Display Commands

AMPL offers a rich variety of commands and options to help you examine and report the results of optimization. Section 12.1 introduces display, the most convenient command for arranging set members and numerical values into lists and tables; Sections 12.2 and 12.3 provide a detailed account of display options that give you more control over how the lists and tables are arranged and how numbers appear in them. Section 12.4 describes print and printf, two related commands that are useful for preparing data to be sent to other programs, and for formatting simple reports.

Although our examples are based on the display of sets, parameters and variables — and expressions involving them — you can use the same commands to inspect dual values, slacks, reduced costs, and other quantities associated with an optimal solution; the rules for doing so are explained in Section 12.5.

AMPL also provides ways to access modeling and solving information. Section 12.6 describes features that can be useful when, for example, you want to view a parameter’s declaration at the command-line, display a particular constraint from a problem instance, list the values and bounds of all variables regardless of their names, or record timings of AMPL and solver activities.

Finally, Section 12.7 addresses general facilities for manipulating output of AMPL commands. These include features for redirection of command output, logging of output, and suppression of error messages.

12.1 Browsing through results: the display command

The easiest way to examine data and result values is to type display and a description of what you want to look at. The display command automatically formats the values in an intuitive and familiar arrangement; as much as possible, it uses the same list and table formats as the data statements described in Chapter 9. Our examples use parameters and variables from models defined in other chapters.
As we will describe in more detail in Section 12.7, it is possible to capture the output of display commands in a file, by adding >filename to the end of a display command; this redirection mechanism applies as well to most other commands that produce output.

**Displaying sets**

The contents of sets are shown by typing display and a list of set names. This example is taken from the model of Figure 6-2a:

```ampl
ampl: display ORIG, DEST, LINKS;
set ORIG := GARY CLEV PITT;
set DEST := FRA DET LAN WIN STL FRE LAF;
set LINKS :=
   (GARY,DET) (GARY,LAF) (CLEV,LAN) (CLEV,LAF) (PITT,STL)
   (GARY,LAN) (CLEV,FRA) (CLEV,WIN) (PITT,FRA) (PITT,FRE)
   (GARY,STL) (CLEV,DET) (CLEV,STL) (PITT,WIN);
```

If you specify the name of an indexed collection of sets, each set in the collection is shown (from Figure 6-3):

```ampl
ampl: display PROD, AREA;
set PROD := bands coils;
set AREA[bands] := east north;
set AREA[coils] := east west export;
```

Particular members of an indexed collection can be viewed by subscripting, as in display AREA["bands"].

The argument of display need not be a declared set; it can be any of the expressions described in Chapter 5 or 6 that evaluate to sets. For example, you can show the union of all the sets AREA[p]:

```ampl
ampl: display union {p in PROD} AREA[p];
set union {p in PROD} AREA[p] := east north west export;
```

or the set of all transportation links on which the shipping cost is greater than 500:

```ampl
ampl: display {(i,j) in LINKS: cost[i,j] * Trans[i,j] > 500};
set {(i,j) in LINKS: cost[i,j]*Trans[i,j] > 500} :=
   (GARY,STL) (CLEV,DET) (CLEV,WIN) (PITT,FRA) (PITT,FRE)
   (GARY,LAF) (CLEV,LAN) (CLEV,LAF) (PITT,STL);
```

Because the membership of this set depends upon the current values of the variables Trans[i,j], you could not refer to it in a model, but it is legal in a display command, where variables are treated the same as parameters.

**Displaying parameters and variables**

The display command can show the value of a scalar model component:
or the values of individual components from an indexed collection (Figure 1-6b):

ampl: display avail["reheat"], avail["roll"];
avail[‘reheat’] = 35
avail[‘roll’] = 40

or an arbitrary expression:

ampl: display sin(1)^2 + cos(1)^2;
sin(1)^2 + cos(1)^2 = 1

The major use of display, however, is to show whole indexed collections of data. For “‘one-dimensional”’ data — parameters or variables indexed over a simple set — AMPL uses a column format (Figure 4-6b):

ampl: display avail;
avail [*] :=
  reheat 35
  roll 40
;

For “‘two-dimensional”’ parameters or variables — indexed over a set of pairs or two simple sets — AMPL forms a list for small amounts of data (Figure 4-1):

ampl: display supply;
supply :=
  CLEV bands  700
  CLEV coils  1600
  CLEV plate  300
  GARY bands  400
  GARY coils  800
  GARY plate  200
  PITT bands  800
  PITT coils  1800
  PITT plate  300
;

or a table for larger amounts:

ampl: display demand;
demand [*,*] :
  bands  coils  plate :=
  DET  300  750  100
  FRA  300  500  100
  FRE  225  850  100
  LAF  250  500  250
  LAN  100  400  0
  STL  650  950  200
  WIN  75  250  50
;
You can control the choice between formats by setting option `display_1col`, which is described in the next section.

A parameter or variable (or any other model entity) indexed over a set of ordered pairs is also considered to be a two-dimensional object and is displayed in a similar manner. Here is the display for a parameter indexed over the set `LINKS` that was displayed earlier in this section (from Figure 6-2a):

```ampl
display cost;
cost :=
CLEV DET 9
CLEV FRA 27
CLEV LAF 17
CLEV LAN 12
CLEV STL 26
CLEV WIN 9
GARY DET 14
GARY LAF 8
GARY LAN 11
GARY STL 16
PITT FRA 24
PITT FRE 99
PITT STL 28
PITT WIN 13
;
```

This, too, can be made to appear in a table format, as the next section will show.

To display values indexed in three or more dimensions, AMPL again forms lists for small amounts of data. Multi-dimensional entities more often involve data in large quantities, however, in which case AMPL “slices” the values into two-dimensional tables, as in the case of this variable from Figure 4-6:

```ampl
display Trans;
Trans [CLEV,*,*]
: bands coils plate :=
DET 0 750 0
 FRA 0 0 0
 FRE 0 0 0
 LAF 0 500 0
 LAN 0 400 0
 STL 0 50 0
 WIN 0 250 0

 [GARY,*,*]
 : bands coils plate :=
DET 0 0 0
 FRA 0 0 0
 FRE 225 850 100
 LAF 250 0 0
 LAN 0 0 0
 STL 650 900 200
 WIN 0 0 0
```
At the head of the first table, the template \([\text{CLEV,*,*}]\) indicates that the slice is through \(\text{CLEV}\) in the first component, so the entry in row \(\text{LAF}\) and column \(\text{coils}\) says that \(\text{Trans["CLEV","LAF","coils"]}\) is 500. Since the first index of \(\text{Trans}\) is always \(\text{CLEV}\), \(\text{GARY}\) or \(\text{PITT}\) in this case, there are three slice tables in all. But AMPL does not always slice through the first component; it picks the slices so that the display will contain the fewest possible tables.

A display of two or more components of the same dimensionality is always presented in a list format, whether the components are one-dimensional (Figure 4-4):

```
ampl: display inv0, prodcost, invcost;
    : inv0 prodcost invcost :=
      bands 10 10 2.5
      coils 0 11 3
;  
```

or two-dimensional (Figure 4-6):

```
ampl: display rate, make_cost, Make;
    : rate make_cost Make :=
      CLEV bands 190 190 0
      CLEV coils 130 170 1950
      CLEV plate 160 185 0
      GARY bands 200 180 1125
      GARY coils 140 170 1750
      GARY plate 160 180 300
      PITT bands 230 190 775
      PITT coils 160 180 500
      PITT plate 170 185 500
;  
```

or any higher dimension. The indices appear in a list to the left, with the last one changing most rapidly.

As you can see from these examples, \text{display} normally arranges row and column labels in alphabetical or numerical order, regardless of the order in which they might have been given in your data file. When the labels come from an ordered set, however, the original ordering is honored (Figure 5-3):
ampl: display avail;
avail [*] :=  
27sep 40
04oct 40
11oct 32
18oct 40
;

For this reason, it can be worthwhile to declare certain sets of your model to be ordered, even if their ordering plays no explicit role in your formulation.

