experience accumulates, new ideas are tried, and available computing power further increases.

### **Other pitfalls**

Many other factors can influence the efficiency and success of a nonlinear solver, including the way that the model is formulated and the choice of units (or scaling) for the variables. As a rule, nonlinearities are more easily handled when they appear in the objective function rather than in the constraints. AMPL's option to substitute variables automatically, described earlier in this chapter, may help in this regard. Another rule of thumb is that the values of the variables should differ by at most a few orders of magnitude; solvers can be misled when some variables are, say, in millions and others are in thousandths. Some solvers automatically scale a problem to try to avoid such a situation, but you can help them considerably by judiciously picking the units in which the variables are expressed.

Nonlinear solvers also have many modes of failure besides the ones we have discussed. Some methods of nonlinear optimization can get stuck at stationary points that are not optimal in any sense, can identify a maximum when a minimum is desired (or vice-versa), and can falsely give an indication that there is no feasible solution to the constraints. In these cases your only recourse may be to try a different starting guess; it can sometimes help to specify a start that is feasible for many of the nonlinear constraints. You may also improve the solver's chances of success by placing realistic bounds on the variables. If you know, for instance, that an optimal value of 80 is plausible for some variables, but a value of 800 is not, you may want to give them a bound of 400. (Once an indicated optimum is at hand, you should be sure to check whether these safety bounds have been reached by any of the variables; if so, the bounds should be relaxed and the problem re-solved.)

The intent of this section has been to illustrate that extra caution is advisable in working with nonlinear models. If you encounter a difficulty that cannot be resolved by any of the simple devices described here, you may need to consult a textbook in nonlinear programming, the documentation for the particular solver that you are using, or a numerical analyst versed in nonlinear optimization techniques.

### **Bibliography**

Roger Fletcher, *Practical Methods of Optimization*. John Wiley & Sons (New York, NY, 1987). A concise survey of theory and methods.

Philip E. Gill, Walter Murray and Margaret H. Wright, *Practical Optimization*. Academic Press (New York, NY, 1981). Theory, algorithms and practical advice.

Jorge Nocedal and Stephen J. Wright, *Numerical Optimization*. Springer Verlag (Heidelberg, 1999). A text on methods for optimization of smooth functions.

Richard P. O'Neill, Mark Williard, Bert Wilkins and Ralph Pike, *The Mathematical Programming Model for Allocation of Natural Gas*, *Operations Research* **27**, 5 (1979) pp. 857–873. A source for the nonlinear relationships in natural gas pipeline networks described in Section 18.1.

## Exercises

**18-1.** In the last example of Section 18.4, try some more starting points to see if you can find an even better locally optimal solution. What is the best solution you can find?

**18-2.** The following little model `fence.mod` determines the dimensions of a rectangular field of maximum area that can be surrounded by a fence of given length:

```
param fence > 0;
var Xfield >= 0;
var Yfield >= 0;

maximize Area: Xfield * Yfield;
subject to Enclose: 2*Xfield + 2*Yfield <= fence;
```

It is well known that the optimum field is a square.

(a) When we tried to solve this problem for a fence of 40 meters, with the default initial guess of zero for the variables, we got the following result:

```
ampl: solve;
MINOS 5.5: optimal solution found.
0 iterations, objective 0

ampl: display Xfield, Yfield;
Xfield = 0
Yfield = 0
```

What could explain this unexpected outcome? Try the same problem on any nonlinear solver available to you, and report the behavior that you observe.

(b) Using a different starting point if necessary, run your solver to confirm that the optimal dimensions for 40 meters of fence are indeed 10 10.

(c) Experiment with an analogous model for determining the dimensions of a box of maximum volume that can be wrapped by paper of a given area.

(d) Solve the same problem as in (c), but for wrapping a cylinder rather than a box.

