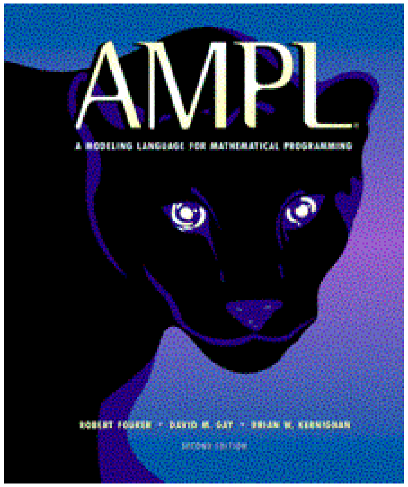
AMPL

A Modeling Language for Large-Scale Optimization



Robert Fourer

AMPL Optimization LLC, www.ampl.com

Department of Industrial Engineering & Management Sciences,
Northwestern University, Evanston, IL 60208-3119, USA

Department of Economics, University of Chicago — 21 April 2005

Development History

Research projects since 1985

Bell Laboratories Computing Sciences Research Center,
David Gay and *Brian Kernighan*

NU IE & MS Department,
National Science Foundation grants,
Robert Fourer

. . . all code after 1987 written by Gay

Lucent Technologies divestiture 1996

Lucent retains Bell Laboratories

Bell Laboratories retains AMPL

Commercialization History

Sold by licensed vendors since 1992

CPLEX Optimization, subsequently ILOG/CPLEX

4-6 much smaller companies, *including in Europe:*

- MOSEK (Denmark)
- OptiRisk Systems (UK)

AMPL Optimization LLC formed 2002

Lucent assigns

vendor agreements, trademark, web domain

Lucent retains

ownership of AMPL and gets a small royalty

. . . two years to negotiate!

Commercialization History *(cont'd)*

Current members of LLC

Fourer, professor at Northwestern

Kernighan, professor at Princeton

Gay, researcher at Sandia National Laboratory

Current situation

Sandia licenses the AMPL source code

. . . another year to negotiate!

AMPL Optimization LLC is gradually arranging to
sell solvers, provide marketing and maintenance

Marketing Strategy

Goals

Clearest *and* most powerful language

Tutorial but comprehensive textbook

Broad base of satisfied users and consultants

Automated benchmarking services

Moderate price

Advantages

Marketing and support can be decentralized

New AMPL company can be expanded gradually

Disadvantages

Not much known about the user base

Development of new features can be hard to coordinate

Market Position

Competition from . . .

Other modeling languages & systems

(AIMMS, MPL, GAMS, LPL)

Proprietary systems of established solver vendors

(ILOG/OPL Studio, Dash/MOSEL, LINGO)

Other software used as a modeling system

(Excel/Frontline, MATLAB/Tomlab)

Outline

The basics: model, data, solution

A simple example

A set-intensive example

Complementarity problems

Stochastic programming

Combinatorial optimization

The NEOS Server

Ex 1: The McDonald's Diet Problem

Foods:

QP	Quarter Pounder
FR	Fries, small
MD	McLean Deluxe
SM	Sausage McMuffin
BM	Big Mac
1M	1% Lowfat Milk
FF	Filet-O-Fish
OJ	Orange Juice
MC	McGrilled Chicken

Nutrients:

Prot	Protein
Iron	Iron
VitA	Vitamin A
Cals	Calories
VitC	Vitamin C
Carb	Carbohydrates
Calc	Calcium

McDonald's Diet Problem Data

	QP	MD	BM	FF	MC	FR	SM	1M	OJ	
Cost	**1.8**	**2.2**	**1.8**	**1.4**	**2.3**	**0.8**	**1.3**	**0.6**	**0.7**	**Need:**
Protein	28	24	25	14	31	3	15	9	1	**55**
Vitamin A	15	15	6	2	8	0	4	10	2	**100**
Vitamin C	6	10	2	0	15	15	0	4	120	**100**
Calcium	30	20	25	15	15	0	20	30	2	**100**
Iron	20	20	20	10	8	2	15	0	2	**100**
Calories	510	370	500	370	400	220	345	110	80	**2000**
Carbo	34	35	42	38	42	26	27	12	20	**350**

Formulation: Too General

Minimize cx

Subject to $Ax = b$

$x \geq 0$

Formulation: Too Specific

$$
\begin{array}{lrrrrrrrrrl}
\textit{Minimize} & 1.84\, x_{QP} & +\, 2.19\, x_{MD} & +\, 1.84\, x_{BM} & +\, 1.44\, x_{FF} & +\, 2.29\, x_{MC} & +\, 0.77\, x_{FR} & +\, 1.29\, x_{SM} & +\, 0.60\, x_{1M} & +\, 0.72\, x_{OJ} & \\
\textit{Subject to} & 28\, x_{QP} & +\, 24\, x_{MD} & +\, 25\, x_{BM} & +\, 14\, x_{FF} & +\, 31\, x_{MC} & +\, 3\, x_{FR} & +\, 15\, x_{SM} & +\, 9\, x_{1M} & +\, 1\, x_{OJ} & \geq 55 \\
 & 15\, x_{QP} & +\, 15\, x_{MD} & +\, 6\, x_{BM} & +\, 2\, x_{FF} & +\, 8\, x_{MC} & +\, 0\, x_{FR} & +\, 4\, x_{SM} & +\, 10\, x_{1M} & +\, 2\, x_{OJ} & \geq 100 \\
 & 6\, x_{QP} & +\, 10\, x_{MD} & +\, 2\, x_{BM} & +\, 0\, x_{FF} & +\, 15\, x_{MC} & +\, 15\, x_{FR} & +\, 0\, x_{SM} & +\, 4\, x_{1M} & +\, 120\, x_{OJ} & \geq 100 \\
 & 30\, x_{QP} & +\, 20\, x_{MD} & +\, 25\, x_{BM} & +\, 15\, x_{FF} & +\, 15\, x_{MC} & +\, 0\, x_{FR} & +\, 20\, x_{SM} & +\, 30\, x_{1M} & +\, 2\, x_{OJ} & \geq 100 \\
 & 20\, x_{QP} & +\, 20\, x_{MD} & +\, 20\, x_{BM} & +\, 10\, x_{FF} & +\, 8\, x_{MC} & +\, 2\, x_{FR} & +\, 15\, x_{SM} & +\, 0\, x_{1M} & +\, 2\, x_{OJ} & \geq 100 \\
 & 510\, x_{QP} & +\, 370\, x_{MD} & +\, 500\, x_{BM} & +\, 370\, x_{FF} & +\, 400\, x_{MC} & +\, 220\, x_{FR} & +\, 345\, x_{SM} & +\, 110\, x_{1M} & +\, 80\, x_{OJ} & \geq 2000 \\
 & 34\, x_{QP} & +\, 35\, x_{MD} & +\, 42\, x_{BM} & +\, 38\, x_{FF} & +\, 42\, x_{MC} & +\, 26\, x_{FR} & +\, 27\, x_{SM} & +\, 12\, x_{1M} & +\, 20\, x_{OJ} & \geq 350
\end{array}
$$

Algebraic Model

Given $\mathcal{F}$, a set of foods

$\mathcal{N}$, a set of nutrients

and $a_{ij} \geq 0$, the units of nutrient i in one serving of food j, for each $i \in \mathcal{N}$ and $j \in \mathcal{F}$

$b_i > 0$, units of nutrient i required, for each $i \in \mathcal{N}$

$c_j > 0$, cost per serving of food j, for each $j \in \mathcal{F}$

Define $x_j \geq 0$, servings of food j to be purchased, for each $j \in \mathcal{F}$

Minimize $\Sigma_{j \in \mathcal{F}} \; c_j \, x_j$

Subject to $\Sigma_{j \in \mathcal{F}} \; a_{ij} \, x_j \geq b_i$, for each $i \in \mathcal{N}$