Displaying indexed expressions

The display command can show the value of any arithmetic expression that is valid in an AMPL model. Single-valued expressions pose no difficulty, as in the case of these three profit components from Figure 4-4:

ampl: display sum {p in PROD, t in 1..T} revenue[p,t]*Sell[p,t],
ampl? sum {p in PROD, t in 1..T} prodcost[p]*Make[p,t],
ampl? sum {p in PROD, t in 1..T} invcost[p]*Inv[p,t];
sum{p in PROD, t in 1 .. T} revenue[p,t]*Sell[p,t] = 787810
sum{p in PROD, t in 1 .. T} prodcost[p]*Make[p,t] = 269477
sum{p in PROD, t in 1 .. T} invcost[p]*Inv[p,t] = 3300

Suppose, however, that you want to see all the individual values of revenue[p,t] * Sell[p,t]. Since you can type display revenue, Sell to display the separate values of revenue[p,t] and Sell[p,t], you might want to ask for the products of these values by typing:

ampl: display revenue * Sell;
syntax error
context:  display revenue >>> * <<< Sell;

AMPL does not recognize this kind of array arithmetic. To display an indexed collection of expressions, you must specify the indexing explicitly:

ampl: display {p in PROD, t in 1..T} revenue[p,t]*Sell[p,t];
revenue[p,t]*Sell[p,t] [*,*] (tr) :
  bands coils :=
  1  150000  9210
  2  156000  87500
  3  37800   129500
  4  54000   163800
;

To apply the same indexing to two or more expressions, enclose a list of them in parentheses after the indexing expression:
ampl: display \((p \in \text{PROD}, t \in 1..T)\)

ampl?

\((\text{revenue}\[p,t]\times \text{Sell}\[p,t], \text{prodcost}\[p]\times \text{Make}\[p,t])\);  

: \text{revenue}\[p,t]\times \text{Sell}\[p,t] \text{prodcost}\[p]\times \text{Make}\[p,t] :=  

bands 1 150000 59900  
bands 2 156000 60000  
bands 3 37800 14000  
bands 4 54000 20000  
coils 1 9210 15477  
coils 2 87500 15400  
coils 3 129500 38500  
coils 4 163800 46200  

;

An indexing expression followed by an expression or parenthesized list of expressions is treated as a single display item, which specifies some indexed collection of values. A display command may contain one of these items as above, or a comma-separated list of them.

The presentation of the values for indexed expressions follows the same rules as for individual parameters and variables. In fact, you can regard a command like

\[\text{display revenue, Sell}\]

as shorthand for

ampl: display \((p \in \text{PROD}, t \in 1..T) (\text{revenue}\[p,t], \text{Sell}\[p,t])\);  

: \text{revenue}\[p,t] \text{Sell}\[p,t] :=  

bands 1 25 6000  
bands 2 26 6000  
bands 3 27 1400  
bands 4 27 2000  
coils 1 30 307  
coils 2 35 2500  
coils 3 37 3500  
coils 4 39 4200  

;

If you rearrange the indexing expression so that \(t \in 1..T\) comes first, however, the rows of the list are instead sorted first on the members of \(1..T\):

ampl: display \((t \in 1..T, p \in \text{PROD}) (\text{revenue}\[p,t], \text{Sell}\[p,t])\);  

: \text{revenue}\[p,t] \text{Sell}\[p,t] :=  

1 bands 25 6000  
1 coils 30 307  
2 bands 26 6000  
2 coils 35 2500  
3 bands 27 1400  
3 coils 37 3500  
4 bands 27 2000  
4 coils 39 4200  

;

This change in the default presentation can only be achieved by placing an explicit indexing expression after display.
In addition to indexing individual display items, you can specify a set over which the whole display command is indexed — that is, you can ask that the command be executed once for each member of an indexing set. This feature is particularly useful for rearranging slices of multidimensional tables. When, earlier in this section, we displayed the three-dimensional variable Trans indexed over \{ORIG,DEST,PROD\}, AMPL chose to slice the values through members of ORIG to produce a series of two-dimensional tables.

What if you want to display slices through PROD? Rearranging the indexing expression, as in our previous example, will not reliably have the desired effect; the display command always picks the smallest indexing set, and where there is more than one that is smallest, it does not necessarily choose the first. Instead, you can say explicitly that you want a separate display for each \( p \) in PROD:

```ampl
ampl: display {p in PROD}:
    Trans[i,j,'bands'] [:,]* (tr)
:    CLEV GARY PITT :=
  DET 0 0 300
  FRA 0 0 300
  FRE 0 225 0
  LAF 0 250 0
  LAN 0 0 100
  STL 0 650 0
  WIN 0 0 75
;

Trans[i,j,'coils'] [:,]* (tr)
:    CLEV GARY PITT :=
  DET 750 0 0
  FRA 0 0 500
  FRE 0 850 0
  LAF 500 0 0
  LAN 400 0 0
  STL 50 900 0
  WIN 250 0 0
;

Trans[i,j,'plate'] [:,]* (tr)
:    CLEV GARY PITT :=
  DET 0 0 100
  FRA 0 0 100
  FRE 0 100 0
  LAF 0 0 250
  LAN 0 0 0
  STL 0 200 0
  WIN 0 0 50
;
```

As this example shows, if a display command specifies an indexing expression right at the beginning, followed by a colon, the indexing set applies to the whole command. For
SECTION 12.2  FORMATTING OPTIONS FOR DISPLAY  227

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display_1col</td>
<td>maximum elements for a table to be displayed in list format (20)</td>
</tr>
<tr>
<td>display_transpose</td>
<td>transpose tables if rows – columns &lt; display_transpose (0)</td>
</tr>
<tr>
<td>display_width</td>
<td>maximum line width (79)</td>
</tr>
<tr>
<td>gutter_width</td>
<td>separation between table columns (3)</td>
</tr>
<tr>
<td>omit_zero_cols</td>
<td>if not 0, omit all-zero columns from displays (0)</td>
</tr>
<tr>
<td>omit_zero_rows</td>
<td>if not 0, omit all-zero rows from displays (0)</td>
</tr>
</tbody>
</table>

**Table 12-1:** Formatting options for display (with default values).

each member of the set, the expressions following the colon are evaluated and displayed separately.

### 12.2 Formatting options for display

The display command uses a few simple rules for choosing a good arrangement of data. By changing several options, you can control overall arrangement, handling of zero values, and line width. These options are summarized in Table 12-1, with default values shown in parentheses.

**Arrangement of lists and tables**

The display of a one-dimensional parameter or variable can produce a very long list, as in this example from the scheduling model of Figure 16-5:

```plaintext
ampl: display required;
required [*] :=
Fri1 100
Fri2  78
Fri3  52
Mon1 100
Mon2  78
Mon3  52
Sat1 100
Sat2  78
Thu1 100
Thu2  78
Thu3  52
Tue1 100
Tue2  78
Tue3  52
Wed1 100
Wed2  78
Wed3  52
;
```
The option display_1col can be used to request a more compact format:

    ampl: option display_1col 0;
    ampl: display required;
    required [*] :=
    Fri1 100 Mon1 100 Sat1 100 Thu2 78 Tue2 78 Wed2 78
    Fri2 78 Mon2 78 Sat2 78 Thu3 52 Tue3 52 Wed3 52
    Fri3 52 Mon3 52 Thu1 100 Tue1 100 Wed1 100
    ;

The one-column list format is used when the number of values to be displayed is less than or equal to display_1col, and the compact format is used otherwise. The default for display_1col is 20; set it to zero to force the compact format, or to a very large number to force the list format.

Multi-dimensional displays are affected by option display_1col in an analogous way. The one-column list format is used when the number of values is less than or equal to display_1col, while the appropriate compact format — in this case, a table — is used otherwise. We showed an example of the difference in the previous section, where the display for supply appeared as a list because it had only 9 values, while the display for demand appeared as a table because its 21 values exceed the default setting of 20 for option display_1col.

