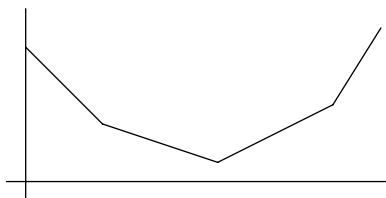# 17

Piecewise-Linear Programs

Several kinds of linear programming problems use functions that are not really linear, but are pieced together from connected linear segments:

These ''piecewise-linear'' terms are easy to imagine, but can be hard to describe in conventional algebraic notation. Hence AMPL provides a special, concise way of writing them.

This chapter introduces AMPL's piecewise-linear notation through examples of piecewise-linear objective functions. In Section 17.1, terms of potentially many pieces are used to describe costs more accurately than a single linear relationship. Section 17.2 shows how terms of two or three pieces can be valuable for such purposes as penalizing deviations from constraints, dealing with infeasibilities, and modeling ''reversible'' activities. Finally, Section 17.3 describes piecewise-linear functions that can be written with other AMPL operators and functions; some are most effectively handled by converting them to the piecewise-linear notation, while others can be accommodated only through more extensive transformations.

Although the piecewise-linear examples in this chapter are all easy to solve, seemingly similar examples can be much more difficult. The last section of this chapter thus offers guidelines for forming and using piecewise-linear terms. We explain how the easy cases can be characterized by the convexity or concavity of the piecewise-linear terms.
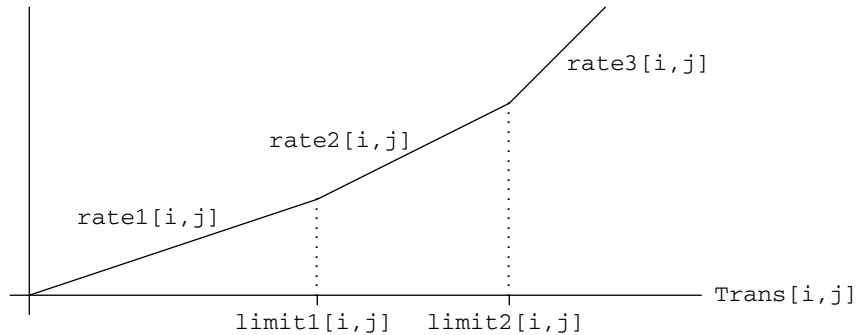
Figure 17-1: Piecewise-linear function, with three slopes.

17.1 Cost terms

Piecewise-linearities are often employed to give a more realistic description of costs than can be achieved by linear terms alone. In this kind of application, piecewise-linear terms serve much the same purpose as nonlinear ones, but without some of the difficulties to be described in Chapter 18.

To make the comparison explicit, we will use the same transportation example as in Chapter 18. We introduce AMPL's notation for piecewise-linear terms with a simple example that has a fixed number of cost levels (and linear pieces) for each shipping link. Then we show how an extension of the notation can use indexing expressions to specify a varying number of pieces controlled through the data.

Fixed numbers of pieces

In a linear transportation model like Figure 3-1a, any number of units can be shipped from a given origin to a given destination at the same cost per unit. More realistically, however, the most favorable rate may be available for only a limited number of units; shipments beyond this limit pay higher rates. As an example, imagine that three cost rate levels are specified for each origin-destination pair. Then the total cost of shipments along a link increases with the amount shipped in a piecewise-linear fashion, with three pieces as shown in Figure 17-1.

To model the three-piece costs, we replace the parameter `cost` of Figure 3-1a by three rates and two limits:

```
param rate1 {i in ORIG, j in DEST} >= 0;
param rate2 {i in ORIG, j in DEST} >= rate1[i,j];
param rate3 {i in ORIG, j in DEST} >= rate2[i,j];

param limit1 {i in ORIG, j in DEST} > 0;
param limit2 {i in ORIG, j in DEST} > limit1[i,j];
```

Shipments from `i` to `j` are charged at `rate1[i,j]` per unit up to `limit1[i,j]` units, then at `rate2[i,j]` per unit up to `limit2[i,j]`, and then at `rate3[i,j]`. Normally `rate2[i,j]` would be greater than `rate1[i,j]` and `rate3[i,j]` would be greater than `rate2[i,j]`, but they may be equal if the link from `i` to `j` does not have three distinct rates.

In the linear transportation model, the objective is expressed in terms of the variables and the parameter `cost` as follows:

```
var Trans {ORIG,DEST} >= 0;

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

We could express a piecewise-linear objective analogously, by introducing three collections of variables, one to represent the amount shipped at each rate:

```
var Trans1 {i in ORIG, j in DEST} >= 0, <= limit1[i,j];
var Trans2 {i in ORIG, j in DEST} >= 0, <= limit2[i,j]
                                          - limit1[i,j];
var Trans3 {i in ORIG, j in DEST} >= 0;

minimize Total_Cost:
   sum {i in ORIG, j in DEST} (rate1[i,j] * Trans1[i,j]
      + rate2[i,j] * Trans2[i,j] + rate3[i,j] * Trans3[i,j]);
```

But then the new variables would have to be introduced into all the constraints, and we would also have to deal with these variables whenever we wanted to display the optimal results. Rather than go to all this trouble, we would much prefer to describe the piecewise-linear cost function explicitly in terms of the original variables. Since there is no standard way to describe piecewise-linear functions in algebraic notation, AMPL supplies its own syntax for this purpose.

The piecewise-linear function depicted in Figure 17-1 is written in AMPL as follows:

```
<<limit1[i,j], limit2[i,j];
  rate1[i,j], rate2[i,j], rate3[i,j]>> Trans[i,j]
```

The expression between `<<` and `>>` describes the piecewise-linear function, and is followed by the name of the variable to which it applies. (You can think of it as ''multiplying'' `Trans[i,j]`, but by a series of coefficients rather than just one.) There are two parts to the expression, a list of *breakpoints* where the slope of the function changes, and a list of the *slopes* — which in this case are the cost rates. The lists are separated by a semicolon, and members of each list are separated by commas. Since the first slope applies to values before the first breakpoint, and the last slope to values after the last breakpoint, the number of slopes must be one more than the number of breakpoints.

Although the lists of breakpoints and slopes are sufficient to describe the piecewise-linear cost function for optimization, they do not quite specify the function uniquely. If we added, say, 10 to the cost at every point, we would have a different cost function even though all the breakpoints and slopes would be the same. To resolve this ambiguity,

```
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 rate1 {i in ORIG, j in DEST} >= 0;
param rate2 {i in ORIG, j in DEST} >= rate1[i,j];
param rate3 {i in ORIG, j in DEST} >= rate2[i,j];

param limit1 {i in ORIG, j in DEST} > 0;
param limit2 {i in ORIG, j in DEST} > limit1[i,j];

var Trans {ORIG,DEST} >= 0;      # units to be shipped

minimize Total_Cost:
   sum {i in ORIG, j in DEST}
      <<limit1[i,j], limit2[i,j];
         rate1[i,j], rate2[i,j], rate3[i,j]>> Trans[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 17-2: Piecewise-linear model with three slopes (`transpl1.mod`).

AMPL assumes that a piecewise-linear function evaluates to zero at zero, as in Figure 17-1. Options for other possibilities are discussed later in this chapter.