Algebraic Model in AMPL

```
set NUTR;    # nutrients
set FOOD;    # foods

param amt {NUTR,FOOD} >= 0;    # amount of nutrient in each food
param nutrLow {NUTR} >= 0;     # lower bound on nutrients in diet
param cost {FOOD} >= 0;        # cost of foods

var Buy {FOOD} >= 0 integer;   # amounts of foods to be bought

minimize TotalCost: sum {j in FOOD} cost[j] * Buy[j];

subject to Need {i in NUTR}:
   sum {j in FOOD} amt[i,j] * Buy[j] >= nutrLow[i];
```

Data for the AMPL Model

```
param: FOOD:             cost :=
  "Quarter Pounder"    1.84     "Fries, small"        .77
  "McLean Deluxe"      2.19     "Sausage McMuffin"   1.29
  "Big Mac"            1.84     "1% Lowfat Milk"      .60
  "Filet-O-Fish"       1.44     "Orange Juice"        .72
  "McGrilled Chicken"  2.29 ;

param: NUTR: nutrLow :=
   Prot  55   VitA 100   VitC  100
   Calc 100   Iron 100   Cals 2000   Carb 350 ;

param amt (tr):          Cals  Carb  Prot  VitA  VitC  Calc  Iron :=
   "Quarter Pounder"      510    34    28    15     6    30    20
   "McLean Deluxe"        370    35    24    15    10    20    20
   "Big Mac"              500    42    25     6     2    25    20
   "Filet-O-Fish"         370    38    14     2     0    15    10
   "McGrilled Chicken"    400    42    31     8    15    15     8
   "Fries, small"         220    26     3     0    15     0     2
   "Sausage McMuffin"     345    27    15     4     0    20    15
   "1% Lowfat Milk"       110    12     9    10     4    30     0
   "Orange Juice"          80    20     1     2   120     2     2 ;
```

Continuous-Variable Solution

```
ampl: model mcdiet1.mod;
ampl: data mcdiet1.dat;

ampl: solve;

MINOS 5.5: ignoring integrality of 9 variables
MINOS 5.5: optimal solution found.
7 iterations, objective 14.8557377

ampl: display Buy;

Buy [*] :=
    1% Lowfat Milk   3.42213
           Big Mac   0
      Filet-O-Fish   0
      Fries, small   6.14754
 McGrilled Chicken   0
     McLean Deluxe   0
      Orange Juice   0
   Quarter Pounder   4.38525
   Sausage McMuffin  0
```

Integer-Variable Solution

```
ampl: option solver cplex;

ampl: solve;

CPLEX 7.0.0: optimal integer solution; objective 15.05
41 MIP simplex iterations
23 branch-and-bound nodes

ampl: display Buy;

Buy [*] :=
      1% Lowfat Milk   4
             Big Mac   0
        Filet-O-Fish   1
        Fries, small   5
   McGrilled Chicken   0
       McLean Deluxe   0
        Orange Juice   0
     Quarter Pounder   4
    Sausage McMuffin   0
```

Same for 63 Foods, 12 Nutrients

```
ampl: reset data;
ampl: data mcdiet2.dat;

ampl: option solver minos;
ampl: solve;
MINOS 5.5: ignoring integrality of 63 variables
MINOS 5.5: optimal solution found.
16 iterations, objective -1.786806582e-14

ampl: option omit_zero_rows 1;
ampl: display Buy;
Buy [*] :=
                  Bacon Bits   55
              Barbeque Sauce   50
           Hot Mustard Sauce   50
```

Improved Algebraic Model

```
set NUTR;    # nutrients
set FOOD;    # foods

param nutrLo {NUTR} >= 0;
param nutrHi {i in NUTR} >= nutrLo[i];
                              # requirements for nutrients

param foodCost {FOOD} >= 0;   # costs of foods
param foodLim {FOOD} >= 0;    # limits on food amounts

param amt {NUTR,FOOD} >= 0;   # amounts of nutrient in foods

var Buy {FOOD} integer >= 0, <= foodLim[j];
                              # amounts of foods to be bought

minimize TotalCost: sum {j in FOOD} foodCost[j] * Buy[j];

subject to Need {i in NUTR}:
   nutrLo[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= nutrHi[i];
```

Improved Algebraic Model *(cont'd)*

```
set F_SAL within FOOD;        # Salads
set F_SAL_DRE within FOOD;    # Salad dressings
set F_SAL_TOP within FOOD;    # Salad toppings

param amt_sal_dre {F_SAL} > 0;
param amt_sal_top {F_SAL} > 0;
                   # Limits on dressings & toppings per serving

subject to SaladDressingLimit:
   sum {j in F_SAL_DRE} Buy[j]
      <= sum {j in F_SAL} amt_sal_dre[j] * Buy[j];

subject to SaladToppingLimit:
   sum {j in F_SAL_TOP} Buy[j]
      <= sum {j in F_SAL} amt_sal_top[j] * Buy[j];

set DRINKS within FOOD;     # Drinks
param drinkNum > 0;         # Number of drinks required in diet

subject to DrinkLimit:
   sum {j in DRINKS} Buy[j] = drinkNum;
```

Improved Algebraic Model *(cont'd)*

```
set F_NUG within FOOD;       # Chicken McNuggets foods
set F_NUG_SCE within FOOD;   # Chicken McNuggets sauces

param amt_nug_sce {F_NUG} > 0;
                             # Limits on sauces per serving

subject to NuggetSauceLimit:
   sum {j in F_NUG_SCE} Buy[j]
      <= sum {j in F_NUG} amt_nug_sce[j] * Buy[j];

param fracCalFat >= 0, <= 1;
                  # Fraction of calories that may be from fat

subject to CalFatLimit:
   sum {j in FOOD} amt['CalFat',j] * Buy[j]
      <= fracCalFat * sum {j in FOOD} amt['Cal',j] * Buy[j];
```

Improved Solution

```
ampl: model diet2.mod;
ampl: data diet2.dat;

ampl: option solver cplex;
ampl: solve;

CPLEX 9.0.0: optimal integer solution; objective 9.06
720 MIP simplex iterations
414 branch-and-bound nodes

ampl: option omit_zero_rows 1;
ampl: display Buy;

Buy [*] :=
                     Cheerios  1
                 Cheeseburger  1
           'Chocolate Shake'  1
    'Cinnamon Raisin Danish'  1
                     Croutons  1
            'English Muffin'  1
  'H-C Orange Drink (large)'  1
                    Hamburger  2
              'Orange Juice'  1
                'Side Salad'  1 ;
```

Ex 2: Airline Fleet Assignment

```
set FLEETS;
set CITIES;
set TIMES circular;

set FLEET_LEGS within
   {f in FLEETS, c1 in CITIES, t1 in TIMES,
                 c2 in CITIES, t2 in TIMES: c1 <> c2 and t1 <> t2};

          # (f,c1,t1,c2,t2) represents the availability of fleet f
          # to cover the leg that leaves c1 at t1 and
          # whose arrival time plus turnaround time at c2 is t2

set LEGS = setof {(f,c1,t1,c2,t2) in FLEET_LEGS} (c1,t1,c2,t2);

          # the set of all legs that can be covered by some fleet
```

Airline Fleet Assignment *(cont'd)*

```
set SERV_CITIES {f in FLEETS} =
   union {(f,c1,c2,t1,t2) in FLEET_LEGS} {c1,c2};

         # for each fleet, the set of cities that it serves

set OP_TIMES {f in FLEETS, c in SERV_CITIES[f]} circular by TIMES =
   setof {(f,c,c2,t1,t2) in FLEET_LEGS} t1 union
   setof {(f,c1,c,t1,t2) in FLEET_LEGS} t2;

         # for each fleet and city served by that fleet,
         # the set of active arrival & departure times at that city,
         # with arrival time adjusted for the turn requirement

param leg_cost {FLEET_LEGS} >= 0;
param fleet_size {FLEETS} >= 0;
```

