Using AMPL Under UNIX

This booklet supplements *AMPL: A Modeling Language for Mathematical Programming* for users of AMPL on computers running the UNIX® operating system. The AMPL software consists of an implementation of the AMPL command environment and language processor that may be used with a variety of solvers. All features of AMPL and the solvers are supported. This booklet describes the installation and use of AMPL under UNIX. Separate booklets are available for each solver; they contains advice on using the solvers, and on options specific to an individual solver or algorithm.

Student and professional editions of AMPL are available on a variety of computers, in conjunction with a variety of solvers. For full information, contact:

Boyd & Fraser Publishing Company
55 Ferncroft Road
Danvers, MA 01923

800-225-3782; 508-777-9069; fax 508-777-9068

Electronic mail: `ampl@research.att.com`

Using AMPL

The AMPL software described in this booklet runs under any version of the UNIX operating system.

Section §1 explains how to install AMPL. The remainder of this booklet covers system-specific aspects of running AMPL under UNIX, including options to the `ampl` command, memory requirements, and management of files.

§1 Installation

The AMPL software distribution consists of AMPL itself, any solvers that may be included, and a directory `Models` of models and data from the AMPL book. These files are distributed on floppy disk in `tar` format. To extract the files, put the distribution disk into a floppy drive, `cd` to a suitable directory (for example, `ampl` or your own `bin`), and type the command

```
tar xvf /dev/rfd0c
```

The `tar` command extracts the AMPL software into the current directory. (The name of the floppy disk drive, here shown as `/dev/rfd0c`, may be different on different systems.)

The material on the distribution disk is compressed. To uncompress, type the command

```
uncompress *.Z
```

To test that the installation is complete, try solving one of the small sample problems from the distribution disk:

```
% ampl
ampl: model Models/steel.mod;
ampl: data Models/steel.dat;
ampl data: solve;
MINOS 5.4: optimal solution found.
2 iterations, objective 192000
ampl: option solver osl;
ampl: solve;
OSL 1.2:
 The primal algorithm has been chosen
OSL 1.2: optimal solution; objective 192000
1 simplex iterations
ampl: quit;
%
```

(You may have a different set of solvers on your system.)

Eventually you will want to install these programs in some more public directory, such as `/usr/local/bin`; in any case, you may wish to add your new AMPL directory to your search path, so that you can invoke AMPL from any directory.

§2 Running AMPL

Once the installation is complete, you can run AMPL by typing `ampl` at the prompt, as shown in the preceding example. (We have shown the prompt as `%`, which is typical if you use the C shell; if you use the Bourne shell or the Korn shell, the prompt is likely to be `$`.) The AMPL command `quit` returns you to the UNIX prompt. For many modeling projects, this is all you need to know about invoking AMPL.

The `ampl` command has longer forms that are convenient or necessary in certain circumstances. The general form of the command is:

`ampl` *switch-list*$_{opt}$ *filename-list*$_{opt}$

The optional *switch-list* contains items that turn certain AMPL features on or off. These items begin with the `-` character. Such items are often called ''command-line options'', but here we call them ''switches'' to distinguish them from the ''options'' controlled by AMPL's `option` command. The items in the *filename-list* are filenames, or the `-` character alone. One or more spaces must separate each item from the next. We describe the switches first, and then the use of the filenames.

AMPL can also be instructed through UNIX environment variables to initialize certain options, or to read an initialization file automatically upon invocation. These features are explained at the end of this section.

Switches

Many command-line switches are simply synonyms for options that can be set within the AMPL command environment. For example, the `-P` switch,

```
% ampl -P
ampl:
```

has the same effect as setting the `presolve` option to zero:

```
% ampl
ampl: option presolve 0;
ampl:
```

Table 1 lists all of these alternatives. The `-v` switch reports the version of AMPL you are running, and the `-?` switch provides a summary of command-line switches.

