XV International Conference on Stochastic Programming,

28 July – 2 August 2019

Forthcoming AMPL Updates and Possible Relevance to Stochastic Programming

David M. Gay

AMPL Optimization, Inc.

Albuquerque, New Mexico, U.S.A.

dmg@ampl.com

http://www.ampl.com

Existing stochastic facilities: random variables

The "random" keyword makes a variable *random*. Internally, random variables act like nonlinear variables.

```
var x random;
```

Declarations may specify a value (with `=` or `default`):

```
var y random = Uniform01();
```

or subsequently be assigned:

```
let x := Normal(0,2);
```

"random variable" versus "random parameter"

For people who prefer "random parameter", there is now an automatic conversion to random variable...

```
ampl:  param p random;
Caution:  Converting p to a random variable.
context:  param p >>> random; <<<
ampl:  show p;
var p random;
```

The warning can be suppressed by specifying

```
option randparam_warn 0;.
```

Dependent random variables

Dependent random variables may be declared in `var` ... `=` and `var` ... `default` declarations:

```
var x random;

var y = x + 1;
```

Random variables may appear as variables in constraint and objective declarations:

```
s.t. Demand: sum {i in A} build[i] >= y;
```

Seeing random variable values

Printing commands see random variables as strings expressing distributions...

```
var x random = Normal01();

var y = x + Uniform(3,5);

display x, y;
```

gives

```
x = 'Normal01()'

y = 'Uniform(3, 5) + x'
```

Sampling random variables

```
display {1..5} (Sample(x), Sample(y));
```

gives

```
:    Sample(x) Sample(y)     :=
1     1.51898    3.62453
2    -3.65725    2.50557
3    -0.412257   5.4215
4     0.726723   2.89672
5    -0.606458   3.776
;
```

Builtin functions of random variables

Builtin functions for solvers to interpret:

- `Expected`(ξ)
- `Moment`(ξ, n), $n = 1, 2, 3, ...$
- `Percentile`(ξ, p), $0 \leq p \leq 100$
- `Sample`(ξ)
- `StdDev`(ξ)
- `Variance`(ξ)
- `Probability`$(logical\ condition)$

"System suffix" `.stage` marks what happens when

Reserved suffix `.stage`, e.g.,

```
set A; set Stages;

var x {A, s in Stages} suffix stage s;
```

or

```
var x {A, s in Stages};

...

let {a in A, s in Stages}

        x[a,s].stage := s;
```

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.

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.

Recursive function example: factorial

```
function fact(n)
{
        if n <= 2 then {
                if n <= 1 then
                        let n := 1;
                return n;
                }
        return n * fact(n-1);
        }
```

Running the *fact* function

```
ampl: display{i in 0..5} fact(i);
fact(i) [*] :=
0    1
1    1
2    2
3    6
4   24
5  120
;
```

"show" command for a function

```
ampl: show fact;
function fact(param n)
{
 if n <= 2 then {
  if n <= 1 then
   let n := 1;
  return n;
  }
 return n*fact(n - 1);
 }
```

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.

Keyword *new*

File *foo*:

```
param a; let a := 4.2;
function foo(a,b) { return a + b; }
```

Invoking "`ampl foo`" gives output

```
foo1, line 2 (offset 32):
        syntax error
context:  function  >>> foo(a, <<< b) { return a +
```

But "`function foo(new a, b)`"
or "`function foo(param a, b)`" is OK.

Local context

```
param a; let a := 4.2;
function foo(new a,b) { return a + b; }
show foo;
display a;
```

...gives output

```
  function foo(param a, param b)
  {
   return a + b;
   }
a = 4.2
```

Declarations within contexts now allowed

```
param a := 1.2;
{ param a := 3.4;
  print 'Inner context 1: a =', a;
  { param a := 5.6;
    print 'Inner context 2: a =', a;
   }
  print 'Back to inner context 1: a =', a;
  }
print 'Outermost context: a =', a;
```

But we may forbid variable, constraint, and objective declarations in inner contexts.

Commands within function bodies

For AMPL functions used in AMPL scripts, all commands are allowed in function bodies. 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 are not allowed to have OUT args.

Declaring arguments to functions

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`.

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.

Ackermann's function

Famous example from recursive-function theory:

```
# recursive Ackermann's function
function Acker(m,n)
{
        if m == 0 then return n + 1;
        if n == 0 then return Acker(m-1,1);
        return Acker(m-1, Acker(m,n-1));
        }
```

Recursive param variant of Ackermann's function

```
param nM default 20;
param nN default 200000;
set M = 0 .. nM;
set N = 0 .. nN;
param Ackermann{m in M, n in N} =
        if m == 0 then n + 1
        else if n == 0 then Ackermann[m-1,1]
        else Ackermann[m-1, Ackermann[m,n-1]];

#Ackermann[4,1] = 65533
#Ackermann[4,2] runs out of 32-bit memory
```

Returning sets

```
set A; set B; set C;
function foo(set S, set T) returns set
{ return S union T; }
data; set A := a b c; set B := x y;
display A, B;
display foo(A, B);
```

gives output

```
set A := a b c;
set B := x y;
set foo(A,B) := a b c x y;
```

Soon... Returning tuples as well

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
```

Should we (later) allow other atomic types?

Allowing sets as atoms would permit true sets of sets.

Adding tuples as atoms might help express some scenario trees. We would presumably use square brackets to extract subtuples of tuples.

Rational numbers would permit instantiating some problems without roundoff errors, which might be of use in some mathematical research.

Other possible new atomic types: complex and rational complex.

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.

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.

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 can 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`.

Status

The initial slides in this talk show output on my (dmg's) laptop. Returning tuples, and functions should (I hope!) work soon. Various issues are still outstanding.

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`

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
```