Airline Fleet Assignment *(cont'd)*

```
minimize Total_Cost;

node Gate {f in FLEETS, c in SERV_CITIES[f], OP_TIMES[f,c]};
          # for each fleet and city served by that fleet,
          # a node for each possible time

arc Fly {(f,c1,t1,c2,t2) in FLEET_LEGS} >= 0, <= 1,
   from Balance[f,c1,t1], to Balance[f,c2,t2],
   obj Total_Cost leg_cost[f,c1,t1,c2,t2];
          # arcs for fleet/flight assignments

arc Sit {f in FLEETS, c in SERV_CITIES[f], t in OP_TIMES[f,c]} >= 0,
   from Balance[f,c,t], to Balance[f,c,next(t)];
          # arcs for planes on the ground
```

Airline Fleet Assignment *(cont'd)*

```
subj to Service {(c1,t1,c2,t2) in LEGS}:
   sum {(f,c1,t1,c2,t2) in FLEET_LEGS} Fly[f,c1,t1,c2,t2] = 1;

          # each leg must be served by some fleet

subj to Capacity {f in FLEETS}:
   sum {(f,c1,t1,c2,t2) in FLEET_LEGS:
      ord(t2,TIMES) < ord(t1,TIMES)} Fly[f,c1,t1,c2,t2] +
   sum {c in SERV_CITIES[f]} Sit[f,c,last(OP_TIMES[f,c])] <= fleet_size[f];

          # number of planes used is the number in the air at the
          # last time (arriving "earlier" than they leave)
          # plus the number on the ground at the last time in each city
```

Airline Fleet Assignment Data

```
set FLEETS := 72S 73S L10 ;
set CITIES := ATL CVG DFW ;

set TIMES := 1200a 1210a 1220a 1230a 1240a 1250a
              100a  110a  120a  130a  140a  150a
              200a  210a  220a  230a  240a  250a
              300a  310a  320a  330a  340a  350a

set FLEET_LEGS :=
         (72S,ATL,*,CVG,*)  630a  740a   830a  950a  1210p  130p
         (72S,ATL,*,CVG,*)  120p  240p   430p  600p   640p  810p
         (72S,ATL,*,CVG,*)  850p 1010p  1150p  100a
         (73S,ATL,*,CVG,*)  630a  740a   830a  950a  1210p  130p

param leg_cost :=
         [72S,ATL,*,CVG,*]  630a  740a 33   830a  950a 33  1210p  130p 33
         [72S,ATL,*,CVG,*]  120p  240p 33   430p  600p 33   640p  810p 33
         [72S,ATL,*,CVG,*]  850p 1010p 33  1150p  100a 33
         [73S,ATL,*,CVG,*]  630a  740a 30   830a  950a 30  1210p  130p 30

param fleet_size :=  72S 6  73S 6  L10 2 ;
```

Airline Fleet Assignment Solution

```
ampl: model fleet.mod;
ampl: data fleet.dat;

ampl: option solver kestrel;
ampl: option kestrel_options 'solver pcx';

ampl: option show_stats 1;

ampl: solve;

327 variables, all linear
258 constraints; 790 nonzeros
        211 linear network constraints
        47 general linear constraints
1 linear objective; 116 nonzeros.

Job has been submitted to Kestrel
Kestrel/NEOS Job number    : 458598
Kestrel/NEOS Job password  : lggrLQxk

Check the following URL for progress report :
     http://www-neos.mcs.anl.gov/neos/neos-cgi/
          check-status.cgi?job=458598&pass=lggrLQxk

In case of problems, e-mail :
     neos-comments@mcs.anl.gov

Intermediate Solver Output: ...
```

Airline Fleet Assignment Solution *(cont'd)*

```
Executing algorithm...

Before Scaling: ScaleFactor = 0.0
Cholesky factor will have density  0.11448
  FOUND      5 TINY DIAGONALS; REPLACED WITH INF

Maximum Gondzio corrections = 0

Iter    Primal       Dual      (PriInf  DualInf)  log(mu) dgts   Merit
  0   1.0426e+04   1.7577e+03  (3.1e+00 8.9e-02)    0.77    0   1.8e+01
  FOUND      3 TINY DIAGONALS; REPLACED WITH INF
  1   6.0005e+03   1.8285e+03  (1.5e+00 1.5e-02)    0.30    0   8.7e+00
  FOUND      4 TINY DIAGONALS; REPLACED WITH INF
  2   2.4219e+03   1.9639e+03  (1.3e-01 8.4e-04)   -0.49    0   9.3e-01
  FOUND      3 TINY DIAGONALS; REPLACED WITH INF
  3   2.1674e+03   2.0302e+03  (4.2e-02 1.5e-04)   -1.12    1   2.8e-01
  FOUND      2 TINY DIAGONALS; REPLACED WITH INF
  4   2.0584e+03   2.0393e+03  (4.5e-03 3.9e-05)   -1.73    2   3.8e-02
  FOUND      4 TINY DIAGONALS; REPLACED WITH INF
  5   2.0442e+03   2.0439e+03  (5.5e-05 3.7e-07)   -3.66    3   4.6e-04
  FOUND      1 TINY DIAGONALS; REPLACED WITH INF
  6   2.0440e+03   2.0440e+03  (1.7e-10 8.5e-13)   -9.23    9   1.4e-09

--termination with OPTIMAL status

Finished call

Optimal solution found.
```

Airline Fleet Assignment Solution *(cont'd)*

```
ampl: option display_eps .00001, omit_zero_rows 1, display_1col 100000;

ampl: display {f in FLEETS}:
ampl?    {(f,c1,t1,c2,t2) in FLEET_LEGS} Fly[f,c1,t1,c2,t2];

Fly['72S',c1,t1,c2,t2] :=
CVG 110p  DFW 220p    1
CVG 640p  DFW 800p    1
CVG 850a  DFW 1010a   1
DFW 1050a CVG 200p    1
DFW 440p  CVG 800p    1
DFW 820p  CVG 1140p   1
;

Fly['73S',c1,t1,c2,t2] :=
ATL 1010a DFW 1110a   1
ATL 1010p DFW 1120p   1
ATL 1140p DFW 1250a   1
ATL 1150p CVG 100a    1
ATL 120p  CVG 240p    1
ATL 120p  DFW 230p    1
ATL 1210p CVG 130p    1
ATL 430p  CVG 600p    1
ATL 630a  CVG 740a    1   ...
```

Complementarity Problems

Definition

Collections of complementarity conditions:

- Two inequalities must hold,
  at least one of them with equality

Applications

Equilibrium problems in economics and engineering

Optimality conditions for nonlinear programs,
bi-level linear programs, bimatrix games, . . .