Switch	Option	Interpretation
`-C`*n*	`Cautions` *n*	$n = 0$: suppress caution messages $n = 1$: report caution messages (default) $n = 2$: treat cautions as errors
`-e`*n*	`eexit` *n*	$n > 0$: abandon a command after n errors $n < 0$: abort AMPL after $\lvert n \rvert$ errors $n = 0$: report any number of errors
`-L`	`linelim 1`	when using ‘‘defining’’ equations to substitute a linear expression for a variable, make an explicit substitution, so that the resulting model can be recognized as linear
`-P`	`presolve 0`	turn off the presolve phase
`-s`	`randseed ''`	use current time for random number seed
`-s`*n*	`randseed` *n*	use n for random number seed
`-S`	`substout 1`	use ‘‘defining’’ equations to eliminate variables
`-T`	`gentimes 1`	show the time taken to generate each model component
`-t`	`times 1`	show the time taken in each model translation phase

Table 1: Command-line switches.

Filenames

If you give one or more filenames on the command line, AMPL will read model declarations, data statements and commands from these files in the order given, instead of entering the usual command environment. This provides a way of running AMPL as a ‘‘batch’’ program, rather than interactively. For example, the command

```
% ampl diet.mod case2/diet.dat /tmp/diet.run
%
```

accomplishes the same thing as

```
% ampl
ampl: include diet.mod;
ampl: include case2/diet.dat;
ampl: include /tmp/diet.run;
ampl: quit;
%
```

We assume that `diet.mod` contains declarations for an AMPL model, `case2/diet.dat` contains a `data` statement followed by data for a particular case, and `/tmp/diet.run` contains AMPL commands for solving the resulting mathematical program and reporting the results. This is only one possible arrangement, however; the contents of all three files might instead be concatenated into one, and in general any command-line file may contain a combination of declarations, data and commands. (We have used filenames compatible with the ones employed on MS-DOS systems, but `ampl` accepts any valid UNIX filenames.)

To combine reading from command-line files with interactive operation of AMPL, place the character `-` in the *filename-list* to indicate where interactive operation is to begin. Most commonly, it appears at the end; to read files `diet.mod` and `case2/diet.dat`, and then give further instructions interactively, you would type:

```
% ampl diet.mod case2/diet.dat -
ampl:
```

A `-` character can precede another filename, however, if you want to type a few lines interactively before proceeding to read the next file:

```
% ampl diet.mod case2/diet.dat - /tmp/diet.run
ampl: let n_max['NA'] := 50000;
ampl: end;
%
```

The `end` command terminates the interactive environment, after which commands are read from `/tmp/diet.run`. The `quit` command, by contrast, would terminate AMPL before the last file could be read.

Environment variables and initialization files

Within a UNIX shell you may define ''environment variables'' that retain their values as long as the shell is active. When you invoke AMPL, each environment variable is interpreted as an AMPL option. For example, you could set the AMPL options `display_width` and `osl_options` by defining the corresponding environment variables like this in a C shell session,

```
% setenv display_width 60
% setenv osl_options 'dual scale 1'
% ampl
```

or like this in a Bourne or Korn shell session:

```
$ display_width=60; export display_width
$ osl_options='dual scale 1'; export osl_options
$ ampl
```

AMPL's option command confirms that the desired settings have been made:

```
ampl: option display_width, osl_options;
option display_width 60;
option osl_options 'dual scale 1';
```

Option values ''inherited'' from environment variables in this way always override any default values that the options would otherwise have.

This feature can provide a convenient means to redefine a few AMPL option defaults. You need only define the appropriate environment variables once, when you log in or start up a new shell window. After that, their values will be inherited each time you invoke AMPL. The inheritance only works in one direction, so that changing an option's value in AMPL never affects any environment variable in UNIX.

If you have many option settings or other initializations that you want to carry out each time AMPL is invoked, you may wish to keep a list of startup commands in a file. AMPL takes the name of this file from the environment variable `OPTIONS_IN`, if such a variable has been defined. In the C shell, the command is

```
% setenv OPTIONS_IN amplinit
```

With the Bourne and Korn shells, it is

```
$ OPTIONS_IN=amplinit; export OPTIONS_IN
```

AMPL will execute the contents of the file `amplinit` before reading any other files or prompting for any commands. The effect is the same as if `include amplinit` were the first command processed by AMPL.