Since a parameter or variable indexed over a set of ordered pairs is also considered to be two-dimensional, the value of display_1col affects its display as well. Here is the table format for the parameter cost indexed over the set LINKS (from Figure 6-2a) that was displayed in the preceding section:

    ampl: option display_1col 0;
    ampl: display cost;
    cost [*,*] (tr) :=
    CLEV GARY PITT :=
    DET 9 14 .
    FRA 27 . 24
    FRE . . 99
    LAF 17 8 .
    LAN 12 11 .
    STL 26 16 28
    WIN 9 . 13
    ;

A dot (.) entry indicates a nonexistent combination in the index set. Thus in the GARY column of the table, there is a dot in the FRA row because the pair (GARY,FRA) is not a member of LINKS; no cost["GARY","FRA"] is defined for this problem. On the other hand, LINKS does contain the pair (GARY,LAF), and cost["GARY","LAF"] is shown as 8 in the table.

In choosing an orientation for tables, the display command by default favors rows over columns; that is, if the number of columns would exceed the number of rows, the table is transposed. Thus the table for demand in the previous section has rows labeled by the first coordinate and columns by the second, because it is indexed over DEST with
7 members and then PROD with 3 members. By contrast, the table for cost has columns labeled by the first coordinate and rows by the second, because it is indexed over ORIG with 3 members and then DEST with 7 members. A transposed table is indicated by a (tr) in its first line.

The transposition status of a table can be reversed by changing the display_transpose option. Positive values tend to force transposition:

```ampl
ampl: option display_transpose 5;
ampl: display demand;
demand [*,*] (tr)

: DET FRA FRE LAF LAN STL WIN :=
bands 300 300 225 250 100 650 75
coils 750 500 850 500 400 950 250
plate 100 100 100 250 0 200 50
;
```

while negative values tend to suppress it:

```ampl
ampl: option display_transpose -5;
ampl: display cost;
cost [*,*]

: DET FRA FRE LAF LAN STL WIN :=
CLEV 9 27 . 17 12 26 9
GARY 14 . . 8 11 16 .
PITT . 24 99 . . 28 13
;
```

The rule is as follows: a table is transposed only when the number of rows minus the number of columns would be less than display_transpose. At its default value of zero, display_transpose gives the previously described default behavior.

**Control of line width**

The option display_width gives the maximum number of characters on a line generated by display (as seen in the model of Figure 16-4):

```ampl
ampl: option display_width 50, display_1col 0;
ampl: display required;
required [*] :=

Fri1 100 Mon3 52 Thu3 52 Wed2 78
Fri2 78 Sat1 100 Tue1 100 Wed3 52
Fri3 52 Sat2 78 Tue2 78
Mon1 100 Thu1 100 Tue3 52
Mon2 78 Thu2 78 Wed1 100
;
```

When a table would be wider than display_width, it is cut vertically into two or more tables. The row names in each table are the same, but the columns are different:
ampl: option display_width 50; display cost;


cost [*,*] 

: C118 C138 C140 C246 C250 C251 D237 D239 :=

Couillard 6 9 8 7 11 10 4 5
Daskin 11 8 7 6 9 10 1 5
Hazen 9 10 11 1 5 6 2 7
Hopp 11 9 8 10 6 5 1 7
Iravani 3 2 8 9 10 11 1 5
Linetsky 11 9 10 5 3 4 6 7
Mehrotra 6 11 10 9 8 7 1 2
Nelson 11 5 4 6 7 8 1 9
Smilowitz 11 9 10 8 6 5 7 3
Tamhane 5 6 9 8 4 3 7 10
White 11 9 8 4 6 5 3 10

: D241 M233 M239 :=

Couillard 3 2 1
Daskin 4 2 3
Hazen 8 3 4
Hopp 4 2 3
Iravani 4 6 7
Linetsky 8 1 2
Mehrotra 5 4 3
Nelson 10 2 3
Smilowitz 4 1 2
Tamhane 11 2 1
White 7 2 1

On the other hand, where the headings are narrower than the values, you may be able to
squeeze more on a line by reducing the option gutter_width — the number of spaces
between columns — from its default value of 3 to 2 or 1.
**Suppression of zeros**

In some kinds of linear programs that have many more variables than constraints, most of the variables have an optimal value of zero. For instance in the assignment problem of Figure 3-2, the optimal values of all the variables form this table, in which there is a single 1 in each row and each column:

```
ampl: display Trans;
Trans [",,]
: Coullard Daskin Hazen Hopp Iravani Linetsky Mehrotra Nelson Smilowitz Tamhane White :=
  C118 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  C138 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
  C140 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
  C246 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  C250 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  C251 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  D237 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  D239 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  D241 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  M233 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  M239 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
;```

By setting `omit_zero_rows` to 1, all the zero values are suppressed, and the list comes down to the entries of interest:

```
ampl: option omit_zero_rows 1;
ampl: display Trans;
Trans :=
  Coullard C118 1
  Daskin D241 1
  Hazen C246 1
  Hopp D237 1
  Iravani C138 1
  Linetsky C250 1
  Mehrotra D239 1
  Nelson C140 1
  Smilowitz M233 1
  Tamhane C251 1
  White M239 1
;```

If the number of nonzero entries is less than the value of `display_1col`, the data is printed as a list, as it is here. If the number of nonzeros is greater than `display_1col`, a table format would be used, and the `omit_zero_rows` option would only suppress table rows that contain all zero entries.

For example, the display of the three-dimensional variable `Trans` from earlier in this chapter would be condensed to the following:
A corresponding option `omit_zero_cols` suppresses all-zero columns when set to 1, and would eliminate two columns from `Trans[CLEV,*,*]`.

### 12.3 Numeric options for display

The numbers in a table or list produced by `display` are the result of a transformation from the computer’s internal numeric representation to a string of digits and symbols. AMPL’s options for adjusting this transformation are shown in Table 12-2. In this section we first consider options that affect only the appearance of numbers, and then options that affect underlying solution values as well.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>display_eps</code></td>
<td>smallest magnitude displayed differently from zero (0)</td>
</tr>
<tr>
<td><code>display_precision</code></td>
<td>digits of precision to which displayed numbers are rounded; full precision if 0 (6)</td>
</tr>
<tr>
<td><code>display_round</code></td>
<td>digits left or (if negative) right of decimal place to which displayed numbers are rounded, overriding <code>display_precision</code> (&quot;&quot;)</td>
</tr>
<tr>
<td><code>solution_precision</code></td>
<td>digits of precision to which solution values are rounded; full precision if 0 (0)</td>
</tr>
<tr>
<td><code>solution_round</code></td>
<td>digits left or (if negative) right of decimal place to which solution values are rounded, overriding <code>solution_precision</code> (&quot;&quot;)</td>
</tr>
</tbody>
</table>

*Table 12-2: Numeric options for `display` (with default values).*
Appearance of numeric values

In all of our examples so far, the display command shows each numerical value to the same number of significant digits:

\[
\text{ampl: display } \{p \text{ in PROD, } t \text{ in } 1..T\} \text{ Make}[p,t]/\text{rate}[p];
\]
\[
\text{Make}[p,t]/\text{rate}[p] [*,*] \text{ (tr)}
\]
\[
: \text{ bands coils :=}
\]
\[
1 \text{ 29.95 10.05}
\]
\[
2 \text{ 30 10}
\]
\[
3 \text{ 20 12}
\]
\[
4 \text{ 32.1429 7.85714}
\]
\[
;
\]
\[
\text{ampl: display } \{p \text{ in PROD, } t \text{ in } 1..T\} \text{ prodcost}[p]*\text{Make}[p,t];
\]
\[
\text{prodcost}[p]*\text{Make}[p,t] [*,*] \text{ (tr)}
\]
\[
: \text{ bands coils :=}
\]
\[
1 \text{ 59900 15477}
\]
\[
2 \text{ 60000 15400}
\]
\[
3 \text{ 40000 18480}
\]
\[
4 \text{ 64285.7 12100}
\]
\[
;
\]

(see Figures 6-3 and 6-4). The default is to use six significant digits, whether the result comes out as 7.85714 or 64285.7. Some numbers seem to have fewer digits, but only because trailing zeros have been dropped; 29.95 represents the number that is exactly 29.9500 to six digits, for example, and 59900 represents 59900.0.