Classical Linear Complementarity

Economic equilibrium

```
set PROD;   # products
set ACT;    # activities

param cost {ACT} > 0;       # cost per unit of each activity
param demand {PROD} >= 0;   # units of demand for each product

param io {PROD,ACT} >= 0;   # units of each product from
                            # 1 unit of each activity

var Price {i in PROD};
var Level {j in ACT};

subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j] * Level[j] >= demand[i];

subject to Lev_Compl {j in ACT}:
   Level[j] >= 0 complements
      sum {i in PROD} Price[i] * io[i,j] <= cost[j];
```

. . . complementary slackness conditions
for an equivalent linear program

Mixed Linear Complementarity

Economic equilibrium with bounded variables

```
set PROD;    # products
set ACT;     # activities

param cost {ACT} > 0;         # cost per unit
param demand {PROD} >= 0;     # units of demand

param io {PROD,ACT} >= 0;     # units of product per unit of activity

param level_min {ACT} > 0;    # min allowed level for each activity
param level_max {ACT} > 0;    # max allowed level for each activity

var Price {i in PROD};
var Level {j in ACT};

subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j] * Level[j] >= demand[i];

subject to Lev_Compl {j in ACT}:
   level_min[j] <= Level[j] <= level_max[j] complements
      cost[j] - sum {i in PROD} Price[i] * io[i,j];
```

. . . complementarity conditions
for optimality of an equivalent* bounded-variable *linear program

Nonlinear Complementarity

Economic equilibrium with price-dependent demands

```
set PROD;    # products
set ACT;     # activities

param cost {ACT} > 0;         # cost per unit
param demand {PROD} >= 0;     # units of demand

param io {PROD,ACT} >= 0;     # units of product per unit of activity

param demzero {PROD} > 0;     # intercept and slope of the demand
param demrate {PROD} >= 0;    # as a function of price

var Price {i in PROD};
var Level {j in ACT};

subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j] * Level[j]
         >= demzero[i] + demrate[i] * Price[i];

subject to Lev_Compl {j in ACT}:
   Level[j] >= 0 complements
      sum {i in PROD} Price[i] * io[i,j] <= cost[j];
```

. . . not equivalent to a linear program

Operands to `complements`: always 2 inequalities

Two single inequalities

single-ineq1 **`complements`** *single-ineq2*

Both inequalities must hold, at least one at equality

One double inequality

double-ineq **`complements`** *expr*
expr **`complements`** *double-ineq*

The double-inequality must hold, and

if at lower limit then $expr \geq 0$,
if at upper limit then $expr \leq 0$,
if between limits then $expr = 0$

One equality

equality **`complements`** *expr*
expr **`complements`** *equality*

The equality must hold ***(included for completeness)***

Solvers

“Square” systems

of variables =
  # of complementarity constraints +
  # of equality constraints

Transformation to a simpler canonical form required

MPECs

Mathematical programs with equilibrium constraints

No restriction on numbers of variables & constraints

Objective functions permitted

. . . solvers continuing to emerge

Stochastic Programs

Extensions within AMPL *(proposed)*

Allow random distributions for some problem data

Make distributions available to solvers

Extensions using AMPL *(substantially implemented)*

Add special expressions and conventions for stages & scenario trees

Compile to standard AMPL

Generate problem descriptions for various solvers

- SAMPL
- StAMPL

Random Entities

Distributions set in the model

```
param avail_mean >= 0;
param avail_var >= 0;
param avail {1..T} random
   := Normal (avail_mean, avail_var);
```

Distributions assigned as data

```
param mktbas {PROD} >= 0;
param grow_min {PROD} >= 0;
param grow_max {PROD} >= 0;
var Market {PROD,1..T} random;
.......
let {p in PROD} Market[p,1] := mktbas[p];
let {p in PROD, t in 2..T} Market[p,t] :=
   else Market[p,t-1] + Uniform (grow_min[p], grow_max[p]);
```

Parameters or Variables?

Modeled like "random" parameters

Specify distributions in place of fixed data values

Instantiate the same model with different distributions

Processed like "defined" variables

Save a symbolic definition rather than a specific sample

Record in expression tree passed to solver driver

Evaluate (sample) as directed by solver

New Expression Types

Discrete distributions

```
Discrete (1/3, 20, 1/3, 50, 1/3, 175)
Discrete ( {s in SCEN} (prob[s],demand[s]) )
```

Stochastic objectives

Default: expected value of objective

Explicit: using functions **`Expected_Value`** and **`Variance`**

Further Concerns

Modeling

Recourse variables indicated by user-defined **`.stage`** suffix

Chance constraints defined by
new function **`Probability (`***logical-expression***`)`**

Processing

For **`Discrete`**, **`Uniform`**, and other (half-) bounded distributions,
AMPL's presolve phase may eliminate constraints.

Jacobian entries indicate
which constraints involve which random entities

SAMPL

Patrick Valente, Gautam Mitra, Mustapha Sadki

Brunel University, Uxbridge, Middlesex, UK

- P. Valente, G. Mitra, M. Sadki and R. Fourer, "Extending Algebraic Modelling Languages for Stochastic Programming" (2004).

SAMPL

Stages

Two-stage recourse model

```
suffix stage IN;

var x {Prod, Fact, t in Time, Scen} >=0,
                          suffix stage if t = 1 then 1 else 2;

var y {Prod, Fact, t in Time, Scen} >=0,
                          suffix stage if t = 1 then 1 else 2;

.......
```

Multi-stage recourse model

```
suffix stage IN;

var x {Prod, Fact, t in Time, Scen} >=0, suffix stage t;

var y {Prod, Fact, t in Time, Scen} >=0, suffix stage t;

.......
```

SAMPL

Scenarios

Scenario set

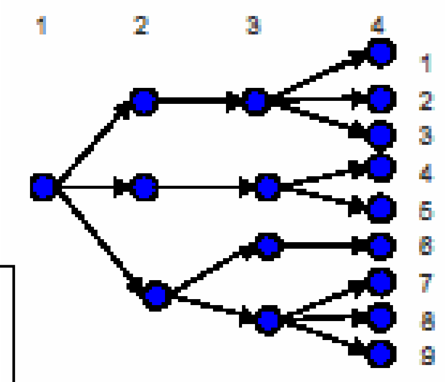
```
param NS > 1;

scenarioset Sc = 1..NS;
```

Scenario probabilities

```
probability param Pr {Sc} = 1 / card(Sc);   # uniform case
```

SAMPL

Scenario Trees: General Form



Bundle form

```
tree FourStageExample :=
bundles {
   (1,1),
   (2,1),(2,4),(2,6),
   (3,1),(3,4),(3,6),(3,7),
   (4,1),(4,2),(4,3),(4,4),(4,5),(4,6),(4,7)(4,8),(4,9)
};
```

Treelist form (if scenario paths never cross)

```
tree FourStageEXample := tlist {1,4,4,2,4,2,3,4,4};
```

SAMPL

Scenario Trees: Standard Forms

Uniform branching at each stage

```
multibranch (n1,n2,..., nST);
```

Uniform branching at all stages

```
nway (n);
```

Binary at all stages

```
binary;
```

Two-stage

```
twostage;
```

Random Parameters

Declaration

```
random param d {Time,Scen} >=0;  # demand
```

Compact data

```
random param dem :=

   1 1  10.0
   2 1   5.0
   2 3  15.0
   3 1   2.5
   3 2   7.5
   3 3   7.5
   3 4  22.5 ;
```

Expanded data

```
random param dem (tr):=

      1   2    3

   1  10   5   2.5

   2  10   5   7.5

   3  10  15   7.5

   4  10  15  22.5 ;
```

SAMPL

Chance Constraints

Random parameter in constraint

```
random param d {Prod,Deal,Time,Scen} >= 0  # demand

subj to satisfy_demand {j in Prod, k in Deal, t in Time, s in Scen}:

   sum {i in Fact} z[j,i,k,t,s] = d[j,k,t,s];
```

Conversion to a chance constraint

```
param beta := 0.9;

chance {j in Prod, k in Deal, t in Time,s in Scen}

   satisfy_demand[j,k,t,s] >= beta;
```

Further Issues

Communication with solvers

Need a standard form for communicating
stochastic programming instances to solvers

Existing “SMPS” form is outdated and inadequate
(and not entirely standard)

Communication with scenario generators

Independent generators:
consistency with model must be ensured somehow

Integrated generator:
modeling language calls generator as needed

Integration in a modeling environment

SPInE: scenario generation, modeling, solving, results analysis

StAMPL

Leo Lopes

PhD, Northwestern University;
Assistant Professor, University of Arizona

- R. Fourer and L. Lopes, "StAMPL: A Filtration-Oriented Modeling Tool for Stochastic Programming" (2003).