**18-3.** A falling object on a nameless planet has been observed to have approximately the following heights  $h_j$  at (mostly) one-second intervals  $t_j$ :

$t_j$	0.0	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	9.5	10.0
$h_j$	100	95	87	76	66	56	47	38	26	15	6	0

According to the laws of physics on this planet, the height of the object at any time should be given by the formula

$$h_j = a_0 - a_1 t_j + \frac{1}{2} a_2 t_j^2,$$

where  $a_0$  is the initial height,  $a_1$  is the initial velocity, and  $a_2$  is the acceleration due to gravity. But since the observations were not made exactly, there exists no choice of  $a_0$ ,  $a_1$ , and  $a_2$  that will

cause all of the data to fit this formula exactly. Instead, we wish to estimate these three values by choosing them so as to minimize the sum of squares

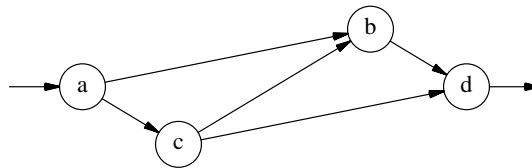
$$\sum_{j=1}^n [h_j - (a_0 + a_1 t_j + \frac{1}{2} a_2 t_j^2)]^2.$$

where  $t_j$  and  $h_j$  are the observations from the  $j$ th entry of the table, and  $n$  is the number of observations. This expression measures the error between the ideal formula and the observed behavior.

(a) Create an AMPL model that minimizes the sum of squares for any number  $n$  of observations  $t_j$  and  $h_j$ . This model should have three variables and an objective function, but no constraints.

(b) Use AMPL data statements to represent the sample observations given above, and solve the resulting nonlinear program to determine the estimates of  $a_0$ ,  $a_1$ , and  $a_2$ .

**18-4.** This problem involves a very simple traffic flow network:



Traffic proceeds in the direction of the arrows, entering at intersection  $a$ , exiting at  $d$ , and passing through  $b$  or  $c$  or both. These data values are given for the roads connecting the intersections:

From	To	Time	Capacity	Sensitivity
$a$	$b$	5.0	10	0.1
$a$	$c$	1.0	30	0.9
$c$	$b$	2.0	10	0.9
$b$	$d$	1.0	30	0.9
$c$	$d$	5.0	10	0.1

To be specific, we imagine that the times are in minutes, the capacities are in cars per minute, and the sensitivities are in minutes per (car per minute).

The following AMPL statements can be used to represent networks of this kind:

```

set inters;      # road intersections

param EN symbolic in inters;  # entrance to network
param EX symbolic in inters;  # exit from network

set roads within {i in inters, j in inters: i <> EX and j <> EN};

param time {roads} > 0;
param cap {roads} > 0;
param sens {roads} > 0;

```

(a) What is the shortest path, in minutes, from the entrance to the exit of this network? Construct a shortest path model, along the lines of Figure 15-7, that verifies your answer.

(b) What is the maximum traffic flow from entrance to exit, in cars per minute, that the network can sustain? Construct a maximum flow model, along the lines of Figure 15-6, that verifies your answer.

(c) Question (a) above was concerned only with the speed of traffic, and question (b) only with the volume of traffic. In reality, these quantities are interrelated. As the traffic volume on a road increases from zero, the time required to travel the road also increases.

Travel time increases only moderately when there are just a few cars, but it increases very rapidly as the traffic approaches capacity. Thus a nonlinear function is needed to model this phenomenon. Let us define  $T[i, j]$ , the travel time on road  $(i, j)$ , by the following constraints:

```
var X {roads} >= 0;    # cars per minute entering road (i,j)
var T {roads};         # travel time for road (i,j)

subject to Travel_Time {(i,j) in roads}:
    T[i,j] = time[i,j] + (sens[i,j]*X[i,j]) / (1-X[i,j]/cap[i,j]);
```

You can confirm that the travel time on  $(i, j)$  is close to  $\text{time}[i, j]$  when the road is lightly used, but goes to infinity as the use approaches  $\text{cap}[i, j]$  cars per minute. The magnitude of  $\text{sens}[i, j]$  controls the rate at which travel time increases, as more cars try to enter the road.

