Hooking Your Solver to AMPL

David M. Gay

AMPL Optimization, Inc.
900 Sierra Place SE
Albuquerque, NM 87108

ABSTRACT

This report tells how to make solvers work with AMPL's `solve` command. It describes an interface library, `amplsolver.a`, whose source is available from the AMPL web site in `http://ampl.com/netlib/ampl` (where updates first appear) and from *netlib*. Examples include programs for listing LPs, automatic conversion to the LP dual (shell-script as solver), solvers for various nonlinear problems (with first and sometimes second derivatives computed by automatic differentiation), and getting C or Fortran 77 for nonlinear constraints, objectives, and their first derivatives. Drivers for various well known linear, mixed-integer, and nonlinear solvers provide more examples.

CONTENTS

1. Introduction

The AMPL modeling system [6, 7] lets you express constrained optimization problems in an algebraic notation close to conventional mathematics. AMPL's `solve` command causes AMPL to instantiate the current problem, send it to a solver, and attempt to read a solution computed by the solver (for use in subsequent commands, e.g., to print values computed from the solution). This technical report tells how to arrange for your own solver to work with AMPL's `solve` command. See the AMPL web site (http://ampl.com) for much more information about AMPL.

Stub`.nl` files

AMPL runs solvers as separate programs and communicates with them by writing and reading files. The files have names of the form *stub*`.`*suffix*; AMPL usually chooses the *stub* automatically, but one can specify the *stub* explicitly with AMPL's `write` command. Before invoking a solver, AMPL normally writes a file named *stub*`.nl`. This file contains a description of the problem to be solved. AMPL invokes the solver with two arguments, the *stub* and a string whose first five characters are `-AMPL`, and expects the solver to write a file named *stub*`.sol` containing a termination message and the solution it has found.

Most linear programming solvers are prepared to read MPS files, which are described, e.g., in chapter 9 of [16]; see also the linear programming FAQ at

```
https://neos-guide.org/content/lp-faq
```

AMPL can be provoked to write an MPS file, *stub*`.mps`, rather than *stub*`.nl`, but MPS files are slower to read and write, entail loss of accuracy (because of rounding to fit numbers into 12-column fields), and can only describe linear and mixed-integer problems (with some differences in interpretations among solvers for the latter). AMPL's *stub*`.nl` files, on the other hand, contain a complete and unambiguous problem description of both linear and nonlinear problems, and they introduce no new rounding errors.

In the following, we assume you are familiar with C and that your solver is callable from C or C++. If your solver is written in some other language, it is probably callable from C, though the details are likely to be system-dependent. If your solver is written in Fortran 77, you can make the details system-independent by running your Fortran source through the Fortran-to-C converter *f2c* [5]; see `http://ampl.com/netlib/f2c` or `http://www.netlib.org/f2c` for more information, including source.

The AMPL/solver interface directory,

```
http://ampl.com/netlib/ampl/solvers
```

which is simply called `solvers`, in most of this document, contains some useful header files and source for a library, `amplsolver.a`, of routines for reading *stub*`.nl` and writing *stub*`.sol` files. Much of the rest of this report is about using routines in `amplsolver.a`. The entire `solvers` directory (without

subdirectories) is available as the gzipped tar file

```
http://ampl.com/netlib/ampl/solvers.tgz
```

A variant called `solvers2`, available as

```
http://ampl.com/netlib/ampl/solvers2.tgz
```

is useful with multiple threads, as explained at the end of this report. For large problems, nonlinear evaluations are often faster with `solvers2` than with `solvers`, and the representations of nonlinearities take less memory, even with a single thread.

Material for many of the examples discussed here is in the `solvers/examples` directory, henceforth simply called ‘‘`examples`’’, available via such URLs as

```
http://ampl.com/netlib/ampl/solvers/examples
http://www.netlib.org/ampl/solvers/examples
```

or as the gzipped tar file `http://ampl.com/netlib/ampl/solvers/examples.tgz`. You will find it helpful to look at the files in `examples` while reading this report.

A ‘‘`README`’’ file in the `solvers` directory discusses compiling `amplsolver.a`, which on MS Windows systems is called `amplsolv.lib`. The instructions below for compiling examples assume you have already built `amplsolver.a` or `amplsolv.lib`, as appropriate.

2. Linear Problems

Row-wise treatment

For simplicity, we first consider linear programming (LP) problems. Solvers can view an LP as the problem of finding $x \in \mathbb{R}^n$ to

$$\begin{aligned} \text{minimize or maximize} \quad & c^T x \\ \text{subject to} \quad & b \le Ax \le d \\ \text{and} \quad & \ell \le x \le u \end{aligned} \tag{LP}$$

where $A \in \mathbb{R}^{m \times n}$, b, $d \in \mathbb{R}^m$, and c, ℓ, $u \in \mathbb{R}^n$. The initial examples of reading linear problems just print out the data (A, b, c, d, ℓ, u) and perhaps the primal and dual initial guesses.

The first example, `examples/lin1.c`, just prints the data. The `examples` directory contains *makefile* variants `makefile.u` for Unix and Linux and `makefile.vc` for MS Windows. These *makefile* variants have rules for compiling and linking various examples. On a Unix or Linux system, initially give the command

```
cp makefile.u makefile
```

and under MS Windows, in a Commands window give the command

```
copy makefile.vc makefile
```

If necessary, you can then make changes to ‘‘`makefile`’’. For example, `makefile.u` and `makefile.vc` assume that `solvers` is the parent directory. If `solvers` is somewhere else, you should change the line

```
S = ..
```

suitably, replacing ‘‘`..`’’ with a relative or absolute path to your `solvers` or `solvers2` directory. After this preparation, under Unix or Linux, invoke

```
make lin1
```

to compile and load the ‘‘`lin1`’’ program. Under MS Windows, the corresponding command would be

```
nmake lin1.exe
```

File `lin1.c` starts with

```
#include "asl.h"
```

i.e., asl.h from your solvers directory, as specified by the ''S = ...'' line in the makefile. The phrase ''asl'' or ''ASL'' that appears in many names stands for *A*MPL/*S*olver interface *L*ibrary). In turn, asl.h includes various standard header files: math.h, stdio.h, string.h, stdlib.h, setjmp.h, and errno.h. Among other things, asl.h defines type ASL for a structure that holds various problem-specific data, and asl.h provides a long list of #defines to facilitate accessing items in an ASL when a pointer asl declared

```
ASL *asl;
```

is in scope. Among the components of an ASL are various pointers and such integers as the numbers of variables (n_var), constraints (n_con), and objectives (n_obj). Most higher-level interface routines have their prototypes in asl.h, and a few more appear in getstub.h, which is discussed later. Also defined in asl.h are the types Long (usually the name of a 32-bit integer type, which is usually long or int), fint (''Fortran integer'', normally a synonym for Long), real (normally a synonym for double), and ftnlen (also normally a synonym for Long, and used to convey string lengths to Fortran 77 routines that follow the *f2c* calling conventions).

The main routine in lin1.c expects to see one command-line argument: the *stub* of file *stub*.nl written by AMPL, as explained above. After checking that it has a command-line argument, the main routine allocates an ASL via

```
asl = ASL_alloc(ASL_read_f);
```

the argument to ASL_alloc determines how nonlinearities are handled and is discussed further below in the section headed ''Reading nonlinear problems''. The main routine appears to pass the *stub* to interface routine jac0dim, with prototype

```
FILE *jac0dim(char *stub, fint stub_len);
```

in reality, a #define in asl.h turns the call

```
jac0dim(stub, stublen)
```

into

```
jac0dim_ASL(asl, stub, stublen) .
```

There are analogous #defines in asl.h for most other high-level routines in amplsolver.a, but for simplicity, we henceforth just show the apparent prototypes (without leading asl arguments). This scheme makes code easier to read and preserves the usefulness of solver drivers written before the _ASL suffix was introduced.

For use with Fortran programs, jac0dim assumes stub is stublen characters long and does not assume stub is null-terminated. After trimming any trailing blanks from stub (by allocating space for ASL field filename, i.e., asl->i.filename_, and copying stub there as the *stub*), jac0dim reads the first part of *stub*.nl and records some numbers in *asl, as summarized in Table 1. If *stub*.nl does not exist, jac0dim by default prints an error message and stops execution (but setting return_nofile to a nonzero value changes this behavior: see Table 2 below).

To read the rest of *stub*.nl, lin1.c calls f_read. As discussed more fully below and shown in Table 6, several routines are available for reading *stub*.nl, one for each possible argument to ASL_alloc; f_read just reads linear problems, complaining and aborting execution if it sees any nonlinearities. To acquire temporary memory — freed before f_read returns — f_read calls Malloc, which appears in most of our examples; Malloc calls malloc and aborts execution if malloc returns 0. To acquire memory that remains allocated when f_read returns, f_read calls M1alloc, which uses Malloc and records the allocation, so all memory allocated by M1alloc can be freed automatically when ASL_free(&asl) is called. The reason for breaking the reading of *stub*.nl into two steps will be seen in more detail below: sometimes it is convenient to modify the behavior of the *stub*.nl reader — here f_read — by allocating problem-dependent arrays before calling it.

AMPL may transmit several objectives. The linear part of each is contained in a list of ograd structures (declared in asl.h; note that asl.h declares

Component	Meaning
`n_var`	number of variables (total)
`nbv`	number of binary variables used linearly
`niv`	number of other integer variables used linearly
`n_con`	number of constraints (total)
`n_obj`	number of objectives (total)
`nlo`	number of nonlinear objectives: they come before linear objectives
`nranges`	number of ranged constraints: $\lvert\{i\colon -\infty < b_i < d_i < \infty\}\rvert$
`nlc`	number of nonlinear general constraints, including nonlinear network constraints
`nlnc`	number of nonlinear network constraints: they come after general nonlinear constraints and before any linear constraints
`nlvb`*	number of variables appearing nonlinearly in both constraints and objectives
`nlvbi`*	number of integer variables appearing nonlinearly in both constraints and objectives
`nlvc`	number of variables appearing nonlinearly in constraints
`nlvci`*	number of integer variables appearing nonlinearly just in constraints
`nlvo`	number of variables appearing nonlinearly in objectives
`nlvoi`*	number of integer variables appearing nonlinearly just in objectives
`lnc`	number of linear network constraints
`nzc`	number of nonzeros in the Jacobian matrix
`nzo`	number of nonzeros in objective gradients
`maxrownamelen`	length of longest constraint or objective name (0 if no *stub*`.row` file)
`maxcolnamelen`	length of longest variable name (0 if no *stub*`.col` file)
`n_conjac[0]` `n_conjac[1]`	Conval and Jacval operate on constraints i for `n_conjac[0]` $\le i <$ `n_conjac[1]`, initially `n_conjac[0]` $= 0$ and `n_conjac[1]` $=$ `n_con` (all constraints)

Table 1: `ASL` *components set by* `jac0dim`.
∗ AMPL versions ≥ 19930630; otherwise `nlvb` = –1.

```
typedef struct ograd ograd;
```

and has similar `typedef`s for all the other structures it declares). `ASL` field `Ograd[`i`]` points to the head of a linked-list of `ograd` structures for objective $i + 1$, so the sequence

```
c = (real *)Malloc(n_var*sizeof(real));
memset(c, 0, n_var*sizeof(real));
if (n_obj)
        for(og = Ograd[0]; og; og = og->next)
                c[og->varno] = og->coef;
```

allocates a scratch vector `c`, initializes it to zero, and (if there is at least one objective) stores the coefficients of the first objective in `c`. (The `varno` values in the `ograd` structure specify 0 for the first variable, 1 for the second, etc.)

Among the arrays allocated in `lin1.c`'s call on `f_read` are an array of alternating lower and upper variable bounds called `LUv` and an array of alternating lower and upper constraint bounds called `LUrhs`. For the present exercise, these arrays could have been declared to be arrays of

```
struct LU_bounds { real lower, upper; };
```

however, for the convenience discussed below of being able to request separate lower and upper bound arrays, both `LUv` and `LUrhs` have type `real*`. Thus the code

```
printf("\nVariable\tlower bound\tupper bound\tcost\n");
for(i = 0; i < n_var; i++)
        printf("%8ld\t%-8g\t%-8g\t%g\n", i+1,
                LUv[2*i], LUv[2*i+1], c[i]);
```

prints the lower and upper bounds on each variable, along with its cost coefficient in the first objective.

For `lin1.c`, the linear part of each constraint is conveyed in the same way as the linear part of the objective, but by a list of `cgrad` structures. These structures have one more field,

```
int goff;
```

than `ograd` structures, to allow a ''columnwise'' representation of the Jacobian matrix in nonlinear problems; the computation of Jacobian elements proceeds ''row-wise''. The final `for` loops of `lin1.c` present the *A* of (LP) row by row. The outer loop compares the constraint lower and upper bounds against `negInfinity` and `Infinity` (declared in `asl.h` and available after `ASL_alloc` has been called) to see if they are $-\infty$ or $+\infty$.

Columnwise treatment

Most LP solvers expect a ''columnwise'' representation of the constraint matrix *A* of (LP). By allocating some arrays (and setting pointers to them in the `ASL` structure), you can make the *stub*`.nl` reader give you such a representation, with subscripts optionally adjusted for the convenience of Fortran. The next examples illustrate this. Their source files are `lin2.c` and `lin3.c` in `examples`, and you can say

```
make lin2 lin3
```

to compile and link them.