StAMPL

Main Features

Non-technical

Fewer indexes

Modular

Fewer conditions

Technical

No non-anticipativity constraints

Recourse and technology matrices are apparent at first inspection

```
definestage 1;

set INSTR;
var Buy{INSTR} >= 0;

param initial_wealth;

subject to InvestAll:
     sum{i in INSTR} Buy[i] = initial_wealth;

##################################################
definestage 2..(stages()-1);

set INSTR;
var Buy{INSTR} >= 0;

param return{INSTR};

subject to ReinvestAll:
     sum{i in INSTR} parent().Buy[i]*return[i] =
     sum{i in INSTR} Buy[i];

##################################################
definestage stages();

set INSTR;
var Shortage >= 0;
var Overage >= 0;

param shortage_penalty;
param overage_reward;
     check: shortage_penalty > overage_reward;
param return{INSTR};
param goal;

maximize Final_Wealth:
     overage_reward*Overage -
shortage_penalty*Shortage;

subject to ReinvestAll:
     sum{i in INSTR} parent().Buy[i]*return[i]
     + Shortage - Overage = goal;
```

StAMPL

Syntax: Stage Definition

***definestage** statement*

Similar to the AMPL **problem** statement

```
definestage 1;

set INSTR;
var Buy{INSTR} >= 0;

param initial_wealth;

subject to InvestAll:
     sum{i in INSTR} Buy[i] = initial_wealth;

####################################################
definestage 2..(stages()-1);

set INSTR;
var Buy{INSTR} >= 0;

param return{INSTR};

subject to ReinvestAll:
     sum{i in INSTR} parent().Buy[i]*return[i] =
     sum{i in INSTR} Buy[i];

####################################################
definestage stages();

set INSTR;
var Shortage >= 0;
var Overage >= 0;

param shortage_penalty;
param overage_reward;
     check: shortage_penalty > overage_reward;
param return{INSTR};
param goal;

maximize Final_Wealth:
     overage_reward*Overage -
shortage_penalty*Shortage;

subject to ReinvestAll:
     sum{i in INSTR} parent().Buy[i]*return[i]
     + Shortage - Overage = goal;
```

Syntax: Connecting Problems

`parent()` *function*

Returns a model object

All components of the parent model can be accessed using the `'.'` (dot) operator

```
definestage 1;

set INSTR;
var Buy{INSTR} >= 0;

param initial_wealth;

subject to InvestAll:
     sum{i in INSTR} Buy[i] = initial_wealth;

##################################################
definestage 2..(stages()-1);

set INSTR;
var Buy{INSTR} >= 0;

param return{INSTR};

subject to ReinvestAll:
     sum{i in INSTR} parent().Buy[i]*return[i] =
     sum{i in INSTR} Buy[i];

##################################################
definestage stages();

set INSTR;
var Shortage >= 0;
var Overage >= 0;

param shortage_penalty;
param overage_reward;
     check: shortage_penalty > overage_reward;
param return{INSTR};
param goal;

maximize Final_Wealth:
     overage_reward*Overage -
shortage_penalty*Shortage;

subject to ReinvestAll:
     sum{i in INSTR} parent().Buy[i]*return[i]
     + Shortage - Overage = goal;
```

Syntax: Stage Information

`stage()` *and* `stages()` *functions*

Return the number of the current stage and the total stages

Uses:

- Defining stages
- Discounting
- Multi-period
- constraints

```
definestage 1;

set INSTR;
var Buy{INSTR} >= 0;

param initial_wealth;

subject to InvestAll:
     sum{i in INSTR} Buy[i] = initial_wealth;

##################################################
definestage 2..(stages()-1);

set INSTR;
var Buy{INSTR} >= 0;

param return{INSTR};

subject to ReinvestAll:
     sum{i in INSTR} parent().Buy[i]*return[i] =
     sum{i in INSTR} Buy[i];

##################################################
definestage stages();

set INSTR;
var Shortage >= 0;
var Overage >= 0;

param shortage_penalty;
param overage_reward;
     check: shortage_penalty > overage_reward;
param return{INSTR};
param goal;

maximize Final_Wealth:
     overage_reward*Overage -
shortage_penalty*Shortage;

subject to ReinvestAll:
     sum{i in INSTR} parent().Buy[i]*return[i]
     + Shortage - Overage = goal;
```

Scenario Trees

Issues

Trees don't fit well with AML paradigms

Generating scenario trees is very specialized

Usually involves developing special routines

Often demands specialized software

Conclusion

Write a general-purpose language library (C++)

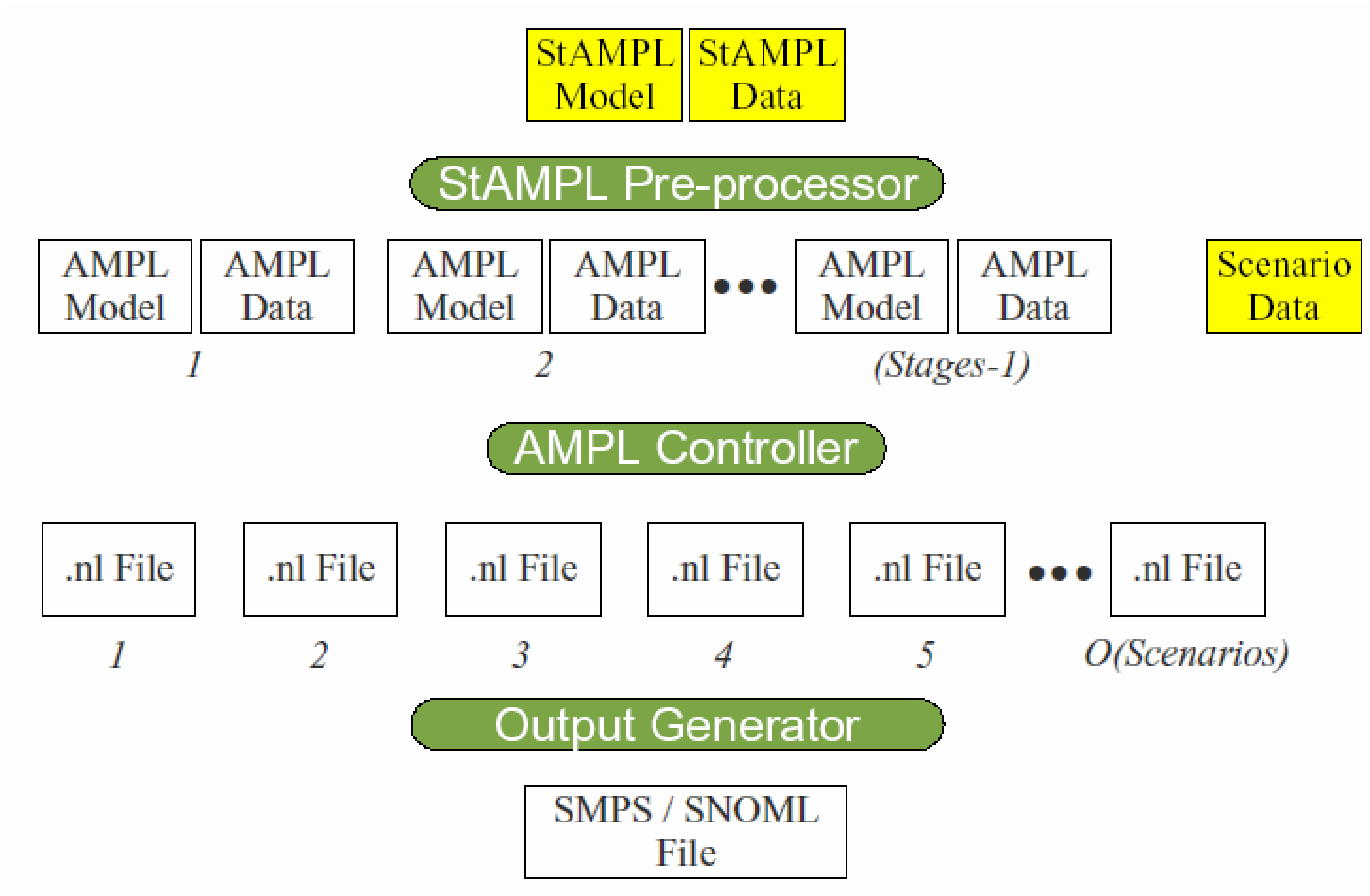
- *The library generates an intermediary file*

Export the library to each necessary environment using

- *SWIG*
- *XML-RPC, .NET*
- *System-specific libraries*

StAMPL

How It Works



Combinatorial Optimization

Formulations

More natural for modelers than integer programs

Independent of solvers

Compatible with existing modeling languages

Solution methods

Theoretically optimal

Based on tree search (like branch & bound)

Sensitive to details of search strategy

Combinatorial

Example: Job Sequencing with Setups

Given

A set of jobs, with
production times, due times and earliness penalties

One machine that processes one job at a time

Setup costs and times between jobs

Precedence relations between certain jobs

Choose

A sequence for the jobs

Minimizing

Setup costs plus earliness penalties

C. Jordan & A. Drexl, A Comparison of Constraint and Mixed Integer Programming Solvers for Batch Sequencing with Sequence Dependent Setups. *ORSA Journal on Computing* **7** (1995) 160–165.