Summing the cost over all links, the piecewise-linear objective function is now written

```
minimize Total_Cost:
   sum {i in ORIG, j in DEST}
      <<limit1[i,j], limit2[i,j];
         rate1[i,j], rate2[i,j], rate3[i,j]>> Trans[i,j];
```

The declarations of the variables and constraints stay the same as before; the complete model is shown in Figure 17-2.

Varying numbers of pieces

The approach taken in the preceding example is most useful when there are only a few linear pieces for each term. If there were, for example, 12 pieces instead of three, a model defining `rate1[i,j]` through `rate12[i,j]` and `limit1[i,j]` through `limit11[i,j]` would be unwieldy. Realistically, moreover, there would more likely be up to 12 pieces, rather than exactly 12, for each term; a term with fewer than 12 pieces could be handled by making some rates equal, but for large numbers of pieces this would

be a cumbersome device that would require many unnecessary data values and would obscure the actual number of pieces in each term.

A much better approach is to let the number of pieces (that is, the number of shipping rates) itself be a parameter of the model, indexed over the links:

```
param npiece {ORIG,DEST} integer >= 1;
```

We can then index the rates and limits over all combinations of links and pieces:

```
param rate {i in ORIG, j in DEST, p in 1..npiece[i,j]}
   >= if p = 1 then 0 else rate[i,j,p-1];

param limit {i in ORIG, j in DEST, p in 1..npiece[i,j]-1}
   > if p = 1 then 0 else limit[i,j,p-1];
```

For any particular origin `i` and destination `j`, the number of linear pieces in the cost term is given by `npiece[i,j]`. The slopes are `rate[i,j,p]` for `p` ranging from 1 to `npiece[i,j]`, and the intervening breakpoints are `limit[i,j,p]` for `p` from 1 to `npiece[i,j]-1`. As before, there is one more slope than there are breakpoints.

To use AMPL's piecewise-linear function notation with these data values, we have to give indexed lists of breakpoints and slopes, rather than the explicit lists of the previous example. This is done by placing indexing expressions in front of the slope and break-point values:

```
minimize Total_Cost:
   sum {i in ORIG, j in DEST}
      <<{p in 1..npiece[i,j]-1} limit[i,j,p];
        {p in 1..npiece[i,j]} rate[i,j,p]>> Trans[i,j];
```

Once again, the rest of the model is the same. Figure 17-3a shows the whole model and Figure 17-3b illustrates how the data would be specified. Notice that since `npiece["PITT","STL"]` is 1, `Trans["PITT","STL"]` has only one slope and no breakpoints; this implies a one-piece linear term for `Trans["PITT","STL"]` in the objective function.

17.2 Common two-piece and three-piece terms

Simple piecewise-linear terms have a variety of uses in otherwise linear models. In this section we present three cases: allowing limited violations of the constraints, analyzing infeasibility, and representing costs for variables that are meaningful at negative as well as positive levels.

Penalty terms for ''soft'' constraints

Linear programs most easily express ''hard'' constraints: that production must be at least at a certain level, for example, or that resources used must not exceed those available. Real situations are often not nearly so definite. Production and resource use may

```
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 npiece {ORIG,DEST} integer >= 1;

param rate {i in ORIG, j in DEST, p in 1..npiece[i,j]}
  >= if p = 1 then 0 else rate[i,j,p-1];

param limit {i in ORIG, j in DEST, p in 1..npiece[i,j]-1}
  > if p = 1 then 0 else limit[i,j,p-1];

var Trans {ORIG,DEST} >= 0;      # units to be shipped

minimize Total_Cost:
   sum {i in ORIG, j in DEST}
      <<{p in 1..npiece[i,j]-1} limit[i,j,p];
        {p in 1..npiece[i,j]} rate[i,j,p]>> Trans[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 17-3a: Piecewise-linear model with indexed slopes (`transp12.mod`).

have certain preferred levels, yet we may be allowed to violate these levels by accepting some extra costs or reduced profits. The resulting ''soft'' constraints can be modeled by adding piecewise-linear ''penalty'' terms to the objective function.

For an example, we return to the multi-week production model developed in Chapter 4. As seen in Figure 4-4, the constraints say that, in each of weeks 1 through T, total hours used to make all products may not exceed hours available:

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

Suppose that, in reality, a larger number of hours may be used in each week, but at some penalty per hour to the total profit. Specifically, we replace the parameter `avail[t]` by two availability levels and an hourly penalty rate:

```
        param avail_min {1..T} >= 0;
        param avail_max {t in 1..T} >= avail_min[t];

        param time_penalty {1..T} > 0;
```

Up to `avail_min[t]` hours are available without penalty in week `t`, and up to `avail_max[t]` hours are available at a loss of `time_penalty[t]` in profit for each hour above `avail_min[t]`.

To model this situation, we introduce a new variable `Use[t]` to represent the hours used by production. Clearly `Use[t]` may not be less than zero, or greater than

```
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 npiece:  FRA DET LAN WIN STL FRE LAF :=
        GARY    3   3   3   2   3   2   3
        CLEV    3   3   3   3   3   3   3
        PITT    2   2   2   2   1   2   1 ;

param rate :=
  [GARY,FRA,*] 1 39   2 50   3 70    [GARY,DET,*] 1 14   2  17   3  33
  [GARY,LAN,*] 1 11   2 12   3 23    [GARY,WIN,*] 1 14   2  17
  [GARY,STL,*] 1 16   2 23   3 40    [GARY,FRE,*] 1 82   2  98
  [GARY,LAF,*] 1  8   2 16   3 24

  [CLEV,FRA,*] 1 27   2 37   3 47    [CLEV,DET,*] 1  9   2  19   3  24
  [CLEV,LAN,*] 1 12   2 32   3 39    [CLEV,WIN,*] 1  9   2  14   3  21
  [CLEV,STL,*] 1 26   2 36   3 47    [CLEV,FRE,*] 1 95   2 105   3 129
  [CLEV,LAF,*] 1  8   2 16   3 24

  [PITT,FRA,*] 1 24   2 34           [PITT,DET,*] 1 14   2  24
  [PITT,LAN,*] 1 17   2 27           [PITT,WIN,*] 1 13   2  23
  [PITT,STL,*] 1 28                  [PITT,FRE,*] 1 99   2 140
  [PITT,LAF,*] 1 20 ;

param limit :=
  [GARY,*,*] FRA 1  500   FRA 2 1000   DET 1  500   DET 2 1000
             LAN 1  500   LAN 2 1000   WIN 1 1000
             STL 1  500   STL 2 1000   FRE 1 1000
             LAF 1  500   LAF 2 1000

  [CLEV,*,*] FRA 1  500   FRA 2 1000   DET 1  500   DET 2 1000
             LAN 1  500   LAN 2 1000   WIN 1  500   WIN 2 1000
             STL 1  500   STL 2 1000   FRE 1  500   FRE 2 1000
             LAF 1  500   LAF 2 1000