The beginning of `lin2.c` differs from that of `lin1.c` in that `lin2.c` executes

```
A_vals = (real *)Malloc(nzc*sizeof(real));
```

before invoking `f_read`. When a *stub*`.nl` reader finds `A_vals` non-null, it allocates integer arrays `A_colstarts` and `A_rownos` and stores the linear part of the constraints columnwise as follows: `A_colstarts` is an array of column offsets, and linear coefficient `A_vals[`*i*`]` appears in row `A_rownos[`*i*`]`; the *i* values for column *j* satisfy `A_colstarts[`*j*`]` $\le$ *i* < `A_colstarts[`*j*+1`]` (in C notation). The column offsets and the row numbers start with the value `Fortran` (i.e., `asl->i.Fortran_`), which is 0 by default — the convenient value for use with C. For Fortran solvers, it is often convenient to set `Fortran` to 1 before invoking a *stub*`.nl` reader. This is illustrated in `lin3.c`, which also illustrates getting separate arrays of lower and upper bounds on the variables and constraints: if `LUv` and `Uvx` are not null, the *stub*`.nl` readers store the lower bound on the variables in `LUv` and the upper bounds in `Uvx`; similarly, if `LUrhs` and `Urhsx` are not null, the *stub*`.nl` readers store the constraint lower bounds in `LUrhs` and the constraint upper bounds in `Urhsx`. Table 2 summarizes these and other `ASL` components that you can optionally set.

Optional ASL components

Table 2 lists some `ASL` (i.e., `asl->i.`...`_`) components that you can optionally set and summarizes their effects.

Component	Type	Meaning
`return_nofile`	`int`	If nonzero, `jac0dim` returns 0 rather than halting execution if *stub*`.nl` does not exist.
`want_derivs`	`int`	If you want to compute nonlinear functions but will never compute derivatives, reduce overhead by setting `want_derivs = 0` (before calling the desired `.nl` reader). Alternatively, if you want nonlinear evaluations to succeed at points where first or second derivatives cannot be computed, set `want_derivs = 2` (before calling the `.nl` reader).
`Fortran`	`int`	Adjustment to `A_colstarts` and `A_rownos`.
`LUv`	`real*`	Array of lower (and, if `UVx` is null, upper) bounds on variables.
`Uvx`	`real*`	Array of upper bounds on variables.
`LUrhs`	`real*`	Array of lower (and, if `Urhsx` is null, upper) constraint bounds.
`Urhsx`	`real*`	Array of upper bounds on constraints.
`X0`	`real*`	Primal initial guess: only retained if `X0` is not null.
`havex0`	`char*`	If not null, `havex0[`*i*`]` != 0 means `X0[`*i*`]` was specified (even if it is zero).
`pi0`	`real*`	Dual initial guess: only retained if `pi0` is not null.
`havepi0`	`char*`	If not null, `havepi0[`*i*`]` != 0 means `pi0[`*i*`]` was specified (even if it is zero).
`want_xpi0`	`int`	`want_xpi0 & 1 == 1` tells *stub*`.nl` readers to allocate `X0` if a primal initial guess is available; `want_xpi0 & 2 == 2` tells *stub*`.nl` readers to allocate `pi0` if a dual initial guess is available; `want_xpi0 & 4 == 4` tells *stub*`.nl` readers to allocate `havex0` or `havepi0` when reading `X0` or `pi0`.
`A_vals`	`real*`	If not null, store linear Jacobian coefficients in `A_vals`, `A_rownos`, and `A_colstarts` rather than in lists of `cgrad` structures.
`A_rownos`	`int*`	Row numbers when `A_vals` is not null; allocated by *stub*`.nl` readers if necessary.
`A_colstarts`	`int*`	Column starts when `A_vals` is not null; allocated by *stub*`.nl` readers if necessary.
`err_jmp`	`Jmp_buf*`	If not null and an error occurs during nonlinear expression evaluation, `longjmp` here (without printing an error message).
`err_jmp1`	`Jmp_buf*`	If not null and an error occurs during nonlinear expression evaluation, `longjmp` here after printing an error message.
`obj_no`	`fint`	Objective number for `writesol()` and `qpcheck()`: 0 = first objective, –1 = no objective, i.e., just find a feasible point.

Table 2: *Optionally settable* ASL *components*.

Example: *linrc*, a ‘‘solver’’ for row-wise printing

It is easy to extend the above examples to show the variable and constraint names used in an AMPL model. When writing *stub*`.nl`, AMPL optionally stores these names in files *stub*`.col` and *stub*`.row`, as described in §A18.4 (page 487–488) of the AMPL book [7]. As an illustration, example file `linrc.c` is a variant of `lin1.c` that shows these names if they are available — and tells how to get them if they are not. Among other embellishments, `linrc.c` uses the value of environment variable `display_width` to decide when to break lines. (By the way, `$display_width` denotes this value, and other environment-variable values are denoted analogously.) Say

```
make linrc
```

to create a `linrc` program based on `linrc.c`, and say

```
linrc '-?'
```

to see a summary of its usage. It can be used stand-alone, or as the ‘‘solver’’ in an AMPL session:

```
ampl: option solver linrc, linrc_auxfiles rc; solve;
```

will send a listing of the linear part of the current problem to the screen, and

```
ampl: solve >foo;
```

will send it to file `foo`. Thus `linrc` can act much like AMPL’s `expand` and `solexpand` commands. See the AMPL book [7] (second edition) for more details on these commands.

Affine objectives: linear plus a constant

Adding a constant to a linear objective makes the problem no harder to solve. (The constant may be stated explicitly in the original model formulation, or may arise when AMPL’s *presolve* phase deduces the values of some variables and removes them from the problem that the solver sees.) For algorithmic purposes, the solver can ignore the constant, but it should take the constant into account when reporting objective values. Some solvers, such as MINOS, make explicit provision for adding a constant to an otherwise linear objective. For other solvers, such as OSL and older versions of CPLEX®, we must resort to introducing a new variable that is either fixed by its bounds (CPLEX) or by a new constraint (OSL). Function `objconst`, with apparent signature

```
real objconst(int objno)
```

returns the constant term for objective `objno` (with $0 \le$ `objno` $<$ `n_obj`) when that objective is linear. It returns 0 if objective `objno` is nonlinear. See the printing of the ‘‘Objective adjustment’’ in `examples/linrc.c` for an example of invoking `objconst`.

Example: shell script as solver for the dual LP

Sometimes it is convenient for the solver AMPL invokes to be a shell script that runs several programs, e.g., to transform *stub*`.nl` to the form the underlying solver expects and to create the *stub*`.sol` that AMPL expects. As an illustration, `examples` contains a shell script called `dminos` that arranges for `minos` to solve the dual of an LP. Why is this interesting? Well, sometimes the dual of an LP is much easier to solve than the original (‘‘primal’’) LP. Because of this, the CPLEX driver (http://ampl.com/netlib/solvers/cplex/cplex.c) has provision for solving the dual LP. (This is not to be confused with using the dual simplex algorithm, which might be applied to either the primal or the dual problem. The old OSL driver also had this provision.) Because `minos` is meant primarily for solving nonlinear problems (whose duals are more elaborate than the dual of an LP), `minos` currently lacks provision for solving dual LPs directly. At the cost of some extra overhead (over converting an LP to its dual within `minos`) and loss of flexibility (of deciding whether to solve the primal or the dual LP after looking at the problem), the `dminos` shell script provides an easy way to see how `minos` would behave on the dual of an LP. And one can use `dminos` to feed dual LPs to other LP solvers that understand *stub*`.nl` files: it’s just a matter of setting the shell variable `$dsolver` (which is discussed below).

The `dminos` shell script relies on a program called `dualconv` whose source, `dualconv.c`, also appears in `examples`. `Dualconv` reads the *stub*`.nl` for an LP and writes a *stub*`.nl` (or *stub*`.mps`) for the dual of the LP. `Dualconv` also writes a *stub*`.duw` file that it can use in a subsequent invocation to translate the *stub*`.sol` file from solving the dual LP into the primal *stub*`.sol` that AMPL expects. Thus `dualconv` is really two programs packaged, for convenience, as one. (Type

```
make dualconv
```

to create `dualconv` and then

```
dualconv '-?'
```

for more detail on its invocation than we discuss below.)

Here is a simplified version of the `dminos` shell script (for Unix systems):

```
#!/bin/sh
dualconv $1
minos $1 -AMPL
dualconv -u $1
rm $1.duw
```

This simplified script and the fancier version shown below use Bourne shell syntax. In this syntax, `$1` is the script's first argument, which should be the *stub*. Thus

```
dualconv $1
```

passes the *stub* to `dualconv`, which overwrites *stub*`.nl` with a description of the dual LP (or complains, as discussed below). If all goes well,

```
minos $1 -AMPL
```

will cause `minos` to write *stub*`.sol`, and

```
dualconv -u $1
```

will overwrite *stub*`.sol` with the form that AMPL expects. Finally,

```
rm $1.duw
```

cleans up: in the usual case where AMPL chooses the *stub*, AMPL removes the intermediate files about which it knows (e.g., *stub*`.nl` and *stub*`.sol`), but AMPL does not know about *stub*`.duw`.

The simplified `dminos` script above does not clean up properly if it is interrupted, e.g., if you turn off your terminal while it is running. Here is the more robust `examples/dminos`:

```
#!/bin/sh
# Script that uses dualconv to feed a dual LP problem to $dsolver
dsolver=${dsolver-minos}
trap "rm -f $1.duw" 1 2 3 4 13
dualconv $1
case $? in 0)
        $dsolver $1 -AMPL
        case $? in 0) dualconv -u $1;; esac
        ;; esac
rc=$?
rm -f $1.duw
exit $rc
```

It starts by determining the name of the underlying solver to invoke:

```
dsolver=${dsolver-minos}
```

is an idiom of the Bourne shell that checks whether `$dsolver` is null; if so, it sets `$dsolver` to `minos`. The line

```
trap "rm -f $1.duw" 1 2 3 4 13
```

arranges for automatic cleanup in the event of various signals. The next line

```
dualconv $1
```

works as before. If all goes well, `dualconv` gives a zero exit code; but if `dualconv` cannot overwrite *stub*`.nl` with a description of the dual LP (e.g., because *stub*`.nl` does not represent an LP), `dualconv` complains and gives return code 1. The next line

```
case $? in 0)
```

checks the return code; only if it is 0 is `$dsolver` invoked. If the latter is happy (i.e., gives zero return code), the line

```
case $? in 0) dualconv -u $1;; esac
```

adjusts *stub*`.sol` appropriately. In any event,

```
rc=$?
```

saves the current return code (i.e., `$?` is the return code from the most recently executed program), since the following clean-up line

```
rm -f $1.duw
```

will change `$?`. Finally,

```
exit $rc
```

uses the saved return code as `dminos`'s return code. This is important, as AMPL only tries to read *stub*`.sol` if the solver gives a 0 return code.

To write *stub*`.sol` files, `dualconv` calls `write_sol`, which appears in most of the subsequent examples and is documented below in the section on ''Writing the *stub*`.sol` file''.

3. Integer and Nonlinear Problems

Ordering of integer variables and constraints

When writing *stub*`.nl`, AMPL orders the variables as shown in Tables 3 and 4 and the constraints as shown in Table 5. These tables also give expressions for how many entities are in each category. Table 4 applies to AMPL versions ≥ 19930630; `nlvb` = –1 signifies earlier versions. For all versions, the first `nlvc` variables appear nonlinearly in at least one constraint. If `nlvo` > `nlvc`, the first `nlvc` variables may or may not appear nonlinearly in an objective, but the next `nlvo` – `nlvc` variables do appear nonlinearly in at least one objective. Otherwise all of the first `nlvo` variables appear nonlinearly in an objective. ''Linear arcs'' are linear variables declared with an `arc` declaration in the AMPL model, and ''nonlinear network'' constraints are nonlinear constraints introduced with a `node` declaration.

Category	Count
nonlinear	max(`nlvc`, `nlvo`); see Table 4.
linear arcs	`nwv`
other linear	`n_var` – (max {`nlvc`, `nlvo`} + `niv` + `nbv` + `nwv`)
linearly used binary	`nbv`
linearly used other integer	`niv`

Table 3: *Ordering of Variables.*

Smoothness	Appearance	Count
continuous	in an objective and in a constraint	`nlvb – nlvbi`
integer	in an objective and in a constraint	`nlvbi`
continuous	just in constraints	`nlvc – (nlvb + nlvci)`
integer	just in constraints	`nlvci`
continuous	just in objectives	max(0, nlvo – nlvc)
integer	just in objectives	`nlvoi`

Table 4: *Ordering of Nonlinear Variables.*

Category	Count
nonlinear general	`nlc` – `nlnc`
nonlinear network	`nlnc`
linear network	`lnc`
linear general	`n_con` – (`nlc` + `lnc`)

Table 5: *Ordering of Constraints*.