Example: Variables and Costs

Either way

`ComplTime[j]` is the completion time of job `j`

Earliness penalty is the sum over jobs `j` of

```
duePen[j] * (dueTime[j] - ComplTime[j])
```

Integer programming formulation

`Seq[i,j]` = 1 iff `i` immediately precedes `j`

Setup cost is the sum over job pairs `(i,j)` of

```
setupCost[i,j] * Seq[i,j]
```

More natural formulation

`JobForSlot[k]` is the job in the `k`th slot in sequence

Setup cost is the sum over slots `k` of

```
setupCost[JobForSlot[k],JobForSlot[k+1]]
```

Example: Production Constraints

Integer programming formulation

For each job **i**, **ComplTime[i] ≤ dueTime[i]**

For each job pair **(i,j)**,

```
ComplTime[i] + setupTime[i,j] + procTime[j] ≤
ComplTime[j] + BIG * (1 - Seq[i,j])
```

More natural formulation

For each slot **k**,

```
ComplTime[JobForSlot[k]] = min (
  dueTime[JobForSlot[k]],
  ComplTime[JobForSlot[k+1]]
    - procTime[JobForSlot[k+1]]
    - setupTime[JobForSlot[k],JobForSlot[k+1]] )
```

Example: Sequencing Constraints

Integer programming formulation

For each job **`i`**,

```
sum {j in JOBS} Seq[i,j] = 1
```

For each job **`i`**,

```
sum {j in JOBS} Seq[j,i] = 1
```

More natural formulation

```
all_different {k in SLOTS} JobForSlot[k]
```

Combinatorial

Representing "Range" Constraints

General format

lower-bound ≤ *linear-expr* + *nonlinear-expr* ≤ *upper-bound*

Arrays of *lower-bound* and *upper-bound* values

Coefficient lists for *linear-expr*

Expression tree for *nonlinear-expr*

Expression tree nodes

Variables, constants

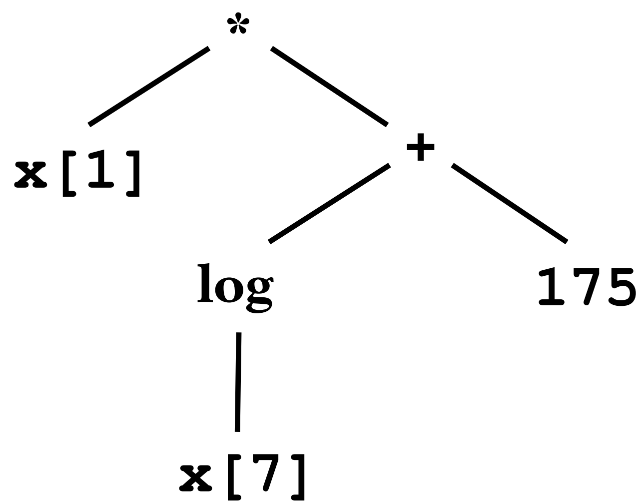
Binary, unary operators

Iterated summation, min, max

Piecewise-linear terms

If-then-else terms

. . . ***single array of variables***



Combinatorial

"Walking the Tree"

Example: AMPL interface to ILOG Concert

Definition of variables

```
IloNumVarArray Var(env, n_var);

for (j = 0; j < n_var - n_var_int; j++)
   Var[j] = IloNumVar(env, loVarBnd[j], upVarBnd[j], ILOFLOAT);

for (j = n_var - n_var_int; j < n_var; j++)
   Var[j] = IloNumVar(env, loVarBnd[j], upVarBnd[j], ILOINT);
```

Combinatorial

Tree Walk *(cont'd)*

Top-level processing of constraints

```
IloRangeArray Con(env, n_con);
for (i = 0; i < n_con; i++) {
   IloExpr conExpr(env);
   if (i < nlc)
      conExpr += build_expr (con_de[i].e);
   for (cg = Cgrad[i]; cg; cg = cg->next)
      conExpr += (cg -> coef) * Var[cg -> varno];
   Con[i] = (loConBnd[i] <= conExpr <= upConBnd[i]);
}
```

Combinatorial

Tree Walk *(cont'd)*

Tree-walk function for expressions

```
IloExpr build_expr (expr *e)
{
   expr **ep;
   IloInt opnum;
   IloExpr partSum;
   opnum = (int) e->op;
   switch(opnum) {
      case PLUS_opno: ...
      case MINUS_opno: ...
      .......
   }
}
```

Tree Walk *(cont'd)*

Tree-walk cases for expression nodes

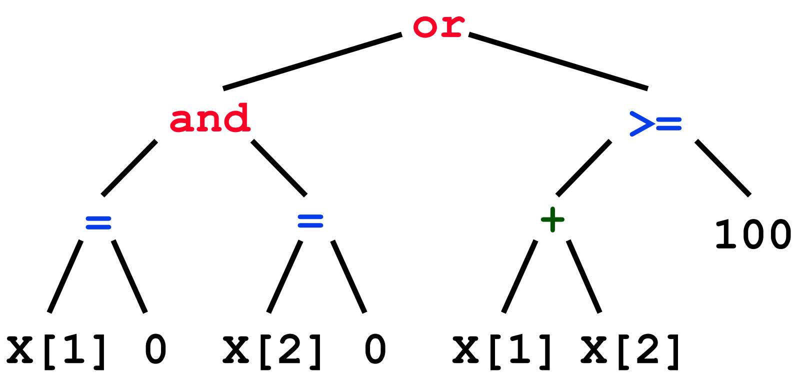
```
switch(opnum) {

 case PLUS_opno:
    return build_expr (e->L.e) + build_expr (e->R.e);

 case SUMLIST_opno:
    partSum = IloExpr(env);
    for (ep = e->L.ep; ep < e->R.ep; *ep++)
       partSum += build_expr (*ep);
    return partSum;

 case LOG_opno:
    return IloLog (build_expr (e->L.e));

 case CONST_opno:
    return IloExpr (env, ((expr_n*)e)->v);

 case VAR_opno:
    return Var[e->a];

 .......

}
```

Combinatorial

Logical Constraints

Simple forms

constraint **and** *constraint*
constraint **or** *constraint*
not *constraint*




```
(X[1] = 0 and X[2] = 0) or X[1] + X[2] >= 100
```

Representation

Expression tree for entire constraint

Constraint nodes whose children are constraint nodes

Constraint nodes whose children are expression nodes

Combinatorial

Tree Walk *(cont'd)*

Tree-walk function for constraints

```
IloConstraint build_constr (expr *e)
{
   expr **ep;
   IloInt opnum;

   opnum = (int) e->op;

   switch(opnum) {

      .......

   }
}
```