  [PITT,*,*] FRA 1 1000   DET 1 1000   LAN 1 1000   WIN 1 1000
             FRE 1 1000 ;
```

Figure 17-3b: Data for piecewise-linear model (`transpl2.dat`).

`avail_max[t]`. In place of our previous constraint, we say that the total hours used to make all products must equal `Use[t]`:
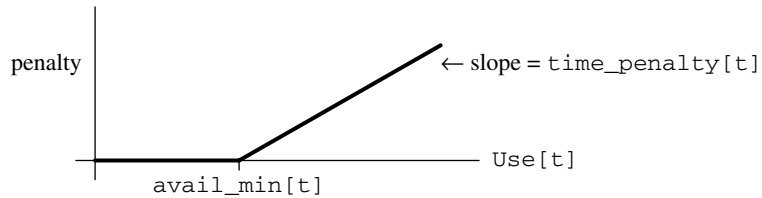
```
    var Use {t in 1..T} >= 0, <= avail_max[t];

    subject to Time {t in 1..T}:
        sum {p in PROD} (1/rate[p]) * Make[p,t] = Use[t];
```

We can now describe the hourly penalty in terms of this new variable. If `Use[t]` is between 0 and `avail_min[t]`, there is no penalty; if `Use[t]` is between `avail_min[t]` and `avail_max[t]`, the penalty is `time_penalty[t]` per hour

Figure 17-4: Piecewise-linear penalty function for hours used.

that it exceeds `avail_min[t]`. That is, the penalty is a piecewise-linear function of `Use[t]` as shown in Figure 17-4, with slopes of 0 and `time_penalty[t]` surrounding a breakpoint at `avail_min[t]`. Using the syntax previously introduced, we can rewrite the expression for the objective function as:

```
maximize Net_Profit:
    sum {p in PROD, t in 1..T} (revenue[p,t]*Sell[p,t] -
        prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t])
 - sum {t in 1..T} <<avail_min[t]; 0,time_penalty[t]>> Use[t];
```

The first summation is the same expression for total profit as before, while the second is the sum of the piecewise-linear penalty functions over all weeks. Between << and >> are the breakpoint `avail_min[t]` and a list of the surrounding slopes, 0 and `time_penalty[t]`; this is followed by the argument `Use[t]`.

   The complete revised model is shown in Figure 17-5a, and our small data set from Chapter 4 is expanded with the new availabilities and penalties in Figure 17-5b. In the optimal solution, we find that the hours used are as follows:

```
ampl: model steelpl1.mod; data steelpl1.dat; solve;
MINOS 5.5: optimal solution found.
21 iterations, objective 457572.8571

ampl: display avail_min,Use,avail_max;
: avail_min  Use avail_max    :=
1     35      35      42
2     35      42      42
3     30      30      40
4     35      42      42
;
```

In weeks 1 and 3 we use only the unpenalized hours available, while in weeks 2 and 4 we also use the penalized hours. Solutions to piecewise-linear programs usually display this sort of solution, in which many (though not necessarily all) of the variables ''stick'' at one of the breakpoints.

```
set PROD;      # products
param T > 0;   # number of weeks

param rate {PROD} > 0;             # tons per hour produced
param inv0 {PROD} >= 0;            # initial inventory
param commit {PROD,1..T} >= 0;  # minimum tons sold in week
param market {PROD,1..T} >= 0;  # limit on tons sold in week

param avail_min {1..T} >= 0;      # unpenalized hours available
param avail_max {t in 1..T} >= avail_min[t]; # total hours avail
param time_penalty {1..T} > 0;

param prodcost {PROD} >= 0;       # cost/ton produced
param invcost {PROD} >= 0;        # carrying cost/ton of inventory
param revenue {PROD,1..T} >= 0; # revenue/ton sold

var Make {PROD,1..T} >= 0;          # tons produced
var Inv {PROD,0..T} >= 0;           # tons inventoried
var Sell {p in PROD, t in 1..T}
   >= commit[p,t], <= market[p,t];  # tons sold

var Use {t in 1..T} >= 0, <= avail_max[t];  # hours used

maximize Total_Profit:
   sum {p in PROD, t in 1..T} (revenue[p,t]*Sell[p,t] -
      prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t])
 - sum {t in 1..T} <<avail_min[t]; 0,time_penalty[t]>> Use[t];

                 # Objective: total revenue less costs in all weeks

subject to Time {t in 1..T}:
   sum {p in PROD} (1/rate[p]) * Make[p,t] = Use[t];

                 # Total of hours used by all products
                 # may not exceed hours available, in each week

subject to Init_Inv {p in PROD}:  Inv[p,0] = inv0[p];

                 # Initial inventory must equal given value

subject to Balance {p in PROD, t in 1..T}:
   Make[p,t] + Inv[p,t-1] = Sell[p,t] + Inv[p,t];

                 # Tons produced and taken from inventory
                 # must equal tons sold and put into inventory
```

Figure 17-5a: Piecewise-linear objective with penalty function (`steelpl1.mod`).

Dealing with infeasibility

The parameters `commit[p,t]` in Figure 17-5b represent the minimum production amounts for each product in each week. If we change the data to raise these commitments:

```
        param commit:     1     2     3     4 :=
              bands   3500  5900  3900  6400
              coils   2500  2400  3400  4100 ;
```

```
param T := 4;
set PROD := bands coils;

param:    rate  inv0  prodcost  invcost :=
  bands   200   10      10        2.5
  coils   140    0      11        3 ;

param: avail_min  avail_max  time_penalty :=
  1         35         42          3100
  2         35         42          3000
  3         30         40          3700
  4         35         42          3100 ;

param revenue:     1     2     3     4 :=
        bands     25    26    27    27
        coils     30    35    37    39 ;

param commit:      1     2     3     4 :=
        bands   3000  3000  3000  3000
        coils   2000  2000  2000  2000 ;

param market:      1     2     3     4 :=
        bands   6000  6000  4000  6500
        coils   4000  2500  3500  4200 ;
```

Figure 17-5b: Data for Figure 17-5a (steelpl1.dat).

then there are not enough hours to produce even these minimum amounts, and the solver reports that the problem is infeasible:

```
ampl: model steelpl1.mod;
ampl: data steelpl2.dat;

ampl: solve;
MINOS 5.5: infeasible problem.
13 iterations
```
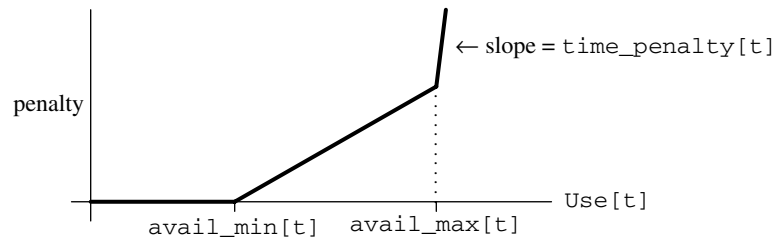
In the solution that is returned, the inventory of coils in the last period is negative:

```
ampl: option display_1col 0;
ampl: display Inv;
Inv [*,*] (tr)
: bands   coils     :=
0   10          0
1    0        937
2    0        287
3    0          0
4    0      -2700
;
```

and production of coils in several periods is below the minimum required:

← slope = `time_penalty[t]`

Figure 17-6: Penalty function for hours used, with two breakpoints.