Reading nonlinear problems

It is convenient to build data structures for computing derivatives while reading a *stub*`.nl` file, and `amplsolver.a` provides several ways of doing this, to suit the needs of various solvers. Table 6 summarizes the available *stub*`.nl` readers and the kinds of nonlinear information they make available. They are to be used with `ASL_alloc` invocations of the form

```
asl = ASL_alloc(ASLtype);
```

Table 6's *ASLtype* column indicates the argument to supply for *ASLtype*. (This argument affects the size of the allocated `ASL` structure. Though we could easily arrange for a single routine to call the reader of the appropriate *ASLtype*, on some systems this would cause many otherwise unused routines from `amplsolver.a` to be linked with the solver. Explicitly calling the relevant reader avoids this problem.)

reader	*ASLtype*	nonlinear information
`f_read`	`ASL_read_f`	no derivatives: linear objectives and constraints only
`fg_read`	`ASL_read_fg`	first derivatives
`fgh_read`*	`ASL_read_fgh`	first derivatives and Hessian-vector products
`pfg_read`*	`ASL_read_pfg`	first derivatives and partially separable structure
`pfgh_read`	`ASL_read_pfgh`	first and second derivatives and partially separable structure

Table 6: *stub*`.nl` *readers*.
∗ Not available in `solvers2`.

All these readers have apparent signature

```
int reader(FILE *nl, int flags);
```

they close the `nl` file and return 0 if all goes well. The bits in the `flags` argument are described by comments in the `enum ASL_reader_flag_bits` declaration in `asl.h`; some of them pertain only to reading partially separable problems, which are discussed later, but others, such as `ASL_return_read_err` and `ASL_rowwise_jac`, are relevant to all the readers. The former governs the readers' behavior if they detect an error. If this bit is 0, the readers print an error message and abort execution; otherwise they return one of the nonzero values in `enum ASL_reader_error_codes`. See `asl.h` for details. Setting the `ASL_rowwise_jac` bit causes computed Jacobian matrices to be stored rowwise rather than columnwise.

Evaluating nonlinear functions

Specific evaluation routines are associated with each *stub*`.nl` reader. For simplicity, the readers supply pointers to the specific routines in the `ASL` structure, and `asl.h` provides macros to simplify calling the specific routines. The macros provide the following apparent signatures and functionality; many of them appear in the examples that follow. Reader `pfg_read` is mainly for debugging and does not provide any evaluation routines; it is used in solver ''`v8`'', discussed below. Reader `fgh_read` is mainly for debugging of Hessian-vector products, but does provide all of the routines described below except for the full Hessian computations (which would have to be done with `n_var` Hessian-vector products). Reader `pfgh_read` generally provides more efficient Hessian computations and provides the full complement of evaluation routines. If you invoke an ''unavailable'' routine, an error message is printed and execution is

aborted.

Many of the evaluation routines have final argument `nerror` of type `fint*`. This argument controls what happens if the routine detects an error. If `nerror` is null or points to a negative value, an error message is printed and, unless `err_jmp1` (i.e., `asl->i.err_jmp1_`) is nonzero, execution is aborted. (You can set `err_jmp1` much the same way that `obj1val_ASL` and `obj1grd_ASL` in file `objval.c` set `err_jmp` to gain control after the error message is printed.) If `nerror` points to a nonnegative value, `*nerror` is set to 0 if no error occurs and to a positive value otherwise.

```
real objval(int nobj, real *X, fint *nerror)
```

returns the value of objective `nobj` (with $0 \le$ `nobj` $<$ `n_obj`) at the point `X`.

```
void objgrd(int nobj, real *X, real *G, fint *nerror)
```

computes the gradient of objective `nobj` and stores it in `G[`i`]`, $0 \le i <$ `n_var`.

```
void conval(real *X, real *R, fint *nerror)
```

evaluates the *bodies* of constraints at point `X` and stores them in `R`. Recall that AMPL puts constraints into the canonical form

left-hand side $\le$ *body* $\le$ *right-hand side*,

with left- and right-hand sides contained in the `LUrhs` and perhaps `Urhsx` arrays, as explained above in the section on ‘‘Row-wise treatment’’. `Conval` operates on constraints i with

`n_conjac[0]` $\le i <$ `n_conjac[1]`

(i.e., all constraints, unless you adjust the `n_conjac` values) and stores the body of constraint i in `R[`i–`n_conjac[0]]`, i.e., it stores the first constraint body it evaluates in `R[0]`.

```
void jacval(real *X, real *J, fint *nerror)
```

computes the Jacobian matrix of the constraints evaluated by `conval` and stores it in `J`, at the `goff` offsets in the `cgrad` structures discussed above. In other words, there is one `goff` value for each nonzero in the Jacobian matrix, and the `goff` values determine where in `J` the nonzeros get stored. The *stub*`.nl` readers compute `goff` values so a Fortran program will see Jacobian matrices stored columnwise, but you can adjust the `goff` fields to make other arrangements.

```
real conival(int ncon, real *X, fint *nerror)
```

evaluates and returns the body of constraint `ncon` (with $0 \le$ `ncon` $<$ `n_con`).

```
void congrd(int ncon, real *X, real *G, fint *nerror)
```

computes the gradient of constraint `ncon` and stores it in `G`. By default, `congrd` sets `G[`i`]`, $0 \le i <$ `n_var`, but if you set `asl->i.congrd_mode` = 1, it will just store the partials that are not identically 0 consecutively in `G`, and if you set `asl->i.congrd_mode` = 2, it will store them at the `goff` offsets of the `cgrad` structures for this constraint.

The following routines that compute Hessian-vector products or Hessians do not have an ‘‘`X`’’ argument and instead use partial derivatives computed in previous objective and constraint evaluations. It is the solver’s responsibility to ensure that nonlinear constraints and objectives were most recently evaluated at the desired vector of primal variables.

```
void hvcomp(real *HV, real *P, int nobj, real *OW, real *Y)
```

stores in `HV` (a full vector of length `n_var`) the Hessian of the Lagrangian times vector `P`. In other words, `hvcomp` computes

`HV` $= W \cdot$ `P`,

where W is the Lagrangian Hessian,

$$W = \nabla^2 \left[\sum_{i=0}^{\texttt{n\_obj}-1} \texttt{OW}[i] f_i + \sigma \sum_{i=0}^{\texttt{n\_con}-1} \texttt{Y}[i] c_i \right], \qquad (*)$$

in which f_i and c_i denote objective function *i* and constraint *i*, respectively, and σ is an extra scaling factor (most commonly $+1$ or -1) that is $+1$ unless specified otherwise by a previous call on `lagscale` (see below). If $0 \le$ `nobj` $<$ `n_obj`, `hvcomp` behaves as though `OW` were a vector of all zeros, except for `OW[nobj]`, which is taken to be 1 if `OW` is null; otherwise, if `OW` is null, `hvcomp` behaves as though it were a vector of all zeros; and if `Y` is null, `hvcomp` behaves as though `Y` were a vector of zeros. *W* is evaluated at the point where the objective(s) and constraints were most recently computed (by calls on `objval` or `objgrd`, and on `conval`, `conival`, `jacval`, or `congrd`, in any convenient order). Normally one computes gradients before dealing with *W*, and if necessary, the gradient computing routines first recompute the objective(s) and constraints at the point specified in their argument lists. The Hessian computations use partial derivatives stored during the objective and constraint evaluations.

To make make it more efficient to call `hvcomp(HV, P, nobj, OW, Y)` several times at the same primal point with different `P` vectors but the same `(nobj, OW, Y)`, some intermediate results may be saved from the first `hvcomp` call at the current point. In the unusual case of calling `hvcomp(HV, P, nobj, OW, Y)` at the same primal point but with a different value of the triple `(nobj, OW, Y)`, it is necessary to first call `hvpinit(ihd_limit, nobj, OW, Y)` with the new values of `nobj`, `OW`, and `Y`. This function has apparent signature

```
void hvpinit(int ihd_lim, int nobj, real *OW, real *Y);
```

The first argument, `ihd_lim`, limits *k* in the size $k \times k$ of Hessians that are precomputed at new primal variable values; `ihd_limit` is #defined in asl.h to be `asl->p.ihd_limit_`, which by default is 12.

```
void duthes(real *H, int nobj, real *OW, real *Y)
```

evaluates and stores in `H` the *d*ense *u*pper *t*riangle of the *Hes*sian of the Lagrangian function *W*. Here and below, arguments `nobj`, `OW` and `Y` have the same meaning as in `hvcomp`, so `duthes` stores the upper triangle by columns in `H`, in the sequence

$$W_{0,0} \quad W_{0,1} \quad W_{1,1} \quad W_{0,2} \quad W_{1,2} \quad W_{2,2} \quad \cdots$$

of length `n_var*(n_var+1)/2` (with 0-based subscripts for *W*).

```
void fullhes(real *H, fint LH, int nobj, real *OW, real *Y)
```

computes the *W* of (∗) and stores it in `H` as a Fortran 77 matrix declared

```
integer LH
double precision H(LH,*)
```

In C notation, `fullhes` sets

$$\texttt{H}[i + j \cdot \texttt{LH}] = W_{i,j}$$

for $0 \le i <$ `n_var` and $0 \le j <$ `n_var`. Both `duthes` and `fullhes` compute the same numbers; `fullhes` first computes the Hessian's upper triangle, then copies it to the lower triangle, so the result is symmetric.

```
fint sphsetup(int nobj, int ow, int y, int uptri)
```

returns the number of nonzeros in the *sp*arse *H*essian *W* of the Lagrangian (∗) (if `uptri` = 0) or its upper triangle (if `uptri` = 1) or its lower triangle (if `uptri` = 2), and stores in fields `sputinfo->hrownos` and `sputinfo->hcolstarts` a description of the sparsity of *W*, as discussed below with `sphes`. For `sphes`'s computation, which determines the components of *W* that could be nonzero, arguments `ow` and `y` indicate whether `OW` and `Y`, respectively, will be zero or nonzero in subsequent calls on `sphes`. In analogy with `hvcomp`, `duthes`, `fullhes` and `sphes`, if $0 \le$ `nobj` $<$ `n_obj`, then `nobj` takes precedence over `ow`.

```
void sphes(real *H, int nobj, real *OW, real *Y)
```

computes the *W* given by (∗) and stores it or its sparse upper triangle in `H`; `sphsetup` must have been called previously with arguments `nobj`, `ow` and `y` of the same sparsity (zero/nonzero structure), i.e., with the same `nobj`, with `ow` nonzero if ever `OW` will be nonzero, and with `y` nonzero if ever `Y` will be nonzero.

Argument `uptri` to `sphsetup` determines whether `sphes` computes W's upper triangle (`uptri` = 1) or all of W (`uptri` = 0); in the latter case, the computation proceeds by first computing the upper triangle, then copying it to the lower triangle, so the result is guaranteed to be symmetric. Fields `sputinfo->hrownos` and `sputinfo->hcolstarts` are pointers to arrays that describe the sparsity of W in the usual columnwise way:

$$\texttt{H}[j] = W_{i,\,\texttt{rownows}[j]}$$

for $0 \le i <$ *n_var* and `hcolstarts`$[i] \le j <$ `hcolstarts`$[i+1]$. Before returning, `sphsetup` adds the `ASL` value `Fortran` to the values in the `hrownos` and `hcolstarts` arrays. The row numbers in `hrownos` for each column are in ascending order.

```
int xknown(real *X)
int xknowne(real *X, fint *nerror)
```

indicate that this `X` will be provided to the function and gradient computing routines in subsequent calls until either another `xknown...` invocation makes a new `X` known, or `xunknown()` is executed. The latter function, with apparent signature

```
void xunknown(void);
```

reinstates the default behavior of checking the `X` arguments against the previous value to see whether common expressions (or, for gradients, the corresponding functions) need to be recomputed. Appropriately calling `xknown...` and `xunknown...` can reduce the overhead in some computations. The `nerror` argument to `xknowne()` and `xknowne_ew()` is set to 0 if all goes well and to 1, 2, or 3 if a shared defined variable cannot be evaluated at this `X`. Values `nerror` = 2 or 3 are only possible when `want_derivs` was set to 2 (or another nonzero even value) before the *stub*`.nl` reader was called. The `xknown()` functions return 0 if the nonlinear components of `X` have not changed and 1 if they have changed since the last call on a routine in `amplsolver.a` that has an ''`x`'' argument.

```
void conscale(int i, real s, fint *nerror)
```

scales function body `i` by `s`, initial dual value `pi0[i]` by 1/`s`, and the lower and upper bounds on constraint `i` by `s`, interchanging these bounds if `s` < 0. This only affects the `pi0`, `LUrhs` and `Urhsx` arrays and the results computed by `conval`, `jacval`, `conival`, `congrd`, `duthes`, `fullhes`, `sphes`, and `hvcomp`. The `write_sol` routine described below takes calls on `conscale` into account.

```
void lagscale(real sigma, fint *nerror)
```

specifies the extra scaling factor $\sigma :=$ `sigma` in the formula (*) for the Lagrangian Hessian.

```
void varscale(int i, real s, fint *nerror)
```

scales variable `i`, its initial value `X0[i]` and its lower and upper bounds by 1/`s`, and it interchanges these bounds if `s` < 0. Thus `varscale` effectively scales the partial derivative of variable `i` by `s`. This only affects the nonlinear evaluation routines and the arrays `X0`, `LUv` and `Uvx`. The `write_sol` routine described below accounts for calls on `varscale`.

Calls on `conscale`, `varscale`, and `lagscale` should be made after the *stub*`.nl` reader has been called.