Combinatorial

Tree Walk *(cont'd)*

Tree-walk cases for constraint nodes

```
switch(opnum) {
 case OR_opno:
    return build_constr (e->L.e) || build_constr (e->R.e);
 case AND_opno:
    return build_constr (e->L.e) && build_constr (e->R.e);
 case GE_opno:
    return build_expr (e->L.e) >= build_expr (e->R.e);
 case EQ_opno:
    return build_expr (e->L.e) == build_expr (e->R.e);
.......
}
```

Combinatorial

Further Logical Constraint Cases

Constraint types

Counting expressions and constraints

Structure (global) constraints

Variables in subscripts

Solver inputs

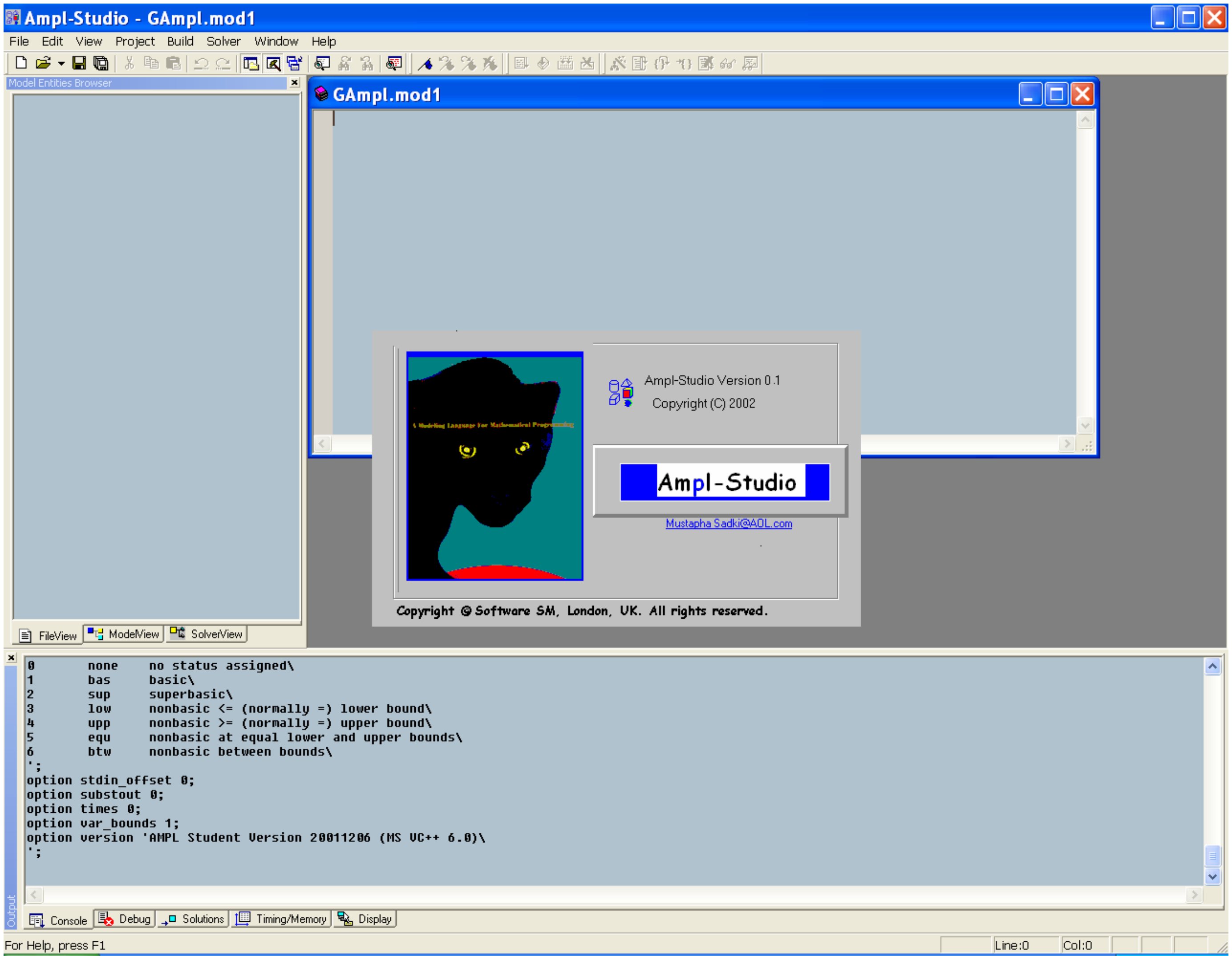
C++ types and operators (ILOG Concert)

Unindexed algebraic input format (BARON)

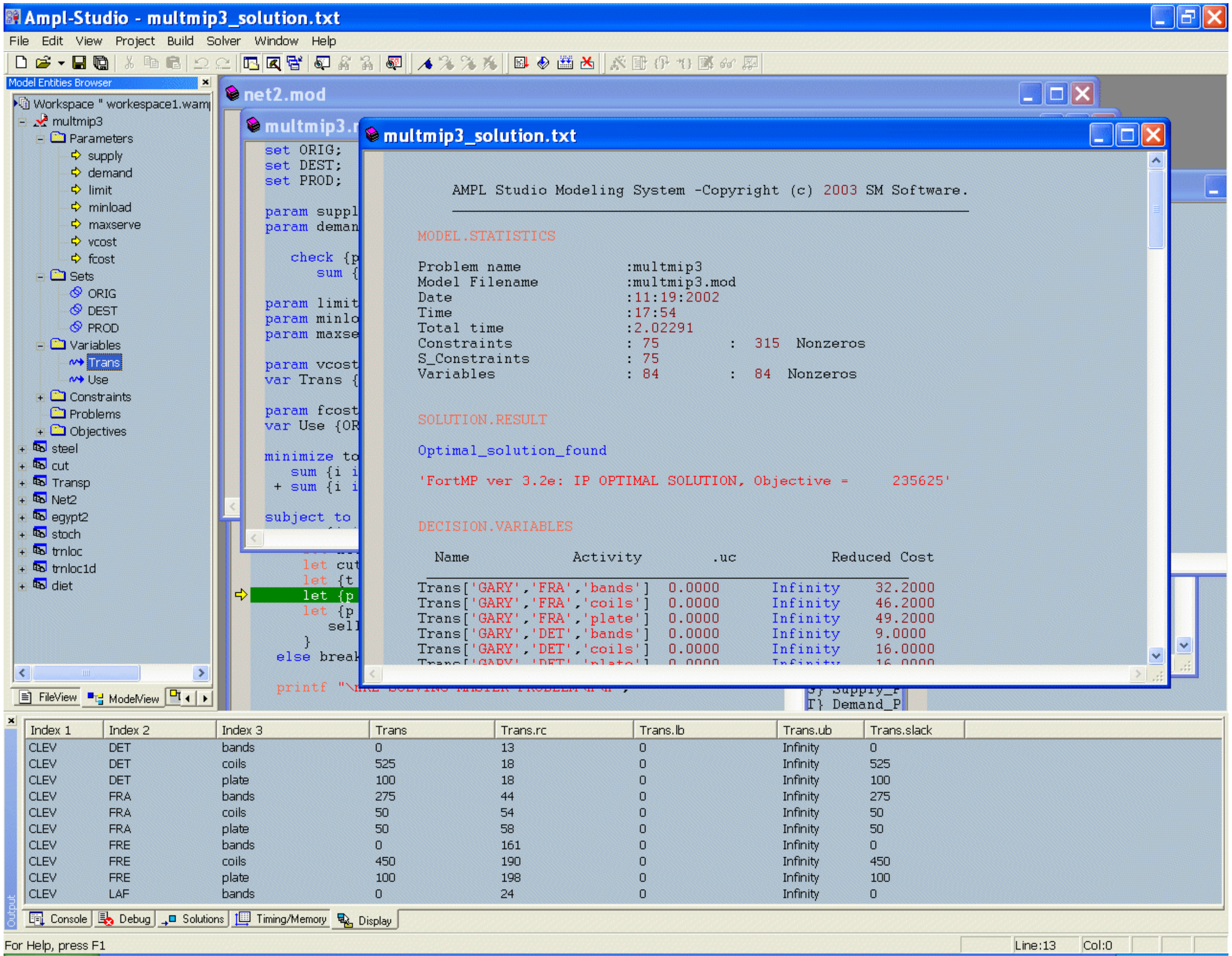
Codelist of 4-tuples (GlobSol)