```
ampl: display commit,Make,market;
:          commit   Make market    :=
bands 1     3500    3490   6000
bands 2     5900    5900   6000
bands 3     3900    3900   4000
bands 4     6400    6400   6500
coils 1     2500    3437   4000
coils 2     2400    1750   2500
coils 3     3400    2870   3500
coils 4     4100    1400   4200
;
```

These are typical of the infeasible results that solvers return. The infeasibilities are scattered around the solution, so that it is hard to tell what changes might be necessary to achieve feasibility. By extending the idea of penalties, we can better concentrate the infeasibility where it can be understood.

Suppose that we want to view the infeasibility in terms of a shortage of hours. Imagine that we extend the piecewise-linear penalty function of Figure 17-4 to the one shown in Figure 17-6. Now `Use[t]` is allowed to increase past `avail_max[t]`, but only with an extremely steep penalty per hour — so that the solution will use hours above `avail_max[t]` only to the extent absolutely necessary.

In AMPL, the new penalty function is introduced through the following changes:
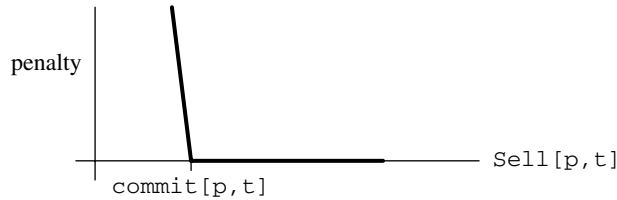
```
var Use {t in 1..T} >= 0;

maximize Total_Profit:
   sum {p in PROD, t in 1..T} (revenue[p,t]*Sell[p,t] -
      prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t])
 - sum {t in 1..T} <<avail_min[t],avail_max[t];
                   0,time_penalty[t],100000>> Use[t];
```

The former bound `avail_max[t]` has become a breakpoint, and to its right a very large slope of 100,000 has been introduced. Now we get a feasible solution, which uses hours as follows:

Figure 17-7: Penalty function for sales.

```
ampl: model steelpl2.mod; data steelpl2.dat; solve;
MINOS 5.5: optimal solution found.
19 iterations, objective -1576814.857

ampl: display avail_max,Use;
: avail_max  Use      :=
1     42        42
2     42        42
3     40        41.7357
4     42        61.2857
;
```

This table implies that the commitments can be met only by adding about 21 hours, mostly in the last week.

Alternatively, we may view the infeasibility in terms of an excess of commitments. For this purpose we subtract a very large penalty from the objective for each unit that Sell[p,t] falls below commit[p,t]; the penalty as a function of Sell[p,t] is depicted in Figure 17-7.

Since this function has a breakpoint at commit[p,t], with a slope of 0 to the right and a very negative value to the left, it would seem that the AMPL representation could be

```
<<commit[p,t]; -100000,0>> Sell[p,t]
```

Recall, however, AMPL's convention that such a function takes the value zero at zero. Figure 17-7 clearly shows that we want our penalty function to take a positive value at zero, so that it will fall to zero at commit[p,t] and beyond. In fact we want the function to take a value of 100000 * commit[p,t] at zero, and we could express the function properly by adding this constant to the penalty expression:

```
<<commit[p,t]; -100000,0>> Sell[p,t] + 100000*commit[p,t]
```

The same thing may be said more concisely by using a second argument that states explicitly where the piecewise-linear function should evaluate to zero:

```
<<commit[p,t]; -100000,0>> (Sell[p,t],commit[p,t])
```

This says that the function should be zero at commit[p,t], as Figure 17-7 shows. In the completed model, we have:

```
    var Sell {p in PROD, t in 1..T} >= 0, <= market[p,t];

    maximize Total_Profit:
       sum {p in PROD, t in 1..T} (revenue[p,t]*Sell[p,t] -
          prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t])
     - sum {t in 1..T} <<avail_min[t]; 0,time_penalty[t]>> Use[t]
     - sum {p in PROD, t in 1..T}
          <<commit[p,t]; -100000,0>> (Sell[p,t],commit[p,t]);
```

The rest of the model is the same as in Figure 17-5a. Notice that `Sell[p,t]` appears in both a linear and a piecewise-linear term within the objective function; AMPL automatically recognizes that the sum of these terms is also piecewise-linear.

This version, using the same data, produces a solution in which the amounts sold are as follows:

```
ampl: model steelpl3.mod; data steelpl2.dat; solve;
MINOS 5.5: optimal solution found.
24 iterations, objective -293856347

ampl: display Sell,commit;
:         Sell commit     :=
bands 1   3500   3500
bands 2   5900   5900
bands 3   3900   3900
bands 4   6400   6400
coils 1      0   2500
coils 2   2400   2400
coils 3   3400   3400
coils 4   3657   4100
;
```

To get by with the given number of hours, commitments to deliver coils are cut by 2500 tons in the first week and 443 tons in the fourth week.

Reversible activities

Almost all of the linear programs in this book are formulated in terms of nonnegative variables. Sometimes a variable makes sense at negative as well as positive values, however, and in many such cases the associated cost is piecewise-linear with a breakpoint at zero.

One example is provided by the inventory variables in Figure 17-5a. We have defined `Inv[p,t]` to represent the tons of product `p` inventoried at the end of week `t`. That is, after week `t` there are `Inv[p,t]` tons of product `p` that have been made but not sold. A negative value of `Inv[p,t]` could thus reasonably be interpreted as representing tons of product `p` that have been sold but not made — tons backordered, in effect. The material balance constraints,

```
    subject to Balance {p in PROD, t in 1..T}:
       Make[p,t] + Inv[p,t-1] = Sell[p,t] + Inv[p,t];
```

remain valid under this interpretation.

This analysis suggests that we remove the `>= 0` from the declaration of `Inv` in our model. Then backordering might be especially attractive if the sales price were expected to drop in later weeks, like this:

```
param revenue:    1     2     3     4 :=
        bands    25    26    23    20
        coils    30    35    31    25 ;
```

When we re-solve with appropriately revised model and data files, however, the results are not what we expect:

```
ampl: model steelpl4.mod; data steelpl4.dat; solve;
MINOS 5.5: optimal solution found.
15 iterations, objective 1194250

ampl: display Make,Inv,Sell;
:         Make     Inv      Sell     :=
bands 0     .        10        .
bands 1    0      -5990     6000
bands 2    0     -11990     6000
bands 3    0     -15990     4000
bands 4    0     -22490     6500
coils 0     .         0        .
coils 1    0      -4000     4000
coils 2    0      -6500     2500
coils 3    0     -10000     3500
coils 4    0     -14200     4200
;
```

The source of difficulty is in the objective function, where `invcost[p] * Inv[p,t]` is subtracted from the sales revenue. When `Inv[p,t]` is negative, a negative amount is subtracted, increasing the apparent total profit. The greater the amount backordered, the more the total profit is increased — hence the odd solution in which the maximum possible sales are backordered, while nothing is produced!

A proper inventory cost function for this model looks like the one graphed in Figure 17-8. It increases both as `Inv[p,t]` becomes more positive (greater inventories) and as `Inv[p,t]` becomes more negative (greater backorders). We represent this piecewise-linear function in AMPL by declaring a backorder cost to go with the inventory cost:
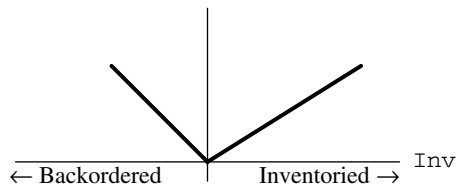
Figure 17-8: Inventory cost function.