Example: nonlinear minimization subject to simple bounds

Our first nonlinear example ignores any constraints other than bounds on the variables and assumes there is one objective to be minimized. This example involves the PORT solver `dmngb`, which amounts to subroutine SUMSL of [8] with added logic for bounds on the variables (as described in [9]). Fortran source for PORT routines mentioned in this report, such as `dmngb`, is available in

`http://ampl.com/netlib/ampl/solvers/examples/fport.zip`

and corresponding C source (via *f2c*) is available in

`http://ampl.com/netlib/ampl/solvers/examples/cport.zip`

The driver for this example is `examples/mng1.c`.

Most of `mng1.c` is specific to `dmngb`. For example, `dmngb` expects subroutine parameters `calcf`

and `calcg` for evaluating the objective function and its gradient. Interface routines `objval` and `objgrd` actually evaluate the objective and its gradient; the `calcf` and `calcg` defined in `mng1.c` simply adjust the calling sequences appropriately. The calling sequences for `objval` and `objgrd` were shown above.

Since `dmngb` is prepared to deal with evaluation errors (which it learns about when argument `*NF` to `calcf` and `calcg` is set to 0), `calcf` and `calcg` pass a pointer to 0 for `nerror`.

The main routine in `mng1.c` is called `MAIN__` rather than `main` because it is meant to be used with an *f2c*-compatible Fortran library. (A C `main` appears in this Fortran library and arranges to catch certain signals and flush buffers. The `main` makes its arguments `argc` and `argv` available in the external cells `xargc` and `xargv`.)

Recall that when AMPL invokes a solver, it passes two arguments: the *stub* and an argument that starts with `-AMPL`. Thus `mng1.c` gets the *stub* from the first command-line argument. Before passing it to `jac0dim`, `mng1.c` calls `ASL_alloc(ASL_read_fg)` to make an `ASL` structure available. `ASL_alloc` stores its return value in the global cell `cur_ASL`. Since `mng1.c` starts with

```
#include "asl.h"
#define asl cur_ASL
```

the value returned by `ASL_alloc` is visible throughout `mng1.c` as ‘‘`asl`’’. This saves the hassle of making `asl` visible to `calcf` and `calcg` by some other means.

The invocation of `dmngb` directly accesses two `ASL` pointers: `X0` and `LUv` (i.e., `asl->i.X0_` and `asl->i.LUv_`). `X0` contains the initial guess (if any) specified in the AMPL model, and `LUv` is an array of lower and upper bounds on the variables. Before calling `fg_read` to read the rest of *stub*`.nl`, `mng1.c` asks `fg_read` to save `X0` (if an initial guess is provided in the AMPL model or data, and otherwise to initialize `X0` to zeros) by executing

```
X0 = (real *)Malloc(n_var*sizeof(real));
```

After invoking `dmngb`, `mng1.c` writes a termination message into the scratch array `buf` and passes it, along with the computed solution, to interface routine `write_sol`, discussed later, which writes the termination message and solution to file *stub*`.sol` in the form that AMPL expects to read them.

The use of `Cextern` in the declaration

```
typedef void (*U_fp)(void);
Cextern int dmngb_(fint *n, real *d, real *x, real *b,
          U_fp calcf, U_fp calcg,
          fint *iv, fint *liv, fint *lv, real *v,
          fint *uiparm, real *urparm, U_fp ufparm);
```

at the start of `mng1.c` permits compiling this example with either a C or a C++ compiler; `Cextern` is `#define`d in `asl.h`.

Example: nonlinear least squares subject to simple bounds

The previous example dealt only with a nonlinear objective and bounds on the variables. The next example deals only with nonlinear equality constraints and bounds on the variables. It minimizes an implicit objective: the sum of squares of the errors in the constraints. The underlying solver, `dn2gb`, again comes from the PORT subroutine library; it is a variant of the unconstrained nonlinear least-squares solver `NL2SOL` [3, 4] that enforces simple bound constraints on the variables. Source for `dn2gb` appears in the files `cport.zip` and `fport.zip` mentioned above.

Source for this example is `examples/nl21.c`. Much like `mng1.c`, it starts with

```
#include "asl.h"
#define asl cur_ASL
```

followed by declarations for the definitions of two interface routines: `calcr` computes the residual vector (vector of errors in the equations), and `calcj` computes the corresponding Jacobian matrix (of first partial derivatives). Again these are just wrappers that invoke `amplsolver.a` routines described above,

`conval` and `jacval`. Parameter `NF` to `calcr` and `calcj` works the same way as in the `calcf` and `calcg` of `mng1.c`. Recall again that AMPL puts constraints into the canonical form

left-hand side ≤ *body* ≤ *right-hand side*.

Subroutine `calcr` calls `conval` to have a vector of `n_con` *body* values stored in array `R`. The `MAIN__` routine in `nl21.c` makes sure the left- and right-hand sides are equal, and passes the vector `LUrhs` of left- and right-hand side pairs as parameter `UR` to `dn2gb`, which passes them unchanged as parameter `UR` to `calcr`. (Of course, `calcr` could also access `LUrhs` directly.) Thus the loop

```
for(Re = R + *N; R < Re; UR += 2)
        *R++ -= *UR;
```

in `calcr` converts the constraint body values into the vector of residuals.

`MAIN__` invokes the interface routine `dense_j()` to tell `jacval` that it wants a dense Jacobian matrix, i.e., a full matrix with explicit zeros for partial derivatives that are always zero. If necessary, `dense_j` adjusts the `goff` components of the `cgrad` structures and tells `jacval` to zero its `J` array before computing derivatives.

Partially separable structure

Many optimization problems involve a *partially separable* objective function, one that has the form

$$f(x) = \sum_{i=1}^{q} f_i(U_i x),$$

in which U_i is an $m_i \times n$ matrix with a small number m_i of rows [13, 14]. Partially separable structure is of interest because it permits better Hessian approximations or more efficient Hessian computations. Many partially separable problems exhibit a more detailed structure, which the authors of `LANCELOT` [2] call ‘‘group partially separable structure’’:

$$f(x) = \sum_{i=1}^{q} \theta_i\left(\sum_{j=1}^{r_i} f_{ij}(U_{ij} x)\right),$$

where $\theta_i : \mathbb{R} \to \mathbb{R}$ is a unary operator. Using techniques described in [12], the *stub*`.nl` readers `pfg_read` and `pfgh_read` discern this latter structure automatically, and the Hessian computations that `pfgh_read` makes available exploit it. Some solvers, such as `LANCELOT` and `VE08` [19], want to see partially separable structure. Driving such solvers involves a fair amount of solver-specific coding. Directory `examples` has drivers for two variants of `VE08`: `ve08` ignores whereas `v8` exploits partially separable structure, using reader `pfg_read`. Directory `solvers/lancelot` contains source for `lancelot`, a solver based on `LANCELOT` that uses reader `pfgh_read`.

Fortran variants

Fortran variants `fmng1.f` and `fnl21.f` of `mng1.c` and `nl21.c` appear in `examples`; the `makefile` has rules to make programs `fmng1` and `fnl21` from them. Both invoke interface routines `jacdim_` and `jacinc_`. The former allocates an `ASL` structure (with `ASL_alloc(ASL_read_fg)`) and reads a *stub*`.nl` file with `fg_read`, and the latter provides arrays of lower and upper constraint bounds, the initial guess, the Jacobian incidence matrix (which neither example uses), and (in the last variable passed to `jacinc_`) the value `Infinity` that represents ∞. These routines have Fortran signatures

```
subroutine jacdim(stub, M, N,  NO, NZ, MXROW, MXCOL)
character*(*) stub
integer M, N, NO, NZ, MXROW, MXCOL

subroutine jacinc(M, N, NZ, JP, JI, X, L, U, Lrhs, Urhs, Inf)
integer M, N, NZ, JP(N+1)
integer*2 JI
double precision X(N), L(N), U(N), Lrhs(M), Urhs(M), Inf
```

`Jacdim_` sets its arguments as shown in Table 7. The values `MXROW` and `MXCOL` are unlikely to be of much interest; `MXROW` is 0 unless AMPL wrote *stub*`.row` (a file of constraint and objective names), in which case `MXROW` is the length of the longest name in *stub*`.row`. Similarly, `MXCOL` is 0 unless AMPL wrote *stub*`.col`, in which case `MXROW` is the length of the longest variable name in *stub*`.col`. Variant `jacinc1_()` of `jacinc_()` has argument `JI` of type `fint*` rather than `short int*`, i.e., of Fortran type `integer` rather than `integer*2`.

`*M`	= `n_con`	= number of constraints
`*N`	= `n_var`	= number of variables
`*NO`	= `n_obj`	= number of objectives
`*NZ`	= `ncz`	= number of Jacobian nonzeros
`*MXROW`	= `maxrownamelen`	= length of longest constraint name
`*MXCOL`	= `maxcolnamelen`	= length of longest variable name

Table 7: *Assignments made by* `jacdim_`.

The Fortran examples call Fortran variants of some of the nonlinear evaluation routines. Table 8 summarizes some Fortran variants; source and a test program for some others (e.g., for evaluating Hessian information) appear in

`http://ampl.com/netlib/ampl/solvers/examples/fortmisc.zip`

In Table 8, Fortran notation appears under ‘‘Fortran variant’’; the corresponding C routines have an underscore appended to their names and are declared in `asl.h`. The Fortran routines shown in Table 8 operate on the `ASL` structure at which `cur_ASL` points. Thus, without help from a C routine to adjust `cur_ASL`, they only deal with one problem at a time. After solving a problem and executing

```
call delprb
```

a Fortran code could call `jacdim` and `jacinc` again to start processing another problem.

Nonlinear test problems

Some nonlinear AMPL models appear in directory

`http://ampl.com/netlib/ampl/models/nlmodels`

The entire `netlib/ampl/models` directory, which has linear models, the `nlmodels` subdirectory, and directory `compl` for complementary problems, is available as the gzipped tar file

`http://ampl.com/netlib/ampl/models.tgz`

4. Advanced Interface Topics

Objective Sense

Whether objective *n* (0 n < `n_obj`) is to be minimized or maximized is specified by the `objtype` array: `objtype[`*n*`]` = 0 if it is to be minimized and `objtype[`*n*`]` = 1 if it is to be maximized.

Accessing names

Functions

```
char *con_name(int n);
char *obj_name(int n);
char *var_name(int n);
```

return the name of constraint *n* (0 ≤ *n* < `n_con`), objective *n* (0 ≤ *n* < `n_obj`), and variable *n* (0 ≤ *n* < `n_var`), respectively. If the relevant `auxfiles` option was suitably set when the `stub.nl` file was written, the names will be from the AMPL model. Otherwise they will be generic names, such as `_scon[1]`.

Routine	Fortran variant
congrd	congrd(N, I, X, G, NERROR)
conival	cnival(N, I, X, NERROR)
conval	conval(M, N, X, R, NERROR)
dense_j	densej()
hvcomp	hvcomp(HV, P, NOBJ, OW, Y)
jacval	jacval(M, N, NZ, X, J, NERROR)
objgrd	objgrd(N, X, NOBJ, G, NERROR)
objval	objval(N, X, NOBJ, NERROR)
writesol	wrtsol(MSG, NLINES, X, Y)
xknown	xknown(X)
xunkno	xunkno()
delprb_	delprb()

Argument	Type	Description
N	integer	number of variables (n_var)
M	integer	number of constraints (n_con)
NZ	integer	number of Jacobian nonzeros (nzc)
NERROR	integer	if ≥ 0, set NERROR to 0 if all goes well and to a positive value if the evaluation fails
I	integer	which constraint
NOBJ	integer	which objective
NLINES	integer	lines in MSG
MSG	character*(*)	solution message, dimension(NLINES)
X	double precision	incoming vector of variables
G	double precision	result gradient vector
J	double precision	result Jacobian matrix
OW	double precision	objective weights
Y	double precision	dual variables
P	double precision	vector to be multiplied by Hessian
HV	double precision	result of Hessian times P

Table 8: *Fortran variants.*

Writing the *stub*`.sol` file

Interface routine `write_sol` returns the computed solution and a termination message to AMPL by writing a *stub*`.sol` file. (When the solver is not invoked by AMPL, the `.sol` file is only written if ‘‘`-AMPL`’’ was given as the second command-line argument or the `wantsol` keyword has an odd value.) This routine has apparent prototype

```
void write_sol(char *msg, real *x, real *y, Option_Info *oi);
```

The first argument is for the (null-terminated) termination message. It should not contain any empty embedded lines (though, e.g., `" \n"`, i.e., a line consisting of a single blank, is fine) and may end with an arbitrary number of newline characters (including none, as in `mng1.c`). The second and third arguments, `x` and `y`, are pointers to arrays of primal and dual variable values to be passed back to AMPL. Either or both may be null (as is `y` in `mng1.c`), which causes no corresponding values to be passed. Normally it is helpful to return the best approximate solution found, but for some errors (such as trouble detected before the solution algorithm can be started) it may be appropriate for both `x` and `y` to be null. The fourth argument points to an optional `Option_Info` structure, which is discussed below in the section on ‘‘Conveying solver options’’. Before calling `write_sol()`, a solver should assign a suitable value to `solve_result_num` (i.e., `asl->i.solve_result_num_`); see pp. 283–284 of [7].