By changing the option display_precision to a value other than six, you can vary the number of significant digits reported:

\[
\text{ampl: option display_precision 3;}
\]
\[
\text{ampl: display Make['bands',4]/rate['bands'],}
\]
\[
\text{ampl? prodcost['bands'] * Make['bands',4];}
\]
\[
\text{Make['bands',4]/rate['bands']} = 32.1
\]
\[
\text{prodcost['bands']*Make['bands',4] = 64300}
\]
\[
\text{ampl: option display_precision 9;}
\]
\[
\text{ampl: display Make['bands',4]/rate['bands'],}
\]
\[
\text{ampl? prodcost['bands'] * Make['bands',4];}
\]
\[
\text{Make['bands',4]/rate['bands']} = 32.1428571
\]
\[
\text{prodcost['bands']*Make['bands',4] = 64285.7143}
\]
\[
\text{ampl: option display_precision 0;}
\]
\[
\text{ampl: display Make['bands',4]/rate['bands'],}
\]
\[
\text{ampl? prodcost['bands'] * Make['bands',4];}
\]
\[
\text{Make['bands',4]/rate['bands']} = 32.14285714285713
\]
\[
\text{prodcost['bands']*Make['bands',4] = 64285.71428571427}
\]

In the last example, a display_precision of 0 is interpreted specially; it tells display to represent numbers as exactly as possible, using however many digits are necessary. (To be precise, the displayed number is the shortest decimal representation that, when correctly rounded to the computer’s representation, gives the value exactly as stored in the computer.)
Displays to a given precision provide the same degree of useful information about each number, but they can look ragged due to the varying numbers of digits after the decimal point. To specify rounding to a fixed number of decimal places, regardless of the resulting precision, you may set the option `display_round`. A nonnegative value specifies the number of digits to appear after the decimal point:

```ampl
ampl: option display_round 2;
ampl: display (p in PROD, t in 1..T) Make[p,t]/rate[p];
Make[p,t]/rate[p] [*,*] (tr)
  : bands coils :=
  1 29.95 10.05
  2 30.00 10.00
  3 20.00 12.00
  4 32.14 7.86
;
```

A negative value indicates rounding before the decimal point. For example, when `display_round` is –2, all numbers are rounded to hundreds:

```ampl
ampl: option display_round -2;
ampl: display (p in PROD, t in 1..T) prodcost[p]*Make[p,t];
prodcost[p]*Make[p,t] [*,*] (tr)
  : bands coils :=
  1 59900 15500
  2 60000 15400
  3 40000 18500
  4 64300 12100
;
```

Any integer value of `display_round` overrides the effect of `display_precision`. To turn off `display_round`, set it to some non-integer such as the empty string ''. Depending on the solver you employ, you may find that some of the solution values that ought to be zero do not always quite come out that way. For example, here is one solver’s report of the objective function terms `cost[i,j] * Trans[i,j]` for the assignment problem of Section 3.3:

```ampl
ampl: option omit_zero_rows 1;
ampl: display (i in ORIG, j in DEST) cost[i,j] * Trans[i,j];
cost[i,j]*Trans[i,j] :=
  Coullard C118 6
  Coullard D241 2.05994e-17
  Daskin D237 1
  Hazen C246 1
  Hopp D237 6.86647e-18
  Hopp D241 4
  ... 9 lines omitted
  White C246 2.74659e-17
  White C251 -3.43323e-17
  White M239 1
;
```
Minuscule values like $6.86647e^{-18}$ and $-3.43323e^{-17}$ have no significance in the context of this problem; they would be zeros in an exact solution, but come out slightly nonzero as an artifact of the way that the solver’s algorithm interacts with the computer’s representation of numbers.

To avoid viewing these numbers in meaningless precision, you can pick a reasonable setting for `display_round` — in this case 0, since there are no digits of interest after the decimal point:

```ampl
    option display_round 0;
    display {i in ORIG, j in DEST} cost[i,j] * Trans[i,j];
    cost[i,j]*Trans[i,j] :=
        Coullard  C118  6
        Coullard  D241  0
        Daskin    D237  1
        Hazen     C246  1
        Hopp      D237  0
        Hopp      D241  4
        Iravani   C118  0
        Iravani   C138  2
        Linetsky  C250  3
        Mehrotra  D239  2
        Nelson    C138  0
        Nelson    C140  4
        Smilowitz M233  1
        Tamhane   C118  -0
        Tamhane   C251  3
        White     C246  0
        White     C251  -0
        White     M239  1
    ;
```

The small numbers are now represented only as 0 if positive or –0 if negative. If you want to suppress their appearance entirely, however, you must set a separate option, `display_eps`:

```ampl
    option display_eps 1e-10;
    display {i in ORIG, j in DEST} cost[i,j] * Trans[i,j];
    cost[i,j]*Trans[i,j] :=
        Coullard  C118  6
        Daskin    D237  1
        Hazen     C246  1
        Hopp      D241  4
        Iravani   C138  2
        Linetsky  C250  3
        Mehrotra  D239  2
        Nelson    C140  4
        Smilowitz M233  1
        Tamhane   C251  3
        White     M239  1
    ;
```
Any value whose magnitude is less than the value of `display_eps` is treated as an exact zero in all output of `display`.

**Rounding of solution values**

The options `display_precision`, `display_round` and `display_eps` affect only the appearance of numbers, not their actual values. You can see this if you try to display the set of all pairs of `i` in `ORIG` and `j` in `DEST` that have a positive value in the preceding table, by comparing `cost[i,j]*Trans[i,j]` to 0:

```
ampl: display {i in ORIG, j in DEST: cost[i,j]*Trans[i,j] > 0};
set {i in ORIG, j in DEST: cost[i,j]*Trans[i,j] > 0} :=
  (Couillard,C118) (Iravani,C118) (Smilowitz,M233)
  (Daskin,D237) (Linetsky,C250) (Tamhane,C251)
  (Hazen,C246) (Mehrotra,D239) (White,M239)
  (Hopp,D237) (Nelson,C138)
  (Hopp,D241) (Nelson,C140);
```

Even though a value like 2.05994e–17 is treated as a zero for purposes of `display`, it tests greater than zero. You could fix this problem by changing `>` above to, say, `>` 0.1. As an alternative, you can set the option `solution_round` so that AMPL rounds the solution values to a reasonable precision when they are received from the solver:

```
ampl: option solution_round 10;
ampl: solve;
MINOS 5.5: optimal solution found.
40 iterations, objective 28
```

```
ampl: display {i in ORIG, j in DEST: cost[i,j]*Trans[i,j] > 0};
set {i in ORIG, j in DEST: cost[i,j]*Trans[i,j] > 0} :=
  (Couillard,C118) (Iravani,C138) (Smilowitz,M233)
  (Daskin,D237) (Linetsky,C250) (Tamhane,C251)
  (Hazen,C246) (Mehrotra,D239) (White,M239)
  (Hopp,D241) (Nelson,C140);
```

The options `solution_precision` and `solution_round` work in the same way as `display_precision` and `display_round`, except that they are applied only to solution values upon return from a solver, and they permanently change the returned values rather than only their appearance.

Rounded values can make a difference even when they are not near zero. As an example, we first use several display options to get a compact listing of the fractional solution to the scheduling model of Figure 16-4:

```
ampl: model sched.mod;
ampl: data sched.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
19 iterations, objective 265.6
```
Each value \( \text{Work}[j] \) represents the number of workers assigned to schedule \( j \). We can get a quick practical schedule by rounding the fractional values up to the next highest integer; using the \( \text{ceil} \) function to perform the rounding, we see that the total number of workers needed should be:

\[
\text{sum}\{j \in \text{SCHEDS}\} \text{ceil}(\text{Work}[j])
\]

If we copy the numbers from the preceding table and round them up by hand, however, we find that they only sum to 271. The source of the difficulty can be seen by displaying the numbers to full precision:

\[
\text{Work}[*] :=
\begin{align*}
10 & \quad 28.8 & 30 & 14.4 & 71 & 35.6 & 106 & 23.2 & 123 & 35.6 \\
18 & \quad 7.6 & 35 & 6.8 & 73 & 28 & 109 & 14.4 \\
24 & \quad 6.8 & 66 & 35.6 & 87 & 14.4 & 113 & 14.4
\end{align*}
\]

Half the problem is due to the minuscule positive value of \( \text{Work}[108] \), which was rounded up to 1. The other half is due to \( \text{Work}[73] \); although it is 28 in an exact solution, it comes back from the solver with a slightly larger value of 28.000000000000018, so it gets rounded up to 29.