```
param invcost {PROD} >= 0;
param backcost {PROD} >= 0;
```

Then the slopes for the `Inv[p,t]` term in the objective are `-backcost[p]` and `invcost[p]`, with the breakpoint at zero, and the correct objective function is:

```
maximize Total_Profit:
   sum {p in PROD, t in 1..T}
      (revenue[p,t]*Sell[p,t] - prodcost[p]*Make[p,t]
       - <<0; -backcost[p],invcost[p]>> Inv[p,t])
 - sum {t in 1..T} <<avail_min[t]; 0,time_penalty[t]>> Use[t];
```

In contrast to our first example, the piecewise-linear function is subtracted rather than added. The result is still piecewise-linear, though; it's the same as if we had added the expression `<<0; backcost[p], -invcost[p]>> Inv[p,t]`.

When we make this change, and add some backorder costs to the data, we get a more reasonable-looking solution. Nevertheless, there remains a tendency to make nothing and backorder everything in the later periods; this is an ''end effect'' that occurs because the model does not account for the eventual production cost of items backordered past the last period. As a quick fix, we can rule out any remaining backorders at the end, by adding a constraint that final-week inventory must be nonnegative:

```
subject to Final {p in PROD}:  Inv[p,T] >= 0;
```

Solving with this constraint, and with `backcost` values of 1.5 for band and 2 for coils:

```
ampl: model steelpl5.mod; data steelpl5.dat; solve;
MINOS 5.5: optimal solution found.
20 iterations, objective 370752.8571

ampl: display Make,Inv,Sell;
:           Make      Inv     Sell       :=
bands 0       .        10        .
bands 1    4142.86      0    4152.86
bands 2    6000         0    6000
bands 3    3000         0    3000
bands 4    3000         0    3000
coils 0       .         0        .
coils 1    2000         0    2000
coils 2    1680      -820    2500
coils 3    2100      -800    2080
coils 4    2800         0    2000
;
```

About 800 tons of coils for weeks 2 and 3 will be delivered a week late under this plan.

17.3 Other piecewise-linear functions

Many simple piecewise-linear functions can be modeled in several equivalent ways in AMPL. The function of Figure 17-4, for example, could be written as

```
if Use[t] > avail_min[t]
   then time_penalty[t] * (Use[t] - avail_min[t]) else 0
```

or more concisely as

```
max(0, time_penalty[t] * (Use[t] - avail_min[t]))
```

The current version of AMPL does not detect that these expressions are piecewise-linear, so you are unlikely to get satisfactory results if you try to solve a model that has expressions like these in its objective. To take advantage of linear programming techniques that can be applied for piecewise-linear terms, you need to use the piecewise-linear terminology

```
<<avail_min[t]; 0,time_penalty[t]>> Use[t]
```

so the structure can be noted and passed to a solver.

The same advice applies to the function abs. Imagine that we would like to encourage the number of hours used to be close to avail_min[t]. Then we would want the penalty term to equal time_penalty[t] times the amount that Use[t] deviates from avail_min[t], either above or below. Such a term can be written as

```
time_penalty[t] * abs(Use[t] - avail_min[t])
```

To express it in an explicitly piecewise-linear fashion, however, you should write it as

```
time_penalty[t] * <<avail_min[t]; -1,1>> Use[t]
```

or equivalently,

```
<<avail_min[t]; -time_penalty[t],time_penalty[t]>> Use[t]
```

As this example shows, multiplying a piecewise-linear function by a constant is the same as multiplying each of its slopes individually.

As a final example of a common piecewise-linearity in the objective, we return to the kind of assignment model that was discussed in Chapter 15. Recall that, for i in the set PEOPLE and j in the set PROJECTS, cost[i,j] is the cost for person i to work an hour on project j, and the decision variable Assign[i,j] is the number of hours that person i is assigned to work on project j:

```
set PEOPLE;
set PROJECTS;

param cost {PEOPLE,PROJECTS} >= 0;
var Assign {PEOPLE,PROJECTS} >= 0;
```

We originally formulated the objective as the total cost of all assignments,

```
sum {i in PEOPLE, j in PROJECTS} cost[i,j] * Assign[i,j]
```

What if we want the fairest assignment instead of the cheapest? Then we might minimize the maximum cost of any one person's assignments:

```
set PEOPLE;
set PROJECTS;

param supply {PEOPLE} >= 0;    # hours each person is available
param demand {PROJECTS} >= 0; # hours each project requires

   check: sum {i in PEOPLE} supply[i]
           = sum {j in PROJECTS} demand[j];

param cost {PEOPLE,PROJECTS} >= 0;    # cost per hour of work
param limit {PEOPLE,PROJECTS} >= 0;   # maximum contributions
                                      # to projects

var M;
var Assign {i in PEOPLE, j in PROJECTS} >= 0, <= limit[i,j];

minimize Max_Cost: M;

subject to M_def {i in PEOPLE}:
   M >= sum {j in PROJECTS} cost[i,j] * Assign[i,j];

subject to Supply {i in PEOPLE}:
   sum {j in PROJECTS} Assign[i,j] = supply[i];

subject to Demand {j in PROJECTS}:
   sum {i in PEOPLE} Assign[i,j] = demand[j];
```

Figure 17-9: Min-max assignment model (`minmax.mod`).

```
minimize Max_Cost:
   max {i in PEOPLE}
      sum {j in PROJECTS} cost[i,j] * Assign[i,j];
```

This function is also piecewise-linear, in a sense; it is pieced together from the linear functions `sum {j in PROJECTS} cost[i,j] * Assign[i,j]` for different people `i`. However, it is not piecewise-linear in the individual variables — in mathematical jargon, it is not separable — and hence it cannot be written using the << . . . >> notation.

This is a case in which piecewise-linearity can only be handled by rewriting the model as a linear program. We introduce a new variable `M` to represent the maximum. Then we write constraints to guarantee that `M` is greater than or equal to each cost of which it is the maximum:

```
var M;
minimize Max_Cost: M;