Suppose that we want to analyze how the network can best handle some number of cars per minute. The objective is to minimize average travel time from entrance to exit:

```
param through > 0;    # cars per minute using the network

minimize Avg_Time:
    (sum {(i,j) in roads} T[i,j] * X[i,j]) / through;
```

The nonlinear expression  $T[i, j] * X[i, j]$  is travel minutes on road  $(i, j)$  times cars per minute entering the road. Hence, the number of cars on road  $(i, j)$ . The summation in the objective thus gives the total cars in the entire system. Dividing by the number of cars per minute getting through, we have the average number of minutes for each car.

Complete the model by adding the following:

- ⌘ A constraint that total cars per minute in equals total cars per minute out at each intersection, except the entrance and exit.
- ⌘ A constraint that total cars per minute leaving the entrance equals the total per minute (represented by `through`) that are using the network.
- ⌘ Appropriate bounds on the variables. (The example in Section 18.4 should suggest what bounds are needed.)

Use AMPL to show that, for the example given above, a throughput of 4.0 cars per minute is optimally managed by sending half the cars along  $a \rightarrow b \rightarrow d$  and half along  $a \rightarrow c \rightarrow d$ , giving an average travel time of about 8.18 minutes.

(d) By trying values of parameter `through` both greater than and less than 4.0, develop a graph of minimum average travel time as a function of throughput. Also, keep a record of which travel routes are used in the optimal solutions for different throughputs, and summarize this information on your graph.

What is the relationship between the information in your graph and the solutions from parts (a) and (b)?

(e) The model in (c) assumes that you can make the cars/drivers take certain routes. For example, in the optimal solution for a throughput of 4.0, no drivers are allowed to ~~go~~ ~~but~~ ~~through~~ ~~to~~ ~~from~~ ~~c~~ to ~~b~~.

What would happen if instead all drivers could take whatever route they pleased? Observation has shown that, in such a case, the traffic tends to reach a *stable* solution in which no route has a travel time less than the average. The optimal solution for a throughput of 4.0 is not stable, since ~~it~~ ~~is~~

you can verify the travel time on  $a \rightarrow c \rightarrow b \rightarrow d$  is only about 7.86 minutes; some drivers would try to cut through if they were permitted.

To find a stable solution using AMPL, we have to add some data specifying the possible routes from entrance to exit:

```
param choices integer > 0;    # number of routes
set route {1..choices} within roads;
```

Here `route` is an indexed collection of sets; for each  $r$  in  $1..choices$ , the expression `route[r]` denotes a different subset of roads that together form a route from EN to EX. For our network, `choices` should be 3, and the `route` sets should be  $\{(a,b), (b,d)\}$ ,  $\{(a,c), (c,d)\}$  and  $\{(a,c), (c,b), (b,d)\}$ . Using these data values, the stability conditions may be ensured by one additional collection of constraints, which say that the time to travel any route is no less than the average travel time for all routes:

```
subject to Stability {r in 1..choices}:
    sum {(i,j) in route[r]} T[i,j] >=
        (sum {(i,j) in roads} T[i,j] * X[i,j]) / through;
```

Show that, in the stable solution for a throughput of 4.0, a little more than 5% of the drivers cut through, and the average travel time increases to about 8.27 minutes. Thus traffic would have been faster if the road from  $c$  to  $b$  had never been built! (This phenomenon is known as Braess's paradox, in honor of a traffic analyst who noticed that when a certain link was added to Munich's road system, traffic seemed to get worse.)

(f) By trying throughput values both greater than and less than 4.0, develop a graph of the stable travel time as a function of throughput. Indicate, on the graph, for which throughputs the stable time is greater than the optimal time.