Locating evaluation errors

If the routines in `amplsolver.a` detect an error during evaluation of a nonlinear expression, they look to see if *stub*`.row` (or, if evaluation of a ‘‘defined variable’’ was in progress, *stub*`.fix`) is available. If so, they use it to report the name of the constraint, objective, or defined variable that they were trying to evaluate. Otherwise they simply report the number of the constraint, objective, or variable in question (first one = 1). This is why AMPL provides the default value `RF` for `$minos_auxfiles`. See the discussion of auxiliary files in §A18.4 of the AMPL book [7]; as documented in *netlib*’s ‘‘`changes from ampl`’’, i.e.,

```
http://ampl.com/netlib/ampl/changes
```

capital letters in `$solver_auxfiles` have the same effect as their lower-case equivalents on nonlinear problems, including problems with integer variables, and have no effect on purely linear problems.

Imported functions

An AMPL model may involve imported functions. If invocations of such functions involve variables, the solver must be able to evaluate the functions. You can tell your solver about the relevant functions by supplying a suitable `funcadd` function, rather than loading a dummy `funcadd` compiled from `solvers/funcadd0.c`. Include file `funcadd.h` gives `funcadd`’s prototype:

```
void funcadd(AmplExports *ae);
```

Among the fields in the `AmplExports` structure are some function pointers, such as

```
void (*Addfunc)(char *name, real (*f)(Arglist*), int type,
                int nargs, void *funcinfo, AmplExports *ae);
```

also in `funcadd.h` are `#define`s that simplify using the function pointers, assuming

```
AmplExports *ae
```

is visible. In particular, `funcadd.h` gives `addfunc` the apparent prototype

```
void addfunc(char *name, real (*f)(Arglist*), int type,
             int nargs, void *funcinfo);
```

To make imported functions known, `funcadd` should call `addfunc` once for each one. The first argument, `name`, is the function’s name in the AMPL model. The second argument points to the function itself. The `type` argument tells whether the function is prepared to accept symbolic arguments (character strings): 0 means ‘‘no’’, 1 means ‘‘yes’’. Argument `nargs` tells how many arguments the function expects; if `nargs` ≥ 0, the function expects exactly that many arguments; otherwise it expects at least –(`nargs` + 1). (Thus `nargs` = –1 means 0 or more arguments, `nargs` = –2 means 1 or more, etc. The argument counting and type checking occur when the *stub*`.nl` file is subsequently read.) Finally, argument `funcinfo` is for the function to use as it sees fit; it will subsequently be passed to the function in field `funcinfo` of struct `arglist`.

When an imported function is invoked, it always has a single argument, `al`, which points to an `arglist` structure. This structure is designed so the same imported function can be linked with AMPL (in case AMPL needs to evaluate the function); the final `arglist` components are relevant only to AMPL. The function receives `al->n` arguments, `al->nr` of which are numeric; for 0 ≤ *i* < `al->n`,

if `al->at[`*i*`]` ≥ 0, argument *i* is `al->ra[al->at[`*i*`]]`

if `al->at[`*i*`]` < 0, argument *i* is `al->sa[`–`(al->at[`*i*`] + 1)]`.

If `al->derivs` is nonzero, the function must store its first partial derivative with respect to `al->ra[`*i*`]` in `al->derivs[`*i*`]`, and if `al->hes` is nonzero (which is possible only with `fgh_read` or `pfgh_read`), it must also store the upper triangle of its Hessian matrix in `al->hes`, i.e., for

0 ≤ *i* ≤ *j* < `al->nr`

it must store its second partial with respect to `al->ra[`*i*`]` and `al->ra[`*j*`]` in

`al->hes[`*i* + ½*j*(*j*+1)`]`. Sometimes partials with respect to some arguments are not needed, in which case `al->dig` is nonzero, and `al->dig[`*i*`]` is nonzero if partials with respect to

`al->ra[`*i*`]` will not be used. It is always safe to provide all partials.

If the function does any printing, it should initially say

```
AmplExports *ae = al->AE;
```

to make special variants of `printf` available.

See `solvers/funcadd.c` for an example `funcadd`. The `mng`, `mnh` and `nl2` examples mentioned below illustrate linking with this `funcadd`.

When an imported function is called by AMPL, `ae->asl` is null and calls on `getenv(name)`, which is #defined to be `(*ae->Getenv)(name)`, return values affected by AMPL's `option` commands. When an imported function is called by a solver, `ae->asl` is the solvers's ASL pointer.

If an imported function cannot perform as expected, it should set `al->Errmsg` to a string explaining the trouble. If the function returns a function value but cannot compute first derivatives, `acl->Errmsg` should start with a single single quote (’); if the function returns a function value and first derivatives but cannot compute second derivatives, `al->Errmsg` should start with a double-quote character ("). If need be, memory *n* bytes long to hold this string can be allocated by calling `ae->Tempmem(al->TMI, n)`, which returns a `void*` value. The `ae->Tempmem()` function can also be called to obtain memory for any other use during the current invocation of an imported function. Such memory is automatically freed after the function returns — and after use is made of `al->Errmsg`.

Complementarity Constraints

Constraints involving complementarity conditions are declared with the `complements` keyword in an AMPL model, which causes the *stub*`.nl` readers to allocate and populate a `cvar` array that indicates which constraints are involved in complementarity conditions. Suppose `cvar[`*i*`]` = *j*. If *j* = 0, then constraint *i* is an ordinary algebraic constraint, but if *j* > 0, then the constraint complements variable *j*−1. In this latter case, if the constraint has one finite bound, then the variable will also have one finite bound, and either the constraint or the variable must be at its bound. If the constraint has distinct finite lower and upper bounds, then the variable has no explicit bounds, but must be nonnegative if the constraint is at its lower bound, nonpositive if the constraint is at its upper bound, and zero if the constraint is strictly slack.

Some solvers are not prepared to handle general complementarity conditions, but do handle conditions of the form $v_i \geq 0$, $v_j \geq 0$, $min(v_i, v_j) = 0$. Such solvers should set the `ASL_cc_simplify` bit in the flags argument to the *stub*`.nl` reader, which causes the reader to allocate arrays `asl->i.ccind1` and `asl->i.ccind2` and to adjust the problem so there are `n_cc` complementarity conditions of the form

$$\textit{variable}\ \texttt{asl->ccind1[}i\texttt{]} \geq 0$$

$$\textit{variable}\ \texttt{asl->ccind2[}i\texttt{]} \geq 0$$

and

$$min(\textit{variable}\ \texttt{asl->ccind1[}i\texttt{]}, \textit{variable}\ \texttt{asl->ccind2[}i\texttt{]}) = 0.$$

Suffixes

AMPL models can declare suffixes, which are names for auxiliary values associated with variables, constraints, objectives, and problems. Some of these values may be inputs to the solver, others may be values computed by the solver and returned to the AMPL session, and some may be both. For example, some solvers allow one to specify an incoming basis and to return a final basis; basic variables and constraints are conventionally indicated by `.sstatus` suffixes on variables and constraints. AMPL can send initial `.sstatus` values to the solver and retrieve updated `.sstatus` values from the `.sol` file that the solver writes. Some suffixes assume only a small number of integer values or ranges of such values, and sometimes it is convenient to have symbolic descriptions of these (ranges of) values. For example, when AMPL begins execution, option `sstatus_table` has the value

```
0   none    no status assigned\
```

```
1   bas     basic\
2   sup     superbasic\
3   low     nonbasic <= (normally =) lower bound\
4   upp     nonbasic >= (normally =) upper bound\
5   equ     nonbasic at equal lower and upper bounds\
6   btw     nonbasic between bounds\
```

For suffixes that have an associated ‘‘`_table`’’ option value (which one can supply and modify as desired), AMPL uses the value in the second column of the table as the displayed value of the suffix when its numeric value is at most the value in the first column (and exceeds the first-column value of the previous row, if any, of the table); one can print or display the suffix’s numeric value by appending ‘‘`_num`’’ to the suffix’s name. When changing a suffix’s value in a ‘‘`let`’’ command, one can use either a numeric or string for the new value. For example:

```
ampl: display x.sstatus, x.sstatus_num;
x.sstatus = none
x.sstatus_num = 0

ampl: let x.sstatus := 3;
ampl: display x.sstatus, x.sstatus_num;
x.sstatus = low
x.sstatus_num = 3

ampl: let x.sstatus := 'upp';
ampl: display x.sstatus, x.sstatus_num;
x.sstatus = upp
x.sstatus_num = 4
```

Before accessing incoming status values or supplying outgoing status values, a solvers must invoke

```
suf_declare(suftab, nsuftab);
```

in which `suftab` is an array of `SufDecl` structures:

```
struct SufDecl {
        /* Pass array of SufDecl's to suf_declare(). */
        char *name;
        char *table;
        int kind;
        int nextra;
        };
```

and `nsuftab` is the number of `SufDecl` values in the `suftab` array. Usually the `table` value is null, but it can be the value desired for the suffix’s `_table` option when the suffix is returned to the AMPL session. The `kind` value should be one of

```
ASL_Sufkind_var  = 0,
ASL_Sufkind_con  = 1,
ASL_Sufkind_obj  = 2,
ASL_Sufkind_prob = 3,
```

to indicate a suffix on variables, constraints, objectives, or problems, respectively, possibly or-ed with one or more of

```
ASL_Sufkind_real = 4,    /* use SufDesc.u.r rather than .i */
ASL_Sufkind_iodcl = 8,   /* declare as INOUT suffix */
ASL_Sufkind_output = 16, /* return this suffix to AMPL */
ASL_Sufkind_input = 32,  /* input values were received from AMPL */
ASL_Sufkind_outonly = 64 /* reject as an input value */
```

The *stub*`.nl` reader sets the ASL_Sufkind_input bit for suffixes retrieved from the incoming *stub*`.nl` file.

Suffixes are stored in integer arrays unless `ASL_Sufkind_real` is specified. Suffixes like `sstatus` that can appear on more than one kind of entity must have a separate `SufDecl` entry for each kind. For example, the CPLEX driver `http://ampl.com/netlib/ampl/solvers/cplex/cplex.c` has a `suftab` declaration with many `SufDecl` values, including

```
{ "iis", iis_table, ASL_Sufkind_var | ASL_Sufkind_outonly },
{ "iis", 0, ASL_Sufkind_con | ASL_Sufkind_outonly },
{ "lazy", 0, ASL_Sufkind_con },
...
{ "sstatus", 0, ASL_Sufkind_var, 1 },
{ "sstatus", 0, ASL_Sufkind_con, 1 },
```

Thus `.iis` can be a suffix on both constraints and variables. This suffix is for describing an ‘‘irreducible infeasible’’ set of mutually inconsistent constraints and variables. The `iis_table` permits using symbolic names for the suffix values. It is only necessary to supply `iis_table` once. Its declaration is

```
 static char iis_table[] = "\n\
0   non     not in the iis\n\
1   low     at lower bound\n\
2   fix     fixed\n\
3   upp     at upper bound\n\
4   mem     member\n\
5   pmem    possible member\n\
6   plow    possibly at lower bound\n\
7   pupp    possibly at upper bound\n\
8   bug\n";
```

To access incoming suffix values, one invokes `suf_get()` with apparent prototype

```
SufDesc *suf_get(const char *sufname, int kind);
```

For example, to see if there is an incoming `.lazy` suffix (as declared in the partial `suftab` declaration shown above), one would use coding of the form

```
SufDesc *lazy;
...
lazy = suf_get("lazy", ASL_Sufkind_con);
```

If the resulting `lazy->u.i` value is null, then the `.lazy` suffix is not available. Otherwise `lazy->u.i` is an arrary (of integers) giving the `.lazy` suffix values on constraints. If `ASL_Sufkind_real` had been specified for this suffix, then `lazy->u.r` would be the (possibly null) array of ‘‘real’’ values for the suffix. By or-ing `ASL_Sufkind_input` onto the `kind` argument to `suf_get`, e.g.,

```
lazy = suf_get("lazy", ASL_Sufkind_con | ASL_Sufkind_input);
```

one can request that null be returned for `lazy` if incoming `.lazy` values are not present in the *stub*`.nl` file.

To return suffix values to AMPL for a suffix not declared with `ASL_Sufkind_output` in its `suftab` entry, one invokes apparent prototype

```
SufDesc *suf_iput(const char *sufname, int kind, int *val);
```

for an integer-valued suffix or

```
SufDesc *suf_rput(const char *sufname, int kind, real *val);
```

for a ‘‘real’’ valued one. The `val` argument should be the arrary that will contain the desired outgoing suffix values. The value returned by `suf_iput` or `suf_rput` is that which `suf_get(...)` would have returned.

