

Update on the AMPL[®]/Solver Interface Library

Robert Fourer

Dept. of Industrial Engineering
and Management Sciences

Northwestern University

`4er@iems.nwu.edu`

David M. Gay

AMPL Optimization LLC

`dmg@acm.org`

`dmg@ampl.com`

Outline

1. Components of AMPL
2. Problem areas (historical view)
3. Design flexibility
4. Documented solver-interface library
5. Undocumented stuff
6. Work in progress
7. Other environments, e.g.,
MATLAB, Java, VB
8. Summary

1. Components of AMPL

- a. *Model* = declared problem
 keyboard-math notation
 general indexing sets
- b. Commands to show state
 display, print, printf
- c. Commands to change state
 — retaining declared relations

d. Separate data specification

data sections

read table

database or spreadsheet

etc. via custom handlers

let

call

2. Problem Areas (historical view)

Linear

Integer

Nonlinear

AD for derivatives

auto. use of partial separability
for Hessians (by AD)

Complementarity

Forthcoming

logic programming

stochastic programming

3. Design Flexibility

Processes files

stdin = interactive

or server mode for

GUIs, COM objects

Separate solvers

general solver interface lib.

solvers can run elsewhere

Imported function libraries

Table handlers

Separation of model & data, and automatic recomputations permit

- interactive model development
- solving similar problems:
 - same model
 - but updated data
 - or all new data

4. Documented Solver-Interface Library

Readers read .nl files and set up data structures for efficient evaluation of desired information:

f_read	for LPs and MILPs
qp_read	for QPs
fg_read	for f and ∇f
pfgh_read	for f , ∇f , and $\nabla^2 f$

Library routines compute $f, \nabla f$ for

- objectives
- constraint bodies
 - * all at once
 - * singly
 - * subarray

- Hessian of the Lagrangian

$$H = \nabla^2 f + \sum_i \lambda_i \nabla^2 c_i$$

Readers provide Jacobian sparsity.

sphsetup gives sparsity of Hessian H .

5. Undocumented stuff

Statuses and suffixes

Returning (new) suffixes

Symbolic suffixes

solve_result and

solve_result_num

Example of (basis) statuses...

```
ampl: model diet.mod; data diet2a.dat;  
solve;  
MINOS 5.5: optimal solution found.  
13 iterations, objective 118.0594032  
display Buy;  
Buy [*] :=  
BEEF      5.36061  
  CHK      2  
FISH      2  
  HAM     10  
  MCH     10  
  MTL     10  
  SPG     9.30605  
  TUR      2  
;
```

```
ampl: minimize mtl: Buy['MTL'];
ampl: solve mtl;
MINOS 5.5: optimal solution found.
1 iterations, objective 5.810623557
ampl: display Buy.lb, Buy, Buy.ub,
ampl? Buy.status;

:      Buy.lb      Buy      Buy.ub  Buy.status
BEEF      2      10      10      upp
CHK       2       2      10      low
FISH      2       2      10      low
HAM       2      10      10      upp
MCH       2      10      10      upp
MTL       2      5.81062      10      bas
SPG       2      8.85604      10      bas
TUR       2       2      10      low
;
```

```
ampl:  display Diet.body, Diet,  
ampl?  Diet.status;
```

:	Diet.body	Diet	Diet.status
A	1956.29	0	bas
B1	1036.26	0	bas
B2	700	0.404585	low
C	1682.51	0	bas
CAL	19794.6	0	bas
NA	50000	-0.00306905	upp
;			

```

ampl: option *status_table;
option astatus_table '\
0      in      normal state (in problem)\
1      drop    removed by drop command\
2      pre     eliminated by presolve\
3      fix     fixed by fix command\
4      sub     defined variable, substituted out\
5      unused  not used in current problem\
6      log     logical constraint in current problem\
option sstatus_table '\
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 bound\
6      btw     nonbasic between bounds\

```



```
ampl: print $solve_result_table;
```

0	solved
100	solved?
200	infeasible
300	unbounded
400	limit
500	failure

```
ampl: display solve_result,
```

```
ampl: solve_result_num;
```

```
solve_result = solved
```

```
solve_result_num = 0
```

```
ampl: reset data; data diet2.dat;  
ampl: solve;  
MINOS 5.5: infeasible problem.  
9 iterations  
Objective = Total_Cost  
ampl: display solve_result,  
ampl? solve_result_num;  
solve_result = infeasible  
solve_result_num = 200
```

To set **solve_result**, solvers assign
solve_result_num.

Example of solver-declared suffix:

```
ampl: option solver cplex;
ampl: option cplex_options 'iisfind=1';
ampl: solve;
CPLEX 8.0.0: iisfind=1
CPLEX 8.0.0: infeasible problem.
0 simplex iterations (0 in phase I)
Returning iis of 7 variables and 2 constraints.
constraint.dunbdd returned
6 extra dual simplex iterations for ray (4 in pha

suffix iis symbolic OUT;
```

```
option iis_table '\
0          non      not in the iis\
1          low      at lower bound\
2          fix      fixed\
3          upp      at upper bound\
suffix dunbdd OUT;
Objective = Total_Cost
```

```
ampl: display Buy.iis;
```

```
Buy.iis [*] :=
```

```
BEEF    upp
```

```
  CHK    low
```

```
FISH    low
```

```
  HAM    upp
```

```
  MCH    non
```

```
  MTL    upp
```

```
  SPG    low
```

```
  TUR    low
```

```
;
```

```
ampl: display Diet.iis;  
Diet.iis [*] :=  
    A    non  
    B1   non  
    B2   low  
    C    non  
CAL     non  
    NA   upp  
;
```

```
ampl: display {i in 1.._nvars:
ampl? P_var[i].iis != 'non'}
ampl? (_varname[i], _var[i].iis);
:      _varname[i]      _var[i].iis      :=
1      "Buy['BEEF']"      upp
2      "Buy['CHK']"      low
3      "Buy['FISH']"      low
4      "Buy['HAM']"      upp
6      "Buy['MTL']"      upp
7      "Buy['SPG']"      low
8      "Buy['TUR']"      low
;
```

6. Work in Progress

Logic programming extensions:

New logical operators

numberof

$\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$

exactly, atleast, atmost

alldiff

count

Solver interface library evaluates new logical operators

— but *treewalk* may be needed to build solver data structures.

Example: Fourer's interface to ILOG Solver.

Example: interface to Globsol (still in progress).

Plan for *stochastic programming* is to introduce random variables, assigned distributions by **let**. Solvers could do their own sampling or ask the interface library to sample from the distributions specified by in the AMPL session.

Solvers would treat derived random variables much like defined variables.

7. Other environments (MATLAB, Java, etc.)

Mex functions

amplfunc (dense Jacobians)

and

spamfunc (sparse Jacobians)

make problem information (f , ∇f , etc.)

available to MATLAB — described since 1997
in *Hooking Your Solver to AMPL*.

Similar arrangements can be made for Java by Java Native Interface. Had preliminary JNI interface in 2000; not yet pursued further, in part because of no perceived interest.

John Chinneck reports an interface to VB.

Similar interface to .NET seems plausible.

8. Summary

Some new stuff, such as
 statuses, suffixes, **solve_result**
work now but need better documentation.

Logic-programming extensions are partly done;
variables in subscripts are
 not yet done and
 will require interface library extensions.

Some interfacing to other systems works now; more would be good. Watch

`http://www.ampl.com`

for news of enhancements. Other pointers:

`http://www.ampl.com/hooking.html`

for *Hooking Your Solver to AMPL* and

`http://www.ampl.com/BOOK`

for AMPL book info.