ISMP 2018, Bordeaux, 2–6 July 2018 and

EURO 2018, Valencia, 8–11 July 2018

Adding Functions to AMPL

David M. Gay

AMPL Optimization, Inc.

Albuquerque, New Mexico, U.S.A.

dmg@ampl.com

http://www.ampl.com

A Language for Mathematical Programming

AMPL is a language for expressing mathematical programming problems involving finitely many constraints and objectives over finitely many continuous or discrete variables. Model entities include

- sets
- parameters
- variables
- constraints and objectives
- *functions*

Small nonlinear example using a function

```
ampl:  var x; s.t. c: sin(x) = .5;
ampl:  solve;
MINOS 5.51: optimal solution found.
...
ampl:  print x; print asin(.5);
0.523598775598298 8
0.523598775598298 9
ampl:  display x - asin(.5);
x - asin(0.5) = -1.11022e-16
```

Load Library

When the builtin functions do not suffice, AMPL's `load` command can introduce libraries of "imported" functions that have been compiled from suitable programming languages, such as C, C++, and Fortran. For example, the GNU Scientific Library, compiled for use with `load`, is available from `https://ampl.com/resources/extended-function-library` .

More generally, `https://ampl.com/netlib/ampl/solvers/funclink` provides details for compiling your own function library.

Load library example

Example: `gsl_log1p`(x) computes $log(1+x)$, avoiding the roundoff error that would occur in computing $1+x$ for small $|x|$.

```
ampl:  load amplgsl.dll;
ampl:  function gsl_log1p;
ampl:  display log(1 + 5e-16), gsl_log1p(5e-16);
log(1 + 5e-16) =  4.44089e-16
gsl_log1p(5e-16) =  5e-16
ampl:  print log(1 + 1.2e-17), gsl_log1p(1.2e-17);
0 1.2e-17
```

Out args

An imported function can have "out args", arguments to which the function provides values. For example, if file `foo` contains

```
load swap.dll;
param a; param b;
data; param a := 1.2; param b := 3.4;
function swap(INOUT, INOUT);
display a, b;
display swap(a,b); ## or "call swap(a,b);"
display a, b;
```

then invoking "`ampl foo`" produces the output...

Output from "ampl foo"

```
a = 1.2
b = 3.4

swap(a, b) = 1

a = 3.4
b = 1.2
```

Why should functions be expressed in AMPL?

- Many MIP solvers permit "callback" functions to influence their solution algorithms. Introducing functions expressed in AMPL would permit making better interfaces to such solvers.
- AMPL functions might simplify some scripts.
- AMPL functions might help express some nonlinear problems.

AMPL function "swap"

Syntax for "swap" in AMPL:

```
function swap(param a INOUT, param b INOUT)
   returns ()
{ param t;
  let t := a;
  let a := b;
  let b := t;
  }
```

Use:

```
call swap(x,y);
```

cleaner AMPL function "swap"

Syntax for a cleaner "swap" in AMPL:

```
function swap2(a, b) returns(param,param)
{ return (b,a); }
```

Use:

```
let (x,y) := swap2(x,y);
```

Obvious alternative:

```
let (x,y) := (y,x);
```

Returning sets and tuples

Sometimes it is useful to return sets...

```
set S; set T;
param p{S};
...
let T := argmaxset(p);
# T = {t in S: p[t] == max{i in S} p[i]}
```

or tuples of values and sets

```
let (t,T) := argmax(p);
# T = {t in S: p[t] == max{i in S} p[i]}
# and t = p[s] for s in T
```

Contexts

A function body must be a new context: it would make no sense for names used locally in the function to be visible outside the function. But it is convenient to let functions access values from the surrounding context:

```
param a;
function foo(b) { return a + b; }
```

Declarations in contexts

While extending AMPL to accept function declarations, it is only a small step further to allow declarations within a context generally. Such declarations disappear when the context ends. For example ... (next slide)

Inner context declaration example

```
param a := 1.23;
{ # new context
  param a := 4.56;
  display a;
  } # end of context
display a;
```

produces output

```
a = 4.56
a = 1.23
```

Parsing challenge with domains

AMPL function declarations have long been allowed to specify a domain for each argument, e.g.,

```
function hypot(Reals,Reals);
```

It is nice to allow

```
function foo(a,b) { return a + 2*b; }
```

which is easy to handle if the arguments are unbound symbols. The argument list of an AMPL function should be a new context, but for imported functions we must allow set expressions for domains. Various remedies are possible; for now, new keyword "`new`" indicates that a parameter name has a new meaning.