The easiest way to ensure that our arithmetic works correctly in this case is again to set solution_round before solve:

\[
\text{option solution_round 10;}
\text{solve;}
\text{MINOS 5.5: optimal solution found.}
19 iterations, objective 265.6
\]

If we copy the numbers from the preceding table and round them up by hand, however, we find that they only sum to 271. The source of the difficulty can be seen by displaying the numbers to full precision:

\[
\text{Work}[*] :=
\begin{align*}
10 & \quad 28.799999999999997 & 73 & 28.000000000000018 \\
18 & \quad 7.599999999999998 & 87 & 14.399999999999995 \\
24 & \quad 6.799999999999999 & 95 & -5.876671973951407e-15 \\
30 & \quad 14.400000000000001 & 106 & 23.200000000000006 \\
35 & \quad 6.799999999999999 & 108 & 4.685288280240683e-16 \\
55 & \quad -4.939614313857677e-15 & 109 & 14.4 \\
66 & \quad 35.6 & 113 & 14.4 \\
71 & \quad 35.599999999999994 & 123 & 35.599999999999999
\end{align*}
\]
We picked a value of 10 for `solution_round` because we observed that the slight inaccuracies in the solver’s values occurred well past the 10th decimal place.

The effect of `solution_round` or `solution_precision` applies to all values returned by the solver. To modify only certain values, use the assignment (let) command described in Section 11.3 together with the rounding functions in Table 11-1.

### 12.4 Other output commands: `print` and `printf`

Two additional AMPL commands have much the same syntax as `display`, but do not automatically format their output. The `print` command does no formatting at all, while the `printf` command requires an explicit description of the desired formatting.

#### The `print` command

A `print` command produces a single line of output:

```ampl
display

ampl: print sum {p in PROD, t in 1..T} revenue[p,t]*Sell[p,t],
ampl? sum {p in PROD, t in 1..T} prodcost[p]*Make[p,t],
ampl? sum {p in PROD, t in 1..T} invcost[p]*Inv[p,t];
787810 269477 3300

ampl: print {t in 1..T, p in PROD} Make[p,t];
5990 1407 6000 1400 1400 3500 2000 4200
```

or, if followed by an indexing expression and a colon, a line of output for each member of the index set:

```ampl

ampl: print {t in 1..T, p in PROD} Make[p,t];
5990 1407 6000 1400 1400 3500 2000 4200
```

Printed entries are normally separated by a space, but option `print_separator` can be used to change this. For instance, you might set `print_separator` to a tab for data to be imported by a spreadsheet; to do this, type `option print_separator "→"`, where `→` stands for the result of pressing the tab key.

The keyword `print` (with optional indexing expression and colon) is followed by a print item or comma-separated list of print items. A print item can be a value, or an indexing expression followed by a value or parenthesized list of values. Thus a print item is much like a display item, except that only individual values may appear; although you can say `display rate`, you must explicitly specify `print {p in PROD} rate[p]`. Also a set may not be an argument to `print`, although its members may be:
SECTION 12.4 OTHER OUTPUT COMMANDS: PRINT AND PRINTF

ampl: print PROD;
syntax error
context: print >>> PROD; <<<

ampl: print {p in PROD} (p, rate[p]);
bands 200 coils 140

Unlike display, however, print allows indexing to be nested within an indexed item:

ampl: print {p in PROD} (p, rate[p], {t in 1..T} Make[p,t]);
bands 200 5990 6000 1400 2000 coils 140 1407 1400 3500 4200

The representation of numbers in the output of print is governed by the print_precision and print_round options, which work exactly like the display_precision and display_round options for the display command. Initially print_precision is 0 and print_round is an empty string, so that by default print uses as many digits as necessary to represent each value as precisely as possible. For the above examples, print_round has been set to 0, so that the numbers are rounded to integers.

Working interactively, you may find print useful for viewing a few values on your screen in a more compact format than display produces. With output redirected to a file, print is useful for writing unformatted results in a form convenient for spreadsheets and other data analysis tools. As with display, just add >filename to the end of the print command.

The printf command

The syntax of printf is exactly the same as that of print, except that the first print item is a character string that provides formatting instructions for the remaining items:

ampl: printf "Total revenue is $%6.2f.\n",
ampl: sum {p in PROD, t in 1..T} revenue[p,t]*Sell[p,t];
Total revenue is $787810.00.

The format string contains two types of objects: ordinary characters, which are copied to the output, and conversion specifications, which govern the appearance of successive remaining print items. Each conversion specification begins with the character % and ends with a conversion character. For example, %6.2f specifies conversion to a decimal representation at least six characters wide with two digits after the decimal point. The complete rules are much the same as for the printf function in the C programming language; a summary appears in Section A.16 of the Appendix.

The output from printf is not automatically broken into lines. A line break must be indicated explicitly by the combination \n, representing a "newline" character, in the format string. To produce a series of lines, use the indexed version of printf:
This `printf` is executed once for each member of the indexing set preceding the colon; for each \( t \) in \( 1 \ldots T \) the format is applied again, and the `\n` character generates another line break.

The `printf` command is mainly useful, in conjunction with redirection of output to a file, for printing short summary reports in a readable format. Because the number of conversion specifications in the format string must match the number of values being printed, `printf` cannot conveniently produce tables in which the number of items on a line may vary from run to run, such as a table of all `Make[p,t]` values.

### 12.5 Related solution values

Sets, parameters and variables are the most obvious things to look at in interpreting the solution of a linear program, but AMPL also provides ways of examining objectives, bounds, slacks, dual prices and reduced costs associated with the optimal solution.

As we have shown in numerous examples already, AMPL distinguishes the various values associated with a model component by use of “qualified” names that consist of a variable or constraint identifier, a dot (`.`), and a predefined “suffix” string. For instance, the upper bounds for the variable `Make` are called `Make.ub`, and the upper bound for `Make["coils",2]` is written `Make["coils",2].ub`. (Note that the suffix comes after the subscript.) A qualified name can be used like an unqualified one, so that `display Make.ub` prints a table of upper bounds on the `Make` variables, while `display Make, Make.ub` prints a list of the optimal values and upper bounds.

#### Objective functions

The name of the objective function (from a `minimize` or `maximize` declaration) refers to the objective’s value computed from the current values of the variables. This name can be used to represent the optimal objective value in `display`, `print`, or `printf`:

```ampl
ampl: print 100 * Total_Profit /
```

```ampl
ampl? sum {p in PROD, t in 1..T} revenue[p,t] * Sell[p,t];
```

```ampl
65.37528084182735
```

If the current model declares several objective functions, you can refer to any of them, even though only one has been optimized.
**Bounds and slacks**

The suffixes `.lb` and `.ub` on a variable denote its lower and upper bounds, while `.slack` denotes the difference of a variable’s value from its nearer bound. Here’s an example from Figure 5-1:

```ampl
display Buy.lb, Buy, Buy.ub, Buy.slack;
: Buy.lb Buy Buy.ub Buy.slack :=
  BEEF 2 10 10 0
  CHK 2 2 10 0
  FISH 2 10 10 0
  HAM 2 2 10 0
  MCH 2 2 10 0
  MTL 2 6.23596 10 3.76404
  SPG 2 5.25843 10 3.25843
  TUR 2 2 10 0;
```

The reported bounds are those that were sent to the solver. Thus they include not only the bounds specified in `>=` and `<=` phrases of `var` declarations, but also certain bounds that were deduced from the constraints by AMPL’s presolve phase. Other suffixes let you look at the original bounds and at additional bounds deduced by presolve; see the discussion of presolve in Section 14.1 for details.