Compact, flexible NOP format (GLOPT)

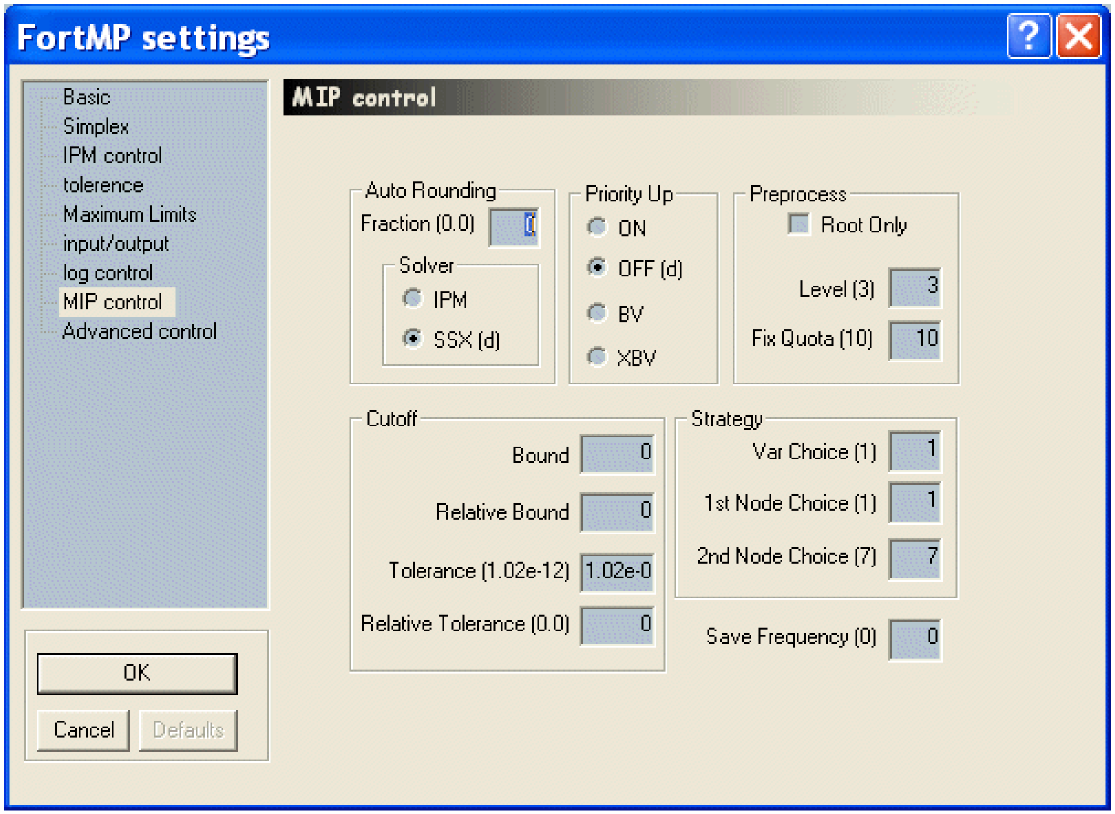
AMPL Studio / Optirisk Systems Ltd.



AMPL Studio *(continued)*



AMPL Studio *(continued)*



AMPL Studio / **Optirisk Systems Ltd.**

Windows IDE for AMPL

Manage projects, edit files

Set solver options and solve

View results

Run command scripts

COM objects for AMPL

Embed AMPL in applications

CONOPT / ARKI Consulting & Development A/S

Local optimization of smooth nonlinear problems

- Large and sparse problems
- Highly nonlinear functions

Multi-method architecture

- Extended generalized reduced gradient method
- Special phase 0
- Linear mode iterations
- Sequential linear programming
- Sequential quadratic programming

. . . can take advantage of 2nd derivatives

KNITRO / Ziena Optimization Inc.

For all smooth nonlinear optimization problems

Interior-point / barrier

- KNITRO/InteriorCG (handles large/dense Hessians)
- KNITRO/InteriorDirect (handles ill-conditioned problems)

Active-set SLQP (new October 2004!)

- KNITRO/Active (good for warm starts)

Trust-region approach

Supported by global convergence theory

Numerous options

1st or 2nd derivatives, exact or approximated

Feasibility of iterates

KNITRO Interfaces

C/C++/Fortran

Easily integrated within existing applications
via callable library

AMPL (or GAMS)

Flexible and powerful syntax

Derivatives computed automatically

Focus on modeling and analysis of results

Ideal for prototyping

MATLAB (through TOMLAB)

Excel (through Frontline Systems solver)

KNITRO Derivative Options

First derivative options

User or modeling language provides exact derivatives

KNITRO computes finite difference derivatives (forward or centered)

Second derivative options

User or modeling language provides exact derivatives

User or modeling language
provides exact Hessian-vector products

KNITRO computes Hessian-vector products

Dense quasi-Newton (BFGS or SR1)

Limited-memory BFGS

KNITRO Feasible Option

Concepts

By default constraints may be violated
during the optimization process

Feasible option enforces feasibility
with respect to inequalities,
given initial point satisfying inequalities

Advantages

Constraints may be undefined outside feasible region

Allows early termination with feasible solution

AMPL Solver Support

Full list at www.ampl.com/solvers.html

Linear programming: MINOS, PCx

Linear & linear integer programming: CPLEX, FortMP, lp_solve, MINTO, MOSEK, SOPT, XA, Xpress-MP

Quadratic & convex programming: LOQO, OOQP

Quadratic & quadratic integer programming: CPLEX, FortMP, MOSEK, OOQP, Xpress-MP

Differentiable nonlinear programming: CONOPT, DONLP2, IPOPT, KNITRO, LOQO, MINOS, SNOPT

Nondifferentiable and global nonlinear programming: ACRS, CONDOR, MINLP

Complementarity: PATH

Problem analysis: MProbe

NEOS **www-neos.mcs.anl.gov/neos/**

A general-purpose optimization server

- Over 45 solvers in all
  - Linear, linear network, linear integer
  - Nonlinear, nonlinear integer, nondifferentiable & global
  - Stochastic, semidefinite, semi-infinite, complementarity
- Commercial as well as experimental solvers
- Central scheduler with distributed solver sites

A research project

- Currently free of charge
- Supported through the Optimization Technology Center
  of Northwestern University & Argonne National Laboratory

. . . 4402 submissions last week
. . . as many as 11906 submissions in a week

AMPL Solver Support . . . via NEOS

Full list at www-neos.mcs.anl.gov/neos/server-solver-types.html

Linear programming: MINOS, PCx

Linear & linear integer programming: FortMP, MINTO, MOSEK

Quadratic & convex programming: LOQO, OOQP

Quadratic & quadratic integer programming: FortMP, OOQP

Differentiable nonlinear programming:
FILTER, IPOPT, KNITRO, LANCELOT, LOQO, MINOS, SNOPT

Nondifferentiable and global nonlinear programming:
ACRS, CONDOR, MINLP, MLOCPSOA

Complementarity: PATH

Design

Flexible architecture

Central controller and scheduler machine

Distributed solver sites

Numerous formats

Low-level formats: MPS, SIF, SDPA

Programming languages: C/ADOL-C, Fortran/ADIFOR

High-level modeling languages: AMPL, GAMS

Varied submission options

E-mail – **Web forms** – **Direct function call**

TCP/IP socket-based submission tool: Java or tcl/tk

. . . more in next week's presentation