Example of "new"

```
param a; param b;
function foo(a,b) { return a + 2*b; }
```

produces a syntax error message, but

```
param a; param b;
function foo(new a,b) { return a + 2*b; }
```

is fine; "new" is only needed before the first argument. Keyword "new" also applies to inner contexts, affecting whether a redefinition warning or error is issued, depending on the (new) option `hidewarn` setting.

Closures

For use in callbacks, our plan is to convey "closures" with functions in .nl files. This will permit the functions to access values from outer contexts. Whether changes to these values are communicated back to the AMPL session will be governed by a new option, just as option `send_suffixes` determines whether suffix values in `.sol` files are returned to the AMPL session.

Restrictions on AMPL functions

AMPL functions visible to solvers, whether in callbacks or nonlinear expressions, will not be allowed to declare variables or execute commands other than `let`, `return`, and flow-of-control commands. In addition, functions in nonlinear expressions will not be allowed to have OUT args.

AD for AMPL functions

AMPL itself (aside from imported functions) has been a *primitive recursive* language. For example, `.nl` files do not contain loops — all loops have been expanded by the AMPL processor before it writes the `.nl` file. This allows the AMPL/solver interface library (ASL) to set up structures needed for automatic differentiation in the course of reading the `.nl` file. Imported functions participate by providing first and possibly second partial derivatives for their numeric arguments.

AD for AMPL functions (cont'd)

AMPL functions appearing in objectives and constraints could be fully (and mutually) recursive, which will require the ASL to use techniques commonly used in various AD packages (such as ADOL-C and Sacado) to store partials and other details in dynamically allocated arrays. This is more general but also somewhat slower and takes more memory.

Recursive functions versus recursive sets and params

AMPL has long allowed recursive set and parameter definitions, such as

```
param factorial{i in integer[0, Infinity)}
      = if i < 2 then i else i*factorial[i-1];
```

Recursive set and parameter definitions effectively cache their computed values, so are automatically efficient. A purely recursive function may be much *less* efficient.

Function arguments in AMPL scripts

Imported functions (made available with `load` commands) can presently only be called with numeric or string arguments. Given the declarations

```
set S; param p{S};
function foo;
```

imported function foo could only be provided all of param p by a call of the form

```
call foo(card(S), {i in S} p[i]);
```

(next slide)

Function arguments in AMPL scripts (cont'd)

But an AMPL function foo, declared with

```
function foo(param p{dimen 1});
```

or

```
function foo(param p{dimen 1}) { ... }
```

or

```
set S; # ...
function foo(param p{S}) { ... }
```

could simply be called by

```
call foo(p);
```

Function arguments in AMPL scripts (cont'd)

Within `foo`, `p`'s declared and actual indexing sets could be accessed by the (provisionally named) new builtin functions `dind` and `indx`, as in

```
dind(p)
```

and

```
indx(p)
```

so

```
sum{i in indx(p)} p[i]
```

would be the sum of all components of `p`.

Function arguments in AMPL scripts (cont'd)

Also allowed as arguments to and return values from AMPL functions will be sets, indexed collections of sets, and functions. Later we may add tuples as atoms, allowing sets to contain any of the possible argument types. Other possible atoms include complex, rational, and complex rational numbers. (Other long-intended extensions, such as variables in subscripts, may happen first.)

Aside on `dind` and `indx`

Given the AMPL input

```
set S; param p{S};
data; set S := a b c;
param p := a 1 c 2.5;
```

we would have

```
indx(p) = {'a', 'c'}
dind(p) = {'a', 'b', 'c'}
```

Declaring function arguments

Arguments can be declared immediately:

```
function foo(set S, param p{S})
```

or first listed, then declared:

```
function foo(p, S; set S; param p{S})
```

either of which could restrict the indexing set of `p` in the body of `foo`, in which

```
indx(p) = S.
```

`S` would have to be a subset of the indexing set of the var or param passed as `p`.

Status

AMPL functions are mostly implemented and should be working soon. We intend to make 64-bit "beta" binaries for Linux, MacOSX, and MS Windows available from the AMPL web site, `https://ampl.com`. The "beta" binaries will use the usual AMPL licensing mechanism and will work with current AMPL licenses. Initially AMPL functions will only work in AMPL scripts. Extensions to the solver-interface library (ASL) are further in the future.

Some references

The AMPL web site

`https://ampl.com`

has more on AMPL, including pointers to papers on AMPL and on the AMPL/solver interface library (ASL). When available, pointers to beta copies of AMPL with function extensions will appear on the AMPL web site.

For more on AD (automatic differentiation), see

`http://www.autodiff.org`