subject to M_def {i in PEOPLE}:
   M >= sum {j in PROJECTS} cost[i,j] * Assign[i,j];
```

Because `M` is being minimized, at the optimal solution it will in fact equal the maximum of `sum {j in PROJECTS} cost[i,j] * Assign[i,j]` over all `i` in `PEOPLE`. The other constraints are the same as in any assignment problem, as shown in Figure 17-9.

This kind of reformulation can be applied to any problem that has a ''min-max'' objective. The same idea works for the analogous ''max-min'' objective, with `maximize` instead of `minimize` and with `M <= ...` in the constraints.

17.4 Guidelines for piecewise-linear optimization

AMPL's piecewise-linear notation has the power to specify a variety of useful functions. We summarize below its various forms, most of which have been illustrated earlier in this chapter.

Because this notation is so general, it can be used to specify many functions that are not readily optimized by any efficient and reliable algorithms. We conclude by describing the kinds of piecewise-linear functions that are most likely to appear in tractable models, with particular emphasis on the property of convexity or concavity.

Forms for piecewise-linear expressions

An AMPL piecewise-linear term has the general form

> *<<breakpoint-list; slope-list>> pl-argument*

where *breakpoint-list* and *slope-list* each consist of a comma-separated list of one or more items. An item may be an individual arithmetic expression, or an indexing expression followed by an arithmetic expression. In the latter case, the indexing expression must be an ordered set; the item is expanded to a list by evaluating the arithmetic expression once for each set member (as in the example of Figure 17-3a).

After any indexed items are expanded, the number of slopes must be one more than the number of breakpoints, and the breakpoints must be nondecreasing. The resulting piecewise-linear function is constructed by interleaving the slopes and breakpoints in the order given, with the first slope to the left of the first breakpoint, and the last slope to the right of the last breakpoint. By indexing breakpoints over an empty set, it is possible to specify no breakpoints and one slope, in which case the function is linear.

The *pl-argument* may have one of the forms

> *var-ref*
> (*arg-expr*)
> (*arg-expr, zero-expr*)

The *var-ref* (a reference to a previously declared variable) or the *arg-expr* (an arithmetic expression) specifies the point where the piecewise-linear function is to be evaluated. The *zero-expr* is an arithmetic expression that specifies a place where the function is zero; when the *zero-expr* is omitted, the function is assumed to be zero at zero.

Suggestions for piecewise-linear models

As seen in all of our examples, AMPL's terminology for piecewise-linear functions of variables is limited to describing functions of individual variables. In model declarations, no variables may appear in the *breakpoint-list*, *slope-list* and *zero-expr* (if any), while an *arg-expr* can only be a reference to an individual variable. (Piecewise-linear expressions in commands like display may use variables without limitation, however.)

A piecewise-linear function of an individual variable remains such a function when multiplied or divided by an arithmetic expression without variables. AMPL also treats a sum or difference of piecewise-linear and linear functions of the same variable as representing one piecewise-linear function of that variable. A *separable* piecewise-linear function of a model's variables is a sum or difference (using +, − or sum) of piecewise-linear or linear functions of the individual variables. Optimizers can effectively handle these separable functions, which are the ones that appear in our examples.

A piecewise-linear function is convex if successive slopes are nondecreasing (along with the breakpoints), and is concave if the slopes are nonincreasing. The two kinds of piecewise-linear optimization most easily handled by solvers are minimizing a separable convex piecewise-linear function, and maximizing a separable concave piecewise-linear function, subject to linear constraints. You can easily check that all of this chapter's examples are of these kinds. AMPL can obtain solutions in these cases by translating to an equivalent linear program, applying any LP solver, and then translating the solution back; the whole sequence occurs automatically when you type solve.

Outside of these two cases, optimizing a separable piecewise-linear function must be viewed as an application of integer programming — the topic of Chapter 20 — and AMPL must translate piecewise-linear terms to equivalent integer programming forms. This, too, is done automatically, for solution by an appropriate solver. Because integer programs are usually much harder to solve than similar linear programs of comparable size, however, you should not assume that just any separable piecewise-linear function can be readily optimized; a degree of experimentation may be necessary to determine how large an instance your solver can handle. The best results are likely to be obtained by solvers that accept an option known (mysteriously) as ''special ordered sets of type 2''; check the solver-specific documentation for details.

The situation for the constraints can be described in a similar way. However, a separable piecewise-linear function in a constraint can be handled through linear programming only under a restrictive set of circumstances:

- If it is convex and on the left-hand side of a ≤ constraint (or equivalently, the right-hand side of a ≥ constraint);
- If it is concave and on the left-hand side of a ≥ constraint (or equivalently, the right-hand side of a ≤ constraint).

Other piecewise-linearities in the constraints must be dealt with through integer programming techniques, and the preceding comments for the case of the objective apply.

If you have access to a solver that can handle piecewise-linearities directly, you can turn off AMPL's translation to the linear or integer programming form by setting the

option `pl_linearize` to 0. The case of minimizing a convex or maximizing a concave separable piecewise-linear function can in particular be handled very efficiently by piecewise-linear generalizations of LP techniques. A solver intended for nonlinear programming may also accept piecewise-linear functions, but it is unlikely to handle them reliably unless it has been specially designed for ''nondifferentiable'' optimization.

The differences between hard and easy piecewise-linear cases can be slight. This chapter's transportation example is easy, in particular because the shipping rates increase along with shipping volume. The same example would be hard if economies of scale caused shipping rates to decrease with volume, since then we would be minimizing a concave rather than a convex function. We cannot say definitively that shipping rates ought to be one way or the other; their behavior depends upon the specifics of the situation being modeled.

In all cases, the difficulty of piecewise-linear optimization gradually increases with the total number of pieces. Thus piecewise-linear cost functions are most effective when the costs can be described or approximated by relatively few pieces. If you need more than about a dozen pieces to describe a cost accurately, you may be better off using a nonlinear function as described in Chapter 18.

Bibliography

Robert Fourer, ''A Simplex Algorithm for Piecewise-Linear Programming III: Computational Analysis and Applications.'' Mathematical Programming **53** (1992) pp. 213–235. A survey of conversions from piecewise-linear to linear programs, and of applications.

Robert Fourer and Roy E. Marsten, ''Solving Piecewise-Linear Programs: Experiments with a Simplex Approach.'' ORSA Journal on Computing **4** (1992) pp. 16–31. Descriptions of varied applications and of experience in solving them.

Spyros Kontogiorgis, ''Practical Piecewise-Linear Approximation for Monotropic Optimization.'' INFORMS Journal on Computing **12** (2000) pp. 324–340. Guidelines for choosing the breakpoints when approximating a nonlinear function by a piecewise-linear one.

Exercises

17-1. Piecewise-linear models are sometimes an alternative to the nonlinear models described in Chapter 18, replacing a smooth curve by a series of straight-line segments. This exercise deals with the model shown in Figure 18-4.

(a) Reformulate the model of Figure 18-4 so that it approximates each nonlinear term

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

by a piecewise-linear term having three pieces. Set the breakpoints at `(1/3) * limit[i,j]` and `(2/3) * limit[i,j]`. Pick the slopes so that the approximation equals the original nonlinear term when `Trans[i,j]` is 0, `1/3 * limit[i,j]`, `2/3 * limit[i,j]`, or `11/12 * limit[i,j]`; you should find that the three slopes are 3/2, 9/2 and 36 in every term, regardless

of the size of `limit[i,j]`. Finally, place an explicit upper limit of `0.99 * limit[i,j]` on `Trans[i,j]`.

(b) Solve the approximation with the data given in Figure 18-5, and compare the optimal shipment amounts to the amounts recommended by the nonlinear model.

(c) Formulate a more sophisticated version in which the number of linear pieces for each term is given by a parameter `nsl`. Pick the breakpoints to be at `(k/nsl) * limit[i,j]` for k from 1 to `nsl-1`. Pick the slopes so that the piecewise-linear function equals the original nonlinear function when `Trans[i,j]` is `(k/nsl) * limit[i,j]` for any k from 0 to `nsl-1`, or when `Trans[i,j]` is `(nsl-1/4)/nsl * limit[i,j]`.

Check your model by showing that you get the same results as in (b) when `nsl` is 3. Then, by trying higher values of `nsl`, determine how many linear pieces the approximation requires in order to determine all shipment amounts to within about 10% of the amounts recommended by the original nonlinear model.

17-2. This exercise asks how you might convert the demand constraints in the transportation model of Figure 3-1a into the kind of ''soft'' constraints described in Section 17.2.

Suppose that instead of a single parameter called `demand[j]` at each destination `j`, you are given the following four parameters that describe a more complicated situation:

`dem_min_abs[j]`	absolute minimum that must be shipped to `j`
`dem_min_ask[j]`	preferred minimum amount shipped to `j`
`dem_max_ask[j]`	preferred maximum amount shipped to `j`
`dem_max_abs[j]`	absolute maximum that may be shipped to `j`

There are also two penalty costs for shipment amounts outside of the preferred limits:

`dem_min_pen`	penalty per unit that shipments fall below `dem_min_ask[j]`
`dem_max_pen`	penalty per unit that shipments exceed `dem_max_ask[j]`

Because the total shipped to `j` is no longer fixed, a new variable `Receive[j]` is introduced to represent the amount received at `j`.

(a) Modify the model of Figure 3-1a to use this new information. The modifications will involve declaring `Receive[j]` with the appropriate lower and upper bounds, adding a three-piece piecewise-linear penalty term to the objective function, and substituting `Receive[j]` for `demand[j]` in the constraints.

(b) Add the following demand information to the data of Figure 3-1b:

	dem_min_abs	dem_min_ask	dem_max_ask	dem_max_abs
FRA	800	850	950	1100
DET	975	1100	1225	1250
LAN	600	600	625	625
WIN	350	375	450	500
STL	1200	1500	1800	2000
FRE	1100	1100	1100	1125
LAF	800	900	1050	1175

Let `dem_min_pen` and `dem_max_pen` be 2 and 4, respectively. Find the new optimal solution. In the solution, which destinations receive shipments that are outside the preferred levels?

17-3. When the diet model of Figure 2-1 is run with the data of Figure 2-3, there is no feasible solution. This exercise asks you to use the ideas of Section 17.2 to find some good near-feasible solutions.

(a) Modify the model so that it is possible, at a very high penalty, to purchase more than the specified maximum of a food. In the resulting solution, which maximums are exceeded?

(b) Modify the model so that it is possible, at a very high penalty, to supply more than the specified maximum of a nutrient. In the resulting solution, which maximums are exceeded?

(c) Using extremely large penalties, such as 10^{20} may give the solver numerical difficulties. Experiment to see how available solvers behave when you use penalty terms like 10^{20} and 10^{30}.

17-4. In the model of Exercise 4-4(b), the change in crews from one period to the next is limited to some number M. As an alternative to imposing this limit, suppose that we introduce a new variable D_t that represents the change in number of crews (in all shifts) at period t. This variable may be positive, indicating an increase in crews over the previous period, or negative, indicating a decrease in crews.

To make use of this variable, we introduce a defining constraint,

$$D_t = \sum_{s \in S} (Y_{st} - Y_{s,t-1}),$$

for each $t = 1, \ldots, T$. We then estimate costs of c^+ per crew added from period to period, and c^- per crew dropped from period to period; as a result, the following cost must be included in the objective for each month t:

$$c^- D_t, \qquad \text{if } D_t < 0;$$
$$c^+ D_t, \qquad \text{if } D_t > 0.$$

Reformulate the model in AMPL accordingly, using a piecewise-linear function to represent this extra cost.

Solve using $c^- = -20000$ and $c^+ = 100000$, together with the data previously given. How does this solution compare to the one from Exercise 4-4(b)?

17-5. The following ''credit scoring'' problem appears in many contexts, including the processing of credit card applications. A set `APPL` of people apply for a card, each answering a set `QUES` of questions on the application. The response of person `i` to question `j` is converted to a number, `ans[i,j]`; typical numbers are years at current address, monthly income, and a home ownership indicator (say, 1 if a home is owned and 0 otherwise).

To summarize the responses, the card issuer chooses weights `Wt[j]`, from which a score for each person `i` in `APPL` is computed by the linear formula