(g) Suppose now that you have been hired to analyze the possibility of opening an additional winding road, directly from  $a$  to  $d$ , with travel time 5 minutes, capacity 10, and sensitivity 1.5. Working with the models developed above, write an analysis of the consequences of making this change, as a function of the throughput value.

**18-5.** Return to the model constructed in (e) of the previous exercise. This exercise asks about reducing the number of variables by substituting some out, as explained in Section 18.2.

(a) Set the option `show_stats` to 1, and solve the problem. From the extra output you get, verify that there are 10 variables.

Next repeat the session with option `substout` set to 1. Verify from the resulting messages that some of the variables are eliminated by substitution. Which of the variables must these be?

(b) Rather than setting `substout`, you can specify that a variable is to be substituted out by placing an `appropriate = phrase` in its `var` declaration. Modify your model from (a) to use this feature, and verify that the results are the same.

(c) There is a long expression for average travel time that appears twice in this model. Define a new variable `Avg` to stand for this expression. Verify that AMPL also substitutes this variable out when you solve the resulting model, and that the result is the same as before.

**18-6.** In *Modeling and Optimization with GINO*, Judith Liebman, Leon Lasdon, Linus Schrage and Allan Waren describe the following problem in designing a steel tank to hold ammonia. The decision variables are



$T$     the temperature inside the tank  
 $I$     the thickness of the insulation

The pressure inside the tank is a function of the temperature,

$$P = e^{-3950/(T+460)+11.86}$$

It is desired to minimize the cost of the tank, which has three components: insulation cost, which depends on the thickness; the cost of the steel vessel, which depends on the pressure; and the cost of a recondensation process for cooling the ammonia, which depends on both temperature and insulation thickness. A combination of engineering and economic considerations has yielded the following formulas:

$$\begin{aligned} C_I &= 400I^{0.9} \\ C_V &= 1000 + 22(P - 14.7)^{1.2} \\ C_R &= 144(80 - T)/I \end{aligned}$$

(a) Formulate this problem in AMPL as a two-variable problem, and alternatively as a six-variable problem in which four of the variables can be substituted out. Which formulation would you prefer to work with?

(b) Using your preferred formulation, determine the parameters of the least-cost tank.

(c) Increasing the factor 144 in  $C_R$  causes a proportional increase in the recondensation cost. Try several larger values, and describe in general terms how the total cost increases as a function of the increase in this factor.

**18-7.** A social accounting matrix is a table that shows the flows from each sector in an economy to each other sector. Here is simple five-sector example, with blank entries indicating flows known to be zero:

	LAB	H1	H2	P1	P2	total
LAB		15	3	130	80	220
H1	?					?
H2	?					?
P1		15	130		20	190
P2		25	40	55		105

If the matrix were estimated perfectly, it would be *balanced*: each row sum (the flow out of a sector) would equal the corresponding column sum (the flow in). As a practical matter, however, there are several difficulties:

~~1.~~ Some entries, marked ? above, have no reliable estimates.

~~2.~~ In the estimated table, the row sums do not necessarily equal the column sums.

~~3.~~ We have separate estimates of the total flows into (or out of) each sector, shown to the right of the rows in our table. These do not necessarily equal the sums of the estimated rows or columns.

Nonlinear programming can be used to adjust this matrix by finding the balanced matrix that is closest, in some sense, to the one given.

For a set  $S$  of sectors, let  $E_T \subseteq S$  be the subset of sectors for which we have estimated total flows, and let  $E_A \subseteq S \times S$  contain all sector pairs  $(i, j)$  for which there are known estimates. The given data values are:

$t_i$  estimated row/column sums,  $i \in E_T$   
 $a_{ij}$  estimated social accounting matrix entries,  $(i, j) \in E_A$

Let  $S_A \subseteq S$ .  $S$  contain all row-column pairs  $(i, j)$  for which there should be entries in the matrix  $\frac{1}{2}$  this includes entries that contain ? instead of a number. We want to determine adjusted entries  $A_{ij}$ , for each  $(i, j) \in S_A$ , that are truly balanced:

$$\sum_{j \in S: (i,j) \in S_A} A_{ij} = \sum_{j \in S: (j,i) \in S_A} A_{ji}$$

for all  $i \in S$ . You can think of these equations as the constraints on the variables  $A_{ij}$ .

