IFORS 2026 - Vienna

AMPL Presolve Configuration

Marcos Dominguez,
Gleb Belov,
Bob Fourer,
Filipe Brandao

{marcos,gleb,4er,fdabrandao}@ampl.com

Agenda

0 | Scope: AMPL presolve and reformulation controls

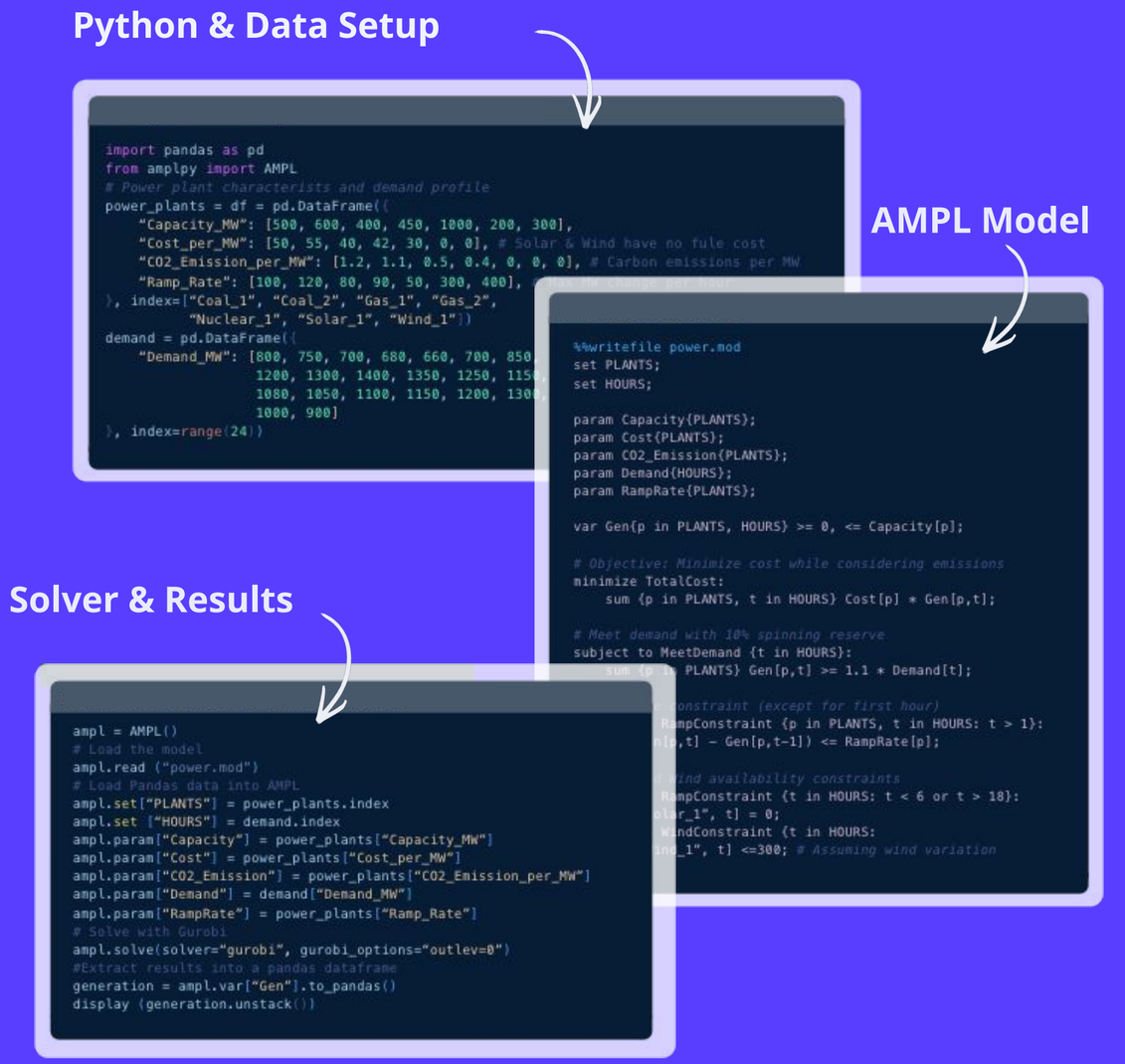
1 | Defined variables: model maintainability and efficiency

2 | Automatic reformulation settings

3 | Numerical tolerances in expressions

What is AMPL?