```
sum {j in QUES} ans[i,j] * Wt[j]
```

The issuer also chooses a cutoff, `Cut`; credit is granted when an applicant's score is greater than or equal to the cutoff, and is denied when the score is less than the cutoff. In this way the decision can be made objectively (if not always most wisely).

To choose the weights and the cutoff, the card issuer collects a sample of previously accepted applications, some from people who turned out to be good customers, and some from people who never paid their bills. If we denote these two collections of people by sets `GOOD` and `BAD`, then the ideal weights and cutoff (for this data) would satisfy

```
sum {j in QUES} ans[i,j] * Wt[j] >= Cut   for each i in GOOD
sum {j in QUES} ans[i,j] * Wt[j] <  Cut   for each i in BAD
```

Since the relationship between answers to an application and creditworthiness is imprecise at best, however, no values of `Wt[j]` and `Cut` can be found to satisfy all of these inequalities. Instead,

the issuer has to choose values that are merely the best possible, in some sense. There are any number of ways to make such a choice; here, naturally, we consider an optimization approach.

(a) Suppose that we define a new variable `Diff[i]` that equals the difference between person `i`'s score and the cutoff:

```
Diff[i] = sum {j in QUES} ans[i,j] * Wt[j] - Cut
```

Clearly the undesirable cases are where `Diff[i]` is negative for `i` in GOOD, and where it is non-negative for `i` in BAD. To discourage these cases, we can tell the issuer to minimize the function

```
sum {i in GOOD} max(0,-Diff[i]) + sum {i in BAD} max(0,Diff[i])
```

Explain why minimizing this function tends to produce a desirable choice of weights and cutoff.

(b) The expression above is a piecewise-linear function of the variables `Diff[i]`. Rewrite it using AMPL's notation for piecewise-linear functions.

(c) Incorporate the expression from (b) into an AMPL model for finding the weights and cutoff.

(d) Given this approach, any positive value for `Cut` is as good as any other. We can fix it at a convenient round number — say, 100. Explain why this is the case.

(e) Using a `Cut` of 100, apply the model to the following imaginary credit data:

```
set GOOD := _17 _18 _19 _22 _24 _26 _28 _29 ;
set BAD  := _15 _16 _20 _21 _23 _25 _27 _30 ;

set QUES := Q1 Q2 R1 R2 R3 S2 T4 ;

param ans:  Q1    Q2    R1    R2    R3    S2    T4 :=
    _15    1.0    10    15    20    10    8     10
    _16    0.0     5    15    40     8    10     8
    _17    0.5    10    25    35     8    10    10
    _18    1.5    10    25    30     8    6     10
    _19    1.5     5    20    25     8    8      8
    _20    1.0     5     5    30     8    8      6
    _21    1.0    10    20    30     8    10    10
    _22    0.5    10    25    40     8    8     10
    _23    0.5    10    25    25     8    8     14
    _24    1.0    10    15    40     8    10    10
    _25    0.0     5    15    15    10    12    10
    _26    0.5    10    15    20     8    10    10
    _27    1.0     5    10    25    10    8      6
    _28    0.0     5    15    40     8    10     8
    _29    1.0     5    15    40     8    8     10
    _30    1.5     5    20    25    10    10    14 ;
```

What are the chosen weights? Using these weights, how many of the good customers would be denied a card, and how many of the bad risks would be granted one?

You should find that a lot of the bad risks have scores right at the cutoff. Why does this happen in the solution? How might you adjust the cutoff to deal with it?

(f) To force scores further away from the cutoff (in the desired direction), it might be preferable to use the following objective,