Special Ordered Sets

Special ordered sets [1] are useful for expressing situations where one of several alternatives must be chosen (type 1 SOS sets) and general piecewise-linear functions (type 2). An SOS set of type 1 is a set of binary variables, exactly one of which can be 1, conveniently expressed as $\sum_i b_i = 1$, and an SOS set of type 2 is an ordered set of binary variables such that at most two are nonzero, and two nonzero such variables are adjacent. A convex piecewise-linear function, when appearing in an objective to be minimized or on the left-hand side of a ≥ constraint, can be expressed by several linear inequality contraints, but a more general piecewise-linear function appearing in such places needs to involve an SOS set of type 2. AMPL uses SOS sets of type 2 to ''linearize'' nonconvex piecewise-linear functions, so they can be treated by a mixed-integer linear programming solver. The resulting *stub*`.nl` file will work with any such solver, but solvers that are prepared to handle SOS sets explicitly may work more efficiently when told about SOS sets. To facilitate this, AMPL attaches suffixes `.sos` and `.sosref` to relevant variables. In AMPL declarations and scripts, one can also supply suffixes `.sosno` and `.ref` to manually express SOS sets. Each has a distinct `.sosno` value, with positive values for SOS type 1 sets and negative values for type 2 sets. Solvers can call `suf_sos()`, with apparent signature

```
int suf_sos(int flags, int *nsosnz, char **sostype,
            int **sospri, int *copri, int **sosbeg,
            int **sosind, real **sosref)
```

to obtain SOS details in the arrays allocated by `suf_sos()` and to remove relevant constraints supplied by AMPL for conveying nonconvex piecewise-linear terms. This function returns the number of SOS sets so treated. The flags argument determines which SOS sets are treated. It is an ''or'' of

```
ASL_suf_sos_explict_free
ASL_suf_sos_ignore_sosno
ASL_suf_sos_ignore_amplso
```

(declared in `asl.h`). `ASL_suf_sos_explict_free` means the caller will explicitly free arrays allocated by `suf_sos()`; otherwise the arrays will be freed automatically when `ASL_free(&asl)` is called. `ASL_suf_sos_ignore_sosno` causes `suf_sos()` to ignore any `.sosno` and `.ref` suffixes, and `ASL_suf_sos_ignore_amplso` causes `suf_sos()` to ignore the `.sos` and `.sosref` suffixes AMPL uses for nonconvex piecewise-linear terms. After the call

```
nsos = suf_sos(flags, &nsosnz, &sostype, &sospri, &copri,
            &sosbeg, &sosind, &sosref);
```

the arrays allocated by `suf_sos()` describe `nsos` SOS sets. A total of `nsosnz` variables are involved in these sets. For $0 \le i <$ `nsos`, set i is of type `sostype[`i`]` and has variables `sosind[`j`]` with weight `sosref[`j`]` for `sosbeg[`i`]` $\le j <$ `sosbeg[`$i+1$`]`.

Solver drivers that use `suf_sos()` include `cplex/cplex.c`, `gurobi/gurobi.c`, and `xpress/xpress.c` in `http://ampl.com/netlib/ampl/solvers`.

Checking for quadratic programs: example of a *DAG* walk

Some solvers make special provision for handling quadratic programming problems, which have the form

$$\begin{aligned} \text{minimize or maximize} \quad & \tfrac{1}{2} x^T Q x + c^T x \\ \text{subject to} \quad & b \le Ax \le d \\ \text{and} \quad & \ell \le x \le u \end{aligned} \tag{QP}$$

in which $Q \in \mathbb{R}^{n \times n}$. Various solvers handle general positive-definite Q matrices, and the old KORBX solver handled positive-definite diagonal Q matrices (''convex separable quadratic programs''). These solvers generally assume the explicit ½ shown above in the (QP) objective.

AMPL considers quadratic forms, such as the objective in (QP), to be nonlinear expressions. To determine whether a given objective function is a quadratic form, it is necessary to walk the directed acyclic graph (*DAG*) that represents the (possibly) nonlinear part of the objective. For just determining whether an

objective or a constraint is quadratic, function `degree` with apparent signature

```
int degree_ASL(int co, void **pv);
```

does a simple graph walk on objective `co` if $0 \le$ `co` $<$ `n_obj` or on constraint $-$(`co`$+1$) if $0 \le -$(`co`$+1) <$ `n_con` and returns

–1 = bad `co` value
0 = constant
1 = linear
2 = quadratic
3 = general nonlinear.

The `pv` argument should be null if `degree(...)` is to be called just once (e.g., to see if the objective is quadratic). If multiple calls are expected (e.g., for objectives and constraints), it may save time to use the pattern

```
void *v = 0;
for(...) { ... degree(asl, co, &v); ... }
if (v) free(v);
```

Function `degree` and the `qpcheck` variants discussed next are meant to be used with a variant of `fg_read` called `qp_read` that has the same prototype as the other *stub*`.nl` readers, and which changes some function pointers to integers for the convenience of the `qpcheck` functions. After `qp_read` returns, you can invoke `degree` or `nqpcheck`, etc., one or more times, but you may not call `objval`, `conval`, etc., until you have called `qp_opify`, with apparent prototype

```
void qp_opify(void)
```

to restore the function pointers. With `solvers2`, one can simply call the desired *stub*`.nl` reader rather than `qp_read()`, and there is no need to call `qp_opify()`, which does nothing when `solvers2` is used.

Function `nqpcheck` (in `solvers/nqpcheck.c`) illustrates a more detailed graph walk that can determine quadratic coefficients. This function has apparent prototype

```
fint nqpcheck(int co, fint **rowqp, fint **colqp, real **delsqp);
```

its first argument indicates the constraint or objective to which it applies: `co` ≥ 0 means objective `co`, and `co` < 0 means constraint $-$(`co` $+ 1$). If the relevant objective or constraint is a quadratic form with Hessian Q, `nqpcheck` returns the number of nonzeros in Q (which is 0 if the function is linear), and sets its pointer arguments to pointers to arrays that describe Q. Specifically, `*delsqp` points to an array of the nonzeros in Q, `*rowqp` to their row numbers (first row = Fortran), and `*colqp` to an array of subscripts, incremented by `Fortran`, of the first entry in `*rowqp` and `*delsqp` for each column, with `(*colqp)[n_var]` giving the subscript just after the last column. `Nqpcheck` sorts the nonzeros in each column of Q by their row indices and returns a symmetric Q. For non-quadratic functions, `npqcheck` returns -1; it returns -2 in the unlikely case that it sees a division by 0, and -3 if `co` is out of range.

For solvers that only deal with one objective, it may be more convenient to call `qpcheck` rather than `nqpcheck; qpcheck` has apparent prototype

```
fint qpcheck(fint **rowqp, fint **colqp, real **delsqp);
```

It looks at objective `obj_no` (i.e., `asl->i.obj_no_`, with default value 0) and complains and aborts execution if it sees something other than a linear or quadratic form. When it sees one of the latter, it gives the same return value as `nqpcheck` and sets its arguments the same way.

Arguments `rowqp`, `colqp`, and `delsqp` to `nqpcheck()` and `qpcheck()` can be null; the return value is unaffected. When these arguments are not null, the memory pointed to by `*rowqp`, `*colqp`, and `*delsqp` is automatically freed when `ASL_free(&asl)` is called. Variant

```
fint mqpcheck(int co, fint **rowqp, fint **colqp, real **delsqp);
```

of nqpcheck() assigns values to *rowqp, *colqp, and *delsqp that are allocated individually by malloc() and must be freed by explicit calls on free().

Some solvers have specific ways of handling quadratic constraints. Such solvers can avoid some overhead by calling

```
ssize_t mqpcheckv(int co, QPinfo **QPIp, void **vp);
```

once per constraint (or objective), with final argument &v and with v initialized to null before the first call. Some auxiliary arrays are constructed and stored in memory pointed to by v and are reused on subsequent calls on mqpcheckv(). After the final such call, one invokes mqpcheckv_free(&v) to dispose of the auxiliary arrays; v is set to null by this call. (In calls on mqpcheckv(), the vp argument can itself be null, in which case the the auxiliary arrays are constructed anew on each call and are freed before mqpcheckv() returns.)

The QPIp argument to mqpcheckv() can be null. When it is not null, it is set to point to a QPinfo structure declared in asl.h:

```
typedef struct QPinfo {
 int nc;         /* number of nonempty columns */
 int nz;         /* number of nonzeros */
 int *colno;     /* column numbers of nonempty columns */
 size_t *colbeg; /* nonzeros for column colno[i]: */
 int *rowno;     /* (rowno[j], delsq[j]) for */
 real *delsq;    /* colbeg[i] <= j < colbeg[i+1], except that */
                 /* values in colno, colbeg, and rowno are */
                 /* incremented by Fortran */
 } QPinfo;
```

On most systems that use 64-bit addressing, size_t is an unsigned 64-bit integer, which allows addressing more than 2^{31} nonzeros, and ssize_t is a signed variant of size_t. (Though size_t is a standard type, on some systems it may be necessary to provide a suitable #define or typedef for ssize_t.) The QPI value returned by mqpcheckv(co, &QPI, vp), and the arrays whose values QPI contains are all allocated by a single malloc() call, so when done with QPI, a solver should call free(QPI) to dispose of QPI and the arrays to which it points.

Before returning quadratic details, the qpcheck() variants adjust constraint bounds and the objconst() function to account for constant terms.

Drivers cplex/cplex.c, gurobi/gurobi.c, and xpress/xpress.c call qp_read and qpcheck, and file examples/qtest.c illustrates invocations of nqpcheck and qp_opify.

More elaborate *DAG* walks are useful in other situations. For example, the nlc program discussed next does a more detailed *DAG* walk.

C or Fortran 77 for a problem instance: *nlc*

Occasionally it may be convenient to turn a *stub*.nl file into C or Fortran. This can lead to faster function and gradient computations — but, because of the added compile and link times, many evaluations are usually necessary before any net time is saved. Program nlc converts *stub*.nl into C or Fortran code for evaluating objectives, constraints, and their derivatives. You can get source for nlc as

```
http://ampl.com/netlib/ampl/solvers/nlc.tgz
```

By default, nlc emits C source for functions feval_ and ceval_; the former evaluates objectives and their gradients, the latter constraints and their Jacobian matrices (first derivatives). These functions have signatures

```
real feval_(fint *nobj, fint *needfg, real *x, real *g);
void ceval_(fint *needfg, real *x, real *c, real *J);
```

For both, x is the point at which evaluations take place, and *needfg tells whether the routines compute function values (if *needfg = 1), gradients (if *needfg = 2), or both (if *needfg = 3). For feval_,

`*nobj` is the objective number (0 for the first objective), and `g` points to storage for the gradient (when `*needfg` = 2 or 3). For `ceval_`, `c` points to storage for the values of the constraint bodies, and `J` points to columnwise storage for the nonzeros in the Jacobian matrix. Auxiliary arrays

```
extern fint funcom_[];
extern real boundc_[], x0comn_[];
extern fint auxcom_[1];
```

describe the problem dimensions, nonzeros in the Jacobian matrix, left- and right-hand sides of the constraints, bounds on the variables, the starting guess, and the number of nonlinear constraints (which come first). Specifically,

`funcom_[0] = n_var` = number of variables;
`funcom_[1] = n_obj` = number of objectives;
`funcom_[2] = n_con` = number of constraints;
`funcom_[3] = nzc` = number of Jacobian nonzeros;
`funcom_[4] = densej` is zero in the default case that the Jacobian matrix is stored sparsely, and is 1 if the full Jacobian matrix is stored (if requested by the `-d` command-line option to `nlc`).

`funcom_[`i`]`, $5 \le i \le 4 +$ `n_obj`, is 1 if the objective is to be maximized and 0 if it is to be minimized. If `densej = funcom_[4]` is 0, then `colstarts` = `funcom_ + n_obj` + 5 and `rownos` = `funcom_ + n_obj + n_var` + 6 are arrays describing the nonzeros in the columns of the Jacobian matrix: the nonzeros for column i (with i = 1 for the first column) are in `J[`j`]` for `colstarts[`$i-1$`]` $- 1 \le j \le$ `colstarts[`i`]` $- 2$, which looks more natural in Fortran notation: the calling sequences are compatible with the *f2c* calling conventions for Fortran.

Bounds are conveyed in `boundc_` as follows:

`boundc_[0]` is the value passed for ∞;
`boundc_ + 1` is an array of lower and upper bounds on the variables, and
`boundc_ + 2*n_var + 1` is an array of lower and upper bounds on the constraint bodies. The initial guess appears in `x0comn_`. Bounds of $\pm\infty$ are rendered as `1.7e308` and `-1.7e308`.

The `-f` command-line option causes `nlc` to emit Fortran 77 equivalents of `feval_` and `ceval_`; they correspond to the Fortran signatures

```
double precision function feval(nobj, needfg, x, g)
integer nobj,needfg
double precision x(*), g(*)
```

and

```
subroutine ceval(needfg, x, c, J)
integer needfg
double precision x(*), c(*), J(*)
```

and the auxiliary arrays are rendered as the COMMON blocks

```
common /funcom/ nvar, nobj, ncon, nzc, densej, colrow
integer nvar, nobj, ncon, nzc, densej, colrow(*)
common /boundc/ bounds
double precision bounds(*)
common /x0comn/ x0
double precision x0(*)
common /auxcom/ nlc
integer nlc  ! number of nonlinear constraints
```

where the *'s have the values described above. (Strictly speaking, it would be necessary to make problem-specific adjustments to the dimensions in other Fortran source that referenced these common blocks, but most systems follow the rule that the array size seen first wins, in which case it suffices to load the object for `feval` and `ceval` first.) The first `nobj` values of the `colrow` array have value 0 or 1 to indicate whether the corresponding objective is to be minimized (0) or maximized (1).

Command-line option -1 causes `nlc` to emit variants `feval0_` and `ceval0_` of `feval_` and

`ceval_` that omit gradient computations. They have signatures

```
real feval0_(fint *nobj, real *x);
void ceval0_(real *x, real *c);
```

With command-line option `-3`, `nlc` produces all four routines (or, if `-f` is also present, equivalent Fortran).