There is no best function for measuring ~~the loss~~ but one popular choice is the sum of squared differences between the estimates and the adjusted values ~~for~~ for both the matrix and the row and column sums ~~is~~ scaled by the estimated values. For convenience, we write the adjusted sums as defined variables:

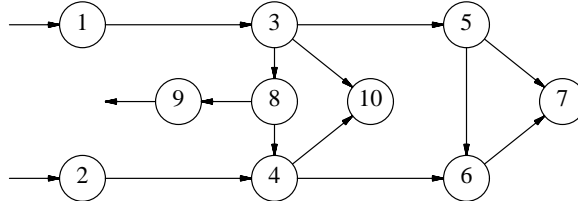
$$T_i = \sum_{j \in S: (i,j) \in S_A} A_{ij}$$

Then the objective is to minimize

$$\sum_{(i,j) \in E_A} (a_{ij} - A_{ij})^2 / a_{ij} + \sum_{i \in E_T} (t_i - T_i)^2 / t_i$$

Formulate an AMPL model for this problem, and determine an optimal adjusted matrix.

**18-8.** A network of pipes has the following layout:



The circles represent joints, and the arrows are pipes. Joints 1 and 2 are sources of flow, and joint 9 is a sink or destination for flow, but flow through a pipe can be in either direction. Associated with each joint  $i$  is an amount  $w_i$  to be withdrawn from the flow at that joint, and an elevation  $e_i$ :

	1	2	3	4	5	6	7	8	9	10
$w_i$	0	0	200	0	0	200	150	0	0	150
$e_i$	50	40	20	20	0	0	0	20	20	20

Our decision variables are the flows  $F_{ij}$  through the pipes from  $i$  to  $j$ , with a positive value representing flow in the direction of the arrow, and a negative value representing flow in the opposite direction. Naturally, flow in must equal flow out plus the amount withdrawn at every joint, except for the sources and the sink.

The ~~head loss~~ of a pipe is a measure of the energy required to move a flow through it. In our situation, the head loss for the pipe from  $i$  to  $j$  is proportional to the square of the flow rate:

$$H_{ij} = Kc_{ij}F_{ij}^2 \text{ if } F_{ij} > 0,$$

$$H_{ij} = Kc_{ij}F_{ij}^2 \text{ if } F_{ij} < 0,$$

where  $K = 4.96407 \cdot 10^{-6}$  is a conversion constant, and  $c_{ij}$  is a factor computed from the diameter, friction, and length of the pipe:

from	to	$c_{ij}$
1	3	6.36685
2	4	28.8937
3	10	28.8937
3	5	6.36685
3	8	43.3406
4	10	28.8937
4	6	28.8937
5	6	57.7874
5	7	43.3406
6	7	28.8937
8	4	28.8937
8	9	705.251

For two joints  $i$  and  $j$  at the same elevation, the pressure drop for flow from  $i$  to  $j$  is equal to the head loss. Both pressure and head loss are measured in feet, so that after correcting for differences in elevation between the joints we have the relation:

$$H_{ij} = (P_i + e_i) - (P_j + e_j)$$

Finally, we wish to maintain the pressure at both the sources and the sink at 200 feet.

(a) Formulate a general AMPL model for this situation, and put together data statements for the data given above.

(b) There is no objective function here, but you can still employ a nonlinear solver to seek a feasible solution. By setting the option `show_stats` to 1, confirm that the number of variables equals the number of equations, so that there are no ~~12~~ degrees of freedom ~~in~~ the solution. (This does not guarantee that there is just one solution, however.)

Check that your solver finds a solution to the equations, and display the results.