If you want AMPL to remember all your option settings from one invocation to the next, first set up an options command file by running a short AMPL session like this:

```
% ampl
ampl: option OPTIONS_INOUT 'amplopt';
ampl: quit;
%
```

AMPL writes to the specified file, `amplopt` in this case, a list of `option` commands that set all options to the values they had when you typed `quit`. To use this file, set the corresponding environment variable to the same filename; for the C shell:

```
% setenv OPTIONS_INOUT amplopt
```

and for Bourne and Korn shells:

```
$ OPTIONS_INOUT=amplopt; export OPTIONS_INOUT
```

At each subsequent invocation of AMPL, the previous option settings will be read from the specified file; and each time thereafter that you quit AMPL, the current option settings — including any changes you might have made — will be written back to the file. The effect is to preserve all option values from one invocation to the next.

To save the trouble of typing the commands to set your AMPL environment variables each time you log in, place the commands in your initialization file (normally `.cshrc` for the C shell or `.profile` for other shells).

The `OPTIONS_IN` file is read before the `OPTIONS_INOUT` file, if both are specified.

§3 Memory

The amount of memory (not disk space) required by AMPL includes an initial memory region upon invocation, plus additional memory for generating instances and running solvers. The total memory requirement thus varies according to the problem that you are trying to solve, and tends to increase as an AMPL session proceeds.

Most obviously, a larger problem instance will require more memory to generate and solve. The size depends upon several factors, including the number of variables, the number of objectives and constraints, and the number of terms in the objective and constraint expressions.

AMPL's memory requirements also depend in a less straightforward way upon the complexity of a model. Memory space is needed for every model component, including sets that are not directly used to index anything, parameters that are defined in terms of other data, and variables that are later removed by the presolve phase. The memory requirement for processing a model can thus be much larger than the size of the resulting instance would suggest. For example, when a `var` declaration is indexed over a Cartesian product of many sets,

```
var Ship {i in ORIG, j in DEST, p in PROD, q in AREA[p], t in 1..T} ...
```

some memory must be set aside for each resulting variable, even if the great majority of these variables are later fixed to zero by the constraints and eliminated by presolve. If the total memory is insufficient, you may need to reformulate the model so that the variable is declared over a subset of tuples `(i,j,p,q,t)`, as explained in Chapter 6 of the AMPL book.

Since the need for memory ultimately depends upon the model, the data, and any commands you give, AMPL cannot determine in advance how much memory it might require. Most often, insufficient memory is reflected in slow performance, since the operating system must page information in and out. If the system imposes limits on process size, you might also encounter an out-of-memory message, either directly after `solve` or another AMPL command (if AMPL cannot get enough memory), or after some solver output (if the solver cannot get enough memory). Certain algorithms within some solvers may have particularly large additional requirements for memory; for details, see the discussions of the algorithms in solver-specific booklets.

Clearly there is no single best way to remedy a problem of insufficient memory. We suggest that you first review your model and data, to ensure that you are not creating unnecessarily great requirements for memory. If the out-of-memory message comes from the solver, you should also investigate whether solver-specific directives might reduce the memory needed.

If no reformulations or new directives seem to help, try scaling back your data to see how large a problem does fit in your computer's memory. Such an exercise may provide some estimate of how much more memory you would need, or may help you to discover which components of your model are making the greatest demands for memory allocations.

§4 Files

When you type `solve`, AMPL generates an instance in memory, then writes a description to one or more temporary files. The solver reads these files, applies an algorithm, and writes its results to another temporary file, which is in turn read by AMPL. Finally the temporary files are all deleted. In normal use these steps occur automatically, and you need not give any thought to them.

This section describes two circumstances in which you may want to be more aware of the files that AMPL is creating: when the temporary files do not fit in the file system, and when you want to save a solution for later inspection.