Two equal bounds denote a fixed variable, which is normally eliminated by presolve. Thus in the planning model of Figure 4-4, the constraint `Inv[p,0] = inv0[p]` fixes the initial inventories:

```ampl
display {p in PROD} (Inv[p,0].lb, inv0[p], Inv[p,0].ub);
: Inv[p,0].lb inv0[p] Inv[p,0].ub :=
  bands 10 10 10
  coils 0 0 0;
```

In the production-and-transportation model of Figure 4-6, the constraint

```
sum {i in ORIG} Trans[i,j,p] = demand[j,p]
```

leads presolve to fix three variables at zero, because `demand["LAN","plate"]` is zero:

```ampl
display {i in ORIG} (Trans[i,"LAN","plate"].lb, Trans[i,"LAN","plate"].ub);
: Trans[i,"LAN","plate"].lb Trans[i,"LAN","plate"].ub :=
  CLEV 0 0
  GARY 0 0
  PITT 0 0;
```

As this example suggests, presolve’s adjustments to the bounds may depend on the data as well as the structure of the constraints.

The concepts of bounds and slacks have an analogous interpretation for the constraints of a model. Any AMPL constraint can be put into the standard form
where the \textit{body} is a sum of all terms involving variables, while the \textit{lower bound} and \textit{upper bound} depend only on the data. The suffixes .lb, .body and .ub give the current values of these three parts of the constraint. For example, in the diet model of Figure 5-1 we have the declarations

\begin{verbatim}
subject to Diet_Min {i in MINREQ}:
  sum {j in FOOD} amt[i,j] * Buy[j] >= n_Min[i];

subject to Diet_Max {i in MAXREQ}:
  sum {j in FOOD} amt[i,j] * Buy[j] <= n_Max[i];
\end{verbatim}

and the following constraint bounds:

\begin{verbatim}
ampl: display Diet_Min.lb, Diet_Min.body, Diet_Min.ub;
: Diet_Min.lb Diet_Min.body Diet_Min.ub :=
  A  700 1013.98 Infinity
  B1  0  605  Infinity
  B2  0  492.416 Infinity
  C  700  700  Infinity
  CAL 16000 16000 Infinity
;

ampl: display Diet_Max.lb, Diet_Max.body, Diet_Max.ub;
: Diet_Max.lb Diet_Max.body Diet_Max.ub :=
  A  -Infinity 1013.98 20000
  CAL  -Infinity 16000 24000
  NA  -Infinity 43855.9 50000
;
\end{verbatim}

Naturally, <= constraints have no lower bounds, and >= constraints have no upper bounds; AMPL uses -Infinity and Infinity in place of a number to denote these cases. Both the lower and the upper bound can be finite, if the constraint is specified with two <= or >= operators; see Section 8.4. For an = constraint the two bounds are the same.

The suffix .slack refers to the difference between the body and the nearer bound:

\begin{verbatim}
ampl: display Diet_Min.slack;
Diet_Min.slack [*] :=
  A  313.978
  B1  605
  B2  492.416
  C  0
  CAL  0
;
\end{verbatim}

For constraints that have a single <= or >= operator, the slack is always the difference between the expressions to the left and right of the operator, even if there are variables on both sides. The constraints that have a slack of zero are the ones that are truly constraining at the optimal solution.
**Dual values and reduced costs**

Associated with each constraint in a linear program is a quantity variously known as the dual variable, marginal value or shadow price. In the AMPL command environment, these dual values are denoted by the names of the constraints, without any qualifying suffix. Thus for example in Figure 4-6 there is a collection of constraints named Demand:

\[
\text{subject to Demand \{j in DEST, p in PROD\}:} \\
\quad \text{sum \{i in ORIG\} Trans[i,j,p] = demand[j,p];}
\]

and a table of the dual values associated with these constraints can be viewed by

```
ampl: display Demand;
Demand [*,*] :
    bands  coils  plate :=
DET  201  190.714  199
FRA  209  204   211
FRE  266.2 273.714  285
LAF  201.2 198.714  205
LAN  202  193.714   0
STL  206.2 207.714  216
WIN  200  190.714  198
```

Solvers return optimal dual values to AMPL along with the optimal values of the “primal” variables. We have space here only to summarize the most common interpretation of dual values; the extensive theory of duality and applications of dual variables can be found in any textbook on linear programming.

To start with an example, consider the constraint Demand["DET","bands"] above. If we change the value of the parameter demand["DET","bands"] in this constraint, the optimal value of the objective function Total_Cost changes accordingly. If we were to plot the optimal value of Total_Cost versus all possible values of demand["DET","bands"], the result would be a cost “curve” that shows how overall cost varies with demand for bands at Detroit.

Additional computation would be necessary to determine the entire cost curve, but you can learn something about it from the optimal dual values. After you solve the linear program using a particular value of demand["DET","bands"], the dual price for the constraint tells you the slope of the cost curve, at the demand’s current value. In our example, reading from the table above, we find that the slope of the curve at the current demand is 201. This means that total production and shipping cost is increasing at the rate of $201 for each extra ton of bands demanded at DET, or is decreasing by $201 for each reduction of one ton in the demand.

As an example of an inequality, consider the following constraint from the same model:

\[
\text{subject to Time \{i in ORIG\}:} \\
\quad \text{sum \{p in PROD\} (1/rate[i,p]) * Make[i,p] <= avail[i];}
\]
Here it is revealing to look at the dual values together with the slacks:

```
ampl: display Time, Time.slack;
:       Time      Time.slack :=
CLEV  -1522.86     0
GARY  -3040        0
PITT   0          10.5643
;
```

Where the slack is positive, the dual value is zero. Indeed, the positive slack implies that the optimal solution does not use all of the time available at PITT; hence changing `avail[*PITT*]` somewhat does not affect the optimum. On the other hand, where the slack is zero the dual value may be significant. In the case of GARY the value is −3040, implying that the total cost is decreasing at a rate of $3040 for each extra hour available at GARY, or is increasing at a rate of $3040 for each hour lost.

In general, if we plot the optimal objective versus a constraint’s constant term, the curve will be convex piecewise-linear (Figure 12-1) for a minimization, or concave piecewise-linear (the same, but upside-down) for a maximization.

In terms of the standard form `lower bound ≤ body ≤ upper bound` introduced previously, the optimal dual values can be viewed as follows. If the slack of the constraint is positive, the dual value is zero. If the slack is zero, the body of the constraint must equal one (or both) of the bounds, and the dual value pertains to the equaled bound or bounds. Specifically, the dual value is the slope of the plot of the objective versus the bound, evaluated at the current value of the bound; equivalently it is the rate of change of the optimal objective with respect to the bound value.

A nearly identical analysis applies to the bounds on a variable. The role of the dual value is played by the variable’s so-called reduced cost, which can be viewed from the AMPL command environment by use of the suffix `.rc`. As an example, here are the bounds and reduced costs for the variables in Figure 5-1:
ampl:  display Buy.lb, Buy, Buy.ub, Buy.rc;
:  Buy.lb  Buy  Buy.ub  Buy.rc :=
 BEEF  2  2  10  1.73663
 CHK  2  10  10 -0.853371
 FISH  2  2  10  0.255281
 HAM  2  2  10  0.698764
 MCH  2  2  10  0.246573
 MTL  2  6.23596 10  0
 SPG  2  5.25843 10  0
 TUR  2  2  10  0.343483
;

Since Buy["MTL"] has slack with both its bounds, its reduced cost is zero. Buy["HAM"] is at its lower bound, so its reduced cost indicates that total cost is increasing at about 70 cents per unit increase in the lower bound, or is decreasing at about 70 cents per unit decrease in the lower bound. On the other hand, Buy["CHK"] is at its upper bound, and its negative reduced cost indicates that total cost is decreasing at about 85 cents per unit increase in the upper bound, or is increasing at about 85 cents per unit decrease in the upper bound.