Writing *stub*`.nl` files for debugging

You can use AMPL's `write` command or its `-o` command-line flag to get a *stub*`.nl` (and any other needed auxiliary files) for use in debugging. Normally AMPL writes a binary-format *stub*`.nl`, which corresponds to a command-line `-ob`*stub* argument. Such files are faster to read and write, but slightly less convenient for debugging, in that `write_sol` notes the format of *stub*`.nl` (binary or ASCII — by looking at `binary_nl`) and writes *stub*`.sol` in the same format. To get ASCII format files, either issue an AMPL `write` command of the form

```
write g stub;
```

or use the `-og`*stub* command-line option. Your solver should see exactly the same problem, and AMPL should get back exactly the same solution, whether you use binary or ASCII format *stub*`.nl` and *stub*`.sol` files (if your computer has reasonable floating-point arithmetic).

With AMPL versions ≥ 19970214, binary *stub*`.nl` files written on one machine with binary IEEE-arithmetic can be read on any other.

Use with MATLAB® or Octave

It is easy to use AMPL with MATLAB or Octave. This requires the help of a *mex* file that reads *stub*`.nl` files, writes *stub*`.sol` files, and provides function, gradient, and Hessian values. Example file `amplfunc.c` is source for an `amplfunc.mex` that looks at its left- and right-hand sides to determine what it should do and works as follows:

```
[x,bl,bu,v,cl,cu] = amplfunc('stub')
```

or

```
[x,bl,bu,v,cl,cu,cv] = amplfunc('stub')
```

reads `stub.nl` and sets

`x` = primal initial guess,
`bl` = lower bounds on the primal variables,
`bu` = upper bounds on the primal variables,
`v` = dual initial guess (often a vector of zeros),
`cl` = lower bounds on constraint bodies,
`cu` = upper bounds on constraint bodies, and (when present)
`cv` = variables complementing constraints:
`cv[`*i*`]` > 0 means constraint *i* complements variable `cv[`*i*`]` – 1.

```
[f,c] = amplfunc(x,0)
```

sets

`f` = value of first objective at *x* and
`c` = values of constraint bodies at *x*.

```
[g,Jac] = amplfunc(x,1)
```

sets

`g` = gradient of first objective at *x* and
`Jac` = Jacobian matrix of constraints at *x*.

```
W = amplfunc(Y)
```

sets `W` to the Hessian of the Lagrangian (equation (*) in the section ‘‘Evaluating Nonlinear Functions’’ above) for the first objective at the point *x* at which the objective and constraint bodies were most recently evaluated. Finally,

```
[] = amplfunc(msg,x,v)
```

calls `write_sol(msg,x,v,0)` to write the *stub*`.sol` file, with

`msg` = termination message (a string),
`x` = optimal primal variables, and
`v` = optimal dual variables.

It is often convenient to use `.m` files to massage problems to a desired form. To illustrate this, the `examples` directory offers the following files (which are simplified forms of files used in joint work with Michael Overton and Margaret Wright):

• `init.m`, which expects variable `pname` to have been assigned a *stub* (a string value), reads *stub*`.nl`, and puts the problem into the form

$$\begin{aligned} &\text{minimize } f(x) \\ &\text{s.t. } c(x) = 0 \\ &\text{and } d(x) \geq 0. \end{aligned}$$

For simplicity, the example `init.m` assumes that the initial `x` yields `d(x)` > 0. A more elaborate version of `init.m` is required in general.

• `evalf.m`, which provides `[f,c,d] = evalf(x)`.

• `evalg.m`, which provides `[g,A,B] = evalg(x)`, where `A = c'(x)` and `B = d'(x)` are the Jacobian matrices of `c` and `d`.

• `evalw.m`, which computes the Lagrangian Hessian, `W = evalw(y,z)`, in which `y` and `z` are vectors of Lagrange multipliers for the constraints

$$c(x) = 0$$

and

$$d(x) \geq 0,$$

respectively.

• `enewt.m`, which uses `evalf.m`, `evalg.m` and `evalw.m` in a simple, non-robust nonlinear interior-point iteration that is meant mainly to illustrate setting up and solving an extended system involving the constraint Jacobian and Lagrangian Hessian matrices.

• `savesol.m`, which writes file *stub*`.sol` to permit reading a computed solution into an AMPL session.

• `hs100.amp`, an AMPL model for test problem 100 of Hock and Schittkowski [15].

• `hs100.nl`, derived from `hs100.amp`. To solve this problem, start MATLAB and type

```
pname = 'hs100';
init
enewt
savesol
```

`Amplfunc.c` provides dense Jacobian matrices and Lagrangian Hessians; `spamfunc.c` is a variant that provides sparse Jacobian matrices and Lagrangian Hessians. To see an example of using `spamfunc`, change all occurrences of ‘‘`amplfunc`’’ to ‘‘`spamfunc`’’ in the `.m` files.

5. Utility Routines and Interface Conventions

–AMPL Flag

Sometimes it is convenient for a solver to behave differently when invoked by AMPL than when invoked ‘‘stand-alone’’. This is why AMPL passes a string that starts with `-AMPL` as the second command-line argument when it invokes a solver. As a simple example, `nl21.c` turns `dn2gb`’s default printing off when it sees `-AMPL`, and it only invokes `write_sol` when this flag is present.

Conveying solver options

Most solvers have knobs (tolerances, switches, algorithmic options, etc.) that one might want to turn. An AMPL convention is that appending `_options` to the name of a solver gives the name of an environment variable (AMPL option) in which the solver looks for knob settings. Thus a solver named `wondersol` would take knob settings from `$wondersol_options` (the value of environment variable `wondersol_options`). For interactive use, it’s usually a good idea for a solver to print its name and perhaps version number when it starts, and to echo nondefault knob settings to confirm that they’ve been seen and accepted. It’s also conventional for the `msg` argument to `write_sol` to start with the solver’s name and perhaps version number. Since AMPL echoes the `write_sol`’s `msg` argument when it reads the solution, a minor problem arises: if there are no nondefault knob settings, an interactive user would see the solver’s name printed twice in a row. To keep this from happening, you can set `need_nl` (i.e., `asl->i.need_nl_`) to a positive value; this causes `write_sol` to insert that many backspace characters at the beginning of *stub*`.sol`. Usually this is done as follows: initially you execute, e.g.,

```
need_nl = printf("wondersol 3.2: ");
```

(Note that `printf` returns the number of characters it transmits — exactly what we need.) Subsequently, if you echo any options or otherwise print anything, also set `need_nl` to 0.

Conventionally, $*solver*`_options` may contain keywords and name-value pairs, separated by white space (spaces, tabs, newlines), with case ignored in names and keywords. For name-value pairs, the usual practice is to allow white space or an = (equality) sign, optionally surrounded by white space, between the name and the value. For debugging, it is sometimes convenient to pass keywords and name-value pairs on the solver’s command line, rather than setting $*solver*`_options` appropriately. The usual practice is to look first in $*solver*`_options`, then at the command-line arguments, so the latter take precedence.

Interface routines `getstub`, `getopts`, and `getstops` facilitate the above conventions. They have apparent prototypes

```
char *getstub (char ***pargv, Option_Info *oi);
int   getopts (char **argv,   Option_Info *oi);
char *getstops(char ***pargv, Option_Info *oi);
```

which you can import by saying

```
include "getstub.h"
```

rather than (or in addition to)

```
include "asl.h"
```

Type `Option_Info` is also declared in `getstub.h`; it is a structure whose initial components are

```
char *sname;          /* invocation name of solver */
char *bsname;         /* solver name in startup "banner" */
char *opname;         /* name of solver_options environment var */
keyword *keywds;      /* key words */
int n_keywds;         /* number of key words */
int want_funcadd;     /* whether funcadd will be called */
char *version;        /* for -v and Ver_key_ASL() */
char **usage;         /* solver-specific usage message */
Solver_KW_func *kwf;  /* solver-specific keyword function */
```

```
Fileeq_func *feq;     /* for n=filename */
keyword *options;     /* command-line options (with -) before stub */
int n_options;        /* number of options */
```

Ordinarily a solver declares

```
static Option_Info Oinfo = { ... };
```

and supplies only the first few fields (in place of ‘‘...’’), relying on the convenience of static initialization setting the remaining fields to zero.

Function `getstub` looks in `*pargv` for the *stub*, possibly preceded by command-line options that start with ‘‘`-`’’; `getstub` provides a small default set of command-line options, which may be augmented or overridden by names in `oi->options`. Among the default command-line options are `'-?'`, which requests a usage summary that reports `oi->sname` as the invocation name of the solver; `'-='`, which summarizes possible keyword values; `-v`, which reports the versions of the solver (supplied by `oi->version`) and of `amplsolver.a` (which is available in cell `ASLdate_ASL`, declared in `asl.h`); and, if `oi->want_funcadd` is nonzero, `-u`, which lists the available imported functions; imported functions are discussed in their own section above. If it finds a *stub*, `getstub` checks whether the next argument begins with `-AMPL` and sets `amplflag` accordingly; if so, it executes

```
if (oi->bsname)
      need_nl = printf("%s: ", oi->bsname);
```

At any rate, it sets `*pargv` to the command-line argument following the *stub* and optional `-AMPL` and returns the *stub*. It returns 0 (null) if it does not find a *stub*.

Function `getopts` looks first in $*solver*`_options`, then at the command line for keywords and optional values; `oi->opname` provides the name of the *solver*`_options` environment variable. `Getopts` is separate from `getstub` because sometimes it is convenient to call `jac0dim`, do some storage allocation, or make other arrangements before processing the keywords. For cases where no such separation is useful, function `getstops` calls `getstub` and `getopts` and returns the *stub*, complaining and exiting if none is found.

Keywords are conveyed in `keyword` structures declared in `getstub.h`:

```
 typedef struct keyword keyword;

 typedef char *Kwfunc(Option_Info *oi, keyword *kw, char *value);

struct keyword {
    char *name;
    Kwfunc *kf;
    void *info;
    char *desc;
    };
```

Array `oi->keywds` describes `oi->n_keywds` keywords that may appear in $*solver*`_options`; these `keyword` structures must be sorted (with comparisons as though by `strcmp`) on their `name` fields, which must be in lower case. Similarly, `oi->options` is an array of `oi->n_options` `keyword`s for initial command-line options, which must also be sorted; often `oi->n_options` = 0. The `desc` field of a `keyword` may be null; it provides a short description of the keyword for use with the `-=` command-line option. If `desc` starts with an = sign, the text in `desc` up to the first space is appended to the keyword in the output of the `-=` command-line option. The `kf` field provides a function that processes the value (if any) of the keyword. Its arguments are `oi` (the `Option_Info` pointer passed to `getstub`), a pointer `kw` to the `keyword` structure itself, and a pointer `value` to the possible value for the keyword (stripped of preceding white space). The `kf` function may use `kw->info` as it sees fit and should return a pointer to the first character in `value` that it has not consumed. Ordinarily `getopts` echoes any keyword assignments it processes (and sets `need_nl` = 0), but the `kf` function can suppress this echoing for a particular assignment by executing

```
oi->option_echo &= ~ASL_OI_echothis;
```

or for all subsequent assignments by executing

```
oi->option_echo &= ~ASL_OI_echo;
```

name	description of value
CK_val	known character value in known place
C_val	character value in known place
DA_val	real (double) value in asl
DK_val	known real (double) value in known place
DU_val	real (double) value: offset from uinfo
D_val	real (double) value in known place
IA_val	int value in asl
IK0_val	int value 0 in known place
IK1_val	int value 1 in known place
IK_val	known int value in known place
IU_val	int value: offset from uinfo
I_val	int value in known place
LK_val	known Long value in known place
LU_val	Long value: offset from uinfo
L_val	Long value in known place
SU_val	short value: offset from uinfo
Ver_val	report version
WS_val	set wantsol in Option_Info

Table 9: *keyword functions in* getstub.h.

For convenience, amplsolver.a provides a variety of keyword-processing functions. Table 9 summarizes these functions; their prototypes appear in getstub.h, which also provides a macro, nkeywds, for computing the n_keywds field of an Option_Info structure from a keyword declaration of the form

```
static keyword keywds[] = { ... };
```

To allow compilation by a K&R C compiler, it is best to cast the info fields to (Char*) (which is (char*) with K&R C and (void*) with ANSI/ISO C and C++). Often it is convenient to use macro KW, defined in getstub.h, for this. An example appears in file tnmain.c, in which the keywds declaration is followed by

```
static Option_Info Oinfo =
    { "tn", "TN", "tn_options", keywds, nkeywds, 1 };
```

Many other examples appear in various subdirectories of *netlib*'s ampl/solvers directory. Occasionally it is necessary to make custom keyword-processing functions, as in the example files keywds.c, rvmsg.c and rvmsg.h, which are discussed further below.

Some solvers, such as minos and npsol, have their own routines for parsing keyword phrases. For such a solver you can initialize oi->kwf with a pointer to a function that invokes it; if getopts sees a keyword that does not appear in oi->keywds, it changes any underscore characters to blanks and passes the resulting phrase to oi->kwf. Some solvers, such as minos, also need a way to associate Fortran unit numbers with file names; oi->feq (if not null) points to a function for doing this. See ampl/solvers/minos/m55.c for an example that uses all 12 of the Option_Info fields shown above, including oi->kwf and oi->feq.

Many solvers allow outlev to appear in $*solver*_options. Generally, outlev = 0 means ‘‘no printed output’’, and larger integers cause the solver to print more information while they work. Another common keyword is maxit, whose value bounds the number of iterations allowed. For stand-alone

invocations (those without `-AMPL`), solvers commonly recognize `wantsol=`*n*, where *n* is the sum of

1 to write a `.sol` file,
2 to print the primal variable values,
4 to print the dual variable values, and
8 to suppress printing the solution message.