Relocating temporary files

If your file system is getting full, it may not have enough space to hold AMPL's temporary files, and you will get an error message to that effect after typing `solve`. To get around this problem, you will have to either free more space, or direct the temporary files to another file system.

The option `TMPDIR` specifies a directory to which temporary files will be written. When it is left at its default setting, the temporary files are written to `/tmp`. If there were also a filesystem `/moretmp` having more free space, for example, you could send the temporary files there by typing

```
ampl: option TMPDIR "/moretmp";
```

You could also send the files to one of your own directories, say `mytemp`, by setting `TMPDIR` to `mytemp`.

Saving solutions

Since a solver called from AMPL normally writes its results into a temporary file, no permanent record is kept. You can save any portion of the results by redirecting the output of `display`, `print` or `printf` to a file (simply append `>` *filename* to the command), but the solution is otherwise inaccessible once you specify `reset` or `quit`.

To save a solution for subsequent examination, you may use the `write` command to override the creation of temporary files. For example, if you have a model in `diet.mod` and data in `diet.dat`, you could type the following:

```
% ampl
ampl: model diet.mod; data diet.dat;
ampl data: write bdietrun;
ampl: solve;
MINOS 5.4: optimal solution found.
6 iterations, objective 88.2
ampl: quit;
% ls -l dietrun*
-rw-r--r--  1 bwk          1221 Nov 16 15:40 dietrun.nl
-rw-r--r--  1 bwk           252 Nov 16 15:40 dietrun.sol
%
```

The first character of the string following `write` is interpreted specially, as explained below. The rest of the string is combined with different extensions to produce the names for the files that AMPL uses. The `ls -l` listing above shows that two files have been created in this case: `dietrun.nl`, which contains the problem description that was sent to the solver, and `dietrun.sol`, which contains the result description that was sent back. Because `write` was used explicitly, these files take the place of the temporary ones that would normally be generated, and they are not automatically deleted.

To view the results in `dietrun.sol` at a later time, you would use the `solution` command:

```
% ampl
ampl: model diet.mod; data diet.dat;
ampl data: solution dietrun.sol;
MINOS 5.4: optimal solution found.
6 iterations, objective 88.2
ampl: display Buy;
Buy [*] :=
BEEF   0
 CHK   0
FISH   0
 HAM   0
 MCH  46.6667
 MTL  -1.07823e-16
 SPG  -1.32893e-16
 TUR   0
;
```

You do have to repeat the `model` and `data` statements, so that AMPL sets up the appropriate instance again. But `solution` then gets the old results directly from the specified file, without running any solver.

When `b` is the first character of the string that follows `write`, as above, AMPL uses a compact and efficient binary format for the files. When `g` appears instead, AMPL uses a compact text format that may be easier to transfer between computers. There is also an `m` option that causes AMPL to write linear problems in the widely recognized ''MPS format''; the resulting filename ends in `mps` rather than `nl`. These files may be useful for communicating test problems to solvers that do not yet have an AMPL interface.

The AMPL option `auxfiles` lets you request creation of several auxiliary files by the `write` command:

key	extension	file contents
`a`	`.adj`	adjustment to objective, e.g., to compensate for fixed variables eliminated by presolve
`c`	`.col`	names of the variables (columns) sent to the solver
`f`	`.fix`	names of variables fixed by presolve, and the values to which they are fixed
`r`	`.row`	names of the constraints (rows) sent to the solver
`s`	`.slc`	names of ''slack'' constraints eliminated by presolve because they can never be binding
`u`	`.unv`	names of variables dropped by presolve because they are never used in the problem instance

If you set `auxfiles` to a string containing one or more of the specified key letters, `write` creates the corresponding file with the specified extension (unless it would be empty). For example, if you type

```
ampl: option auxfiles "cr";
```

before `solve` in our example above, the files `dietrun1.col` and `dietrun1.row` are created and the names of the variables and constraints, respectively, are written to them.