An optimization modeling system to formulate, solve, and scale decision problems.



AMPL Workflow

AMPL/MP Solver Library: high-level expressions supported

Combinatorial and logical

- Conditional operators: `if-then-else; ==> [else], <== [else], <==>`
- Logical operators: `or, and, not; exists, forall`
- Piecewise-linear functions: `abs; min, max; <<breakpoints; slopes>>`
- Counting operators: `count; atmost, atleast, exactly; numberof`
- Relational and comparison operators: `>(=), <(=), (!)=; alldiff`
- Complementarity operator: `complements`
- Set membership: `in`

Nonlinear

- Nonlinear operators and functions: `*, /, ^; exp, log;` trigonometric, hyperbolic
- High-order polynomials
- Conic algebra

Much more than expressions...

AMPL Presolve and Reformulation Options

The following controls apply:

- **Generic presolve and reformulation controls (AMPL compiler):**
  a. Presolve: `presolve, presolve_[int/fix]eps[max], constraint_drop_tol`, …
  b. Reformulations: `pl_linearize, pl_bigM`
  c. Substitutions: `linelim, substout`

- **Solver-specific controls (in MP-based drivers):**
  a. MP presolve: `cvt:pre:all, cvt:pre:feastol[rel], cvt:pre:eqresult`, …
  b. Reformulations: `cvt:socp, cvt:pre:prod, cvt:bigM`, …
    - Expression **type-specific controls**: `acc:abs, acc:max, …, cvt:compl[:tol]`
  a. Substitutions: `cvt:dvelim, cvt:pre:unnest, cvt:expr:nlassign`, …
  b. Solver-specific: `presolve, scale, aggregate, cut:*`, …

Independent vs Defined vars

Defined vs Independent variables

```
set ATTRIBUTES;
set SAMPLES;
param X {ATTRIBUTES, SAMPLES};
param Y {SAMPLES};
# Independent variables
var WEIGHT {ATTRIBUTES} ⩾ 0;

# Defined variables
var ERROR_SAMPLE {d in SAMPLES} = Y[d] - sum {b in ATTRIBUTES} (WEIGHT[b] * X[b, d]);
var REGULARIZER =                          # Another defined variable
   sum {b in ATTRIBUTES}
      (WEIGHT[b] - 1 / card(ATTRIBUTES)) ^ 2 / 5500;

minimize TotalError:
    sum {d in SAMPLES} ERROR_SAMPLE[d]^2 + REGULARIZER;
```

Defined vs Independent variables

```
set ATTRIBUTES;
set SAMPLES;
param X {ATTRIBUTES, SAMPLES};
param Y {SAMPLES};
# Independent variables
var WEIGHT {ATTRIBUTES} ≥ 0;

# Defined variables
var ERROR_SAMPLE {d in SAMPLES} =  Y[d] - sum {b in ATTRIBUTES} (WEIGHT[b] * X[b, d]);
var REGULARIZER =                          # Another defined variable
   sum {b in ATTRIBUTES}
      (WEIGHT[b] - 1 / card(ATTRIBUTES)) ^ 2 / 5500;

minimize TotalError:
    sum {d in SAMPLES} ERROR_SAMPLE[d]^2 + REGULARIZER;
```

Defined vs Independent variables

```
set ATTRIBUTES;
set SAMPLES;
param X {ATTRIBUTES, SAMPLES};
param Y {SAMPLES};
# Independent variables
var WEIGHT {ATTRIBUTES} ⩾ 0;

# Defined variables
var ERROR_SAMPLE {d in SAMPLES} = Y[d] - sum {b in ATTRIBUTES} (WEIGHT[b] * X[b, d]);
var REGULARIZER =                          # Another defined variable
   sum {b in ATTRIBUTES}
      (WEIGHT[b] - 1 / card(ATTRIBUTES)) ^ 2 / 5500;

minimize TotalError:
    sum {d in SAMPLES} ERROR_SAMPLE[d]^2 + REGULARIZER;
```

Defined vs Independent variables

```
set ATTRIBUTES;
set SAMPLES;
param X {ATTRIBUTES, SAMPLES};
param Y {SAMPLES};
# Independent variables
var WEIGHT {ATTRIBUTES} ⩾ 0;

# Defined variables
var ERROR_SAMPLE {d in SAMPLES} = Y[d] - sum {b in ATTRIBUTES} (WEIGHT[b] * X[b, d]);
var REGULARIZER =                          # Another defined variable
   sum {b in ATTRIBUTES}
      (WEIGHT[b] - 1 / card(ATTRIBUTES)) ^ 2 / 5500;

minimize TotalError:
    sum {d in SAMPLES} ERROR_SAMPLE[d]^2 + REGULARIZER;
```

Defined variables: automatic substitution

Defined variables are substituted in linear and quadratic expressions for solving, often resulting in more efficient models.