If the current value of a bound happens to lie right at a breakpoint in the relevant curve — one of the places where the slope changes abruptly in Figure 12-1 — the objective will change at one rate as the bound increases, but at a different rate as the bound decreases. In the extreme case either of these rates may be infinite, indicating that the linear program becomes infeasible if the bound is increased or decreased by any further amount. A solver reports only one optimal dual price or reduced cost to AMPL, however, which may be the rate in either direction, or some value between them.

In any case, moreover, a dual price or reduced cost can give you only one slope on the piecewise-linear curve of objective values. Hence these quantities should only be used as an initial guide to the objective’s sensitivity to certain variable or constraint bounds. If the sensitivity is very important to your application, you can make a series of runs with different bound settings; see Section 11.3 for ways to quickly change a small part of the data. (There do exist algorithms for finding part or all of the piecewise-linear curve, given a linear change in one or more bounds, but they are not directly supported by the current version of AMPL.)

12.6 Other display features for models and instances

We gather in this section various utility commands and other features for displaying information about models or about problem instances generated from models.

Two commands let you review an AMPL model from the command-line: show lists the names of model components and displays the definitions of individual components, while xref lists all components that depend on a given component. The expand command displays selected objectives and constraints that AMPL has generated from a model and data, or analogous information for variables. AMPL’s “generic” names for variables,
constraints, or objectives permit listings or tests that apply to all variables, constraints, or objectives.

**Displaying model components: the show command:**

By itself, the `show` command lists the names of all components of the current model:

```ampl
ampl: model multip3.mod;
ampl: show;
parameters: demand fcost limit maxserve minload supply vcost
sets: DEST ORIG PROD
variables: Trans Use
constraints: Demand Max_Serve Min_Ship Multi Supply
objective: Total_Cost
checks: one, called check 1.
```

This display may be restricted to components of one or more types:

```ampl
ampl: show vars;
variables: Trans Use
ampl: show obj, constr;
objective: Total_Cost
constraints: Demand Max_Serve Min_Ship Multi Supply
```

The `show` command can also display the declarations of individual components, saving you the trouble of looking them up in the model file:

```ampl
ampl: show Total_Cost;
minimize Total_Cost: sum{i in ORIG, j in DEST, p in PROD} vcost[i,j,p] * Trans[i,j,p] + sum{i in ORIG, j in DEST} fcost[i,j] * Use[i,j];
ampl: show vcost, fcost, Trans;
param vcost{ORIG, DEST, PROD} >= 0;
param fcost{ORIG, DEST} >= 0;
var Trans{ORIG, DEST, PROD} >= 0;
```

If an item following `show` is the name of a component in the current model, the declaration of that component is displayed. Otherwise, the item is interpreted as a component type according to its first letter or two; see Section A.19.1. (Displayed declarations may differ in inessential ways from their appearance in your model file, due to transformations that AMPL performs when the model is parsed and translated.)

Since the `check` statements in a model do not have names, AMPL numbers them in the order that they appear. Thus to see the third check statement you would type

```ampl
ampl: show check 3;
check{p in PROD} :
  sum{i in ORIG} supply[i,p] == sum{j in DEST} demand[j,p];
```

By itself, `show checks` indicates the number of check statements in the model.
Displaying model dependencies: the xref command

The \texttt{xref} command lists all model components that depend on a specified component, either directly (by referring to it) or indirectly (by referring to its dependents). If more than one component is given, the dependents are listed separately for each. Here is an example from \texttt{multmip3.mod}:

\begin{verbatim}
ampl: \texttt{xref demand, Trans;}
# 2 entities depend on demand:
check 1
Demand
# 5 entities depend on Trans:
Total_Cost
Supply
Demand
Multi
Min_Ship
\end{verbatim}

In general, the command is simply the keyword \texttt{xref} followed by a comma-separated list of any combination of set, parameter, variable, objective and constraint names.

Displaying model instances: the expand command

In checking a model and its data for correctness, you may want to look at some of the specific constraints that AMPL is generating. The \texttt{expand} command displays all constraints in a given indexed collection or specific constraints that you identify:

\begin{verbatim}
ampl: \texttt{model nltrans.mod;}
ampl: \texttt{data nltrans.dat;}
ampl: \texttt{expand Supply;}
subject to Supply[‘GARY’]:
Trans[‘GARY’,‘FRA’] + Trans[‘GARY’,‘DET’] +
Trans[‘GARY’,‘LAN’] + Trans[‘GARY’,‘WIN’] +
Trans[‘GARY’,‘STL’] + Trans[‘GARY’,‘FRE’] +
Trans[‘GARY’,‘LAF’] = 1400;
subject to Supply[‘CLEV’]:
Trans[‘CLEV’,‘FRA’] + Trans[‘CLEV’,‘DET’] +
Trans[‘CLEV’,‘LAN’] + Trans[‘CLEV’,‘WIN’] +
Trans[‘CLEV’,‘STL’] + Trans[‘CLEV’,‘FRE’] +
Trans[‘CLEV’,‘LAF’] = 2600;
subject to Supply[‘PITT’]:
Trans[‘PITT’,‘FRA’] + Trans[‘PITT’,‘DET’] +
Trans[‘PITT’,‘LAN’] + Trans[‘PITT’,‘WIN’] +
Trans[‘PITT’,‘STL’] + Trans[‘PITT’,‘FRE’] +
Trans[‘PITT’,‘LAF’] = 2900;
\end{verbatim}

(See Figures 18-4 and 18-5.) The ordering of terms in an expanded constraint does not necessarily correspond to the order of the symbolic terms in the constraint’s declaration.

Objectives may be expanded in the same way:
ampl: `expand Total_Cost;`
minimize Total_Cost:
  39*Trans['GARY','FRA']/(1 - Trans['GARY','FRA']/500) + 14*
  Trans['GARY','DET']/(1 - Trans['GARY','DET']/1000) + 11*
  Trans['GARY','LAN']/(1 - Trans['GARY','LAN']/1000) + 14*
  Trans['GARY','WIN']/(1 - Trans['GARY','WIN']/1000) + 16*
  ... 15 lines omitted
  Trans['PITT','FRE']/(1 - Trans['PITT','FRE']/500) + 20*
  Trans['PITT','LAF']/(1 - Trans['PITT','LAF']/900);

When `expand` is applied to a variable, it lists all of the nonzero coefficients of that variable in the linear terms of objectives and constraints:

ampl: `expand Trans;`
Coefficients of Trans['GARY','FRA']:
Supply['GARY'] 1
Demand['FRA'] 1
Total_Cost 0 + nonlinear
Coefficients of Trans['GARY','DET']:
Supply['GARY'] 1
Demand['DET'] 1
Total_Cost 0 + nonlinear
Coefficients of Trans['GARY','LAN']:
Supply['GARY'] 1
Demand['LAN'] 1
Total_Cost 0 + nonlinear
Coefficients of Trans['GARY','WIN']:
Supply['GARY'] 1
Demand['WIN'] 1
Total_Cost 0 + nonlinear
  ... 17 terms omitted

When a variable also appears in nonlinear expressions within an objective or a constraint, the term `+ nonlinear` is appended to represent those expressions.

The command `expand` alone produces an expansion of all variables, objectives and constraints in a model. Since a single `expand` command can produce a very long listing, you may want to redirect its output to a file by placing `>filename` at the end as explained in Section 12.7 below.

The formatting of numbers in the expanded output is governed by the options `expand_precision` and `expand_round`, which work like the `display` command’s `display_precision` and `display_round` described in Section 12.3.