A special keyword function, `WS_val`, processes `wantsol` assignments, which are interpreted by `write_sol`. Strings `WS_desc_ASL` and `WSu_desc_ASL` provide descriptions of `wantsol` for constrained and unconstrained solvers, respectively, and appear in many of the sample drivers available from *netlib*.

Printing and `Stderr`

To facilitate using AMPL and solvers in some contexts, such as some versions of Microsoft Windows, it is best to route all printing through `printf` and `fprintf`. Because of this, and because some systems furnish a `sprintf` that does not give the return value specified by ANSI/ISO C, `amplsolver.a` provides suitable versions of `printf`, `fprintf`, `snprintf`, `sprintf`, `vfprintf`, `vsnprintf`, and `vsprintf` that function as specified by ANSI/ISO C, except that they do not recognize the `F`, `L`, `j`, `ll`, `t`, or `z` qualifiers or the `%n` format item; as in AMPL, they provide some extensions: they turn `%.0g` and `%.0G` into the shortest decimal string that rounds to the number being converted, and they allow negative precisions for %f. These provisions apply to systems with IEEE, VAX, or IBM mainframe arithmetic, and comments in `solvers/makefile` explain how to use the system's `printf` routines on other systems.

On systems where it is convenient to redirect `stderr`, it is best to write error messages to `stderr`. Unfortunately, redirecting `stderr` is inconvenient on some systems (e.g., Microsoft systems with the usual Microsoft shells). To promote portability among systems, `amplsolver.a` provides access to

```
extern FILE *Stderr,
```

which can be set, as appropriate, to `stderr` or `stdout`. Thus we recommend writing error messages to `Stderr` rather than `stderr`, as is illustrated in various examples discussed above.

Formatting the optimal value and other numbers

An AMPL convention is that solvers should report (in the `msg` argument to `write_sol`) the final objective value to `$objective_precision` significant figures. Interface routines `g_fmtop` and `obj_prec` facilitate this. They have apparent prototypes

```
int g_fmtop(char *buf, double v);
int obj_prec(void);
```

For use as the ‘‘*’’ argument in the format `%.*g`, `obj_prec` returns `$objective_precision`. Occasionally it may be convenient to use `g_fmtop` instead. It stores the appropriate decimal approximation in `buf` (using the same conversion routine as AMPL's printing commands), and returns the number of characters (excluding the terminating null) it has stored in `buf`. The end of `nl21.c` illustrates both the use of `g_fmtop` and of the `Sprintf` in `amplsolver.a`. The latter is there because, contrary to standard (ANSI/ISO) C, the `sprintf` on some systems does not return the count of characters written to its first argument. Ordinarily, `Sprintf` is the `sprintf` described above in the section ‘‘Printing and `stderr`’’, but if you are using the system's `sprintf`, then `Sprintf` is similar to `sprintf`, but only understands `%c`, `%d`, `%ld`, and `%s` (and complains if it sees something else).

Two relatives of `g_fmtop` that are also in `amplsolver.a` are

```
int g_fmt( char *buf, double v);
int g_fmtp(char *buf, double v, int prec);
```

`g_fmtp` rounds its argument to `prec` significant figures unless `prec` is 0, in which case it stores in `buf` the shortest decimal string that rounds to `v` (provided the machine uses IEEE, VAX, or IBM mainframe arithmetic: see [10]); `g_fmt(buf,v)` = `g_fmtp(buf,v,0)`.

If they find an exponent field necessary, both `g_fmtop` and its relatives delimit it with the current value of

```
extern char g_fmt_E;
```

(whose declaration appears in `asl.h`). The default value of `g_fmt_E` is `'e'`.

By default, `g_fmtop` and its relatives only supply a decimal point if it is followed by a digit, but if you set

```
extern int g_fmt_decpt;
```

(declared in `asl.h`) to a nonzero value, they always supply a decimal point when `v` is finite. If you set `g_fmt_decpt` to 2, these routines supply an exponent field for finite `v`. The `nlc` program discussed above uses these features when it writes Fortran.

A ‘‘Solver’’ for Gradients and Hessians: gjh

File `examples/gjh.c` is source for a ‘‘solver’’ that computes objective and constraint gradients and the Lagrangian Hessian at the current (primal and dual) point and writes them to a file that can be read in an AMPL session (via ‘‘include filename’’ or ‘‘model filename;’’) to make these first and second derivative values available as AMPL `param`s. The `solve_message` gives the filename as well as a command for removing the file after reading it.

More examples

Some examples illustrating the above points appear in the `solvers/examples` directory. One such example is `tnmain.c`, a wrapper for Stephen Nash’s `LMQN` and `LMQNBC` [18, 17], which solve unconstrained and simply bounded minimization problems by a truncated Newton algorithm. Since `tnmain.c` calls `getstub`, the resulting solver, `tn`, explains its usage when invoked

```
tn '-?'
```

and summarizes the keywords it recognizes when invoked

```
tn '-='
```

For another example, files `mng.c` and `nl2.c` are for solvers called `mng` and `nl2`, which are more elaborate variants of the `mng1` and `nl21` considered above (source files `mng1.c` and `nl21.c`). Both use auxiliary files `keywds.c`, `rvmsg.c` and `rvmsg.h` to turn the knobs summarized in [11] and pass a more elaborate `msg` to `write_sol`. Their linkage, in `examples/makefile`, also illustrates adding imported functions, which we will discuss shortly. Unlike `mng1`, `mng` checks to see if the objective is to be maximized and internally negates it if so.

File `mnh.c` is a variant of `mng.c` that supplies the analytic Hessian matrix computed by `duthes` to solver `mnh`, based on PORT routine `dmnhb`. For maximum likelihood problems, it is sometimes appropriate to use the Hessian at the solution as an estimate of the variance-covariance matrix; `mnh` offers the option of computing standard-deviation estimates for the optimal solution from this variance-covariance matrix estimate. Specify `stddev=1` in `$mnh_options` or on the command line to exercise this option, or specify `stddev_file=`*filename* to have this information written to a file.

Various subdirectories of

```
http://ampl.com/netlib/ampl/solvers/
```

provide other examples of drivers for linear and nonlinear solvers. See

```
http://ampl.com/netlib/ampl/solvers/README
```

for more details.

Evaluation Test Program

File `et.c` is source for program *et* (*e*valuation *t*ester) for testing various kinds of ASL evaluations. It can use the various readers to read specified *stub*`.nl` files, print and change the current primal and dual variable values, compute objective function and constraint bodies, their derivatives, Lagrangian Hessians and Hessian-vector products, and use finite differences to check computed derivatives. The *et* program reads commands from the standard input or a file or files given on the command line and produces output in response to them. After invoking `et`, type a question mark and carriage return to see a detailed usage summary, or invoke

```
et -?
```

or

```
et --help
```

for a brief usage summary.

Multiple problems and multiple threads

It is possible to have several problems in memory at once, each with its own `ASL` pointer. To free the memory associated with a particular `ASL` pointer `asl`, execute

```
ASL_free(&asl);
```

this call sets `asl` = 0. To allocate problem-specific memory that will be freed by `ASL_free`, call `M1alloc` rather than `Malloc`. Do not pass such memory to `realloc` or `free`.

Variant `solvers2` of the `solvers` directory, with source

```
http://ampl.com/netlib/ampl/solvers2.tgz
```

is designed for use with multiple threads — and for large nonlinear problems is often more efficient than the facilities in the `solvers` directory. For using `solvers` or `solvers2` with multiple threads, `amplsolver.a` should be compiled with `ALLOW_OPENMP` `#define`d if *OPENMP* is available, or with `MULTIPLE_THREADS` `#define`d along with suitable `#define ACQUIRE_DTOA_LOCK(`*n*`)` and `FREE_DTOA_LOCK(`*n*`)` directives to provide exclusive access to a few short critical regions (with distinct values of *n*). A possible approach is first to create `arith.h` by saying ‘‘`make arith.h`’’, then to add

```
#define ALLOW_OPENMP
```

or else

```
#define MULTIPLE_THREADS
```

and suitable definitions of `ACQUIRE_DTOA_LOCK(`*n*`)` and `FREE_DTOA_LOCK(`*n*`)` to the end of `arith.h`, and finally to create `amplsolver.a` by saying ‘‘`make`’’.

With `solvers2`, each thread can use its own *EvalWorkspace* pointer `ew`, acquired by invoking apparent function

```
EvalWorkspace *ew = ewalloc();
```

The problem-specific functions, such as `objval()` and `objgrd()` discussed above have equivalent thread-specific forms with `_ew` appended to the function name and `ew` as the first argument, e.g.,

```
real objval_ew(EvalWorkspace *ew, real *x);
real objvale_ew(EvalWorkspace *ew, real *x, fint *nerror);
int xknown_ew(EvalWorkspace *ew, real *X)
int xknowne_ew(EvalWorkspace *ew, real *X, fint *nerror)
```

see `solvers2/asl.h` for more details.

Acknowledgment

Thanks go to Bob Fourer, Brian Kernighan, Bob Vanderbei, and Margaret Wright for helpful comments.

REFERENCES

[1] E. M. L. BEALE AND J. A. TOMLIN, ‘‘Special Facilities in a General Mathematical System for Non-Convex Problems Using Ordered Sets of Variables,’’ pp. 447–454 in *Proceedings of the Fifth International Conference on Operational Research*, ed. J. Lawrence, Tavistock Publications, London (1970).

[2] A. R. CONN, N. I. M. GOULD, AND PH. L. TOINT, *LANCELOT, a Fortran Package for Large-Scale Nonlinear Optimization (Release A)*, Springer-Verlag, 1992. Springer Series in Computational Mathematics 17.

[3] J. E. DENNIS, JR., D. M. GAY, AND R. E. WELSCH, ‘‘An Adaptive Nonlinear Least-Squares Algorithm,’’ *ACM Trans. Math. Software* **7** (1981), pp. 348–368.

[4] J. E. DENNIS, JR., D. M. GAY, AND R. E. WELSCH, ‘‘Algorithm 573. NL2SOL—An Adaptive Nonlinear Least-Squares Algorithm,’’ *ACM Trans. Math. Software* **7** (1981), pp. 369–383.

[5] S. I. FELDMAN, D. M. GAY, M. W. MAIMONE, AND N. L. SCHRYER, ‘‘A Fortran-to-C Converter,’’ Computing Science Technical Report No. 149 (1990), Bell Laboratories, Murray Hill, NJ.

[6] ROBERT FOURER, DAVID M. GAY, AND BRIAN W. KERNIGHAN, *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press/Wadsworth, 1993. ISBN: 0-89426-232-7.

[7] ROBERT FOURER, DAVID M. GAY, AND BRIAN W. KERNIGHAN, *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press / Brooks/Cole Publishing Company, 2003. second edition, ISBN: 0-534-38809-4.

[8] D. M. GAY, ‘‘ALGORITHM 611—Subroutines for Unconstrained Minimization Using a Model/Trust-Region Approach,’’ *ACM Trans. Math. Software* **9** (1983), pp. 503–524.

[9] D. M. GAY, ‘‘A Trust-Region Approach to Linearly Constrained Optimization,’’ pp. 72–105 in *Numerical Analysis. Proceedings, Dundee 1983*, ed. D. F. Griffiths, Springer-Verlag (1984).

[10] D. M. GAY, ‘‘Correctly Rounded Binary-Decimal and Decimal-Binary Conversions,’’ Numerical Analysis Manuscript 90-10 (11274-901130-10TMS) (1990), Bell Laboratories, Murray Hill, NJ.

[11] D. M. GAY, ‘‘Usage Summary for Selected Optimization Routines,’’ Computing Science Technical Report No. 153 (1990), AT&T Bell Laboratories, Murray Hill, NJ.

[12] D. M. GAY, ‘‘More AD of Nonlinear AMPL Models: Computing Hessian Information and Exploiting Partial Separability,’’ in *Computational Differentiation: Applications, Techniques, and Tools*, ed. George F. Corliss, SIAM (1996).

[13] A. GRIEWANK AND PH. L. TOINT, ‘‘On the Unconstrained Optimization of Partially Separable Functions,’’ pp. 301–312 in *Nonlinear Optimization 1981*, ed. M. J. D. Powell, Academic Press (1982).

[14] A. GRIEWANK AND PH. L. TOINT, ‘‘Partitioned Variable Metric Updates for Large Structured Optimization Problems,’’ *Numer. Math.* **39** (1982), pp. 119–137.

[15] W. HOCK AND K. SCHITTKOWSKI, *Test Examples for Nonlinear Programming Codes*, Springer-Verlag, 1981.

[16] B. A. MURTAGH, in *Advanced Linear Programming: Computation and Practice*, McGraw-Hill, New York (1981).

[17] S. G. NASH, ‘‘Newton-type Minimization via the Lanczos Method,’’ *SIAM J. Num. Anal.* **21** (1984), pp. 770–788.

[18] S. G. NASH, ‘‘User’s Guide for TN/TNBC: Fortran Routines for Nonlinear Optimization,’’ Report 397 (1984), Mathematical Sciences Dept., The Johns Hopkins Univ., Baltimore, MD.

[19] PH. L. TOINT, ‘‘User’s Guide to the Routine VE08 for Solving Partially Separable Bounded Optimization Problems,’’ Technical Report 83/1 (1983), FUNDP, Namur, Belgium.