```
sum {i in GOOD} max(0,-Diff[i]+offset) +
sum {i in BAD} max(0,Diff[i]+offset)
```

where `offset` is a positive parameter whose value is supplied. Explain why this change has the desired effect. Try offset values of 2 and 10 and compare the results with those in (e).

(g) Suppose that giving a card to a bad credit risk is considered much more undesirable than refusing a card to a good credit risk. How would you change the model to take this into account?

(h) Suppose that when someone's application is accepted, his or her score is also used to suggest an initial credit limit. Thus it is particularly important that bad credit risks not receive very large scores. How would you add pieces to the piecewise-linear objective function terms to account for this concern?

17-6. In Exercise 18-3, we suggest a way to estimate position, velocity and acceleration values from imprecise data, by minimizing a nonlinear ''sum of squares'' function:

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

An alternative approach instead minimizes a sum of absolute values:

$$\sum_{j=1}^{n} |h_j - (a_0 - a_1 t_j - \tfrac{1}{2} a_2 t_j^2)|.$$

(a) Substitute the sum of absolute values directly for the sum of squares in the model from Exercise 18-3, first with the `abs` function, and then with AMPL's explicit piecewise-linear notation.

Explain why neither of these formulations is likely to be handled effectively by any solver.

(b) To model this situation effectively, we introduce variables e_j to represent the individual formulas $h_j - (a_0 - a_1 t_j - \tfrac{1}{2} a_2 t_j^2)$ whose absolute values are being taken. Then we can express the minimization of the sum of absolute values as the following constrained optimization problem:

$$\text{Minimize } \sum_{j=1}^{n} |e_j|$$

$$\text{Subject to } e_j = h_j - (a_0 - a_1 t_j - \tfrac{1}{2} a_2 t_j^2), \ j = 1, \ldots, n$$

Write an AMPL model for this formulation, using the piecewise-linear notation for the terms $|e_j|$.

(c) Solve for a_0, a_1, and a_2 using the data from Exercise 18-3. How much difference is there between this estimate and the least-squares one?

Use `display` to print the e_j values for both the least-squares and the least-absolute-values solutions. What is the most obvious qualitative difference?

(d) Yet another possibility is to focus on the greatest absolute deviation, rather than the sum:

$$\max_{j=1,\ldots,n} |h_j - (a_0 - a_1 t_j - \tfrac{1}{2} a_2 t_j^2)|.$$

Formulate an AMPL linear program that will minimize this quantity, and test it on the same data as before. Compare the resulting estimates and e_j values. Which of the three estimates would you choose in this case?

17-7. A planar *structure* consists of a set of *joints* connected by *bars*. For example, in the following diagram, the joints are represented by circles, and the bars by lines between two circles:
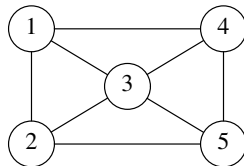
Consider the problem of finding a minimum-weight structure to meet certain external forces. We let J be the set of joints, and $B \subseteq J \times J$ be the set of *admissible* bars; for the diagram above, we could take $J = \{1,2,3,4,5\}$, and

$$B = \{(1,2), (1,3), (1,4), (2,3), (2,5), (3,4), (3,5), (4,5)\}.$$

The ''origin'' and ''destination'' of a bar are arbitrary. The bar between joints 1 and 2, for example, could be represented in B by either $(1,2)$ or $(2,1)$, but it need not be represented by both.

We can use two-dimensional Euclidean coordinates to specify the position of each joint in the plane, taking some arbitrary point as the origin:

a_i^x horizontal position of joint i relative to the origin

a_i^y vertical position of joint i relative to the origin

For the example, if the origin lies exactly at joint 2, we might have

$$(a_1^x, a_1^y) = (0, 2), \ (a_2^x, a_2^y) = (0, 0), \ (a_3^x, a_3^y) = (2, 1),$$
$$(a_4^x, a_4^y) = (4, 2), \ (a_5^x, a_5^y) = (4, 0).$$

The remaining data consist of the external forces on the joints:

f_i^x horizontal component of the external force on joint i

f_i^y vertical component of the external force on joint i

To resist this force, a subset $S \subseteq J$ of joints is fixed in position. (It can be proved that fixing two joints is sufficient to guarantee a solution.)

The external forces induce stresses on the bars, which we can represent as

F_{ij} if > 0, tension on bar (i,j)

         if < 0, compression of bar (i,j)

A set of stresses is in *equilibrium* if the external forces, tensions and compressions balance at all joints, in both the horizontal and vertical components — except at the fixed joints. That is, for each joint $k \notin S$,

$$\sum_{i \in J:(i,k) \in B} c_{ik}^x F_{ik} \ - \ \sum_{j \in J:(k,j) \in B} c_{kj}^x F_{kj} \ = \ f_k^x$$
$$\sum_{i \in J:(i,k) \in B} c_{ik}^y F_{ik} \ - \ \sum_{j \in J:(k,j) \in B} c_{kj}^y F_{kj} \ = \ f_k^y,$$

where c_{st}^x and c_{st}^y are the cosines of the direction from joint s to joint t with the horizontal and vertical axes,

$$c_{st}^x = (a_t^x - a_s^x)/l_{st},$$
$$c_{st}^y = (a_t^y - a_s^y)/l_{st},$$

and l_{st} is the length of the bar (s,t):

$$l_{st} = \sqrt{(a_t^x - a_s^x)^2 + (a_t^y - a_s^y)^2}.$$

In general, there are infinitely many different sets of equilibrium stresses. However, it can be shown that a given system of stresses will be realized in a structure of minimum weight if and only if the cross-sectional areas of the bars are proportional to the absolute values of the stresses. Since the weight of a bar is proportional to the cross section times length, we can take the (suitably scaled) weight of bar (i,j) to be

$$w_{ij} = l_{ij} \cdot |F_{ij}|.$$

The problem is then to find a system of stresses F_{ij} that meet the equilibrium conditions, and that minimize the sum of the weights w_{ij} over all bars $(i,j) \in B$.

(a) The indexing sets for this linear program can be declared in AMPL as:

```
set joints;
set fixed within joints;
set bars within {i in joints, j in joints: i <> j};
```

Using these set declarations, formulate an AMPL model for the minimum-weight structural design problem. Use the piecewise-linear notation of this chapter to represent the absolute-value terms in the objective function.

(b) Now consider in particular a structure that has the following joints:
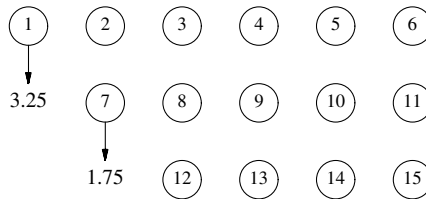
Assume that there is one unit horizontally and vertically between joints, and that the origin is at the lower left; thus $(a_1^x, a_1^y) = (0, 2)$ and $(a_{15}^x, a_{15}^y) = (5, 0)$.

Let there be external forces of 3.25 and 1.75 units straight downward on joints 1 and 7, so that $f_1^y = -3.25$, $f_7^y = -1.75$, and otherwise all $f_i^x = 0$ and $f_i^y = 0$. Let $S = \{6, 15\}$. Finally, let the admissible bars consist of all possible bars that do not go directly through a joint; for example, $(1, 2)$ or $(1, 9)$ or $(1, 13)$ would be admissible, but not $(1, 3)$ or $(1, 12)$ or $(1, 14)$.

Determine all the data for the problem that is needed by the linear program, and represent it as AMPL data statements.

(c) Use AMPL to solve the linear program and to examine the minimum-weight structure that is determined.

Draw a diagram of the optimal structure, indicating the cross sections of the bars and the nature of the stresses. If there is zero force on a bar, it has a cross section of zero, and may be left out of your diagram.

(d) Repeat parts (b) and (c) for the case in which all possible bars are admissible. Is the resulting structure different? Is it any lighter?