- AMPL option `linelim`
- MP solver option `cvt:dvelim`

Experiment with the above model (over 1000 attributes and samples and 1 additional linear constraint) with commercial MIP solvers: times in seconds

	Solver 1	Solver 2	Solver 3
Default (substituted)	0.11 s	0.60 s	1.17 s
cvt:dvelim=0	10.82 s	1.87 s	3.39 s

Independent variables can be substituted using AMPL option substout (non-default).

Avoid substitution

Variables that should not be substituted are best declared as independent:

```
var weight {ASSETS} >=0 <=1;                # Asset weights
var factor_exposure {FACTORS};              # Leave independent
var portfolio_variance =
           sum {f in FACTORS} factor_exposure[f]^2
           + sum {a in ASSETS}
              asset_specific_risks[a] * weight[a]^2;

maximize Utility:
    portfolio_return - risk_free_rate / 2.0 * portfolio_variance;

subject to FactorExposureDefinition {f in FACTORS}:
    factor_exposure[f] ==
        sum {i in ASSETS} weight[i] * factor_loadings[i, f];
```

https://colab.ampl.com/tags/factor-model.html

Defined variables: solution exploration

```
var flow_p_out {p in POOLS} =
    sum {t in TARGETS: (p,t) in POOLS_TARGETS}
        flow_p2t[p,t];
var flow_s2p {(s,p) in SOURCES_POOLS} =
    prop_s2p[s,p] * flow_p_out[p];
```

The values of defined variables are available for browsing:

```
ampl.solve(solver=”scip”, scip_options=”outlev=1”)
df = ampl.getData('flow_s2p').toPandas().unstack()
df = df[df['flow_s2p'] > 1e-6]
display(df)
```

Automatic reformulations

A basic case: indicator

```
var b binary;
s.t. Indicator: b==1 ==> 5*x[1] + 2*x[2] <= 0;
```

- Important: tight bounds on the compared expression, e.g.:

```
var {1..2} x >= 0 <= 15;
```

➔ Bounds can be set only on independent variables!

- Linearized when not supported, or when desired (MP option `acc:indle=0`):

```
s.t. BigM: 5*x[1] + 2*x[2] <= 105*(1-b);
```

Directly supported by many solvers

Disjunction: ||, exists

```
subject to Disjunction:
   (level_A <= 5) || (level_B >= 10);
```

Reformulation for a typical MIP solver:

```
var aux_result {1..3}: binary;
s.t. Term1: aux_result[1] ==> level_A <= 5;
s.t. Term2: aux_result[2] ==> level_B >= 10;
s.t. Disj__:
   aux_result[3] ==> (aux_result[1] || aux_result[2]);
s.t. FixResult: aux_result[3] <==> True;
```

Indicators and OR are linearized if necessary (or desired, options `acc:indle=0, acc:indge=0, acc:or=0`).

Paintshop color change scheduling problem

Different solvers might prefer own ways to receive these constraints!

```
s.t. ColorMAXSpec {s in SLOTS,
                   p in SKID_TYPE_PATTERNS[slot_skid_type[s]]:
                   color_spec_MAX[p] in COLORS_MAX}:
    (slot_pattern[s] == p) ==>                                  # Pattern allocated, then …
        (exists {cm in COLOR_MAX_MACHINES}
            (slot_color_spec_MAX[s, cm] == color_spec_MAX[p]
                && slot_color_MAX_machine_available[s, cm]));
```