The output of `expand` reflects the “modeler’s view” of the problem; it is based on the model and data as they were initially read and translated. But AMPL’s presolve phase (Section 14.1) may make significant simplifications to the problem before it is sent to the solver. To see the expansion of the “solver’s view” of the problem following presolve, use the keyword `solexpend` in place of `expand`.
Generic synonyms for variables, constraints and objectives

Occasionally it is useful to make a listing or a test that applies to all variables, constraints, or objectives. For this purpose, AMPL provides automatically updated parameters that hold the numbers of variables, constraints, and objectives in the currently generated problem instance:

\_nvars number of variables in the current problem
\_ncons number of constraints in the current problem
\_nobjs number of objectives in the current problem

Correspondingly indexed parameters contain the AMPL names of all the components:

\_varname\{1..\_nvars\} names of variables in the current problem
\_conname\{1..\_ncons\} names of constraints in the current problem
\_objname\{1..\_nobjs\} names of objectives in the current problem

Finally, the following synonyms for the components are made available:

\_var\{1..\_nvars\} synonyms for variables in the current problem
\_con\{1..\_ncons\} synonyms for constraints in the current problem
\_obj\{1..\_nobjs\} synonyms for objectives in the current problem

These synonyms let you refer to components by number, rather than by the usual indexed names. Using the variables as an example, \_var\[5\] refers to the fifth variable in the problem, \_var\[5\].ub to its upper bound, \_var\[5\].rc to its reduced cost, and so forth, while \_varname\[5\] is a string giving the variable’s true AMPL name. Table A-13 lists all of the generic synonyms for sets, variables, and the like.

Generic names are useful for tabulating properties of all variables, where the variables have been defined in several different var declarations:

ampl: model net3.mod
ampl: data net3.dat
ampl: solve;
MINOS 5.5: optimal solution found.
3 iterations, objective 1819

ampl: display \{j in 1..\_nvars\}
ampl? (\_varname\[j\],\_var\[j\],\_var\[j\].ub,\_var\[j\].rc);

: \_varname\[j\] \_var\[j\] \_var\[j\].ub \_var\[j\].rc :=
1 "PD_Ship[‘NE’]" 250 250 -0.5
2 "PD_Ship[‘SE’]" 200 250 -1.11022e-16
3 "DW_Ship[‘NE’,’BOS’]" 90 90 0
4 "DW_Ship[‘NE’,’EWR’]" 100 100 -1.1
5 "DW_Ship[‘NE’,’BWI’]" 60 100 0
6 "DW_Ship[‘SE’,’EWR’]" 20 100 2.22045e-16
7 "DW_Ship[‘SE’,’BWI’]" 60 100 2.22045e-16
8 "DW_Ship[‘SE’,’ATL’]" 70 70 0
9 "DW_Ship[‘SE’,’MCO’]" 50 50 0;
Another use is to list all variables having some property, such as being away from the upper bound in the optimal solution:

```ampl
ampl: display {j in 1.._nvars: _var[j] < _var[j].ub - 0.00001} _varname[j];
```

The same comments apply to constraints and objectives. More precise formatting of this information can be obtained with `printf` instead of `display`.

As in the case of the `expand` command, these parameters and generic synonyms reflect the modeler's view of the problem; their values are determined from the model and data as they were initially read and translated. AMPL's presolve phase may make significant simplifications to the problem before it is sent to the solver. To work with parameters and generic synonyms that reflect the solver's view of the problem following presolve, replace `_` by `_s` in the names given above; for example in the case of variables, use `_snvars, _svarname` and `_svar`.

Additional predefined sets and parameters represent the names and dimensions (arities) of the model components. They are summarized in A.19.4.

### Resource listings

Changing option `show_stats` from its default of 0 to a nonzero value requests summary statistics on the size of the optimization problem that AMPL generates:

```ampl
ampl: model steelT.mod;
ampl: data steelT.dat;
ampl: option show_stats 1;
ampl: solve;
```

Presolve eliminates 2 constraints and 2 variables. Adjusted problem:

- 24 variables, all linear
- 12 constraints, all linear; 38 nonzeros
- 1 linear objective; 24 nonzeros.

MINOS 5.5: optimal solution found.
15 iterations, objective 515033

Additional lines report the numbers of integer and variables and nonlinear components where appropriate.

Changing option `times` from its default of 0 to a nonzero value requests a summary of the AMPL translator's time and memory requirements. Similarly, by changing option `gentimes` to a nonzero value, you can get a detailed summary of the resources that AMPL's `genmod` phase consumes in generating a model instance.
When AMPL appears to hang or takes much more time than expected, the display produced by gentimes can help associate the difficulty with a particular model component. Typically, some parameter, variable or constraint has been indexed over a set far larger than intended or anticipated, with the result that excessive amounts of time and memory are required.

The timings given by these commands apply only to the AMPL translator, not to the solver. A variety of predefined parameters (Table A-14) let you work with both AMPL and solver times. For example, _solve_time always equals total CPU seconds required for the most recent solve command, and _ampl_time equals total CPU seconds for AMPL excluding time spent in solvers and other external programs.

Many solvers also have directives for requesting breakdowns of the solve time. The specifics vary considerably, however, so information on requesting and interpreting these timings is provided in the documentation of AMPL’s links to individual solvers, rather than in this book.

12.7 General facilities for manipulating output

We describe here how some or all of AMPL’s output can be directed to a file, and how the volume of warning and error messages can be regulated.

Redirection of output

The examples in this book all show the outputs of commands as they would appear in an interactive session, with typed commands and printed responses alternating. You may direct all such output to a file instead, however, by adding a > and the name of the file:

ampl:  display ORIG, DEST, PROD >multi.out;
ampl:  display supply >multi.out;

The first command that specifies >multi.out creates a new file by that name (or overwrites any existing file of the same name). Subsequent commands add to the end of the file, until the end of the session or a matching close command:

ampl:  close multi.out;

To open a file and append output to whatever is already there (rather than overwriting), use >> instead of >. Once a file is open, subsequent uses of > and >> have the same effect.

Output logs

The log_file option instructs AMPL to save subsequent commands and responses to a file. The option’s value is a string that is interpreted as a filename:

ampl: option log_file ‘multi.tmp’;
The log file collects all AMPL statements and the output that they produce, with a few exceptions described below. Setting log_file to the empty string:

```
   ampl: option log_file '';
```

turns off writing to the file; the empty string is the default value for this option.

When AMPL reads from an input file by means of a model or data command (or an include command as defined in Chapter 13), the statements from that file are not normally copied to the log file. To request that AMPL echo the contents of input files, change option log_model (or input in model mode) or log_data (for input in data mode) from the default value of 0 to a nonzero value.

When you invoke a solver, AMPL logs at least a few lines summarizing the objective value, solution status and work required. Through solver-specific directives, you can typically request additional solver output such as logs of iterations or branch-and-bound nodes. Many solvers automatically send all of their output to AMPL’s log file, but this compatibility is not universal. If a solver’s output does not appear in your log file, you should consult the supplementary documentation for that solver’s AMPL interface; possibly that solver accepts nonstandard directives for diverting its output to files.

**Limits on messages**

By specifying option eexit $n$, where $n$ is some integer, you determine how AMPL handles error messages. If $n$ is not zero, any AMPL statement is terminated after it has produced $\left| n \right|$ error messages; a negative value causes only the one statement to be terminated, while a positive value results in termination of the entire AMPL session. The effect of this option is most often seen in the use of model and data statements where something has gone badly awry, like using the wrong file:

```
   ampl: option eexit -3;
   ampl: model diet.mod;
   ampl: data diet.mod;

   diet.mod, line 4 (offset 32):
    expected ; ( [ ; or symbol
    context:  param cost >>> { <<< FOOD} > 0;

diet.mod, line 5 (offset 56):
    expected ; ( [ ; or symbol
    context:  param f_min >>> { <<< FOOD} >= 0;

diet.mod, line 6 (offset 81):
    expected ; ( [ ; or symbol
    context:  param f_max >>> { <<< j in FOOD} >= f_min[j];

   Bailing out after 3 warnings.
```

The default value for eexit is $-10$. Setting it to 0 causes all error messages to be displayed.

The eexit setting also applies to infeasibility warnings produced by AMPL’s presolve phase after you type solve. The number of these warnings is simultaneously lim-
ited by the value of option `presolve_warnings`, which is typically set to a smaller value; the default is 5.

An AMPL `data` statement may specify values that correspond to illegal combinations of indices, due to any number of mistakes such as incorrect index sets in the model, indices in the wrong order, misuse of `(tr)`, and typing errors. Similar errors may be caused by `let` statements that change the membership of index sets. AMPL catches these errors after `solve` is typed. The number of invalid combinations displayed is limited to the value of the option `bad_subscripts`, whose default value is 3.