Paintshop color change scheduling problem

Different solvers might prefer own ways to receive these constraints!

```
s.t. ColorMAXSpec {s in SLOTS,
                   p in SKID_TYPE_PATTERNS[slot_skid_type[s]]:
                   color_spec_MAX[p] in COLORS_MAX}:
    (slot_pattern[s] == p) ==>                                  # Pattern allocated, then …
        (exists {cm in COLOR_MAX_MACHINES}
            (slot_color_spec_MAX[s, cm] == color_spec_MAX[p]
                && slot_color_MAX_machine_available[s, cm]));
```

Paintshop color change scheduling problem

Different solvers might prefer own ways to receive these constraints!

```
s.t. ColorMAXSpec {s in SLOTS,
                   p in SKID_TYPE_PATTERNS[slot_skid_type[s]]:
                   color_spec_MAX[p] in COLORS_MAX}:
    (slot_pattern[s] == p) ==>                                  # Pattern allocated, then …
        (exists {cm in COLOR_MAX_MACHINES}
            (slot_color_spec_MAX[s, cm] == color_spec_MAX[p]
                && slot_color_MAX_machine_available[s, cm]));
```

Paintshop color change scheduling problem

Different solvers might prefer own ways to receive these constraints!

```
s.t. ColorMAXSpec {s in SLOTS,
                   p in SKID_TYPE_PATTERNS[slot_skid_type[s]]:
                   color_spec_MAX[p] in COLORS_MAX}:
    (slot_pattern[s] == p) ==>                                  # Pattern allocated, then …
        (exists {cm in COLOR_MAX_MACHINES}
            (slot_color_spec_MAX[s, cm] == color_spec_MAX[p]
                && slot_color_MAX_machine_available[s, cm]));
```

Reformulation options

A paintshop color change scheduling problem with 5 pattern types, 4 skid types, and 50 slots, tested with two major MIP solvers.

	Solver 1	Solver 2
High-level (default)	3.87 s	4.22 s
Full linearization	3.17 s	6.70 s

AMPL MP uses a solver's high-level modeling by default.

Performance difference can increase on larger instances.

https://colab.ampl.com/tags/color-change-scheduling.html

Numerical tolerances

Numerical tolerances in expressions

```
var pool_outflow {p in POOLS} =
   sum {t in TARGETS: (p,t) in P2T} flow_p2t[p,t];
var source_pool_flow {(s,p) in S2P} =
   prop_s2p[s,p] * pool_outflow[p];
minimize Objective: ... +
   (... if source_pool_flow[s, p]>0 then pipe_setup_cost);
```

What is the meaning of

`source_pool_flow[s, p]>0 ?`

https://colab.ampl.com/tags/pooling.html

Objective value evaluation in AMPL

```
MP2NL 0.1: Knitro 15.1.0: Terminated by user.
objective 3673625.459; optimality gap Infinity
```

Objective value evaluation in AMPL

```
MP2NL 0.1: Knitro 15.1.0: Terminated by user.
objective 3673625.459; optimality gap Infinity

>>> ampl.get_value('_obj[1]')

3575296.7552816207
```

Objective value evaluation in AMPL

```
MP2NL 0.1: Knitro 15.1.0: Terminated by user.
objective 3673625.459; optimality gap Infinity

>>> ampl.get_value('_obj[1]')

3575296.7552816207

>>> ampl.get_value('min {(s,p) in S2P: source_pool_flow[s,p]>0} source_pool_flow[s,p]')

5.088304459676217e-11
```

What is the meaning of

```
source_pool_flow[s, p]>0 ?
```

Suggested workaround

In the model:

```
param eval_eps {i in 0..1} = 0.01*i;

minimize Objective {i in 0..1}:
   … + if (flow > 0 + eval_eps[i])
       then pipe_setup_cost;
```

In Python:

```
ampl.solve('Objective[0]', solver='mp2nl')
ampl.get_value('Objective[1]');
```

Conclusions

01 Presolve options may affect heavily the performance of your model

02 Reformulate is not easy, and solvers behave differently

03 High-level expressions provide more room for experimentation while increasing readability

04 Understand your model. What does `x>0` actually mean?

References

- AMPL Colaboratory: colab.ampl.com
- AMPL MP Library: mp.ampl.com
- AMPL options: https://dev.ampl.com/ampl/reference/options.html
- Solver options: https://dev.ampl.com/solvers/

(marcos@ampl.com)

Thank You!

Do you have any questions?

Free for Academia

ampl.com/academia

academia@ampl.com

Visit our booth!